pylablib Documentation

Release 1.2.1

Alexey Shkarin

Oct 07, 2021

Contents:

PyLabLib aims to provide support for device control and experiment automation. It interfaces with lots of different *devices*, including several different *camera interfaces*, *translational stages*, *oscilloscopes*, *AWGs*, *sensors*, and more. The interface is implemented in a natural way through Python objects, and is easy to understand. For example, here is a complete script which steps *Thorlabs KDC101* stage by 10000 steps ten times, and each time grabs a frame with *Andor iXon camera*:

```
from pylablib.devices import Thorlabs, Andor  # import the device libraries
import numpy as np  # import numpy for saving

# connect to the devices
with Thorlabs.KinesisMotor("27000000") as stage, Andor.AndorSDK2Camera() as cam:
   # change some camera parameters
   cam.set_exposure(50E-3)
   cam.set_roi(0, 128, 0, 128, hbin=2, vbin=2)
   # start the stepping loop
   images = []
   for _ in range(10):
      stage.move_by(10000)  # initiate a move
      stage.wait_move()  # wait until it's done
      img = cam.snap()  # grab a single frame
      images.append(img)

np.array(images).astype("<u2").tofile("frames.bin")  # save frames as raw binary
```

The list of the devices is constantly expanding.

Additional utilities are added to simplify data acquisition, storage, and processing:

- Simplified data processing utilities: convenient *fitting*, *filtering*, *feature detection*, *FFT* (mostly wrappers around NumPy and SciPy).
- Universal multi-level *dictionaries* which are convenient for *storing* heterogeneous data and settings in human-readable format.
- Assorted functions for dealing with *file system* (creating, moving and removing folders, zipping/unzipping, path normalization), *network* (simplified interface for client and server sockets), *strings* (conversion of various Python objects to and from string), and more.
- Tools for GUI generation and advanced multi-threading built on top of Qt5 *(still in development stage: the documentation is incomplete, and the interfaces can change in later versions)*

The library only works on Python 3, and has been most extensively tested on Windows 10 with 64-bit Python. Linux is, in principle, supported, but devices which require manufacturer-provided DLLs (mostly cameras) might, potentially, have problems.

Note: This is documentation for the newer **1.x** version of the library. The older **0.x** documentation can be found at https://pylablib-v0.readthedocs.io/en/latest/ .

CHAPTER 1

Related projects

Pylablib cam-control - software for universal camera control and camera data acquisition.

1.1 Installation

1.1.1 Standard install

You can install the library from PyPi:

```
pip install pylablib
```

If you already have it installed, you can upgrade it to get the newest version:

```
pip install -U pylablib
```

This will install the full set of dependencies: basic dependencies and computing packages (`numpy`, `scipy`, `pandas`, `numba`, `rpyc`), basic device communication packages (`pyft232`, `pyvisa`, `pyserial`, `pyusb`), and PyQt5-based GUI (`pyqt5` and `pyqtgraph`). You can also install additional device library dependencies (`nidaqmx` and `websocket-client`) using the extra requirements feature of pip:

```
pip install -U pylablib[devio-full]
```

1.1.2 Minimal install

In case you do not want some of these packages installed, or they are unavailable on your platform, you can install a lightweight version of pylablib called `pylablib-lightweight`. It contains exactly the same code, but has only the most basic dependencies (`numpy`, `scipy`, and `pandas`):

```
pip install -U pylablib-lightweight
```

With this, the basic functionality (such as data processing or file IO) will work, but more advanced features such as device communication and GUI, will require additional packages. In most cases, the raised errors will notify which packages are missing. These can be installed either manually, or using the extra requirements:

- `[extra]` extra packages used in some situations: `numba` (speeds up some data processing) and `rpyc` (communication between different PCs)
- `[devio]` basic devio packages: `pyft232`, `pyvisa`, `pyserial`, and `pyusb`
- `[devio-extra]` additional devio packages: `nidaqmx` and `websocket-client`
- `[gui-pyqt5]` PyQt5-based GUI: `pyqt5` and `pyqtgraph`. Should not be used together with `[gui-pyside2]`
- `[gui-pyside2]` PySide2-based GUI: `pyside2` and `pyqtgraph`. Should not be used together with `[gui-pyqt5]`

The options can be combined. For example,

```
pip install pylablib-lightweight[extra,devio,gui-pyside2]
```

installs the dependencies as the usual pylablib distribution, but with PySide2 Qt5 backend instead of PyQt5.

1.1.3 Anaconda install

The package is also available on Anaconda via `conda-forge` channel. To install it, run

```
conda install -c conda-forge pylablib
```

in the Anaconda prompt.

The Anaconda version of pylablib comes with all the standard dependencies except for `pyft232` , `nidaqmx` and `websocket-client`, which are not available on `conda-forge` channel. This means, that *Thorlabs APT/Kinesis*, *NI DAQs*, and some functionality of *M2 Solstis laser* are not accessible. To use those, it is recommended to either install those packages explicitly via `pip` (keep in mind that it can break Anaconda environment), or use a standalone Python distribution.

1.1.4 Usage

To access to the most common features simply import the library:

```python
import pylablib as pll
# Create a parameter dictionary (e.g., for some processing script)
parameters = pll.Dictionary({"par/x":1, "par/y":2, "par/z":[3,4,5], "out":"result"})
pll.save_dict(parameters, "parameters.dat")  # save parameters to a text file
```

More advanced features (e.g., *device communication*) should be imported directly:

```python
from pylablib.devices import Andor  # import Andor devices module
cam = Andor.AndorSDK2Camera()  # connect to Andor SDK2 camera (e.g., iXon)
cam.set_exposure(0.1)  # set exposure to 100ms
frame = cam.snap()  # grab a single frame
cam.close()  # close the connection
```

1.1.5 Dependencies and requirements

The basic package dependencies are NumPy for basic computations and overall array interface, SciPy for advanced computations (interpolation, optimization, special functions), and pandas for heterogeneous tables (`DataFrame`). In addition, it is recommended to have Numba package to speed up some computations. Finally, if you use options for remote computing and communication between different PCs, you need to install RPyC. Note that when installed directly from pip, `numpy` comes with the OpenBLAS version of the linear algebra library; if other version (e.g., Intel MKL) is preferred, it is a good idea to have `numpy` already installed before installing pylablib.

The main device communication packages are PyVISA and pySerial, which cover the majority of devices. Several devices (e.g., *Thorlabs Kinesis* and *Attocube ANC 350*) require additional communication packages: pyft232 and PyUSB. Finally, some particular devices completely or partially rely on specific packages: NI-DAQmx for *NIDAQ* and websocket-client for additional *M2 Solstis* functionality.

Finally, GUI and advanced multi-threading relies on Qt5, which has two possible options. The first (default) option is PyQt5 with sip for some memory management functionality. Note that while newer PyQt5 versions `>=5.11` already come with `PyQt5-sip`, older versions require a separate `sip` installation. Hence, if you use an older `PyQt5` version, you need to install `sip` separately. The second possible Qt5 option is PySide2 with shiboken2. Both PyQt5 and PySide2 should work equally well, and the choice mostly depends on what is already installed, because having both PyQt5 and PySide2 might lead to conflicts. Finally, plotting relies on pyqtgraph, which, starting with version 0.11m is compatible with both PySide2 and PyQt5.

The package has been tested with Python 3.6 through 3.9, and is incompatible with Python 2. The last version officially supporting Python 2.7 is 0.4.0. Furthermore, testing has been mostly performed on 64-bit Python. This is the recommended option, as 32-bit version limitations (most notably, limited amount of accessible RAM) mean that it should only be used when absolutely necessary, e.g., when some required packages or libraries are only available in 32-bit version.

1.1.6 Installing from GitHub

The most recent and extensive, but less tested and documented, version of this library is available on GitHub at https://github.com/AlexShkarin/pyLabLib/. There are several versions of installing it:

- Install using pip using GitHub as a library source:

  ```
  pip install -U git+https://github.com/AlexShkarin/pyLabLib.git
  ```

- Download it as a zip-file and unpack it into any appropriate place (can be folder of the project you're working on, Python `site-packages` folder, or any folder added to `PATH` or `PYTHONPATH` variable).

  To download the code of a specific version, you can choose it in the dropdown *Branch* menu under *Tags* tab. This is the same code as available on PyPi.

  Keep in mind that, unlike the first method, the required packages will not be automatically installed, so this has to be done manually:

  ```
  pip install numpy scipy pandas numba rpyc
  pip install pyft232 pyvisa pyserial pyusb nidaqmx websocket-client
  pip install pyqt5 pyqtgraph
  ```

- Clone the repository to your computer In order to easily get updates in order to easily get updates. For that, you need to install Git (https://git-scm.com/), and use the following commands in the command line (in the folder where you want to store the library):

  ```
  git clone https://github.com/AlexShkarin/pyLabLib
  cd ./pyLabLib
  ```

Whenever you want to update to the most recent version, simply type

```
git pull
```

in the library folder. Keep in mind that any changes that you make to the library code might conflict with the new version that you pull from GitHub, so you should not modify anything in this folder if possible.

1.1.7 Support and feedback

If you have any issues, suggestions, or feedback, you can either raise an issue on GitHub at https://github.com/AlexShkarin/pyLabLib/issues, or send an e-mail to pylablib@gmail.com.

1.2 Devices overview

Basic concepts are described at the *general device communication page*.

Currently supported devices:

- *Cameras*
  - *Andor SDK2* and *Andor SDK3*: variety of Andor (currently part of Oxford Instruments) cameras. Tested with Andor iXon, Luca, and Zyla.
  - *DCAM*: Hamamatsu cameras. Tested with Hamamatsu Orca Flash 4.0 and ImagEM.
  - *NI IMAQ*: National Instruments frame grabbers. Tested with NI PCI-1430 and PCI-1433 frame grabbers together with PhotonFocus MV-D1024E camera.
  - *NI IMAQdx*: National Instruments universal camera interface. Tested with Ethernet-connected PhotonFocus HD1-D1312 camera.
  - *Photon Focus*: Photon Focus pfcam interface. Tested with PhotonFocus MV-D1024E camera connected through either NI frame grabbers (PCI-1430 and PCI-1433) or Silicon Software frame grabbers (microEnable IV AD4-CL).
  - *PCO SC2*: PCO cameras. Tested with pco.edge cameras with CLHS and regular CameraLink interfaces.
  - *Silicon Software*: Silicon Software frame grabbers. Tested with microEnable IV AD4-CL frame grabbers together with PhotonFocus MV-D1024E camera.
  - *Thorlabs Scientific Cameras*: Thorlabs sCMOS cameras. Tested with Thorlabs Kiralux camera.
  - *Uc480*: multiple cameras, including simple Thorlabs and IDS cameras. Tested with IDS SC2592R12M and Thorlabs DCC1545M.
- *Stages*
  - *Attocube ANC300* and *Attocube ANC350*: most common Attocube positioner controllers. Tested with Ethernet and USB connection for ANC300, and USB connection for ANC350.
  - *Thorlabs APT/Kinesis*: basic Thorlabs motorized stages and optomechanics devices. Tested with KDC101 and K10CR1, as well as MFF101 and FW102 (described at a *different page*)
  - *Newport Picomotor*: precision piezo-actuated screws based on slip-stick principle. Tested with Newport 8742 Picomotor driver using Ethernet or USB connection.
  - *Arcus Performax*: fairly common single- and multi-axis motor controllers sold under different brands: Arcus, Nippon Pulse America, or Newmark Systems. Tested with PMX-4EX device with USB connection.

   - *Trinamic*: universal motor controllers and drivers. Tested with a single-axis TMCM-1110 controller with USB connection.
   - *SmarAct*: high-performance piezo sliders. So far only simple open-loop *SCU controllers* are supported. Tested with a standard HCU controller unit.
- *Basic sensors*
   - *HighFinesse*: laser wavelength meters. Tested with WS6 and WS7 USB-controlled devices.
   - *Ophir*: optical power and energy meters. Tested with Ophir Vega.
   - *Lakeshore*: temperature sensors. Tested with Lakeshore 218.
   - *Pfeiffer*: pressure gauges. Tested with TPG261 and DPG202 controllers.
- Lasers
   - *Basic lasers*
      * *Lighthouse Photonics SproutG*
      * *Laser Quantum Finesse*
   - *M2 Solstis laser*
   - *Toptica iBeam Smart laser*
- *Tektronix oscilloscopes*. Tested with TDS2002B, TDS2004B, and DPO2004B.
- *NI DAQs*. Tested with NI USB-6008, NI USB-6343, and NI PCIe-6323.
- *Generic AWGs*. Tested with Agilent 33500 and 33220A, Rigol DG1022, Tektronix AFG1022, GW Instek AFG2225 and AFG2115, and RS Comp AFG21005.
- *Miscellaneous Thorlabs devices*: *MFF101/102* motorized flip mirror mount, *FW102/212* motorized filter wheel, and *MDT693/694* high-voltage source.
- *Miscellaneous OZOptics devices*: *EPC04* fiber polarization controller, *DD100* motorized fiber attenuator, and *TF100* motorized fiber filter.
- *Miscellaneous devices*
   - *Conrad relay board*
   - *Basic Arduino communication*

1.2.1 Basics of device communication

The devices are represented as Python objects. In most cases, one object controls one device, although sometimes one object can be responsible for multiple interconnected devices (e.g., when daisy-chaining of several devices is used, as in *Picomotor stage*). All the device control functions are contained within the class. Occasionally, there are auxiliary function present for listing available devices, dealing with data generated by the device, or adjusting global parameters.

Note: Some specific devices functionality might not be completely covered in the current release. If this is the case for your device, you can let the developers know by raising an issue on GitHub, or sending an e-mail to pylablib@gmail.com.

Connection

The device identifier or address needs to be provided upon the device object creation, after which it is automatically connected. Getting the address usually depends on the kind of device:

- Simple message-style devices, such as AWG, oscilloscopes, sensors and gauges, require an address which depends on the exact connection protocol. For example, serial devices addresses look like `"COM1"`, Visa addresses as `"USB0::0x1313::0x8070::000000::INSTR"`, and network addresses take IP and, possibly, port `"192.168.1.3:7230"`. To get the list of all connected devices, you can run *comm_backend.list_backend_resources()*:

```
>> import pylablib as pll
>> pll.list_backend_resources("serial")  # list serial port resources
['COM38', 'COM1', 'COM36', 'COM3']
>> pll.list_backend_resources("visa")  # note that, by default, visa also␣
↪includes all the COM ports
('USB0::0x1313::0x8070::000000::INSTR',
'ASRL1::INSTR',
'ASRL3::INSTR',
'ASRL10::INSTR',
'ASRL36::INSTR',
'ASRL38::INSTR')
```

  Network devices do not easily provide such functionality (and there are, in principle, many unrelated devices connected to the network), so you might need to learn the device IP elsewhere. Usually, it is set on the device front panel or using some kind of configuration tool and a different connection, such as serial or USB.

  In most cases, the connection address is all you need. However, sometimes the connection might require some additional information. The most common situations are ports for the network connection and baud rates for the serial connections. Ports can be supplied either as a part of the string `"192.168.1.3:7230"`, or as a tuple `("192.168.1.3", 7230)`. The baud rates are, similarly, provided as a tuple: `("COM1", 19200)`. By default, the devices would use the baud rate which is most common for them, but in some cases (e.g., if the device baud rate can be changed), you might need to provide it explicitly. If it is provided incorrectly, then no communication can be done, and requests will typically return a timeout error:

```
>> from pylablib.devices import Ophir
>> meter = Ophir.VegaPowerMeter("COM3")  # for this power meter 9600 baud are␣
↪used by default
>> meter.get_power()  # let us assume that the devices is currently set up with␣
↪38400 baud
...
OphirBackendError: backend exception: 'timeout during read'
>> meter.close()  # need to close the connection before reopening
>> meter = Ophir.VegaPowerMeter(("COM3",38400))  # explicitly specifying the␣
↪correct baud rate
>> meter.get_power()
1E-6
```

- More complicated devices using custom DLLs (usually cameras or some translation stages) will have more unique methods of addressing individual devices: serial number, device index, device ID, etc. In most cases such devices come with `list_devices` or `get_devices_number` functions, which give the necessary information.

After communication is done, the connection needs to be closed, since in most cases it can only be opened in one program or part of the script at a time. It also implies that usually it's impossible to connect to the device while its manufacturer software is still running.

The devices have `open` and `close` methods, but they can also work in together with Python `with` statements:

```
# import Thorlabs device classes
from pylablib.devices import Thorlabs

# connect to FW102 motorized filter wheel
wheel = Thorlabs.FW("COM1")
# set the position
wheel.set_position(1)
# close the connection (until that it's impossible to establish a different␣
↪connection to this device)
wheel.close()

# a better approach
with Thorlabs.FW("COM1") as wheel: # connection is closed automatically when leaving␣
↪the with-block
    wheel.set_position(1)
```

Because the devices are automatically connected on creation, `open` method is almost never called explicitly. It is generally only used to reconnect to the device after the connection has been previously closed, although in this case creating a new device object would work just as well.

Operation

The devices are controlled by calling their methods; attributes and properties are very rarely used. Effort is made to maintain consistent naming conventions, e.g., most getter-methods will start with `get_` and setter methods with `set_` or `setup_` (depending on the complexity of the method). It is also common for setter methods to return the new value as a result, which is useful in CLI operation and debugging. Devices of the same kind have the same names for similar or identical functions: most stages have `move_by`, `jog` and `stop` methods, and cameras have `wait_for_frame` and `read_multiple_images` methods. Whenever it makes sense, these methods will also have the same signatures.

Asynchronous operation and multi-threading

For simplicity of usage and construction, devices interfaces are designed to be synchronous and single-threaded. Asynchronous operation can be achieved by explicit usage of Python multi-threading. Furthermore, the device classes are not designed to be thread safe, i.e., it is not recommended to use the same device simultaneously from two separate threads. However, non-simultaneous calling of device methods from different threads (synchronized, e.g., using locks) or simultaneous usage of several separate devices of the same class is supported.

Error handling

Errors raised by the devices are usually specific to the device and manufacturer, e.g., *AttocubeError* or *TrinamicError*. These can be obtained from the module containing the device class, or from the class itself as `Error` attribute:

```
>> from pylablib.devices import Attocube
>> atc = Attocube.ANC300("192.168.1.1")
>> atc.disable_axis(1)
>> atc.move_by(1,10)   # move on a disabled axis raises an error for ANC300
...
AttocubeError: Axis in wrong mode
>> try:
..     atc.move_by(1,10)
```

(continues on next page)

(continued from previous page)

```
.. except atc.Error:  # could also write "except Attocube.AttocubeError"
..     print("Can not move")
Can not move
```

All of the device errors inherit from `DeviceError`, which in turn is a subclass of `RuntimeError`. Therefore, one can also use those exception classes instead:

```
>> import pylablib as pll
>> try:
..     atc.move_by(1,10)
.. except pll.DeviceError:
..     print("Can not move")
Can not move
```

Getting more information

A lot of information about the devices can be gained just from their method names and descriptions (docstrings). There are several ways of getting these:

- In many cases your IDE (PyCharm, Spyder, VS Code with installed Python extension) supports code inspection. In this case, the list of methods will usually pop up after you time the device object name and a dot (such as `cam.`), and the method docstring will show up after you type the method name and parenthesis (such as `cam.get_roi(`). However, sometimes it might take a while for these pop-ups to show up.
- You can use console, such as Jupyter QtConsole, Jupyter Notebook, or a similar console built into the IDE. Here the list of methods can be obtained using the autocomplete feature: type name of the class or object with a dot (such as `cam.`) and then press `Tab`. The list of all methods should appear. To get the description of a particular class or method, type it with a question mark (such as `cam?` or `cam.get_roi?`) and execute the result (`Enter` or `Shift-Enter`, depending on the console). A description should appear with the argument names and the description.
- You can also use the auto-generated documentation within this manual through the search bar: simply type the name of the class or the method (such as `AndorSDK3Camera` or `AndorSDK3Camera.get_roi`) and look through the results. However, the formatting of the auto-generated documentation might be a bit overwhelming.

Universal settings access

All devices have `get_settings` and `apply_settings` methods which, correspondingly, return Python dictionaries with the most common settings or take these dictionaries and apply the contained settings. These can be used to easily store and re-apply device configuration within a script.

Additionally, there is `get_full_info` method, which returns as complete information as possible. It is particularly useful to check the device status and see if it is connected and working properly, and to save the devices configuration when acquiring the data. Finally, the settings can also be accessed through `.dv` attribute, which provides dictionary-like interface:

```
>>> wheel = Thorlabs.FW("COM1")  # connect to FW102 motorized filter wheel
>>> wheel.get_position()
1
>>> wheel.get_settings()
{'pcount': 6,
'pos': 1,
'sensors_mode': 'off',
'speed_mode': 'high',
```

(continues on next page)

(continued from previous page)

```
'trigger_mode': 'in'}
>>> wheel.dv["pos"]
1
>>> wheel.apply_settings({"pos":2})
>>> wheel.get_position()
2
>>> wheel.dv["pos"] = 3
>>> wheel.get_position()
3
>>> wheel.close()
```

By default not all information is shown, as it can take long time (up to several seconds) to obtain it, and it takes a lot of space on the screen. To get a full set of parameters, you can call `get_full_info(-10)` (`-10` here specifies the parameter priority threshold, which is 0 by default, and has the lowest value of `-10`):

```
>> cam = IMAQdx.IMAQdxCamera()
>> cam.get_full_info()
{   'roi': (0, 1312, 0, 1082),
    'acquisition_in_progress': False,
    'frames_status': TFramesStatus(acquired=0, unread=0, skipped=0, buffer_size=0),
    'cls': 'IMAQdxCamera',
    'conn': 'cam0',
    'detector_size': (1312, 1082),
    'device_info': TDeviceInfo(vendor='Photonfocus AG', model='HD1-D1312-80-G2-12',␣
↪serial_number='0000000000000000', bus_type='Ethernet')   }
>>
>> cam.get_full_info(-10)
{   'roi': (0, 1312, 0, 1082),
    'acquisition_in_progress': False,
    'frames_status': TFramesStatus(acquired=0, unread=0, skipped=0, buffer_size=0),
    'camera_attributes': Dictionary('AcquisitionAttributes/AdvancedEthernet/
↪BandwidthControl/ActualPeakBandwidth': 1000.0
        ... lots and lots of attributes
    'OffsetX': 0
    'OffsetY': 0
    'PayloadSize': 1419584
    'PixelFormat': Mono8
    'Width': 1312),
    'cls': 'IMAQdxCamera',
    'conn': 'cam0',
    'detector_size': (1312, 1082),
    'device_info': TDeviceInfo(vendor='Photonfocus AG', model='HD1-D1312-80-G2-12',␣
↪serial_number='0000000000000000', bus_type='Ethernet')   }
```

Dependencies and external software

Many devices require external software not provided with this package.

The simpler devices using serial connection (either with an external USB-to-Serial adapter, or with a similar built-in chip) only need the corresponding drivers: either standard adapter drivers or the ones supplied by the manufacturer, e.g., via Thorlabs APT software. If the device already shows up as a serial communication port in the OS, no additional software is normally needed. Similarly, devices using Ethernet connection do not need any external software, as long as they are properly connected to the network. Finally, devices using Visa connection require NI VISA Runtime, which is freely available from the National Instruments website. See also PyVISA documentation for details.

Devices which require manufacturer DLLs are harder to set up. For most of them, at the very least, you need to install the manufacturer-provided software for communication. Frequently it already includes the necessary libraries, which means that nothing else is required. However, sometimes you would need to download either an additional SDK package, or DLLs directly from the website. Since these libraries take a lot of space and are often proprietary, they are not distributed with the pylablib.

Note that DLLs can have 32-bit and 64-bit version, and this version should agree with the Python version that you use. Unless you have a really good reason to do otherwise, it is strongly recommended to use 64-bit Python, which means that you would need 64-bit DLLs, which is the standard in most cases these days. To check your Python bitness, you can read the prompt when running the Python console, or run `python -c "import platform; print(platform.architecture()[0])"` in the command line.

In addition, you need to provide pylablib with the path to the DLLs. In many cases it checks the standard locations such as the default `System32` folder (used, e.g., in DCAM or IMAQ cameras), paths contained on the `PATH` environment variable, or defaults paths for manufacturer software (such as `C:/Program Files/Andor SOLIS` for Andor cameras). If the software path is different, or if you choose to obtain DLLs elsewhere, you can also explicitly provide path by setting the library parameter:

```
import pylablib as pll
pll.par["devices/dlls/andor_sdk3"] = "D:/Program Files/Andor SOLIS"
from pylablib.devices import Andor
cam = Andor.AndorSDK3Camera()
```

All of these requirements are described in detail for the specific devices.

Starting from Python 3.8 the DLL search path is changed to not include the files contained in `PATH` environment variable and in the script folder. By default, this behavior is still emulated when pylablib searches for the DLLs, since it is required in some cases (e.g., Photon Focus pfcam interface). If needed, it can be turned off (i.e., switched to the new default behavior of Python 3.8+) by setting `pll.par["devices/dlls/add_environ_paths"]=False`.

Advanced examples

Connecting to a Cryomagnetics LM500 level meter and reading out the level at the first channel:

```
from pylablib.devices import Cryomagnetics  # import the device library
with Cryomagnetics.LM500("COM1") as lm:
    level = lm.get_level(1)  # read the level
```

Stepping the M Squared laser wavelength and recording an image from the Andor iXon camera at each step:

```
with M2.Solstis("192.168.1.2", 34567) as laser, Andor.AndorSDK2Camera() as cam:  #␣
↪connect to the devices
    # change some camera parameters
    cam.set_exposure(50E-3)
    cam.set_roi(0, 128, 0, 128, hbin=2, vbin=2)
    cam.setup_shutter("open")
    # start camera acquisition
    wavelength = 770E-9  # initial wavelength (in meters)
    images = []
    cam.start_acquisition()
    while wavelength < 780E-9:
        laser.coarse_tune_wavelength(wavelength)  # tune the laser frequency (using␣
↪coarse tuning)
        time.sleep(0.5)  # wait until the laser stabilizes
        cam.wait_for_frame()  # ensure that there's a frame in the camera queue
        img = cam.read_newest_image()
        images.append(img)
        wavelength += 0.5E-9
```

Available devices

- *Cameras*
  - *Andor SDK2* and *Andor SDK3*: variety of Andor (currently part of Oxford Instruments) cameras. Tested with Andor iXon, Luca, and Zyla.
  - *DCAM*: Hamamatsu cameras. Tested with Hamamatsu Orca Flash 4.0 and ImagEM.
  - *NI IMAQ*: National Instruments frame grabbers. Tested with NI PCI-1430 and PCI-1433 frame grabbers together with PhotonFocus MV-D1024E camera.
  - *NI IMAQdx*: National Instruments universal camera interface. Tested with Ethernet-connected PhotonFocus HD1-D1312 camera.
  - *Photon Focus*: Photon Focus pfcam interface. Tested with PhotonFocus MV-D1024E camera connected through either NI frame grabbers (PCI-1430 and PCI-1433) or Silicon Software frame grabbers (microEnable IV AD4-CL).
  - *PCO SC2*: PCO cameras. Tested with pco.edge cameras with CLHS and regular CameraLink interfaces.
  - *Silicon Software*: Silicon Software frame grabbers. Tested with microEnable IV AD4-CL frame grabbers together with PhotonFocus MV-D1024E camera.
  - *Thorlabs Scientific Cameras*: Thorlabs sCMOS cameras. Tested with Thorlabs Kiralux camera.
  - *Uc480*: multiple cameras, including simple Thorlabs and IDS cameras. Tested with IDS SC2592R12M and Thorlabs DCC1545M.
- *Stages*
  - *Attocube ANC300* and *Attocube ANC350*: most common Attocube positioner controllers. Tested with Ethernet and USB connection for ANC300, and USB connection for ANC350.
  - *Thorlabs APT/Kinesis*: basic Thorlabs motorized stages and optomechanics devices. Tested with KDC101 and K10CR1, as well as MFF101 and FW102 (described at a *different page*)
  - *Newport Picomotor*: precision piezo-actuated screws based on slip-stick principle. Tested with Newport 8742 Picomotor driver using Ethernet or USB connection.
  - *Arcus Performax*: fairly common single- and multi-axis motor controllers sold under different brands: Arcus, Nippon Pulse America, or Newmark Systems. Tested with PMX-4EX device with USB connection.
  - *Trinamic*: universal motor controllers and drivers. Tested with a single-axis TMCM-1110 controller with USB connection.
  - *SmarAct*: high-performance piezo sliders. So far only simple open-loop *SCU controllers* are supported. Tested with a standard HCU controller unit.
- *Basic sensors*
  - *HighFinesse*: laser wavelength meters. Tested with WS6 and WS7 USB-controlled devices.
  - *Ophir*: optical power and energy meters. Tested with Ophir Vega.
  - *Lakeshore*: temperature sensors. Tested with Lakeshore 218.
  - *Pfeiffer*: pressure gauges. Tested with TPG261 and DPG202 controllers.
- Lasers
  - *Basic lasers*
    - *Lighthouse Photonics SproutG*
    - *Laser Quantum Finesse*

    - *M2 Solstis laser*
    - *Toptica iBeam Smart laser*
- *Tektronix oscilloscopes*. Tested with TDS2002B, TDS2004B, and DPO2004B.
- *NI DAQs*. Tested with NI USB-6008, NI USB-6343, and NI PCIe-6323.
- *Generic AWGs*. Tested with Agilent 33500 and 33220A, Rigol DG1022, Tektronix AFG1022, GW Instek AFG2225 and AFG2115, and RS Comp AFG21005.
- *Miscellaneous Thorlabs devices*: *MFF101/102* motorized flip mirror mount, *FW102/212* motorized filter wheel, and *MDT693/694* high-voltage source.
- *Miscellaneous OZOptics devices*: *EPC04* fiber polarization controller, *DD100* motorized fiber attenuator, and *TF100* motorized fiber filter.
- *Miscellaneous devices*
    - *Conrad relay board*
    - *Basic Arduino communication*

1.2.2 Cameras

Basic concepts are described at the *cameras communication page*.

Currently supported cameras:

- *Andor SDK2* and *Andor SDK3*: variety of Andor (currently part of Oxford Instruments) cameras. Tested with Andor iXon, Luca, and Zyla.
- *DCAM*: Hamamatsu cameras. Tested with Hamamatsu Orca Flash 4.0 and ImagEM.
- *NI IMAQ*: National Instruments frame grabbers. Tested with NI PCI-1430 and PCI-1433 frame grabbers together with PhotonFocus MV-D1024E camera.
- *NI IMAQdx*: National Instruments universal camera interface. Tested with Ethernet-connected PhotonFocus HD1-D1312 camera.
- *Photon Focus*: Photon Focus pfcam interface. Tested with PhotonFocus MV-D1024E camera connected through either NI frame grabbers (PCI-1430 and PCI-1433) or Silicon Software frame grabbers (microEnable IV AD4-CL).
- *PCO SC2*: PCO cameras. Tested with pco.edge cameras with CLHS and regular CameraLink interfaces.
- *Silicon Software*: Silicon Software frame grabbers. Tested with microEnable IV AD4-CL frame grabbers together with PhotonFocus MV-D1024E camera.
- *Thorlabs Scientific Cameras*: Thorlabs sCMOS cameras. Tested with Thorlabs Kiralux camera.
- *Uc480*: multiple cameras, including simple Thorlabs and IDS cameras. Tested with IDS SC2592R12M and Thorlabs DCC1545M.

Note: General device communication concepts are described on the corresponding *page*.

Cameras control basics

Basic examples

Basic camera usage is fairly straightforward:

```
from pylablib import Andor
cam = Andor.AndorSDK3Camera()  # connect to the camera
cam.set_exposure(10E-3)  # set 10ms exposure
cam.set_roi(0,128,0,128)  # set 128x128px ROI in the upper left corner
images = cam.grab(10)  # grab 10 frames
cam.close()
```

In case you need to grab and process frames continuously, the example is a bit more complicated:

```
with Andor.AndorSDK2Camera() as cam: # to close the camera automatically
    cam.start_acquisition()  # start acquisition (automatically sets it up as well)
    while True:  # acquisition loop
        cam.wait_for_frame()  # wait for the next available frame
        frame=cam.read_oldest_image()  # get the oldest image which hasn't been read␣
↪yet
        # ... process frame ...
```

Some concepts are explained below in more detail.

Basic concepts

Frames buffer

In most cases, the frames acquired by the camera are first temporarily stored in the local camera and / or frame grabber memory, from which they are transferred to the PC RAM by the camera drivers. Afterwards, this memory is made available to all other applications. In principle, it should be enough to store only the most recent frame in RAM, and for the user software to continuously wait for a new frame, immediately read it from RAM and process it. However, such approach is very demanding to the user code: if the new frame is acquired before the previous one is processed or copied, then the RAM data is overwritten, and the old frame is lost. Hence, it is more practical to have a *buffer* of several most recently acquired frames to account for inevitable interruptions in the user wait-read-process loop caused by OS scheduling and by other jobs. In this case, the frames get lost only when the buffer is completely filled, and the oldest frames starts getting overwritten.

When using the camera classes provided by pylablib, you do not need to worry about setting up the buffer yourself, since it is done behind the scene either by the manufacturer's code or by the device class. However, it is important to keep in mind the existence of the buffer when setting up the acquisition, interpreting the buffer and acquired frames status, or identifying the skipped frames.

The size of the buffer can almost always be selected by the user. Typically it is a good idea to have at least 100ms worth of frames there, although, depending on the other jobs performed by the software, it can be larger.

Acquisition setup

Setting up an acquisition process might take a lot of time (up to 10s in more extreme cases). This happens mostly because of the buffer allocation and setting up internal API structures; initiating the acquisition process itself is fairly fast. Hence, it is useful to separate setting up / cleaning up and starting / stopping.

The first two procedures correspond to `setup_acquisition` and `clear_acquisition` method, which are slow, but rarely called. Usually, they only need to be invoked right after connecting to the camera, or when the acquired image size is changed (e.g., due to a change in binning or ROI). Since these methods deal with buffer allocation, in almost all cases they take a parameter specifying buffer size (typically called `nframes`).

The other two procedures correspond to `start_acquisition` and `stop_acquisition` methods. These try to be as fast as possible, as they need to be called any time the acquisition is started or stopped, or when minor parameters (frame rate, exposure, trigger mode) are called.

Region of interest (ROI) and binning

Most cameras allow the user to select only a part of the whole sensor for readout and transfer. Since the readout speed is usually the factor limiting the frame rate, selecting smaller ROI frequently lets you achieve higher frame rate. In addition, it also reduces the size of the frame buffer and the data transfer load. Same goes for binning: many cameras can combine values of several consecutive pixels in the same row or column (or both), which results in smaller images and, depending on the camera architecture, higher signal-to-noise ratio compared to binning in post-processing. Much less frequently you can set up subsampling instead of binning, which skips pixels instead of averaging them together.

Both operations depend very strongly on the exact hardware, so there are typically many associated restriction. The most common are minimal sizes in width and height, positions and sizes being factors of some power of 2 (up to 32 for some cameras), or equal binning for both axes. Device classes will typically round the ROI to the nearest allowed value. Furthermore, the scaling of the maximal frame rate with the ROI size is also hardware-dependent; for example, in many sCMOS chips readout speed only depends on the vertical extent, since the readout is done simultaneously for the whole row. In most cases, it takes some experiments to get a hang of the camera behavior.

Exposure and frame rate

Almost all scientific cameras let user change the exposure, typically in a wide range (down to sub-ms). Frequently they also allow to separately change the frame period (inverse of the frame rate). Usually (but not always) the minimal frame period is set by the exposure plus some readout time, which depends on the ROI and some additional parameters such as pixel clock or simultaneous readout mode. Usually exposure takes priority over the frame period, i.e., if the frame period is set too short, it is automatically adjusted. Notable exception from this rule is *Uc480* interface, where this dependence in reversed.

Triggering

Usually the cameras will have several different options for triggering, i.e., choosing when to start acquiring a new frame or a new batch of frames. The default option is the internal trigger, which means that the internal timer generates trigger event at a constant rate (frame rate). Many cameras will also take an external trigger signal to synchronize acquisition to external events or other cameras. Typically, a rising edge from 0 to 5V on the input will initiate the frame acquisition, but more exotic options (different polarities or levels, exposure control with pulse width, line-readout trigger) can be present.

Application notes and examples

Here we talk more practically about performing tasks common to most cameras.

Simple acquisition

Frame acquisition is, understandably, the most important part of the camera. Basic acquisition can be done without explicitly setting up the acquisition loop, simply by using `ICamera.snap()` and `ICamera.grab()` methods which, correspondingly, grab a single frame or a given number of frames:

```
from pylablib import Andor
cam = Andor.AndorSDK3Camera()  # connect to the camera
img = cam.snap()  # grab a single frame
images = cam.grab(10)  # grab 10 frames (return a list of frames)
cam.close()
```

These allow for quick tests of whether the camera works properly, and for occasional frames acquisition. However, these methods have to start and stop acquisition every time they are called, which for some cameras can take about a second. Hence, if continuous acquisition and high frame rate are required, you would need to set up the acquisition loop.

Acquisition loop

A typical simple acquisition loop has already been shown above:

```
cam.setup_acquisition(nframes=100)  # could be combined with start_acquisition, or␣
↪kept separate
cam.start_acquisition()
while True:  # acquisition loop
    cam.wait_for_frame()  # wait for the next available frame
    frame = cam.read_oldest_image()  # get the oldest image which hasn't been read yet
    # ... process frame ...
    if time_to_stop:
        break
cam.stop_acquisition()
```

It relies on 3 sets of methods. First, starting and stopping acquisition using `start_acquisition` and `stop_acquisition`. As explained above, one also has an option to setup the acquisition first using `setup_acquisition`, which makes the subsequent `start_acquisition` call faster. However, one can also supply the same setup parameters to `start_acquisition` method, which automatically sets up the acquisition if it is not set up yet, or if any parameters are different from the current ones.

Second are the methods for checking on the acquisition process. The method used above is `wait_for_frame`, which by default waits until there is at least one unread frame in the buffer (i.e., it exits immediately if there is already a frame available). Its arguments modify this behavior by changing the point from which the new frame is acquired (e.g., from the current call), or the minimal required number of frames. Alternatively, there is a method `get_new_images_range`, which returns a range of the frame indices which have been acquired but not read. This method allows for a quick check of a number of unread frames without pausing the acquisition.

Finally, there are methods for reading out the frames. The simplest method is `read_oldest_image`, which return the oldest image which hasn't been read yet, and marks it as read. A more powerful is the `read_multiple_images` method, which can return a range of images (by default, all unread images). Both of these methods also take a `peek` argument, which allows one to read the frames without marking them as read.

Frame indexing

Different areas and libraries adopt different indexing convention for 2D arrays. The two most common ones are coordinate-like `xy` (the first index is the `x` coordinate, the second is `y` coordinate, and the origin is in the lower left corner) and matrix-like `ij` (the first index is row, the second index is column, the origin is int the upper right corner). Almost all cameras adopt the `ij` convention. The only exception is Andor SDK2, which uses similar row-column indexing, but counting from the bottom.

By default, the frames returned by the camera are indexed in the preferred convention, to reduce the overhead on re-indexing the frames. It is possible to check and change it using *`ICamera.get_image_indexing()`* and

ICamera.set_image_indexing() methods:

```
>> cam.set_roi(0,256,0,128)  # 256px horizontally, 128 vertically
>> cam.snap().shape  # 128 rows, 256 columns
(128, 256)
>> cam.set_image_indexing("xyb")  # standard xy indexing, starting from the bottom
>> cam.snap().shape
(256, 128)
```

ROI, detector size and frame shape

Both ROI and binning are controlled by one pair of methods `get_roi` and `set_roi` which, depending on whether camera supports binning, take (and return) 4 or 6 arguments: start and stop positions of ROI along both axes and, optionally, binning along the axes:

```
cam.set_roi(0,128,0,256)  # set 128x256px (width x height) ROI in the (typically)␣
↪upper left controller
cam.set_roi(0,128)  # set roi with 128px width and full height (non-supplied␣
↪arguments take extreme values)
cam.set_roi(0,128,0,128,2,2)  # set 128x128px ROI with 2x2 binning; the resulting␣
↪image size is 64x64
```

Regardless of the frame indexing, the first pair of arguments always controls horizontal span, the second pair controls vertical span, and the last pair controls horizontal and vertical binning (if applicable).

In addition, there is a couple of methods to acquire the detector and frame size. The first method is `get_detector_size`. It always returns the full camera detector size as a tuple `(width, height)` and, therefore, is not affected by ROI, binning, and indexing. The second method is `get_data_dimensions`, which returns the shape of the returned frame given the currently set up indexing. The results of this method do depend on the ROI, binning, and indexing:

```
>> cam.get_detector_size()  # (width, height)
(2560, 1920)
>> cam.get_data_dimensions()  # (rows, columns), i.e., (height, width)
(1920, 2560)

>> cam.set_roi(0,256,0,128,2,2)  # 256px horizontally, 128 vertically, 2x2 binning
>> cam.get_detector_size()  # unaffected
(2560, 1920)
>> cam.get_data_dimensions()  # depends on ROI
(64, 128)

>> cam.set_image_indexing("xyb")
>> cam.get_detector_size()  # unaffected
(2560, 1920)
>> cam.get_data_dimensions()  # depends on indexing
(128, 64)
```

Exposure and frame period

In pylablib these parameters are normally controlled by `get_exposure`/`set_exposure` and, correspondingly `get_frame_period`/`set_frame_period` methods. In addition, `get_frame_timings` method provides an overview of all the relevant times. Exposure typically takes priority over frame period: if the frame period is set too small, it becomes the smallest possible for the given exposure; at the same time, if the exposure is set too big, it is still applied, and the frame period becomes the smallest possible with this exposure:

```
>> cam.get_frame_timings()  # frame period is a usually bit larger due to the readout␣
↪time
TAcqTimings(exposure=0.1, frame_period=0.12)

>> cam.set_exposure(0.01)
>> cam.get_frame_timings()  # smaller exposure is still compatible with this frame␣
↪period
TAcqTimings(exposure=0.01, frame_period=0.12)

>> cam.set_frame_period(0)  # effectively means "set the highest possible frame rate"
>> cam.get_frame_timings()
TAcqTimings(exposure=0.01, frame_period=0.03)

>> cam.set_exposure(0.2)
>> cam.get_frame_timings()  # frame period is increased accordingly
TAcqTimings(exposure=0.2, frame_period=0.22)
```

There are exceptions for some camera types, which are discussed separately.

Camera attributes

Some camera interfaces, e.g., *Thorlabs Scientific Cameras*, *PCO SC2*, or *NI IMAQ* are fairly specific, and only apply to a handful of devices with very similar capabilities. In this case, pylablib usually attempts to implement as much of the functionality as possible given the available hardware, and to present it via the camera object methods.

In other cases, e.g., *NI IMAQdx*, *Andor SDK3*, or *DCAM*, the same interface deals with many fairly different cameras. This is especially true for IMAQdx, which covers hundreds of cameras from dozens of manufacturers, all with very different capabilities and purpose. Since managing such cameras can not usually be conformed to a small set of functions, it is implemented through camera attributes mechanism. That is, for each camera the interface defines a set of attributes (sometimes also called properties or features), which can be queried or set by their names, and whose exact meaning and possible values depend on the specific camera.

Typically, cameras dealing with attributes will implement `IAttributeCamera.get_attribute_value()` and `IAttributeCamera.set_attribute_value()` for querying and setting the attributes, as well as dictionary-like `.cav` (stands for "camera attribute value") interface to do the same thing:

```
>> cam = Andor.AndorSDK3Camera()
>> cam.get_attribute_value("CameraAcquiring")  # check if the camera is acquiring
0
>> cam.set_attribute_value("ExposureTime", 0.1)  # set the exposure to 100ms
>> cam.cav["ExposureTime"]  # get the exposure; could also use cam.get_attribute_
↪value("ExposureTime")
0.1
```

Additionally, there are `IAttributeCamera.get_all_attribute_values()` and `IAttributeCamera.set_all_attribute_values()` which get and set all camera attributes (possibly only within the given branch, if camera attributes form a hierarchy). Finally, methods `IAttributeCamera.get_attribute()` and `IAttributeCamera.get_all_attributes()`, together with the corresponding `.ca` interface, allow to query specific attribute objects, which provide additional information about the attributes: whether they are writable or readable, their range, description, possible values, types, etc.:

```
>> cam = DCAM.DCAMCamera()
>> attr=cam.ca["EXPOSURE TIME"]  # get the exposure attribute
DCAMAttribute(name='EXPOSURE TIME', id=2031888, min=0.001, max=10.0, unit=1)
>> attr.max
```

(continues on next page)

(continued from previous page)

```
10.0
>> attr.set_value(0.1)  # same as cam.set_attribute_value("EXPOSURE TIME", 0.1)
```

Note that, depending on the camera, the attribute properties (especially minimal and maximal value) can depend on the other camera attributes. For example, minimal exposure can depend on the frame size:

```
>> cam = DCAM.DCAMCamera()
>> attr=cam.ca["EXPOSURE TIME"]  # get the exposure attribute
DCAMAttribute(name='EXPOSURE TIME', id=2031888, min=0.001, max=10.0, unit=1)
>> attr.min
0.001
>> cam.set_roi(0, 0, 0, 0)  # set the minimal possible ROI
(0, 4, 0, 4, 1, 1)
>> attr.min  # minimal value hasn't been updated yet
0.001
>> attr.update_limits()  # update the property limits
>> attr.min  # now the minimal possible exposure is smaller
7.795e-05
```

If the documentation is not available (as is the case for, e.g., some IMAQdx cameras), the best way to learn about the attributes is to use the native software (whenever available) to modify camera settings and then check how the attributes change. Besides that, it is always useful to check attribute description (available for IMAQdx parameter), their range, and the available values for enum attributes.

Trigger setup

The trigger is usually set up using `set_trigger_mode` method, although it might be different if more specialized modes are used. When external trigger is involved, most of the code (such as acquisition set up and start) stays the same. The only difference is the rate at which the frames are generated:

```
frame = cam.snap()  # starts acquiring immediately, returns after a single frame
cam.set_trigger_mode("ext") # set up the trigger mode
frame = cam.snap()
#  after cam.snap() is called, the execution will wait
#  for an external trigger pulse to acquire the frame and return
```

Frame metainfo

Many cameras supply additional information together with the frames. Most frequently it contains the internal framestamp and timestamp (which are useful for tracking missing frames), but sometimes it also includes additional information such as frame size or location, status, or auxiliary input bits. To get this information, you can supply the argument `return_info=True` to the `read_multiple_images` method. In this case, instead of a single list of frames, it will return a tuple of two lists, where the second list contains this metainfo.

There are several slightly different metainfo formats, which can be set using *`ICamera.set_frame_info_format()`* method. The default representation is a (possibly nested) named tuple, but it is also possible to represent it as a flat list, flat dictionary, or a numpy array. The exact structure and values depend on the camera.

Keep in mind, that for some camera interfaces (e.g., *Uc480* or *Silicon Software*) obtaining the additional information might take relatively long, even longer than the proper frame readout. Hence, at higher frame rates it might become a bottleneck, and would need to be turned off.

Related projects

Pylablib cam-control is a standalone software package which builds on camera classes included in pylablib. It provides an easy way to detect and control many different cameras and acquire their data. In addition, it supports custom on-line image processing, flexible data acquisition, and control by external software using a TCP/IP server.

Currently supported cameras

- *Andor SDK2* and *Andor SDK3*: variety of Andor (currently part of Oxford Instruments) cameras. Tested with Andor iXon, Luca, and Zyla.
- *DCAM*: Hamamatsu cameras. Tested with Hamamatsu Orca Flash 4.0 and ImagEM.
- *NI IMAQ*: National Instruments frame grabbers. Tested with NI PCI-1430 and PCI-1433 frame grabbers together with PhotonFocus MV-D1024E camera.
- *NI IMAQdx*: National Instruments universal camera interface. Tested with Ethernet-connected PhotonFocus HD1-D1312 camera.
- *Photon Focus*: Photon Focus pfcam interface. Tested with PhotonFocus MV-D1024E camera connected through either NI frame grabbers (PCI-1430 and PCI-1433) or Silicon Software frame grabbers (microEnable IV AD4-CL).
- *PCO SC2*: PCO cameras. Tested with pco.edge cameras with CLHS and regular CameraLink interfaces.
- *Silicon Software*: Silicon Software frame grabbers. Tested with microEnable IV AD4-CL frame grabbers together with PhotonFocus MV-D1024E camera.
- *Thorlabs Scientific Cameras*: Thorlabs sCMOS cameras. Tested with Thorlabs Kiralux camera.
- *Uc480*: multiple cameras, including simple Thorlabs and IDS cameras. Tested with IDS SC2592R12M and Thorlabs DCC1545M.

Note: General camera communication concepts are described on the corresponding *page*

Andor cameras

Andor implements two completely separate interfaces for different cameras. The older one, called SDK2, or simply SDK, provides interface for the older cameras: iXon, iKon, iStart, iDus, iVac, Luca, Newton. The details of this SDK are available in the manual.

The newer SDK, called SDK3, covers newer cameras: Zyla, Neo, Apogee, Sona, Marana, and Balor. The manual describes the cameras and capabilities in more details.

The required DLLs are distributed with Andor Solis or the corresponding Andor SKD. In most cases, you have Andor Solis already installed to provide the drivers and to communicate with the cameras to begin with.

Andor SDK 2

This is an older SDK, which mainly involves older cameras. It has been tested with Andor iXon and Andor Luca.

The code is located in `pylablib.devices.Andor`, and the main camera class is `pylablib.devices.Andor.AndorSDK2Camera`.

Software requirements

The required DLL can have different names depending on the Solis version and SDK bitness. For 64-bit version it will be called `atmcd64d.dll` or `atmcd64d_legacy.dll`. For 32-bit version, correspondingly, `atmcd32d.dll` or `atmcd32d_legacy.dll`. By default, library searches for DLLs in `Andor Solis` and `Andor SDK` folder in `Program Files` folder (or `Program files (x86)`, if 32-bit version of Python is running), as well as in the folder containing the script. If the DLLs are located elsewhere, the path can be specified using the library parameter `devices/dlls/andor_sdk2`:

```
import pylablib as pll
pll.par["devices/dlls/andor_sdk2"] = "path/to/dlls"
from pylablib.devices import Andor
cam = Andor.AndorSDK2Camera()
```

Connection

The cameras are identified by their index, starting from zero. To get the total number of cameras, you can run *Andor.get_cameras_number_SDK2*:

```
>> from pylablib.devices import Andor
>> Andor.get_cameras_number_SDK2()
2
>> cam1 = Andor.AndorSDK2Camera(idx=0)
>> cam2 = Andor.AndorSDK2Camera(idx=1)
>> cam1.close()
>> cam2.close()
```

Warning: It is important to close all camera connections before finishing your script. Otherwise, DLL resources might become permanently blocked, and the only way to solve it would be to restart the PC.

Operation

The operation of these cameras is relatively standard. They support all the standard methods for dealing with ROI and exposure, starting and stopping acquisition, and operating the frame reading loop. However, there's a couple of differences from the standard libraries worth highlighting:

- Since the manufacturer DLLs do not provide methods to get most of the camera parameters (such as exposure or ROI), it is impossible to know them when connecting the camera. To get around it, the camera is put into a "default" state any time the connection is opened.
- When applicable, it is important to properly set the cooling setpoint and the fan mode. By default, the fan is turned off, and the cooling is set to the 20'th percentile of the whole range (e.g., -80C for Andor iXon). It is possible to pass these parameters on camera creation:

  ```
  cam = Andor.AndorSDK2Camera(temperature=-50, fan_mode="on")
  ```

- Often cameras have a lot of different readout parameters: channel, amplifier, vertical and horizontal scan speed, etc. These parameters greatly affect the camera sensitivity and readout speed. Upon the connection, the parameter are typically set to the slowest mode. To get the list of all possible parameter combinations, you can use *AndorSDK2Camera.get_all_amp_modes()* and *AndorSDK2Camera.get_max_vsspeed()*. Afterwards, you can set them using *AndorSDK2Camera.set_amp_mode()* and *AndorSDK2Camera.set_vsspeed()*.

- The default shutter parameter is `"closed"`. This preserves camera from possible high illumination, but can lead to confusion, if you expect to see some image.
- This SDK does not allow for specifying number of frames in the frames buffer. However, the parameters chosen by the SDK are usually reasonable (at least a second worth of acquisition).
- Some cameras (e.g., iXon) have lots of readout (full frame, ROI, full vertical binning, etc.) and acquisition modes (single, continuous, accumulating, kinetic cycle, etc.). They are described in details in the manual.

Andor SDK 3

This is a newer SDK, which covers the newer cameras. It has been tested with Andor Zyla.

The code is located in *pylablib.devices.Andor*, and the main camera class is *pylablib.devices.Andor.AndorSDK3Camera*.

Software requirements

This library requires several DLLs all located in the same folder: `atcore.dll`, `atblkbx.dll`, `atcl_bitflow.dll`, `atdevapogee.dll`, `atdevregcam.dll`, `atusb_libusb.dll`, `atusb_libusb10.dll`. Same as for SDK2, pylablib looks for DLLs in `Andor Solis` and `Andor SDK3` folders in `Program Files` folder (or `Program files (x86)`, if 32-bit version of Python is running), as well as in the folder containing the script. A custom DLLs path can be specified using the library parameter `devices/dlls/andor_sdk3`:

```
import pylablib as pll
pll.par["devices/dlls/andor_sdk3"] = "path/to/SDK3/dlls"
from pylablib.devices import Andor
cam = Andor.AndorSDK3Camera()
```

Connection

The cameras are identified by their index, starting from zero. To get the total number of cameras, you can run *Andor.get_cameras_number_SDK3*:

```
>> from pylablib.devices import Andor
>> Andor.get_cameras_number_SDK3()
2
>> cam1 = Andor.AndorSDK3Camera(idx=0)
>> cam2 = Andor.AndorSDK3Camera(idx=1)
>> cam1.close()
>> cam2.close()
```

Operation

The operation of these cameras is also relatively standard. They support all the standard methods for dealing with ROI and exposure, starting and stopping acquisition, and operating the frame reading loop. However, there's a couple of differences from the standard libraries worth highlighting:

- The SDK also provides a universal interface for getting and setting various *camera attributes* (called "features" in the documentation) using their name. You can use *AndorSDK3Camera.get_attribute_value()* and *AndorSDK3Camera.set_attribute_value()* for that, as well as `.cav` attribute which gives a dictionary-like access:

```
>> cam = Andor.AndorSDK3Camera()
>> cam.get_attribute_value("CameraAcquiring")  # check if the camera is␣
↪acquiring
0
>> cam.set_attribute_value("ExposureTime", 0.1)  # set the exposure to␣
↪100ms
>> cam.cav["ExposureTime"]  # get the exposure; could also use cam.get_
↪attribute_value("ExposureTime")
0.1
```

Some values serve as commands; these can be invoked using `AndorSDK3Camera.call_command()` method. To see all available attributes, you can call `AndorSDK3Camera.get_all_attributes()` to get a dictionary with attribute objects, and `AndorSDK3Camera.get_all_attribute_values()` to get the dictionary of attribute values. The attribute objects provide additional information: their kind, whether they are implemented, readable, or writable, what are their limits or possible values, etc:

```
>> cam = Andor.AndorSDK3Camera()
>> attr = cam.get_attribute("SensorTemperature")
>> attr.readable
True
>> attr.writable
False
>> (attr.min, attr.max)
(-100.0, 50.0)
```

The description of the attributes is given in manual.

- USB cameras can, in principle, generate data at higher rate than about 320Mb/s that the USB3 bus supports. For example, Andor Zyla with 16 bit readout has a single full frame size of 8Mb, which puts the maximal USB throughput at about 40FPS. At the same time, the camera itself is capable of reading up to 100FPS at the full frame. Hence, it is possible to overflow the camera internal buffer (size on the order of 1Gb) regardless of the PC performance. If this happens, the acquisition process halts and needs to be restarted. You can check the number of buffer overflows using `AndorSDK3Camera.get_missed_frames_status()`, and reset this counter using `AndorSDK3Camera.reset_overflows_counter()`; the counter is also automatically resets on acquisition clearing, but not stopping.

Furthermore, the class implements different strategies when encountering overflow while waiting for a new frame. The specific strategy is selected using `AndorSDK3Camera.set_overflow_behavior()`, and it can be `"error"` (raise `AndorFrameTransferError`, which is the default behavior), `"restart"` (restart the acquisition and immediately raise timeout error), or `"ignore"` (ignore the overflow, which will eventually lead to a timeout error, as the new frames are no longer generated).

Note: General camera communication concepts are described on the corresponding *page*.

DCAM cameras interface

DCAM is the interface used in Hamamatsu cameras. It has been tested with Hamamatsu Orca Flash and ImagEM.

The code is located in `pylablib.devices.DCAM`, and the main camera class is `pylablib.devices.DCAM.DCAMCamera`.

Software requirements

These cameras require `dcamapi.dll`, which is installed with most of Hamamatsu software (such as HoKaWo or HiPic), as well as with the freely available DCAM API, which also includes all the necessary drivers. Keep in mind, that you also need to install the drivers for required corresponding camera type (USB, Ethernet, IEEE 1394). These drivers are in the same installer, but need to be installed separately. After installation, the DLL is automatically added to the `System32` folder, where pylablib looks for it by default. If the DLL is located elsewhere, the path can be specified using the library parameter `devices/dlls/dcamapi`:

```
import pylablib as pll
pll.par["devices/dlls/dcamapi"] = "path/to/dlls"
from pylablib.devices import DCAM
cam = DCAM.DCAMCamera()
```

Connection

The cameras are identified by their index, starting from zero. To get the total number of cameras, you can run *`DCAM.get_cameras_number()`*:

```
>> from pylablib.devices import DCAM
>> DCAM.get_cameras_number()
2
>> cam1 = DCAM.DCAMCamera(idx=0)
>> cam2 = DCAM.DCAMCamera(idx=1)
>> cam1.close()
>> cam2.close()
```

Operation

The operation of these cameras is relatively standard. They support all the standard methods for dealing with ROI and exposure, starting and stopping acquisition, and operating the frame reading loop. The SDK also provides a universal interface for getting and setting various *camera attributes* (called "properties" in the documentation) using their name. You can use *`DCAMCamera.get_attribute_value()`* and *`DCAMCamera.set_attribute_value()`* for that, as well as `.cav` attribute which gives a dictionary-like access:

```
>> cam = DCAM.DCAMCamera()
>> cam.get_attribute_value("BINNING")  # get the camera binning (no binning, by␣
↪default)
1
>> cam.set_attribute_value("EXPOSURE TIME", 0.1)  # set the exposure to 100ms
>> cam.cav["EXPOSURE TIME"]  # get the exposure; could also use cam.get_attribute_
↪value("EXPOSURE TIME")
0.1
```

To see all available attributes, you can call *`DCAMCamera.get_all_attributes()`* to get a dictionary with attribute objects, and *`DCAMCamera.get_all_attribute_values()`* to get the dictionary of attribute values, with an option of representing enum attributes either as text or as integer values. The attribute objects provide additional information: attribute range, step, and units:

```
>> cam = DCAM.DCAMCamera()
>> attr = cam.get_attribute("EXPOSURE TIME")
>> (attr.min, attr.max)
(0.001, 10.0)
```

Additionally, there's a couple of differences from the standard libraries worth highlighting:

- The library supports only symmetric binning, i.e., the binning factor is the same in both directions. For compatibility *DCAMCamera.get_roi()* and *DCAMCamera.set_roi()* still return and accept both binning parameters independently, but they are always the same when returned, and `vbin` is ignored when set.
- By default, the SDK does not provide independent control of the frame period and the exposure. Hence, `set_frame_period` method is unavailable, and the frame rate is defined solely by the exposure.

Note: General camera communication concepts are described on the corresponding *page*

NI IMAQ frame grabbers interface

NI IMAQ is the interface from National Instruments, which is used in a variety of frame grabbers. It has been tested with NI PCI-1430 and PCI-1433 frame grabbers together with PhotonFocus MV-D1024E camera.

The code is located in *pylablib.devices.IMAQ*, and the main camera class is *pylablib.devices.IMAQ.IMAQCamera*.

Software requirements

This interfaces requires `imaq.dll`, which is installed with the freely available Vision Acquisition Software, which also includes all the necessary drivers. After installation, the DLL is automatically added to the `System32` folder, where pylablib looks for it by default. If the DLL is located elsewhere, the path can be specified using the library parameter `devices/dlls/niimaq`:

```
import pylablib as pll
pll.par["devices/dlls/niimaq"] = "path/to/dlls"
from pylablib.devices import IMAQ
cam = IMAQ.IMAQCamera()
```

Connection

The cameras are identified by their name, which usually looks like `"img0"`. To get the list of all cameras, you can use NI MAX (Measurement and Automation Explorer), or *IMAQ.list_cameras()*:

```
>> from pylablib.devices import IMAQ
>> IMAQ.list_cameras()
['img0', 'img1']
>> cam1 = IMAQ.IMAQCamera('img0')
>> cam2 = IMAQ.IMAQCamera('img1')
>> cam1.close()
>> cam2.close()
```

Operation

Unlike most camera classes, the frame grabber interface only deals with the frame transfer between the camera and the PC over the CameraLink interface. Therefore, in can not directly control camera parameters such as exposure, frame rate, triggering, ROI, etc. Some similar-looking parameters are still present, but they have a different meaning:

- External trigger controls frame *transfer*, not frame *acquisition*, which is defined by the camera. By default, when the internal frame grabber trigger is used, the frame grabber transfer rate is synchronized to the camera, so every frame gets transferred. However, if the external transfer trigger is used and it is out of sync with the camera, it can result in duplicate or missing frames.
- ROI is defined within the transferred image, whose size itself is determined by the camera ROI. Hence, e.g., if the camera chip is 1024x1024px and its roi is 512x512, then the frame grabber ROI can go only up to 512x512. Any attempts to set it higher result in the frozen acquisition, as the frame grabber expects a larger frame than it receives, and waits forever to get the rest.

The SDK also provides a universal interface for getting and setting various camera attributes using their name. You can use `IMAQCamera.get_grabber_attribute_value()` and `IMAQCamera.set_grabber_attribute_value()` for that:

```
>> cam = IMAQ.IMAQCamera()
>> cam.get_grabber_attribute_value("FRAMEWAIT_MSEC")  # frame read request timeout
1000
```

To get a all available attributes as a dictionary, you can call `IMAQCamera.get_all_grabber_attribute_values()`. Their meaning, as well as descriptions of trigger modes and other settings, is explained in the manual supplied with the Vision Acquisition Software.

Fast buffer readout mode

At high frame rates (above ~10kFPS) dealing with each frame individually becomes too slow for Python. Hence, there is an option to read out and process frames in larger 'chunks', which are 3D numpy arrays with the first axis enumerating the frame index. This approach leverages the ability to store several frame buffers in the contiguous memory locations (resulting in a single 3D array), and it essentially eliminates the overhead for dealing with multiple frames at high frame rates, as long as the total data rate is manageable (typically below 600Mb/s).

This option can be accessed by supplying `fastbuff=True` in `IMAQCamera.read_multiple_images()`. In this case, instead of a list of individual frames (which is the standard behavior), the method returns list of chunks about 1Mb in size, which contain several consecutive frames. Otherwise the method behaves identically to the standard one.

Communication with the camera and camera files

The frame grabber needs some basic information about the camera: sensor size, bit depth, data transfer format, timeouts, aux lines mapping, etc. In NI MAQ this information is contained in the so-called camera files. These files can be assigned to cameras in the NI MAX, and are usually supplied by NI or by the camera manufacturer. In addition, NI MAX allows one to adjust some settings within these files, which are read-only within the NI IMAQ software. These include frame timeout and camera bit depth.

The communication with the camera itself greatly varies between different cameras. Some will have additional connection to control the parameters. Others use serial communication built into the CameraLink interface. This communication can be set up with `IMAQCamera.setup_serial_params()` and used via `IMAQCamera.serial_read()` and `IMAQCamera.serial_write()`. The communication protocols are camera-dependent, and are frequently described in the camera manual. However, some other cameras (e.g., Photon Focus) use proprietary communication protocol. In this case, they provide their own DLLs, which independently use NI-provided DLLs for serial communication (most notably, `clallserial.dll`) to communicate with the camera. In this case, one needs to maintain two independent connections: one directly to the NI frame grabber to obtain the frame data, and one to the manufacturer library to control the camera. This is the way it is implemented in *PhotonFocus* camera interface.

Known issues

- Sometimes when the acquisition is stopped and restarted without being cleared, the acquired frame counter does not refresh. This might show up as the software not reporting any new frames. It has been tracked down to a very low (~1ms) frame read timeout. Hence, it is recommended to keep this timeout at least at 500ms.
- If you are unable to access full camera sensor size, check the camera file (it can be opened in the text editor). `MaxImageSize` parameter defines the maximal allowed image size, and it should be equal to the camera sensor size.
- Same goes for bitness. If the camera bitness is higher than set up in the frame grabber, a single camera pixel gets treated as several pixels by the frame grabber, typically resulting in 1px-wide vertical stripes on the image. In the opposite case, the frame grabber expects more bytes than the camera sends, it never receives the full frame, and the acquisition times out.
- Keep in mind that as long as the frame grabber is accessed in NI MAX, it is blocked from use in any other software. Hence, you need to close NI MAX before running your code.
- As mentioned above, ROI is defined within a frame transferred by the camera. Hence, if it includes pixels with positions outside of the transferred frame, the acquisition will time out. For example, suppose the camera sensor is 1024x1024px, and the *camera* ROI is selected to be central 512x512 region. As far as the frame grabber is concerned, now the camera sensor size is 512x512px. Hence, if you try to set the same *frame grabber* ROI (i.e., 512x512 starting at 256,256), it will expect at least 768x768px frame. Since the frame is, actually, 512x512px, the acquisition will time out. The correct solution is to set frame grabber ROI from 0 to 512px on both axes. In general, it is a good idea to always follow this pattern: control ROI only on camera, and always set frame grabber ROI to cover the whole transfer frame.
- Some frame grabbers have a limit on the data transfer rate (for one model observed to be about 200 Mb/s). If the camera data generation rate exceeds it (e.g., it produces 1024x1024px 16-bit frames at >100FPS), then the camera will raise `IMG_ERR_FIFO` error shortly after the acquisition start. In this case, you will need to reduce the data rate by reducing the frame rate or frame size (through ROI, binning, or bitness).

Note: General camera communication concepts are described on the corresponding *page*

NI IMAQdx cameras interface

NI IMAQdx is the interface provided by National Instruments and which supports a wide variety of cameras. It is completely separate from IMAQ, and it supports different communication interfaces: USB, Ethernet and FireWire. It has been tested with Ethernet-connected PhotonFocus HD1-D1312 camera.

The code is located in *`pylablib.devices.IMAQdx`*, and the main camera class is *`pylablib.devices.IMAQdx.IMAQdxCamera`*.

Software requirements

These cameras require `imaqdx.dll`, which is installed with the freely available Vision Acquisition Software. However, the IMAQdx part of the software is proprietary, and requires purchase to use. If the software license is invalid, then any attempt to communicate with cameras will result in `License not activated` error (although simply listing the cameras still works). After installation, the DLL is automatically added to the `System32` folder, where pylablib looks for it by default. If the DLL is located elsewhere, the path can be specified using the library parameter `devices/dlls/niimaqdx`:

```
import pylablib as pll
pll.par["devices/dlls/niimaqdx"] = "path/to/dlls"
from pylablib.devices import IMAQdx
cam = IMAQdx.IMAQdxCamera()
```

Connection

The cameras are identified by their name, which usually looks like `"cam0"`. To get the list of all cameras, you can use NI MAX (Measurement and Automation Explorer), or *`IMAQdx.list_cameras()`*:

```
>> from pylablib.devices import IMAQdx
>> IMAQdx.list_cameras()
['cam0', 'cam1']
>> cam1 = IMAQdx.IMAQdxCamera('cam0')
>> cam2 = IMAQdx.IMAQdxCamera('cam1')
>> cam1.close()
>> cam2.close()
```

Operation

The operation of these cameras is relatively standard. They support all the standard methods for dealing with ROI, starting and stopping acquisition, and operating the frame reading loop. The SDK also provides a universal interface for getting and setting various *camera attributes* using their name. You can use *`IMAQdxCamera.get_attribute_value()`* and *`IMAQdxCamera.set_attribute_value()`* for that, as well as `.cav` attribute which gives a dictionary-like access:

```
>> cam = IMAQdx.IMAQdxCamera()
>> cam.get_attribute_value("StatusInformation/AcqInProgress")  # check if the camera␣
↪is acquiring
0
>> cam.set_attribute_value("Width", 512)  # set the ROI width to 512px
>> cam.cav["Width"]  # get the exposure; could also use cam.get_attribute_value("Width
↪")
512
```

To see all available attributes, you can call *`IMAQdxCamera.get_all_attributes()`* to get a dictionary with attribute objects, and *`IMAQdxCamera.get_all_attribute_values()`* to get the dictionary of attribute values. The attribute objects provide additional information: attribute kind (integer, enum, string, etc.), range (either numerical range, or selection of values for enum attributes), description string, etc.:

```
>> cam = IMAQdx.IMAQdxCamera()
>> attr = cam.get_attribute("Width")
>> attr.description
'Width of the Image provided by the device (in pixels).'
>> attr.writable
True
>> (attr.min, attr.max)
(448, 1312)
```

Since these properties vary a lot between different cameras, it is challenging to write a universal class covering a large range of cameras. Hence, currently the universal class only has the basic camera parameter control such as ROI (without binning) and acquisition status. For many specific cameras you might need to explore the attributes tree (either using the Python class and, e.g., a console, or via NI MAX) and operate them directly in your code.

Known issues

- It seems like sometimes the camera communication settings might be interfering with its operation. It can show up in an unexpected way, e.g., as an `Attribute value is out of range` error when starting acquisition. If it looks like this might be the case, it is a good idea to open the camera in NI MAX (note that Ethernet cameras are listed under `Network Devices`, not in the general device list) and try to snap a single frame. NI MAX might report some problems with the settings and suggest resolution methods. Once the camera is operational, you can close NI MAX and save the camera settings (request is shown upon closing).
- In general, Ethernet cameras work better with larger packet sizes. However, packets above 1500 bits (so-called jumbo packets) are not supported by all network adapters by default. If this is the case, any attempt to acquire images causes `IMAQdxErrorTestPacketNotReceived` error. One way to deal with that is to set the packet size to 1500, which is done automatically when `small_packet=True` is supplied upon the camera creation. The other is to enable jumbo packets in the adapter properties (in Windows this is done in Device Manager).
- Currently only the basic unpacked monochrome pixel formats are supported: `Mono8`, `Mono10`, `Mono12`, `Mono16`, and `Mono32`. The reason is that even nominally well-defined types (e.g., `Mono12Packed`) have different formats for different cameras. Currently any unsupported format will raise an error on readout by default. It it still possible to read these out as raw frame data in the form of 1D or 2D numpy `'u1'` array by enabling raw frame readout using *`IMAQdxCamera.enable_raw_readout()`* method:

```
>> cam = IMAQdx.IMAQdxCamera()
>> cam.get_detector_size()  # 1280px x 1024px frame
(1280, 1024)
>> cam.set_attribute_value("PixelFormat", "BGRA 8 Packed")  # unsupported format
>> cam.snap().shape
...
IMAQdxError: pixel format BGRA 8 Packed is not supported
>> cam.enable_raw_readout("frame")  # frame data is returned as a flat array
>> cam.snap().shape  # 1280 * 1024 * 4 = 5242880 bytes
(5242880,)
```

Note: General camera communication concepts are described on the corresponding *page*.

Photon Focus pfcam interface

Photon Focus CameraLink cameras transfer their data to the PC using frame grabbers (e.g., via *NI IMAQ* or *Silicon Photonics* interfaces). Hence, the camera control is done through the serial port built into the CameraLink interface. However, the cameras use a closed binary protocol, so all the control is done through the pfcam library provided by Photon Focus. It relies on the libraries exposed by the frame grabber manufacturers (e.g., the standard `cl*serial.dll`) to communicate with the camera directly, meaning that the pfcam user simply calls its method, and all the communication happens behind the scenes.

In principle, pfcam can work with any frame grabber. Because of that, there are two different kinds of classes for this camera. To start with, there is *`.PhotonFocus.IPhotonFocusCamera`*, which provides interface for addressing camera properties, but can not handle actual frame acquisition. Using this class directly leads to errors in any frame data related methods (e.g., `wait_for_frame`, or `read_multiple_images`), and it is mostly intended to serve as a base class to be combined with the actual frame grabber. Two such combined classes are already provided: *`.PhotonFocus.PhotonFocusIMAQCamera`* for National Instruments frame grabbers using the *NI IMAQ* interface and *`.PhotonFocus.PhotonFocusSiSoCamera`* for *Silicon Photonics* frame grabbers. Both classes are complete and ready to use. In addition to combining camera and frame grabber control, they also implement basic consistency support, such as automatic adjustment of frame grabber ROI and data transfer format.

Software requirements

These cameras require `pfcam.dll`, which is installed with freely available (upon registration) PFInstaller. In addition, this DLL requires `comdll.dll` and the DLLs referring to a particular camera, e.g., `mv_d1024e_160.dll`. After installation, the path to the DLLs (all located by default in `Photonfocus/PFRemote/bin` folder in `Program Files`) is automatically added to system `PATH` variable, which is one of the places where pylablib looks for it by default. If the DLLs are located elsewhere, the path can be specified using the library parameter `devices/dlls/pfcam`:

```
import pylablib as pll
pll.par["devices/dlls/pfcam"] = "path/to/dlls"
from pylablib.devices import PhotonFocus
cam = PhotonFocus.PhotonFocusIMAQCamera()
```

Connection

The camera class requires two pieces of information. First is the frame grabber interface connection: either NI IMAQ interface name (e.g., `"img0"`) identified as described in the *NI IMAQ* documentation, or Silicon Software board and applet described in *Silicon Software* documentation. The second piece of information is the pfcam port, which is either a number starting from zero indexing the port in the ports list, or a tuple `(manufacturer, port)`, e.g., `("National Instruments", "port0")`. To list all of the connected pfcam-compatible cameras, you can use the PFRemote software (the interface number is given in parentheses after every connection option in the list) or run `PhotonFocus.list_cameras()`:

```
>> from pylablib.devices import PhotonFocus, IMAQ
>> IMAQ.list_cameras()  # get all IMAQ frame grabber devices
['img0.iid']
>> PhotonFocus.list_cameras()  # by default, get only the ports which support pfcam␣
↪interface
[(1, TCameraInfo(manufacturer='National Instruments', port='port0', version=5,␣
↪type=0))]
>> cam = PhotonFocus.PhotonFocus.PhotonFocusIMAQCamera(imaq_name="img0.iid", pfcam_
↪port=("National Instruments", "port0"))
>> cam.close()
>> cam = PhotonFocus.PhotonFocus.PhotonFocusIMAQCamera(imaq_name="img0.iid", pfcam_
↪port=1)  # same effect as above
>> cam.close()
```

Operation

The operation of these cameras is relatively standard. They support all the standard methods for dealing with ROI and exposure, starting and stopping acquisition, and operating the frame reading loop. However, there's a couple of differences from the standard libraries worth highlighting:

- The SDK also provides a universal interface for getting and setting various *camera attributes* (called "properties" in the documentation) using their name. You can use `IPhotonFocusCamera.get_attribute_value()` and `IPhotonFocusCamera.set_attribute_value()` for that, as well as `.cav` attribute which gives a dictionary-like access:

```
>> cam = PhotonFocus.PhotonFocusIMAQCamera()
>> cam.get_attribute_value("Window/W")  # get the ROI width
256
>> cam.set_attribute_value("ExposureTime", 0.1)  # set the exposure to 100ms
```

(continues on next page)

(continued from previous page)

```
>> cam.cav["ExposureTime"]  # get the exposure; could also use cam.get_attribute_
↪value("ExposureTime")
0.1
```

Some values (e.g., `Window.Max` or `Reset`) serve as commands; these can be invoked using `PhotonFocusIMAQCamera.call_command()` method. To see all available attributes, you can call `IPhotonFocusCamera.get_all_attributes()` to get a dictionary with attribute objects, and `IPhotonFocusCamera.get_all_attribute_values()` to get the dictionary of attribute values. The attribute objects provide additional information: attribute range, step, and units:

```
>> cam = PhotonFocus.PhotonFocusIMAQCamera()
>> attr = cam.get_attribute("Window/W")
>> attr.writable
True
>> (attr.min, attr.max)
(16, 1024)
```

- `PhotonFocus.PhotonFocusIMAQCamera` supports all of `IMAQ.IMAQCamera` features, such as trigger control and fast buffer acquisition. Some methods have been modified to make them more convenient: e.g., `PhotonFocusIMAQCamera.set_roi()` method sets the camera ROI and automatically adjusts the frame grabber ROI to match.
- Same is true for `PhotonFocus.PhotonFocusSiSoCamera`, which, e.g., provides access to all of the frame grabber variables.
- The camera supports a status line, which replaces the bottom one or two rows of the frame with encoded frame-related data such as frame number and timestamp. You can use `PhotonFocus.get_status_lines()` function to identify and extract the data in the status lines from the supplied frames. In addition, you can use `PhotonFocus.remove_status_line()` to remove the status lines in several possible ways: zeroing out, masking with the previous frame, cutting off entirely, etc.
- If several PhotonFocus cameras are connected, you need to correctly associate different PFCam ports with the corresponding frame grabbers. To do that, you can use the function `PhotonFocus.check_grabber_association()`.

Note: General camera communication concepts are described on the corresponding *page*

PCO SC2 cameras interface

SC2 is the interface used with PCO cameras. It has been tested with pco.edge cameras with CLHS and regular CameraLink interfaces. A detailed description of the interface is given in the manual.

The code is located in `pylablib.devices.PCO`, and the main camera class is `pylablib.devices.PCO.PCOSC2Camera`.

Software requirements

These cameras require `SC2_Cam.dll`, which is installed with the freely available pco.camware and pco.sdk tools. By default, the library searches for DLLs in `Digital Camera Toolbox/Camware4` and `PCO Digital Camera Toolbox/pco.sdk/bin` folder in `Program Files` folder (or `Program files (x86)`, if 32-bit version of Python is running), as well as in the folder containing the script. If the DLLs are located elsewhere, the path can be specified using the library parameter `devices/dlls/pco_sc2`:

```
import pylablib as pll
pll.par["devices/dlls/pco_sc2"] = "path/to/dlls"
from pylablib.devices import PCO
cam = PCO.PCOSC2Camera()
```

Connection

The cameras are identified by their index, starting from zero, and, possibly, by their interface. To get the total number of connected cameras, you can run *PCO.get_cameras_number*:

```
>> from pylablib.devices import PCO
>> PCO.get_cameras_number()
2
>> cam1 = PCO.PCOSC2Camera(idx=0)
>> cam2 = PCO.PCOSC2Camera(idx=1)
>> cam1.close()
>> cam2.close()
```

Operation

The operation of these cameras is relatively standard. They support all the standard methods for dealing with ROI and exposure, starting and stopping acquisition, and operating the frame reading loop. The class also provides read-access to all of the relevant camera data using *PCOSC2Camera.get_full_camera_data()*. This method returns data in the internal manufacturer format; to interpret it, you should consult the manual.

Known issues

- Some cameras support only ROIs which are symmetric with respect to vertical flip. In other words, if the camera detector has vertical size of 2160px, the vertical ROI should always have the form `(x0, 2160-x0)`. It is still possible to set non-symmetric ROI, but it is achieved by the software clipping, while the camera still reads out the smallest symmetric ROI contained the selected one. As a result, the readout time for the same ROI size strongly depends on the ROI position. For example, while vertical ROI of `(0, 8)` has only 8 pixel rows, it is not symmetric, and requires reading the whole frame; hence, it will be as slow as the full-frame acquisition. On the other hand, ROI of `(1076, 1084)` is symmetric, so the camera does read out only 8 rows. This results in vastly faster readout time. You can use *PCOSC2Camera.requires_symmetric_roi()* to check if the symmetric ROI is required.

Note: General camera communication concepts are described on the corresponding *page*

Silicon Software frame grabbers interface

Silicon Software produces a range of frame grabbers, which can be used to control different cameras with a CameraLink interface. It has been tested with microEnable IV AD4-CL frame grabber together with PhotonFocus MV-D1024E camera.

The code is located in *pylablib.devices.SiliconSoftware*, and the main camera class is *pylablib.devices.SiliconSoftware.SiliconSoftwareCamera*.

Software requirements

This interfaces requires `fglib5.dll`, which is installed with the freely available (upon registration) Silicon Software Runtime Environment (the newest version for 64-bit Windows is 5.7.0), which also includes all the necessary drivers. After installation, the path to the DLL (located by default in `SiliconSoftware/Runtime5.7.0/bin` folder in `Program Files`) is automatically added to system `PATH` variable, which is one of the places where pylablib looks for it by default. If the DLL is located elsewhere, the path can be specified using the library parameter `devices/dlls/sisofgrab`:

```
import pylablib as pll
pll.par["devices/dlls/sisofgrab"] = "path/to/dlls"
from pylablib.devices import SiliconSoftware
cam = SiliconSoftware.SiliconSoftwareCamera()
```

Connection

Figuring out the connection parameters is a multi-stage process. First, one must identify one of several boards. The boards can be identified using *`SiliconSoftware.list_boards`* function. Second, one must select an applet. These provide different board readout modes and, for Advanced Applets, various post-processing capabilities. These applets can be identified using *`SiliconSoftware.list_applets`* method, or directly from the Silicon Software RT microDisplay software supplied with the runtime. The choice depends on the color mode (color vs. gray-scale and different bitness), readout mode (area or line), and camera connection (single, double, or quad). Finally, depending on the board and the camera connection, one of several ports must be selected. For example, if the frame grabber has two connectors, but the camera only uses a single interface, then the double camera applet (e.g., `DualAreaGray16`) must be selected, and the port should specify the board connector (0 for `A`, 1 for `B`):

```
>> from pylablib.devices import SiliconSoftware
>> SiliconSoftware.list_boards()  # first list the connected boards
[TBoardInfo(name='mE4AD4-CL', full_name='microEnable IV AD4-CL')]
>> SiliconSoftware.list_applets(0)  # list all applets on the first board
[ ...,
TAppletInfo(name='DualAreaGray16', file='DualAreaGray16.dll'),
... ]
>> cam = SiliconSoftware.SiliconSoftwareCamera(0, 'DualAreaGray16')  # connect to the␣
↪first board (port 0 by default)
>> cam.close()
```

Note that currently the code is organized in such a way, that only one port on a single board can be in use at one time.

Operation

Unlike most camera classes, the frame grabber interface only deals with the frame transfer between the camera and the PC over the CameraLink interface. Therefore, in can not directly control camera parameters such as exposure, frame rate, triggering, ROI, etc. Some similar-looking parameters are still present, but they have a different meaning:

- External trigger controls frame *transfer*, not frame *acquisition*, which is defined by the camera. By default, when the internal frame grabber trigger is used, the frame grabber transfer rate is synchronized to the camera, so every frame gets transferred. However, if the external transfer trigger is used and it is out of sync with the camera, it can result in duplicate or missing frames.
- ROI is defined within the transferred image, whose size itself is determined by the camera ROI. Hence, e.g., if the camera chip is 1024x1024px and its roi is 512x512, then the frame grabber ROI can go only up to 512x512. Any attempts to set it higher result in frame being misshapen or having random data outside of the image area.

The SDK also provides a universal interface for getting and setting various *attributes* using their name. You can use `SiliconSoftwareCamera.get_grabber_attribute_value()` and `SiliconSoftwareCamera.set_grabber_attribute_value()` for that, as well as `.gav` attribute which gives a dictionary-like access:

```
>> cam = SiliconSoftware.SiliconSoftwareCamera()
>> cam.get_grabber_attribute_value("CAMERA_LINK_CAMTYP")  # get the camera data format
'FG_CL_SINGLETAP_8_BIT'
>> cam.set_grabber_attribute_value("WIDTH", 512)  # set the readout frame width to␣
↪512px
>> cam.gav["WIDTH"]  # get the width; could also use cam.get_grabber_attribute_value(
↪"WIDTH")
512
```

To see all available attributes, you can call `SiliconSoftwareCamera.get_all_grabber_attributes()` to get a dictionary with attribute objects, and `SiliconSoftwareCamera.get_all_grabber_attribute_values()` to get the dictionary of attribute values. The attribute objects provide additional information: attribute kind (integer, string, etc.), range (either numerical range, or selection of values for enum attributes), description string, etc.:

```
>> cam = SiliconSoftware.SiliconSoftwareCamera()
>> attr = cam.get_grabber_attribute("BITALIGNMENT")
>> attr.values
{1: 'FG_LEFT_ALIGNED', 0: 'FG_RIGHT_ALIGNED'}
```

The parameter can also be inspected in the Silicon Software RT microDisplay software.

Fast buffer readout mode

At high frame rates (above ~10kFPS) dealing with each frame individually becomes too slow for Python. Hence, there is an option to read out and process frames in larger 'chunks', which are 3D numpy arrays with the first axis enumerating the frame index. This approach leverages the ability to store several frame buffers in the contiguous memory locations (resulting in a single 3D array), and it essentially eliminates the overhead for dealing with multiple frames at high frame rates, as long as the total data rate is manageable (typically below 600Mb/s).

This option can be accessed by supplying `fastbuff=True` in `SiliconSoftwareCamera.read_multiple_images()`. In this case, instead of a list of individual frames (which is the standard behavior), the method returns list of chunks about 1Mb in size, which contain several consecutive frames. Otherwise the method behaves identically to the standard one.

Communication with the camera

The frame grabber needs some basic information about the camera: sensor size, bit depth, data transfer format, timeouts, aux lines mapping. This information can be specified using the grabber attributes. The most important transfer parameters are the number of taps and the bitness of the transferred data, which can be set up using `SiliconSoftwareCamera.setup_camlink_pixel_format()`. The values for this parameters can usually be obtained from the camera manuals.

Known issues

- The maximal frame rate is limited for some boards (at least for the tested microEnable IV AD4-CL board) by about 20kFPS. It seems to be relatively independent of the frame size, i.e., it is not the data transfer rate issue. One possible way to get around it is to use line readout applet, e.g., `DualLineGray16`, and set the frame height to be the integer multiple of the camera frame. This will combine several camera frames into a single frame-grabber frame, effectively lowering the frame rate at avoiding the issue. However, this sometimes leads to incorrect frame splitting: the top line of the "combined" frame does not coincide with the top line of the original camera frame, so all frames are shifted cyclically by some number of rows. Hence, it might require some post-processing with frames merging and re-splitting.

- As mentioned above, ROI is defined within a frame transferred by the camera. Therefore, if it includes pixels with positions outside of the transferred frame, the acquisition will be faulty. For example, suppose the camera sensor is 1024x1024px, and the *camera* ROI is selected to be central 512x512 region. As far as the frame grabber is concerned, now the camera sensor size is 512x512px. Hence, if you try to set the same *frame grabber* ROI (i.e., 512x512 starting at 256,256), it will expect 768x768px frame. Since the frame is, actually, 512x512px, the returned frame will partially contain random data. The correct solution is to set frame grabber ROI from 0 to 512px on both axes. In general, it is a good idea to always follow this pattern: control ROI only on camera, and always set frame grabber ROI to cover the whole transfer frame.

Note: General camera communication concepts are described on the corresponding *page*

Thorlabs Scientific Cameras interface

This is the interface used in Thorlabs scientific sCMOS cameras such as Kiralux or Zelux. It has been tested with Thorlabs Kiralux camera.

The code is located in `pylablib.devices.Thorlabs`, and the main camera class is `pylablib.devices.Thorlabs.ThorlabsTLCamera`.

Software requirements

These cameras require `thorlabs_tsi_camera_sdk.dll`, as well as several additional DLLs: `thorlabs_unified_sdk_kernel.dll`, `thorlabs_unified_sdk_main.dll`, `thorlabs_tsi_usb_driver.dll`, `thorlabs_tsi_usb_hotplug_monitor.dll`, `thorlabs_tsi_cs_camera_device.dll`, `tsi_sdk.dll`, `tsi_usb.dll`. All of them is automatically installed with the freely available ThorCam tools. By default, the library searches for DLLs in `Thorlabs/Scientific Imaging/ThorCam` folder in `Program Files` folder (or `Program files (x86)`, if 32-bit version of Python is running), as well as in the folder containing the script. If the DLLs are located elsewhere, the path can be specified using the library parameter `devices/dlls/thorlabs_tlcam`:

```
import pylablib as pll
pll.par["devices/dlls/thorlabs_tlcam"] = "path/to/dlls"
from pylablib.devices import Thorlabs
cam = Thorlabs.ThorlabsTLCamera()
```

Connection

The cameras are identified by their serial number. To list all of the connected cameras, you can run `Thorlabs.list_cameras_tlcam`:

```
>> from pylablib.devices import Thorlabs
>> Thorlabs.list_cameras_tlcam()
['12001', '12002']
>> cam1 = Thorlabs.ThorlabsTLCamera(serial="12001")
>> cam2 = Thorlabs.ThorlabsTLCamera(serial="12002")
```

(continues on next page)

(continued from previous page)

```
>> cam1.close()
>> cam2.close()
```

If no serial is provided, the software connects to the first available camera.

Operation

The operation of these cameras is relatively standard. They support all the standard methods for dealing with ROI and exposure, starting and stopping acquisition, and operating the frame reading loop.

Warning: The library appears to be not entirely stable: every time acquisition start is issued, there is small (0.1-1%) chance that it will not actually start, which results in timeout errors. Furthermore, there are occasional crashes on the SDK unloading (i.e., camera closing), especially when acquisition has been started and stopped multiple times. It is unclear, what is the cause of this behavior, but it seems to originate from the manufacturer's DLL (bare-bones example and the native Python library reproduce this behavior). Hence, it might be different with different DLL versions.

Note: The DLL prints some debug information in the console when camera list is requested and when the camera is opened. At the moment, it is unclear how to get rid of it.

Note: General camera communication concepts are described on the corresponding *page*

Uc480 camera interface

This is the interface used in multiple cameras, including simple Thorlabs and IDS cameras. It has been tested with IDS SC2592R12M and Thorlabs DCC1545M.

The code is located in `pylablib.devices.uc480`, and the main camera class is `pylablib.devices.uc480.UC480Camera`.

Software requirements

These cameras require `uc480.dll`, which is automatically installed with the freely available ThorCam tools. By default, the library searches for DLLs in `Thorlabs/Scientific Imaging/ThorCam` folder in `Program Files` folder (or `Program files (x86)`, if 32-bit version of Python is running), as well as in the folder containing the script. If the DLLs are located elsewhere, the path can be specified using the library parameter `devices/dlls/uc480`:

```python
import pylablib as pll
pll.par["devices/dlls/uc480"] = "path/to/dlls"
from pylablib.devices import uc480
cam = uc480.UC480Camera()
```

Connection

The cameras are identified by their camera ID or device ID (both starting from 1). Device ID corresponds to the connection order of the cameras: it is guaranteed to be unique, but will change if the camera is disconnected and reconnected again. On the other hand, camera ID is tied to the camera, but it is set to 1 by default for all cameras, and needs to be manually assigned using `UC480Camera.set_camera_id()`. Alternatively, one can use other characteristics (model or serial number) as a unique identifier. To list all of the connected cameras together with their basic information, you can run `uc480.list_cameras()`:

```
>> from pylablib.devices import uc480
>> uc480.list_cameras()
[TCameraInfo(cam_id=4, dev_id=1, sens_id=11, model='SC2592R12M', serial_number=
↪'1234567890', in_use=False, status=0)]
>> cam = uc480.UC480Camera(cam_id=4)  # connect to the camera using cam_id
>> cam.close()
>> cam = uc480.UC480Camera(dev_id=1)  # connecting to the same camera using dev_id
>> cam.close()
>> cam = uc480.UC480Camera()  # connecting to the first available camera
>> cam.close()
```

If `cam_id = 0` is provided (default), the software connects to the first available camera.

Operation

The operation of these cameras is relatively standard. They support all the standard methods for dealing with ROI and exposure, starting and stopping acquisition, and operating the frame reading loop. However, there's a couple of differences from the standard libraries worth highlighting:

- Some cameras support both binning (adding several pixels together) and subsampling (skipping some pixels). However, only one can be enabled at a time. They can be set independently using, correspondingly, `UC480Camera.get_binning()`/`UC480Camera.set_binning()` and `UC480Camera.get_subsampling()`/`UC480Camera.set_subsampling()`. They can also be set as binning factors in `UC480Camera.get_roi()`/`UC480Camera.set_roi()`. Whether binning or subsampling is set there can be determined by the `roi_binning_mode` parameter supplied on creation.
- Uc480 API supports many different pixel modes, including packed ones. However, pylablib currently supports only monochrome unpacked modes.
- Occasionally (especially at high frame rates) frames get skipped during transfer, before they are placed into the frame buffer by the camera driver. This can happen in two different ways. First, the frame is simply dropped without any indication. This typically can not be detected without using the framestamp contained in the frame info, as the frames flow appear to be uninterrupted. In the second way, the acquisition appears to get "restarted" (the internal number of acquired frames is dropped to zero), which is detected by the library. In this case there are several different ways the software can react, which are controlled using `UC480Camera.set_frameskip_behavior()`.

  The default way to address this "restart" event (`"ignore"`) is to ignore it and only adjust the internal acquired frame counter; this manifests as quietly dropped frames, exactly the same as the first kind of event. In the other method (`"skip"`), some number of frames are marked as skipped, so that the difference between the number of acquired frames and the internal framestamp is kept constant. This makes the gap explicit in the camera frame counters. Finally (`"error"`), the software can raise `uc480FrameTransferError` when such event is detected, which can be used to, e.g., restart the acquisition.

  One needs to keep in mind, that while the last two methods make "restarts" more explicit, they do not address the first kind of events (quiet drops). The most direct way to deal with them is to use frame information by setting `return_info=True` in frame reading methods like `read_multiple_images`. This information contains the internal camera framestamp, which lets one detect any skipped frames.

1.2.3 Stages

Basic concepts are described at the *stages communication page*.

Currently supported stages:

- *Attocube ANC300* and *Attocube ANC350*: most common Attocube positioner controllers. Tested with Ethernet and USB connection for ANC300, and USB connection for ANC350.
- *Thorlabs APT/Kinesis*: basic Thorlabs motorized stages and optomechanics devices. Tested with KDC101 and K10CR1, as well as MFF101 and FW102 (described at a *different page*)
- *Newport Picomotor*: precision piezo-actuated screws based on slip-stick principle. Tested with Newport 8742 Picomotor driver using Ethernet or USB connection.
- *Arcus Performax*: fairly common single- and multi-axis motor controllers sold under different brands: Arcus, Nippon Pulse America, or Newmark Systems. Tested with PMX-4EX device with USB connection.
- *Trinamic*: universal motor controllers and drivers. Tested with a single-axis TMCM-1110 controller with USB connection.
- *SmarAct*: high-performance piezo sliders. So far only simple open-loop *SCU controllers* are supported. Tested with a standard HCU controller unit.

Note: General device communication concepts are described on the corresponding *page*.

Stages control basics

Basic example

Almost all stages implement the same basic functionality for moving, stopping, homing, and querying the status:

```
stage = Thorlabs.KinesisMotor("27000001")  # connect to the stage
stage.home()  # home the stage
stage.wait_for_home()  # wait until homing is done
stage.move_by(1000)  # move by 1000 steps
stage.wait_move()  # wait until moving is done
stage.jog("+")  # initiate jog (continuous move) in the positive direction
time.sleep(1.)  # wait for 1 second
stage.stop()  # stop the motion
stage.close()
```

Some stages will miss some of this functions (e.g., no homing), but if it's present, it works roughly in the same manner.

Some concepts are explained below in more detail.

Basic concepts

Counters, encoders, homing, and limit switches

Stages have two basic strategies for keeping track of the position. The first one is counting the steps. The problem with it is that once the device is powered up, its position in unknown. Hence, it requires some kind of homing procedure, which usually involves moving to a predefined position and zeroing out the step counter there. This position is defined by the hardware, usually in the form of a limit switch: a physical switch located at the end of the stage travel range, which changes the state when the stage reaches its position. It also usually automatically turns off the motion when tripped, to prevent the motor from overheating or the stage from breaking.

When stepper motors are used, the size of each step (or microstep, if used) is a reasonably well-defined fraction of a turn, so counting them gives fairly reproducible results. On the other hand, piezo slip-stick sliders (such as Attocube, SmarAct, or Picomotor) have inherently unreliable steps size which depends on, e.g., load, direction, position, temperature, or other environmental factors. In this case steps counting, while possible, usually leads to long-term drifts.

If the reliable counting is impossible, like in the case of sliders or regular DC (as opposed to stepper) motors, the manufacturer might add a hardware position readout. It can be digital (encoder) or analog (e.g., resistive, capacitive, or optical readout). The first kind is generally simpler, cheaper and more reliable, but the second one can provide much higher resolution, and can work in more extreme environments (high vacuum, cryogenics). In both cases, the controllers would typically have some kind of feedback loop to smoothly control the motion speed and direction to approach a given position.

Steps and real coordinates

Almost all stages allow control or readout of position in motor steps, encoder steps, or some other internal units. It is usually not straightforward, or sometimes even impossible, to convert those to real units. In cases where it is possible, it is defined by the motor gearbox and the screw pitch (for linear stages); in most cases, this ratio is provided in the motor or translation stage manual (which can be different from the motor controller manual, and the two might even be completely independent). Sometimes, one even has to do explicit calculations, e.g., getting the number of microsteps per revolution from the controller and motor manufacturer, and the displacement per step from the stage manufacturer.

Speed control

In many cases, the motor speed is ramped up and down linearly rather than abruptly; hence, both the “cruising” speed and the ramping acceleration can, in principle, be configured. Usually they are defined in, respectively, steps/s and steps/s^2, although sometimes internal units have to be used.

Application notes and examples

Here we talk more practically about using pyalblib to perform common tasks.

Motion

The most standard motion methods are `move_to`, which moves to a specified position, `move_by`, which moves by a specified distance or number of steps, and `jog`, which moves continuously in a given direction until stopped or run into a limit switch. If both `move_to` and `move_by` are present, they usually perform the same operation under the hood: `stage.move_by(s)` and `stage.move_to(stage.get_position()+s)` yield the same result.

In almost all cases these commands are asynchronous, in the sense that they simply initialize the motion and continue immediately:

```
>> stage.move_by(1000)
>> stage.is_moving()  # the stage is moving, but the execution continues
True
>> time.sleep(1.)
>> stage.is_moving()  # after 1s the motion is done
False
```

To stop immediately (which is usually only used with `jog` commands) you can use the `stop` method. In some cases, there are two different stop kinds: “soft” with a ramp-down, or “hard” which immediately ceases motion.

Status and synchronization

Since the motion commands are asynchronous, the devices provide two methods to synchronize it with the script execution. The first one, `is_moving`, checks if the stage is currently in motion. The second one, `wait_move`, pauses the execution until the stage motion is finished.

In addition, many stages provide methods to obtain additional information, e.g., `get_status` (which, usually, returns state of motion, limit switches, possible errors, etc.), or `get_current_speed`.

Position readout

If a stage has position readout (either hardware sensor, or step counting), it is implemented with the `get_position` method. In most cases, it will be accompanied with the `set_position_reference` method, which lets one change the currently stored position, effectively adding an offset to all further position readings:

```
>> stage.get_position()
10000
>> stage.set_position_reference(20000)  # change current reference to
>> stage.get_position()  # note that it reacts immediately, unlike move_to; no␣
↪physical motion happened
20000
>> stage.move_to(21000)  # move by 1000 steps; equivalent to .move_by(1000), or .move_
↪to(11000) before the reference change
```

Note that it only changes the internal counter state, and does not cause any stage motion (which is performed by `move_to`).

Axis selection

Many controllers support simultaneous control of several different motors. In this case, all of their methods take an additional `axis` (in most cases) or `channel` argument, which specify the exact motor. In cases where usually only one motor is controlled (e.g., TMCM1110 or Thorlabs KDC101), this parameters is set to the default value, and is closer to the end of the parameter list. If having multiple controlled stages is the default (e.g., Attocube ANC350 or Arcus Performax), this parameter is usually the first one, and it has to be specified. In this cases, the methods frequently allow to set this parameter to `"all"`, which means that the action is performed for all axes, or the results is returned for all axes (usually in a form of a list or a dictionary).

The channels are usually specified by their index starting from 0 or 1, although some stages adopt a different labeling (e.g., Arcus Performax labels them as X, Y, Z, and U). The exact specification is given in the specific class description.

Homing

As mentioned above, often stages require homing to get absolute position readings. It needs to be done every time the stage is power-cycled, but the homing parameters usually persist between different re-connections.

If homing is implemented, it is done using the `home` method. In addition, there can also be an `is_homed` method, which checks if the homing has already been performed. If the method is present, then by default `home` will not execute if `is_homed` returns `True`, unless forced.

Some stages do not have an explicit homing method, but can be manually homed by, e.g., running the stage to the limit switch and setting the position reference to 0.

Note: General stage communication concepts are described on the corresponding *page*

Attocube positioners

Attocube has two main positioner controllers: ANC300 and ANC350. These cover different but somewhat overlapping positioner classes, and have fairly different programming interfaces.

Attocube ANC300

This controller is aimed at open-loop (i.e., no position readout) positioners. It is a chassis with a single PC communication module and up to 7 individual piezo control modules: ANM150 (only stepping), ANM200 (only scanning), or ANM250 (stepping and scanning).

The device class is *`pylablib.devices.Attocube.ANC300`*.

Software requirements

The controller has several communication modes: USB, RS232, and Ethernet. USB mode requires a driver supplied with the controller (or downloaded from the controller itself using its Ethernet connection and HTTP port), which makes ANC300 appear as a virtual COM port. RS232 requires a USB-to-RS232 adapter, which usually manifests in the same way. Finally, Ethernet connection works like any other networks device. The controller has been tested with USB and Ethernet communication modes (RS232 is identical to USB, so it should operate as well).

Of all of these modes only USB requires specialized drivers, and the other two are usually available purely through the built-in OS capabilities.

Connection

The device is identified by its communication address. It can be either a serial port (e.g., `"COM5"`), or an IP address (e.g., `"192.168.1.100"`); see *connection description* for more information. The backend is chosen automatically based on the connection parameter. Additionally, Ethernet connection requires a password; by default, the standard Attocube password `"123456"` is used, but if you specified a custom password, you need to provide it upon connection:

```
>> from pylablib.devices import Attocube
>> atc1 = Attocube.ANC300("COM5")  # USB or RS232 connection
>> atc2 = Attocube.ANC300("192.168.1.1", pwd="root")  # Ethernet connection; no need␣
↪to provide a password, if it is default
```

(continues on next page)

(continued from previous page)

```
>> atc1.close()
>> atc2.close()
```

Note that since Ethernet inherently supports multiple connections, it is possible to control the same devices in multiple scripts at the same time.

Operation

This controller has several features and differences compared to most other stages and sliders:

- The controller is inherently multi-axis, hence it always take the axis as the first argument. The axes are numbered starting from 1, and are addressed according to the chassis spaces, so some can be skipped or missing. To update the list of connected axes, use `ANC300.update_available_axes()` (called automatically on connection).
- Different control modules provide different functionality. Hence, not all methods would work for all axes: offset voltage commands such as `ANC300.set_offset()` do not work with ANM150 module, while stepping commands such as `ANC300.move_by()` do not work with ANM200 module. To get the module kinds and serial numbers, use `ANC300.get_axis_serial()`.
- The most important stepping parameters are step voltage amplitude and step frequency (number of steps per second). These can be controlled with, correspondingly, `ANC300.get_voltage()`/`ANC300.set_voltage()` and `ANC300.get_frequency()`/`ANC300.set_frequency()`.
- Different axes can be enabled and disabled (i.e., connected or grounded) using `ANC300.enable_axis()` and `ANC300.disable_axis()`. Disabling an axis completely shuts off the connection to the positioner, which usually reduces the noise. In addition, there can be different operation modes for only offset, only stepping, or combination of the two.
- It is possible to measure the positioner capacitance using `ANC300.get_capacitance()`, which is useful in identifying breaks or shorts in the wiring or faults in the piezos. By default, this method simply returns the last measured value. To re-measure, call it with `measure=True`. Note that after the measurement is done, the axis is automatically disabled, and needs to be enabled explicitly:

  ```
  >> atc = ANC300("COM5")
  >> atc.get_capacitance(1, measure=True)  # get the capacitance (in F) on the␣
  ↪first axis; the method waits until the measurement is done (about 1s)
  200E-9
  >> atc.is_enabled(1)
  False
  ```

  Note that this is also the only way to know if there is an actual positioner connected to the given control module.

Attocube ANC350

This controller is aimed at closed-loop (i.e., with position readout) positioners. It can control up to 3 positioners.

The device class is `pylablib.devices.Attocube.ANC350`.

Software requirements

The controller has USB and Ethernet modes. USB mode requires a driver supplied with the controller. The communication is done via PyUSB, which means that it does not require any additional Attocube DLLs, although you might need to install libusb (see PyUSB for more details). Ethernet control is supplied as an additional purchaseable option and can be configured using the supplied Daisy control software.

This device has only been tested with a USB connection.

Connection

When using a USB connection, the device is identified by its index among all the connected ANC350 devices. To get the total number of devices, you can use `Attocube.get_usb_devices_number_ANC350`:

```
>> from pylablib.devices import Attocube
>> Attocube.get_usb_devices_number_ANC350()
2
>> atc1 = Attocube.ANC350()  # use 0 index by default
>> atc2 = Attocube.ANC350(1)
>> atc1.close()
>> atc2.close()
```

Ethernet connection should work in the same manner as any other similar devices, i.e., the address and, possibly, the port should be provided.

Operation

This controller has several features and differences compared to most other stages and sliders:

- The controller is inherently multi-axis, hence it always take the axis as the first argument. The axes are numbered 0 through 2. You can check if the slide is connected to the given axis using `ANC350.is_connected()`.
- Different axes can be enabled and disabled (i.e., connected or grounded) using `ANC300.enable_axis()` and `ANC300.disable_axis()`. Disabling an axis completely shuts off the connection to the positioner, which usually reduces the noise.
- It is also possible to control the sensor voltage using `ANC350.get_sensor_voltage()`/`ANC350.set_sensor_voltage()` methods. Reducing this voltage lowers the heating produced by the sensor, which becomes especially important at very low (<1K) temperatures.
- The most important stepping parameters are step voltage amplitude and step frequency (number of steps per second). These can be controlled with, correspondingly, `ANC350.get_voltage()`/`ANC350.set_voltage()` and `ANC350.get_frequency()`/`ANC350.set_frequency()`.
- It is possible to measure the positioner capacitance using `ANC350.get_capacitance()`, which is useful in identifying breaks or shorts in the wiring. By default, this method simply returns the last measured value. To re-measure, call it with `measure=True`.
- Fine positioning is performed using the position readout and the feedback loop. Then a `move_to`/`move_by` command is issued, this feedback loop is activated, and the positioner tries to reach and stay at the current position. You can use `ANC350.is_target_reached()` to check if the target is reached, `ANC350.get_target_position()` to get the target, and `ANC350.get_precision()`/`ANC350.set_precision()` to control the target precision.
- In addition, there is a method `ANC350.move_by_steps()`, which mimics `ANC300.move_by()` by moving for a given number of steps instead of a given distance. However, due to implementation limitations, this method is synchronous, i.e., it waits until all steps are performed. Nevertheless, `ANC350.jog()` is still asynchronous.

Note: General stage communication concepts are described on the corresponding *page*

Thorlabs APT/Kinesis devices

Thorlabs has a variety of APT/Kinesis devices for various motion-related functionality (mostly motor controllers and piezo drivers), which share the same API. The library uses an older and more low-level APT protocol to communicate with these devices. So far it has been only implemented for motor controllers and some *specialized devices* and tested with KDC101 and K10CR1 controllers.

The main device classes are `pylablib.devices.Thorlabs.BasicKinesisDevice` for a generic Kinesis/APT devices and `pylablib.devices.Thorlabs.KinesisMotor` aimed at motor controllers such as K10CR1 or KDC101.

Software requirements

The connection is done using Thorlabs APT protocol, so it needs the corresponding APT drivers. Pylablib communicates directly with the FTDI USB-to-RS232 using pyft232 chip inside the controller, so it bypasses most of the Thorlabs software. This means that it does not need any Thorlabs-supplied DLLs, but it also means that it can not work with the simulated devices, since these are simulated on a level above the direct serial communication.

In some cases `pyft232` library can not find the required `ftd2xx.dll` library, which leads to an error. There are several ways to get around this. First, you can install the FTDI drivers from the manufacturer's website. Setup executable for Windows automatically places the necessary DLL into the `System32` folder, where `pyft232` can discover them. Alternatively, you can copy the DLLs there yourself from the Thorlabs APT installation. Their default location is `Program Files\Thorlabs\APT\Drivers\APT\USB Driver\amd64` for 64-bit version or `Program Files\Thorlabs\APT\Drivers\APT\USB Driver\i386` for 32-bit version. Note that in the first case the file is called `ftd2xx64.dll`, and you will need to rename it to `ftd2xx.dll` when copying to the `System32` folder.

Connection

The devices are identified by their address, which correspond to their serial numbers. To get the list of all the connected devices, you can use `Thorlabs.list_kinesis_devices`:

```
>> from pylablib.devices import Thorlabs
>> Thorlabs.list_kinesis_devices()
[('27500001', 'Kinesis K-Cube  DC Driver')]
>> stage = Thorlabs.KinesisMotor("27500001")
>> stage.close()
```

Operation

This controller has several features and differences compared to most other stages and sliders:

- There are several different ways to specify the stage calibration, which are controlled by the `scale` parameter supplied upon the connection. By default (`scale = "step"`), it accepts and returns position in motor steps, velocity in steps/s and acceleration in steps/s^2 (scaling coefficients for the latter two are determined from the controller model). If `scale = "stage"`, the class attempts to autodetect the stage and use meters or degrees instead of steps; in addition you can supply the stage name (e.g., `"MTS25-Z8"`) as a scale instead of relying on the autodetection. If there is no calibration for the stage that you have, you can instead supply a single scaling factor, which specifies the number of steps per physical unit (e.g., for "MTS25-Z8" stage and mm units, one would supply `scale = 34304`). The stage scaling can be obtained from the APT manual. Finally, one can supply a 3-tuple of scales for position, velocity and acceleration (all relative to the internal units). The details are given in the APT manual. To ensure that the units have been applied and/or autodetected correctly, you can use `KinesisMotor.get_scale()`, `KinesisMotor.get_scale_units()` and `KinesisMotor.get_stage()` methods.

- By default, the controllers are treated as single-axis. If several axes are supported, they can be specified using `channel` argument.
- The motor power-up parameters for homing, jogging, limit switches, etc., can be different from the parameters showing up in the APT/Kinesis controller. This can lead to problems if, e.g., homing speed is too low, so the motor appears stationary while homing. You should make sure to check those parameters using `KinesisMotor.get_velocity_parameters()`, `KinesisMotor.get_jog_parameters()`, `KinesisMotor.get_homing_parameters()`, `KinesisMotor.get_gen_move_parameters()`, and `KinesisMotor.get_limit_switch_parameters()`.

Note: General stage communication concepts are described on the corresponding *page*

Newport Picomotor controller

Newport Picomotor is a series of actuators, usually in a screw format, based on the slip-stick piezo actuation mechanism (similar to, e.g., Attocubes). Operating them requires a driver/controller to output specific voltage pulses. The basic modern open-loop controller is Newport 8742, which can drive up to 4 actuators (but only one at a time), supports connection via USB or Ethernet, and can be daisy-chained to communicate with several controllers through one connection. The class has been tested with this controller and a single standard actuator.

The device class is `pylablib.devices.Newport.Picomotor8742`.

Software requirements

The controller has two communication modes: USB, and Ethernet. USB mode requires a driver supplied with the freely available PicomotorApp software, while Ethernet connection works like any other networks device and does not require any additional software. The controller has been tested both with USB and Ethernet communication modes.

Connection

When using the USB connection, the device is identified by its index, starting from 0. To get the number of connected devices, you can use `Newport.get_usb_devices_number_picomotor`:

```
>> from pylablib.devices import Newport
>> Newport.get_usb_devices_number_picomotor()
2
>> stage1 = Newport.Picomotor8742()
>> stage2 = Newport.Picomotor8742(1)
>> stage1.close()
>> stage2.close()
```

Ethernet connection requires a host name or an IP address. Both can be set up by first connecting the device via USB or by using the PicomotorApp software (in the `Setup -> Ethernet` menu). After that, they can be supplied to the class instead of index:

```
>> from pylablib.devices import Newport
>> stage1 = Newport.Picomotor8742("8742-12345")  # by default, all host names start␣
↪with 8742
>> stage1.close()
```

Operation

This controller has several features and differences compared to most other stages and sliders:

- The controller is inherently multi-axis, hence it always take the axis as the first argument. The axes are labeled numerically starting from 1 (i.e., 1, 2, 3, and 4). The list of all axes is related to the exact controller, an can be obtained using `Picomotor8742.get_all_axes()`.
- There is an option to auto-detect motors and their kind using `Picomotor8742.autodetect_motors()` method. However, since it involves stepping the motor, it usually makes more sense to detect them once and then store them into the non-volatile (i.e., power-independent) memory using `Picomotor8742.save_parameters()`.
- Even open-loop controllers support absolute positioning, which is achieved simply by counting steps in both directions. However, unlike stepper motors or encoders, these steps can be different depending on the direction, position, instantaneous load, speed, etc. Hence, the absolute positions quickly become unreliable. It is, therefore, recommended to generally use relative positioning using `Picomotor8742.move_by()` method.
- As mentioned above, the controller support daisy-chaining using RS-485 connections. It allows to connect several controllers together while still only using a single PC connection. In this case, it is recommended to supply `multiaddr=True` upon connecting to the device. If, in addition `scan=True` is set (default), then upon connection the controller scans for all other connected devices, resolves their address conflicts, and builds the list of the available addresses (address is a number between 1 and 31). The list can later be read using `Picomotor8742.get_addr_map()`, and the network rescanned using `Picomotor8742.scan_devices()`. To refer to a specific device, its address should be specified using `addr` parameter of a method; by default it is set to `None`, which selects the device connected to the PC.

Note: General stage communication concepts are described on the corresponding *page*

Arcus Performax positioners

Arcus has several motor controllers and drivers, which are mainly different in their number of axes, communication possibilities, and driving function. They are also distributed under different names, e.g., Nippon Pulse America (NPA) or Newmark Systems. However, the models nomenclature is the same: there is 4EX for 4-axis controllers with USB and RS485 connection, 2EX/2ED for 2-axis controllers with USB and RS485 connections, and 4ET for 4-axis controllers with Ethernet connection. The class has been tested with 4EX and (partially) 2ED controllers with USB connectivity mode, but other controllers and modes mentioned above should also work.

The main device classes are `pylablib.devices.Arcus.Performax4EXStage` or 4-axis controllers and `pylablib.devices.Arcus.Performax2EXStage` for 2-axis controllers. In addition to a different number of axes, they have several syntax differences, so one can not substitute for the other.

Software requirements

The controller has several communication modes: USB, RS485, and Ethernet. USB mode requires a driver supplied with the operation software: Arcus Drivers and Tools, Performax Series Installer, and Performax USB Setup (all obtained at Arcus website). Installing all three seem to be sufficient. Once the appropriate USB drivers are installed, one can connect the device directly via its USB port and use the manufacturer DLLs `PerformaxCom.dll` and `SiUSBXp.dll` to communicate with the device. They can be obtained on the manufacturer's website and placed in the folder with the script, or in the `System32` Windows folder. If the DLL is located elsewhere, the path can be specified using the library parameter `devices/dlls/arcus_performax`:

```
import pylablib as pll
pll.par["devices/dlls/arcus_performax"] = "path/to/dll"
from pylablib.devices import Arcus
stage = Arcus.Performax4EXStage()
```

The controller has only been tested with USB communication.

> **Warning:** There appear to be some issues with Python 3.6 which result in out-of-bounds write, memory corruption, and undefined behavior. Hence, Python 3.7+ is required to work with this device.

Connection

When using the USB connection, the device is identified by its index, starting from 0. To get the list of all the connected devices, you can use *Arcus.list_usb_performax_devices*:

```
>> from pylablib.devices import Arcus
>> Arcus.list_usb_performax_devices()
[(0, '4ex01', 'Performax USB',
 '\\\\?\\usb#vid_1589&pid_a101#4ex01#{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}', '1589',
↪'a101'),
 (1, '4ex21', 'Performax USB',
 '\\\\?\\usb#vid_1589&pid_a101#4ex21#{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}', '1589',
↪'a101')]
>> stage1 = Arcus.Performax4EXStage()
>> stage2 = Arcus.Performax2EXStage(idx=1)
>> stage1.close()
>> stage2.close()
```

Operation

This controller has several features and differences compared to most other stages and sliders:

- The controller is inherently multi-axis, hence it always take the axis as the first argument. The axes are labeled with letters `"x"`, `"y"` for a 2-axis version, or `"x"`, `"y"`, `"z"`, `"u"` for a 4-axis one. The list of all axes is related to the exact controller, an can be obtained using *Performax4EXStage.get_all_axes()*.
- Different axes can be enabled and disabled using *Performax4EXStage.enable_axis()*. Note that disabled axes still behave the same as the enabled ones; e.g., their position will increment as usual, when `move_to` is called. This can lead to some confusion, as the axis appears mostly operational, but the motor does not move.
- In the default controller configuration the limit errors are enabled. In this case, once a single axes reaches the limit switch during motion, it is put into an error state, which immediately stops this an all other axes. Any further motion command on this axis will raise an error, although it is still possible to restart motion on other axes. The axis motion can only be resumed by calling *Performax4EXStage.clear_limit_error()*. If, however, limit errors are disabled, then only the axis which reached the limit is stopped, and all other axes are unaffected. Furthermore, the motion on the offending axis can be resumed without clearing its error status. In many cases the default limit error behavior is undesirable, so the class turns it off upon connection. It can be

subsequently turned on and off using *Performax4EXStage.enable_limit_errors()*, and checked using *Performax4EXStage.limit_errors_enabled()*.

- The controllers also have analog and digital inputs and digital outputs, which can be queried and set with the corresponding commands.
- The controller has an option to connect an encoder for a separate position readout. By default, all of the commands (e.g., for moving, getting position, getting current speed, etc.) still work in the step-counting mode, and the encoder values are only accessed via *Performax4EXStage.get_encoder()*/*Performax4EXStage.set_encoder_reference()*. In principle, there is a closed-loop mode call `StepNLoop`, but it is not currently supported in the code.
- The built-in motion command has 2 modes: relative and absolute. The code sets the absolute mode on connection and assumes it in all commands. However, if the mode changes for any reason, the move commands will stop working properly.

Note: General stage communication concepts are described on the corresponding *page*

Trinamic TMCM-1110 controller

TMCM-1110 is a universal single-axis stepper motor controller from Trinamic. It provides multiple connection options, but so far has only been tested with USB connection.

The main device class is *pylablib.devices.Trinamic.TMCM1110*.

Software requirements

USB connection needs drivers, which are supplied with the freely-available TMCL-IDE, or TMCL-LITE. With those drivers installed, the controllers show up as virtual COM ports. Note that when several devices are connected, they sometimes get assigned conflicting (i.e., overlapping) COM ports. In this case, you might need to manually reassign these in the Device Manager.

Connection

Since the devices are identified as virtual COM ports, they use the standard *connection method*, and all you need to know is their COM-port address (e.g., `COM5`):

```
>> from pylablib.devices import Trinamic
>> stage1 = Trinamic.TMCM1110("COM5")
>> stage2 = Trinamic.TMCM1110("COM8")
>> stage1.close()
>> stage2.close()
```

Operation

This controller has several features and differences compared to most other stages and sliders:

- The controller allows one to control the number of microsteps per step using *TMCM1110.get_microstep_resolution()* and *TMCM1110.set_microstep_resolution()*. Hence, the calibration of the real position to the controller readout position depends on this resolution. Furthermore, changing this resolution does not affect the step counter, meaning that changing it, performing a move, and changing it back will result in a different position. Hence, it is not recommended to change it after homing or referencing the position.

- Similarly, the controller has variable frequency divisors, which control the ratio between internal and real units for the velocity and the acceleration. They are set up together with the maximal velocity and acceleration using `TMCM1110.setup_velocity()` and `TMCM1110.get_velocity_parameters()`, and the conversion factors can be obtained using `TMCM1110.get_acceleration_factor()` and `TMCM1110.get_velocity_factor()`.
- The device has an option of controlling maximal output current using `TMCM1110.setup_current()` and `TMCM1110.get_current_parameters()`. Change them carefully, since the values which are too large can damage the motor. Also take into account, that the currents are defined relative to the maximal output current, which is controlled using the physical jumper on the board.

Note: General stage communication concepts are described on the corresponding *page*

SmarAct positioners

SmarAct has multiple different controller covering different slider kinds. So far only simple controllers (CU/HCU/SCU) are implemented.

SmarAct CU/HCU/SCU

This is a simple controller, which is mostly aimed at open-loop (i.e., no position readout) positioners. It can control up to 3 axes, and connects to the PC via the USB port.

The device class is `pylablib.devices.SmarAct.SCU3D`. Currently only open-loop controllers are supported.

Software requirements

The controller shows up as a virtual COM port, and it has a standard FTDI chip, so it does not need any special drivers. However, to communicate with the device, it still needs `SCU3DControl.dll` library. It is supplied on a CD together with the device, although it might also be possible to request it from SmarAct.

Connection

The devices are identified by their index starting from 0. To get the list of all the connected devices, you can use `SmarAct.list_scu_devices`:

```
>> from pylablib.devices import SmarAct
>> SmarAct.list_scu_devices()
[TDeviceInfo(device_id=0, firmware_version='1.3.0.0', dll_version='4.3.0.0')]
>> stage = SmarAct.SCU3D(idx=0)  # connect to the first device in the list
>> stage.close()
```

Due to the manufacturer's API organization, it is currently only possible to "reserve" all connected stages of the same type simultaneously in one application. This means that no other application can connect to any of the stages as long as at least one stage is being controlled (though it does not make any difference if only one stage is connected).

In addition, currently there is no check on whether the stage is already controlled in the other part of the code. This is in contrast with the vast majority of the devices, which issue a unique handle making it impossible to create two different device objects even within the same application. Hence, one needs to be careful to not connect to the same device twice, which can lead to confusing behavior.

Operation

This controller has several features and differences compared to most other stages and sliders:

- The motion is generally executed in "macrosteps", which is a sequence of several "microsteps" with a given amplitude, frequency, and number. A single macrostep with the defined parameters can be performed with `SCU3D.move_macrostep()`, while `SCU3D.move_by()` executes a series of these macrosteps with one of the predefined sizes (from 0 to 20). These sizes are configured to roughly correspond to the step sizes selectable by the controller, although the agreement is not exact.

1.2.4 Basic sensors

Basic concepts are described at the *basic sensors communication page*.

Currently supported sensors:

- *HighFinesse*: laser wavelength meters. Tested with WS6 and WS7 USB-controlled devices.
- *Ophir*: optical power and energy meters. Tested with Ophir Vega.
- *Lakeshore*: temperature sensors. Tested with Lakeshore 218.
- *Pfeiffer*: pressure gauges. Tested with TPG261 and DPG202 controllers.

Note: General device communication concepts are described on the corresponding *page*.

Basics of sensors communication

Basic example

Basic sensors usually only implement a handful of functions related to reading out the measurements (possibly on different channels) and setting up measurements modes:

```
>> gauge = Pfeiffer.TPG260("COM1")  # connect to the gauge
>> gauge.enable(1)  # enable the first channel (usually it's already enabled)
>> gauge.get_pressure()  # read pressure at the default channel (1)
100E3
>> gauge.close()
```

Application notes and examples

Here we talk more practically about using pylablib to perform commons sensor tasks.

Readout

The main readout methods almost always start with `get_` prefix, e.g., `get_pressure`, `get_temperature`, or `get_level`. In some cases there would be two different measurement modes: one which just reads the latest measurement result, and one which initializes the measurement, waits until it's done, and returns the result. These two approaches may be implemented differently in different devices, and it is addressed in their description:

```
>> meter = Cryomagnetics.LM500("COM1")
>> meter.get_level(1)  # immediately return the latest reading
20.0
>> meter.get_level(1)  # return the same reading
20.0
>> meter.measure_level(1)  # initialize a new measurement; takes some time
19.8
```

Non-numerical values

In some cases the readout method would return a non-numerical values. This usually happens when the sensor readings are outside of its range, or if it is in a wrong state (off, warming up, error, etc.) These cases are documented in the querying method description:

```
>> meter = Ophir.VegaPowerMeter("COM1")
>> meter.get_power()  # power is higher than the current range
'over'
>> meter.set_range_idx(0)  # set the maximal power range
>> meter.get_power()  # now the reading is numerical
10E-3
```

Units

Unless absolutely necessary and obvious, all the redout values are specified in SI units (even, e.g., laser frequency in Hz, or pressure in Pa). In rare cases when the devices allows for selection of readout units (e.g., Pfeiffer TPG260 gauges), it only affects the displayed value, but not the results returned by the corresponding methods:

```
>> gauge = Pfeiffer.TPG260("COM1")
>> gauge.set_units("pa")
>> gauge.get_pressure()
100E3
>> gauge.set_units("mbar")
>> gauge.get_pressure()  # pressure still in Pa
100E3
>> gauge.get_pressure(display_units=True)  # pressure in display units
1000
```

Channel selection

Some gauges support simultaneous readout on several channels. In this case, all of their methods take an additional `channel` (in most cases) argument, which specify the read channel.

The channels are usually specified by their index starting from 0 or 1, although some devices adopt more complicated labeling schemes (e.g., Lakeshore 218 temperature sensor can only assign a sensor type to a group of 4 sensors, which is labeled `"A"` or `"B"`). The exact specification is given in the specific class description.

Currently supported sensors

- *HighFinesse*: laser wavelength meters. Tested with WS6 and WS7 USB-controlled devices.

- *Ophir*: optical power and energy meters. Tested with Ophir Vega.
- *Lakeshore*: temperature sensors. Tested with Lakeshore 218.
- *Pfeiffer*: pressure gauges. Tested with TPG261 and DPG202 controllers.

Note: General sensor communication concepts are described on the corresponding *page*

HighFinesse wavemeters

HighFinesse produces a variety of fiber-coupled wavelength meters. Currently pylablib only deals with WS series which uses a USB connection. The code has been tested with several WS6 and WS7 wavemeters.

The main device class is `pylablib.devices.HighFinesse.WLM`.

Software requirements

HighFinesse employs a fairly unique control system.

First, one needs to install the control software, which is uniquely tied to a particular wavemeter and is supplied with it. In theory, software from another wavemeter might still work, but the results are not guaranteed.

Second, this control software runs an application server which processes all requests from third-party software. This means, that the main application needs to be running to perform any device communication from the code. The code has an option of automatically starting it, but on some occasions it might fail, in which case it is necessary to either manually start it, or supply the location of the executable file.

Note: The control software should keep running the whole time. As soon as it is closed, the device will raise an error.

Finally, one needs the DLL to communicate with this software. It is usually named `wlmData.dll`, and it is located in the main controller software folder either in `Com-Test` (for 32-bit applications) or `Projects/64` (for 64-bit applications).

Connection

The device class makes an attempt to search for the DLL and executable in the standard installation folders, as well as use the DLL in the standard location and its executable auto-detection capabilities. However, depending on the number of installed wavemeters and their installation locations, one needs to provide up to 3 arguments on connection. First, the wavemeter ID, which simply a 1 to 5-digit number (e.g. 1234). It is used to identify the correct instance of the control software, either by searching for the correct folder, or via DLL autostart capabilities. Second, one might need to provide the path to `wlmData.dll` (either including the name, or simply the containing folder). Its location is described in the above section. Finally, you might also need to give the path to the application executable, which is located in the main installation folder and is named `wlm_ws*.exe`, where `*` is the wavemeter generation (e.g., `wlm_ws7.exe` for WS7 wavemeters). Hence, the fully qualified (and, therefore, most robust) instantiation looks like this:

```
>> import os
>> from pylablib.devices import HighFinesse
>> app_folder = r"C:\Program Files\HighFinesse\Wavelength Meter WS7 1234"
>> dll_path = os.path.join(app_folder, "Projects", "64")
>> app_path = os.path.join(app_folder, "wlm_ws7.exe")
```

(continues on next page)

(continued from previous page)

```
>> wm = HighFinesse.WLM(1234, dll_path=dll_path, app_path=app_path)
>> wm.close()
```

A unique property of this device is the ability to control it simultaneously from several applications. Keep this in mind, since it might cause confusion or strange results if the control attempts are not synchronized.

Warning: Communication with several simultaneously running wavemeters from a single application has not been tested, and might not work correctly.

Operation

The operation of the wavemeter is fairly straightforward, but there is a couple of points to keep in mind:

- By default, the main measurement functions (*WLM.get_frequency()* and *WLM.get_wavelength()*) raise an error on over- or under-exposure. If this is undesirable (e.g., the laser has power jumps), one can instead make it return `"over"` or `"under"` on these occasions.
- The measurement result is returned immediately, but it is updated only about every 15-30ms (+ exposure time). Hence, fast consecutive calls to *WLM.get_frequency()* and *WLM.get_wavelength()* will return the same value.
- Multi-channel devices have two working modes: single-channel (when only one channel is enabled at a time) and cycling (the wavemeter constantly cycles through several channels for quasi-simultaneous measurements). Some methods only make sense in one of this modes, e.g., *WLM.set_active_channel()* only works in the single-channel mode, while *WLM.enable_switcher_channel()* only in the multi-channel mode. By default, these methods will automatically switch to the corresponding mode.
- Due to a minor control software bug, change in the exposure on some channels might not be reported until the control software is switched to the corresponding channel's exposure control tab (in the upper right corner). By default, the device class performs this switching any time the exposure value is queried, which solves the issue. However, it does take about 10ms. If it is critical, it's possible to turn of this behavior by setting `auto_channel_tab` attribute to `False`.

Note: General sensor communication concepts are described on the corresponding *page*

Ophir power meters

Ophir produces a variety of power and energy meters with different controllers and measurement heads. The class has been tested with Ophir Vega controller with a photodiode head.

The main device classes are *pylablib.devices.Ophir.OphirDevice* for a generic device and *pylablib.devices.Ophir.VegaPowerMeter* for Vega power meter.

Software requirements

The device provides a bare RS232 interface, so any appropriate USB-to-RS232 adapter should work.

Connection

Since the devices are identified as COM ports, they use the standard *connection method*, and all you need to know is their COM-port address (e.g., `COM5`) and the baudrate, if it is different from the standard one (9600 baud):

```
>> from pylablib.devices import Ophir
>> meter1 = Ophir.VegaPowerMeter("COM5")  # default connection assumes 9600 baud
>> meter2 = Ophir.VegaPowerMeter(("COM6", 19200))  # if the second power meter has a␣
↪different baudrate
>> meter1.close()
>> meter2.close()
```

Operation

The operation of the power meter is fairly straightforward, but there is a couple of points to keep in mind:

- On the Vega controller the results can be sent at most 15 times a second. However, they are not necessarily updated at this rate, so several consecutive request might yield the same result.
- The device provides the way to change the communication baud rate. If the rate is changed, the device is automatically disconnected, and the new object needs to be instantiated with the updated baudrate.
- The device might return `"over"` instead of the power reading on overexposure. To fix that, you can adjust the measurement range using `VegaPowerMeter.set_range_idx()`.

Note: Basic sensors communication concepts are described on the corresponding *page*

Lakeshore temperature sensors

Lakeshore manufactures a range of temperature sensor controllers and resistance bridges, which are also used for temperature sensing. There is some overlap between different products, but they still use fairly distinct interfaces and interaction patterns. The code has been tested with Lakeshore 218 temperature controller.

The main device class is `pylablib.devices.Lakeshore.Lakeshore218`.

Software requirements

The device provides a bare RS232 interface, so any appropriate USB-to-RS232 adapter should work.

Connection

Since the devices are identified as COM ports, they use the standard *connection method*, and all you need to know is their COM-port address (e.g., `COM5`):

```
>> from pylablib.devices import Lakeshore
>> sensor = Lakeshore.Lakeshore218("COM5")
>> sensor.close()
```

Note that the connection uses the standard which is fairly different from most RS232 controllers: 7 data bits, 1 parity bit, and 1 stop bit (as opposed to 8 data bits and no parity bit for most controllers). Hence, it is possible that not all

RS232 controllers can communicate with it. In addition, they might need a null-modem (crossed Rx and Tx lines) RS232 cable.

Operation

The operation of this temperature sensor is fairly straightforward, but there is a couple of points to keep in mind:

- Like most similar devices, querying temperature using *Lakeshore218.get_temperature()* immediately returns the most recently measured value. Re-measurement is periodically initiated by the devices itself.
- It is possible to specify custom response curves by using *Lakeshore218.set_curve_header()* and *Lakeshore218.set_curve()*. However, you need to be careful, as it overwrites the stored user curves.

Note: Basic sensors communication concepts are described on the corresponding *page*

Pfeiffer pressure gauges

Pfeiffer manufactures a range of pressure gauges and controllers with several different standards and communication protocols. The code has been tested with Pfeiffer TPG260 series controller (specifically, TPG261) and Pfeiffer DPG202 controller.

The main device classes are *pylablib.devices.Pfeiffer.TPG260* and *pylablib.devices.Pfeiffer.DPG202*.

Software requirements

The devices provide a bare RS232 interface, so any appropriate USB-to-RS232 adapter should work.

Connection

Since the devices are identified as COM ports, they use the standard *connection method*, and all you need to know is their COM-port address (e.g., `COM5`):

```
>> from pylablib.devices import Pfeiffer
>> gauge = Pfeiffer.TPG260("COM5")
>> gauge.close()
```

Operation

TPG260 series

The operation of this gauge is fairly straightforward, but there is a couple of points to keep in mind:

- On measurement error *TPG260.get_pressure()* returns `None`. To get the underlying issue, you can use *TPG260.get_channel_status()*
- By default, the pressure is always returned in Pa regardless of the display units. This behavior can be overridden by setting `display_units=True` in *TPG260.get_pressure()*.

- In case an error occurs, you can use *TPG260.get_current_errors()* to get the list of currently active errors and *TPG260.reset_error()* to reset them.
- This communication protocol for 350-series gauges (361, 362 and 366) is similar, so the device class should also be able to work with them. However, it has not been tested.

DPG202/TPG202 controller

There is a variety of different controllers which implement a similar protocol: DPG202 and TPG202, as well as a variety of RS485-controlled gauges (e.g., CPT200). It is based on requesting parameters with certain 3-digit numbers. These are fairly consistent between the devices, for example, `312` stands for the software version, `740` for pressure, and `349` for the device name. However, different devices implement different subsets of these parameters. The supplied class provides a generic interface through *DPG202.get_value()* and *DPG202.comm()* methods, which, correspondingly, request or set a value of a given parameter given its number (e.g., `740`) and datatype (e.g., `"string"`, `"u_expo_new"`, or `"u_short_int"`). Both of these pieces of information are usually provided in the controller or gauge manual in the `Parameter overview` (or similar-named) section. Currently the device class provides only the most basic functionality:

```
>> from pylablib.devices import Pfeiffer
>> gauge = Pfeiffer.DPG202("COM5")
>> gauge.get_pressure()  # pressure in Pa
9.78E4
>> gauge.get_value(740,"u_expo_new")  # request the parameter directly, yields␣
↪pressure in mBar
9.78E2
>> gauge.close()
```

Note: General device communication concepts are described on the corresponding *page*.

1.2.5 Basic lasers

Basic example

Basic lasers (such as pump lasers) usually only have very basic power-related functionality: turning it on and off, setting power, and controlling and/or requesting the shutter state:

```
>> laser = LaserQuantum.Finesse("COM1")  # connect to the laser
>> laser.set_output_power(10.)  # set 10W output power
>> laser.enable()  # enable the laser
>> laser.get_output_power()  # laser hasn't ramped up up yet
0.1
>> time.sleep(30.)  # wait until the ramp up is done
>> laser.get_output_power()
10.0
>> laser.enable(False)
>> laser.close()
```

Lighthouse Photonics Sprout

Lighthouse Photonics Sprout laser implements the same basic functionality, with some small additions like reading the interlock status, output mode, temperatures, etc.

The device class is *pylablib.devices.LighthousePhotonics.SproutG*.

Since the device shows up as a COM port, it uses the standard *connection method*, and all you need to know to connect is its COM-port address:

```
from pylablib.devices import LighthousePhotonics
laser = LighthousePhotonics.SproutG("COM1")
laser.close()
```

Laser Quantum Finesse

Laser Quantum Finesse laser implements the same basic functionality, with some small additions like controlling the shutter, reading the driving current, temperatures, etc.

The device class is *pylablib.devices.LaserQuantum.Finesse*.

Since the device shows up as a COM port, it uses the standard *connection method*, and all you need to know to connect is its COM-port address:

```
from pylablib.devices import LaserQuantum
laser = LaserQuantum.Finesse("COM1")
laser.close()
```

Note: General device communication concepts are described on the corresponding *page*.

1.2.6 M2 Solstis laser

Solstis is a Ti:Saph laser produces by M2. It is controlled via IceBloc controller unit, which communicates with the PC via a network connection.

The main laser class is *pylablib.devices.M2.Solstis*.

Software requirements

The device provides a bare network interface, so no additional software is required. However, the device and the local network need to be appropriately configured, such that the PC is and the laser are in the same local network and have static IPs.

In order to access some advanced features, you will need a `websocket-client` package, which is not installed with pylablib by default. You can obtain it from PyPi either separately as

```
pip install websocket-client
```

or with the expanded pylablib version

```
pip install pylablib[devio-full]
```

Connection

The laser is identified by its IP address (typically starting with `192.168.1`, if it is on the local network) and the port:

```
>> from pylablib.devices import M2
>> laser = M2.Solstis("192.168.1.2", 34567)
>> laser.close()
```

The port is set up in the `Remote interface` row of the `Network Settings` menu of the laser web interface. There you also need to provide the correct IP address of the controlling PC and enable the remote interface; otherwise the connection will be rejected by the laser.

In addition, you can enable websocket interface option, which is used to send request directly though the device web interface. It is used for some options which are unavailable otherwise, such as enabling or disable the wavemeter connection, receiving some additional status information, and performing more robust control. Note that for proper operation the web interfaces should be opened in the browser and logged in.

Operation

The method names are pretty self-explanatory, and mostly correspond directly to the operations in the web interface. Note that, due to the remote interface organization, terascan requires two methods to start: first *Solstis.setup_terascan()* to specify parameters, and then *Solstis.start_terascan()* to start it.

One should note, that the device operation is not very stable, and occasionally some errors and crashes arise. These can range from failed wavelength tuning and terascan, to terascans failing in exotic ways (e.g., the remote interface suggests that the scan is in progress while the web interface reports a crash), to complete device failure requiring Ice Bloc power cycling.

The device class attempts to somewhat mitigate it by providing relatively a robust stopping method *Solstis.stop_all_operation()*, which tries to set the devices to the default idle state. It uses web interface to get a better information about the laser crashing and send additional stopping commands. It also performs additional steps to stop scans and put the laser in an operation state after a failure, such as starting quick small fine and terascans, and tuning to a nearby frequency.

Note: General device communication concepts are described on the corresponding *page*.

1.2.7 Toptica iBeam Smart laser

Toptica iBeam Smart is a series of CW diode lasers from Toptica. The software has been tested with the standard 633nm laser.

The main device class is *pylablib.devices.Toptica.TopticaIBeam*.

Software requirements

The device is connected to the PC via RS232 or USB. RS232 simply requires a COM-port controller on the PC, which in most cases is a USB-to-Serial adapter. Such adapters normally come with their standard drivers. The USB version simply involves a built-in USB-to-Serial converter (e.g., a standard FTDI chip), so it also shows up as a virtual COM port. Hence, it requires relatively standard drivers, which are either included with the laser, or can be download from the manufacturer's website, for example, together with the TOPAS control software.

Connection

Since the devices are identified as virtual COM ports, they use the standard *connection method*, and all you need to know is their COM-port address (e.g., `COM5`) and, possibly, baud rate, if it is different from the standard 115200 baud:

```
>> from pylablib.devices import Toptica
>> laser1 = Toptica.TopticaIBeam("COM5")
>> laser2 = Toptica.TopticaIBeam(("COM10",38400))  # in case of 38400 baud connection
>> laser1.close()
>> laser2.close()
```

Operation

Power and output control

Usually the laser has the main power control and one or several (up to 5) output channels, which can be controlled separately. To turn the whole laser on or off, you can use *TopticaIBeam.enable()*, while each channel is controlled using *TopticaIBeam.enable_channel()*. The power is set independently for each channel via *TopticaIBeam.set_channel_power()*. The actual output power can be queried using *TopticaIBeam.get_output_power()*.

Detailed info

The most detailed information about the laser can be obtained using *TopticaIBeam.get_full_data()* method. It outputs a detailed report generated by the laser, which contains most of the adjustable parameters.

Note: General device communication concepts are described on the corresponding *page*.

1.2.8 Tektronix oscilloscopes

Tektronix produces a large number of very widespread oscilloscopes. They have strongly overlapping, though not entirely identical, interfaces. The library has been tested with TDS2002B, TDS2004B, and DBO2014B.

The generic oscilloscope class is *pylablib.devices.Tektronix.ITektronixScope*, and the derived classes for specific devices are *pylablib.devices.Tektronix.TDS2000* of TDS2000 series and *pylablib.devices.Tektronix.DPO2000* for DPO2000/MSO2000 series.

Software requirements

These oscilloscopes use NI VISA communication interface. Hence, it requires NI VISA Runtime, which is freely available from the National Instruments website

Connection

The devices are identified by their VISA connection strings, which typically start with `USB0::0x0699`, e.g., `"USB0::0x0699::0x0364::C000001::INSTR"`. To get a list of all connected VISA-enabled devices, you can run `pylablib.list_backend_resources("visa")`:

```
>> import pylablib as pll
>> pll.list_backend_resources("visa")
('USB0::0x0699::0x0364::C000001::INSTR',)
>> from pylablib.devices import Tektronix
```

(continues on next page)

(continued from previous page)

```
>> osc = Tektronix.TDS2000("USB0::0x0699::0x0364::C000001::INSTR")
>> osc.close()
```

Operation

The method names are usually pretty self-explanatory. A typical operation involves setting up channels, scales, and trigger options, acquiring a waveform, and reading the result:

```
from pylablib.devices import Tektronix
osc = Tektronix.TDS2000("USB0::0x0699::0x0364::C000001::INSTR")  # connect to the
↪oscilloscope
osc.enable_channel([1,2])  # enable channels
osc.set_horizontal_span(0.1)  # set up horizontal and vertical spans
osc.set_vertical_span("CH1", 1)
osc.set_vertical_span("CH2", 0.1)
osc.setup_edge_trigger("CH1", 0., "dc", "rise")  # set up edge trigger on channel 1
↪at 0V threshold
osc.grab_single(wait_timeout=10.)  # grab a single waveform and wait for up to 10s to
↪finish acquisition
sweeps = osc.read_multiple_sweeps([1,2])  # read out the waveforms
osc.close()
```

However, there is a couple of points to keep in mind:

- The acquisition is controlled using `grab_` methods. Generally, the most convenient way is to use *ITektronixScope.grab_single()* to acquire a single waveform (analogous to pressing a `Single` button on the oscilloscope panel). By default, this method waits until the acquisition is complete (i.e., the oscilloscope is triggered and the waveform is completely acquired) before continuing. You can also set `wait=False` to perform other operations in the meantime. The acquisition status can be queried via *ITektronixScope.is_grabbing()*, which returns `True` while the trigger is armed or while the data is recording, and `False` after the acquisition is done.
- It appears that the software trigger does not work some time (~500 ms) after the acquisition is set up. If it is invoked in *ITektronixScope.grab_single()* method by supplying `software_trigger=True`, a 300ms delay is added automatically. However, if you invoke it manually using *ITektronixScope.force_trigger()*, you should keep it in mind.
- The waveform transfer is usually performed via *ITektronixScope.read_sweep()* or *ITektronixScope.read_multiple_sweeps()* methods. Since the waveform is transferred in raw form, it requires a preamble data (vertical and horizontal scales and offsets, data format, etc.) to translate into physical units. By default, it is acquired every time before the waveform transfer, which takes some time (up to ~200ms). Alternatively, one can acquire a preamble once and use it in subsequent reading. This method is faster, but will result in an incorrect scaling if the parameters are changed in the meantime (either remotely, or directly on the oscilloscope):

  ```
  >> wfmpres = osc.osc.get_wfmpre([1,2])
  >> %time sweeps = osc.read_multiple_sweeps([1,2])
  Wall time: 2.2 s
  >> %time sweeps = osc.read_multiple_sweeps([1,2], wfmpres=wfmpres)
  Wall time: 450 ms
  ```

- The device class attempts to determine the number of channels automatically on connection, based on which requests raise device errors. However, this process takes some time, and sometimes can raise errors on not fully SCPI-compliant devices. If that is the case, it is always possible to supply the number of channels on construction:

```
>> osc = Tektronix.TDS2000("USB0::0x0699::0x0364::C000001::INSTR")  # use␣
↪autodetection
>> osc.get_channels_number()
2
>> osc.close()
>> osc = Tektronix.TDS2000("USB0::0x0699::0x0364::C000001::INSTR", nchannels=2)
↪# specify manually
```

Note: General device communication concepts are described on the corresponding *page*.

1.2.9 NI DAQmx interface

National Instruments produces lots of different data acquisition devices, which support digital and analog input and output, both immediate and clocked (depending on the exact device). They are controlled via a very universal NI DAQmx interface. This interface is implemented in python-nidaqmx package, which provides a fairly close to original functionality, but with much more convenient Python wrappers. Pylablib implements a relatively thin wrapper around this package to present it in a way similar to the other device classes, and to simplify common tasks such as setting up voltage and counter input channels.

The main daq class is `pylablib.devices.NI.NIDAQ`. It has been tested with NI PCIe-6323, NI USB-6008, and NI USB-6363.

Software requirements

This interface uses NI DAQmx library, which is freely available on the National Instruments website. Additionally, it needs python-nidaqmx package (not to be confused with pydaqmx). It is not automatically installed with the base version of pylablib, and can be obtained from PyPi either separately as

```
pip install nidaqmx
```

or with the expanded pylablib version

```
pip install pylablib[devio-full]
```

Connection

The devices are identified by their name, such as `"Dev1"`. To list all of the connected devices together with their basic information, you can run `NI.list_nidaqmx_devices`:

```
>> from pylablib.devices import NI
>> NI.list_nidaqmx_devices()
[TDeviceInfo(name='Dev1', model='USB-6008', serial_number='01234567')]
>> daq = NI.NIDAQ("Dev1")
>> daq.close()
```

Operation

The typical use case involves setting up different input and output channels, starting acquisition, and acquiring some number of samples:

```
from pylablib.devices import NI
daq = NI.NIDAQ("Dev1")
daq.add_voltage_input("vin", "ai0")  # add voltage input named "vin" on the terminal "ai0"
daq.add_voltage_input("vin2", "ai1", rng=(-1,1))  # add second channel with a smaller range of +/- 1V
daq.add_digital_input("din", "port0/line0")
daq.setup_clock(100)  # setup 100Hz sampling clock
trace = daq.read(100)  # start acquisition, read finite number of samples, and stop it again
# now do continuous acquisition + processing loop
nsamples = 0
daq.start()  # start continuous acquisition
while nsamples<1000:
    sample = daq.read()
    ... process sample
    nsamples+=1
daq.stop()
```

The class provide basic methods to set up analog, digital, and counter inputs, and analog and digital outputs. All the analog and digital inputs are synchronized to the same clock, which is the default analog input sample clock (`ai/SampleClock`) by default. It is also possible to set up the external clock via `NIDAQ.setup_clock()` and export the sampling clock via `NIDAQ.export_clock()`. Not that not all devices support clocked digital inputs, which means that setting up digital inputs there would raise an error.

By default, the counter inputs are synchronized to the same clock, although it is possible to change that. The counter inputs have 3 modes for output values: bare counter (accumulates the number of counts), differential (number of new counts between the two sampling points), and rate (same as differential, but normalized by the sampling rate). In case of external clock, when the sampling rate is a priori unknown, it might be useful to setup a clock rate counter input to determine this clock rate via `NIDAQ.add_clock_period_input()`.

Acquisition is controlled with `NIDAQ.start()` and `NIDAQ.stop()` methods, and the readout is performed via `NIDAQ.read()`. The result of this is always a 2D numpy array, where the first index corresponds to samples and the second to channels. The order of channels can be obtained from `NIDAQ.get_input_channels()`.

The outputs can be either analog or digital. The digital outputs are always immediate, i.e., they immediately produce and hold the latest output value. The analog outputs can work in two modes: either immediate, or clocked. The mode is set up via `NIDAQ.setup_voltage_output_clock()`. In this case, it is possible to output a list of values, which produces a waveform clocked according to the specified clock: either a separate clock source (default), or the analog input clock, which makes voltage input and output synchronized.

Note: General device communication concepts are described on the corresponding *page*.

1.2.10 Generic AWGs

There is a large variety of Arbitrary Waveform Generators, which have very similar characteristics and communication interface.

The generic AWG class is `pylablib.devices.AWG.GenericAWG`, and the derived classes for specific devices are `pylablib.devices.AWG.Agilent33500` and `pylablib.devices.AWG.Agilent33220A` for two different Agilent AWGs, `pylablib.devices.AWG.RigolDG1000` for Rigol DG1000 series, `pylablib.devices.AWG.TektronixAFG1000` for Tektronix AFG1000 series, `pylablib.devices.AWG.InstekAFG2000` for Instek GW 2000 series, `pylablib.devices.AWG.RSInstekAFG21000` for Iso-Tech 21000 series (a clone of Instek AFG2000, but with a couple of bugs which needs to be worked around), and *pylablib.devices.AWG.InstekAFG2225* for Instek GW 2225 (slightly advanced two-channel version of Instek AFG2000).

Software requirements

Most of these AWGs use NI VISA communication interface. Hence, they require NI VISA Runtime, which is freely available from the National Instruments website. However, Instek and Iso-Tech AWGs show up as virtual COM ports, so they require no additional software.

Connection

The devices are identified by their VISA connection strings, (e.g., `"USB0::0x0699::0x0364::C000001::INSTR"`) or COM-port (e.g., `"COM5"`). To get a list of all connected VISA-enabled devices, you can run `pylablib.list_backend_resources("visa")`:

```
>> import pylablib as pll
>> pll.list_backend_resources("visa")
('USB0::0x09C4::0x0400::DG1D150200000::INSTR',)
>> from pylablib.devices import AWG
>> dev = AWG.RigolDG1000("USB0::0x09C4::0x0400::DG1D150200000::INSTR")
>> dev.close()
```

Operation

The method names are usually pretty self-explanatory. A typical operation involves setting up the function, its parameters, and controlling output:

```
from pylablib.devices import AWG
dev = AWG.RigolDG1000("USB0::0x09C4::0x0400::DG1D150200000::INSTR")  # connect to the device
dev.set_function("square", 2)  # set up square waveform on the second channel
dev.set_duty_cycle(20, 2)
dev.set_output_range((-1, 1), 2)  # set output span from -1V to 1V
dev.enable_output(channel=2)  # enable output
dev.close()
```

However, there is a couple of points to keep in mind:

- Since the same general class architecture supports both single-channel and multichannel devices, the channel argument is usually close to the end of the argument list and is not mandatory. If it is not supplied, it is chosen to be the current default channel (1 upon creation), which can be set using *GenericAWG.select_current_channel()*. Hence, int the example above we can write:

  ```
  dev.select_current_channel(2)  # now all methods assume channel 2
  dev.set_function("square")
  dev.set_duty_cycle(20)
  dev.set_output_range((-1, 1))
  dev.enable_output()
  ```

- Similarly, some methods can be present but not applicable to the particular AWG (e.g., burst trigger related methods, phase synchronization methods, etc.) If this is the case, they will cause an error when called.

Note: General device communication concepts are described on the corresponding *page*.

1.2.11 Miscellaneous Thorlabs devices

Thorlabs has a variety of devices implementing different serial communication protocols, mostly related to optomechanics. Their requirements and general approach are still fairly similar, so they are all collected here.

Software requirements

Most devices provide either a bare RS232 interface, or a USB connection with a built-in USB-to-RS232 chip. In either case, they are automatically recognized as serial ports, and no additional software is required. The only exception on this page is MFF101/102 motorized flip mount, which belongs to the *Kinesis devices* and requires APT software.

Connection

Most of the devices are identified as COM ports, so they use the standard *connection method*, and all you need to know is their COM-port address (e.g., `COM5`):

```
>> from pylablib.devices import Thorlabs
>> wheel = Thorlabs.FW102("COM5")
>> wheel.close()
```

The only exception is MFF101/102, which is identified by its serial number (more details are given at the *Kinesis devices page*).

Operation

MFF101/102 flip mount

The class is provided as *`pylablib.devices.Thorlabs.MFF`*. It allows for control of the flip mirror position, as well as changing its motion parameters and designations of its digital input and output.

FW102/212 filter wheel

The class is proved as *`pylablib.devices.Thorlabs.FW`*.

In addition to setting the position, it allows to adjust speed settings and turn the indicator LED off to minimize light contamination. By default, the wheel also "respects bound" between the first and the last position. Usually, when one orders a move from, e.g., position 2 to 6 on a 6-position wheel, it would go along the shortest route, i.e., position 1. If this is an ND filter wheel (e.g., FW102CNEB), this leads to momentary increase of the transmitted power by ND0.5 (about factor of 3) compared to start and stop positions. To avoid that, the class breaks this move into several shorter (no longer than 1/3 of the wheel) moves, which never cross the boundary between the first and the last position. This takes a bit longer (as it requires several consecutive moves), but is generally safer. This behavior can be turned off by setting `respect_bound=False` on class creation.

Note that older version (1.0) of the filter wheel do not support the full range of options and operate on a slightly different protocol. This leads to crashes on at least some of the methods, e.g., *`FW.get_position()`*. If this is the case, you can try *`pylablib.devices.Thorlabs.FWv1`* instead.

MDT693/694 high-voltage source

The class is proved as *`pylablib.devices.Thorlabs.MDT69xA`*.

The class provides the ability to set and query the voltage on the three channels, as well as to query the total voltage range (it is set by a physical switch on the back panel, and can not be altered remotely).

Note: General device communication concepts are described on the corresponding *page*.

1.2.12 OZ Optics devices

OZ Optics provides a variety of mostly fiber-optics related devices. Pylalbib covers some of its fiber optomechanics solutions: polarization controller, tunable filter and variable attenuator. Their requirements and general approach are fairly similar, so they are all collected here.

Software requirements

All the devices provide either a bare RS232 interface, or a USB connection with built-in USB-to-RS232 chip. In either case, they are automatically recognized as serial ports, and no additional software is required.

Connection

The devices are identified as COM ports, so they use the standard *connection method*, and all you need to know is their COM-port address (e.g., `COM5`):

```
>> from pylablib.devices import OZOptics
>> ctl = OZOptics.EPC04("COM5")
>> ctl.close()
```

Operation

EPC04 fiber polarization controller

The class is proved as *pylablib.devices.OZOptics.EPC04*. It lets the user change the 4 control voltages, switch between DC and AC (scrambling) modes, and change the AC frequency.

DD100 fiber attenuator

The class is proved as *pylablib.devices.OZOptics.DD100*. It simply lets the user query and change the attenuation, as well as home the device. Note that homing is required once after the device power up, and it might in general sweep over the whole range of attenuations.

TF100 fiber filter

The class is proved as *pylablib.devices.OZOptics.TF100*. It simply lets the user query and change the central wavelength, as well as home the device. Note that homing is required once after the device power up, and it might in general sweep over the whole range of wavelengths.

Note: General device communication concepts are described on the corresponding *page*.

1.2.13 Miscellaneous devices

There are several miscellaneous device classes, which are collected in this page. All of them implement straightforward serial communication protocol, so the software requirements and the connection approach is the same for all of them.

Software requirements

All the devices provide either a bare RS232 interface, or a USB connection with a built-in USB-to-RS232 chip. In either way, they are automatically recognized as serial ports, and no additional software is required.

Connection

The devices are identified as COM ports, so they use the standard *connection method*, and all you need to know is their COM-port address (e.g., `COM5`):

```
>> from pylablib.devices import Conrad
>> dev = Conrad.RelayBoard("COM5")
>> dev.close()
```

Operation

Conrad relay board

This is a board, which has several externally-controlled relays.

The class is proved as `pylablib.devices.Conrad.RelayBoard`. It simply lets the user query and set the relay states. It also in principle supports communication with several daisy-chained boards, but it has never been tested.

Generic Arduino class

The class is proved as `pylablib.devices.Arduino.IArduinoDevice`. It implements basic serial communication; the exact command protocol depends on the particular Arduino software written and uploaded by the user.

The main difference from directly using a serial backend is in handling of DTR line, which signal reset to the Arduino board. Unlike the standard backend, connection will not restart the board; instead, there is an explicit `IArduinoDevice.reset_board()` which pulses the DTR line to reset the board.

1.3 Data processing

1.3.1 Fitting

Class `fitting.Fitter` is a user-friendly wrapper around `scipy.optimize.least_squares()` routine. Dealing with fitting is made more convenient in a couple of ways:

- It is easy to specify the x-parameter name (in the case it is not the first parameter), or specify multiple x-parameters;
- All of the fit and fixed parameters are specified by name; it is easy to switch between any parameter being fit or fixed;

- The wrapper automatically handles complex parameters (split into real and imaginary parts), numpy arrays, lists, or tuples (including nested structures);
- The final parameters (fit and fixed) are returned in a single dictionary indexed by their names;
- The wrapper also returns the fit function with all of the parameters bound to the final fit and fixed values;
- The fit function result is flattened during fitting, so it works for functions returning multi-dimensional (for example, 2D) arrays.

Examples

Fitting a Lorentzian:

```
def lorentzian(frequency, position=0., width=1., height=1.):
    return height/(1.+4.*(frequency-position)**2/width**2)

## creating the fitter
# fit_parameters dictionary specifies the initial guess
fit_par = {"position":0.5, "height":1.}
fitter = pll.Fitter(lorentzian, xarg_name="frequency", fit_parameters=fit_par)
# additional fit parameter is supplied during the call
fit_par, fit_func = fitter.fit(xdata, ydata, fit_parameters={"width":1.0})
plot(xdata, ydata)  # plot the experimental data
plot(xdata, fit_func(xdata))  # plot fit result
```

Fitting a sum of complex Lorentzians with the same width:

```
def lorentzian_sum(frequency, positions, width, amplitudes):
    # list of complex lorentzians
    #   positions and amplitudes are lists, one per peak
    lorentzians = [a/(1.+2j*(frequency-p)/width) for (a,p) in zip (amplitudes,
↪positions)]
    return np.sum(lorentzians, axis=0)

## creating the fitter
# fit_parameters dictionary specifies the initial guess
#     (complex initial guess for the "amplitude" parameter hints that this parameter␣
↪is complex)
fit_par = {"positions":[0.,0.5,1.], "amplitudes":[1.+0.j]*3}
fitter = pll.Fitter(lorentzian_sum, xarg_name="frequency", fit_parameters=fit_par)
# fixed parameter is supplied during the call (could have also been supplied on␣
↪Fitter initialization)
fit_par, fit_func = fitter.fit(xdata, ydata, fixed_parameters = {"width":0.3})
plot(xdata, ydata.real)  # plot the experimental data
plot(xdata, fit_func(xdata).real)  # plot fit result
```

Fitting 2D Gaussian and getting the parameter estimation errors:

```
def gaussian(x, y, pos, width, height):
    return np.exp( -((x-pos[0])**2+(y-pos[1])**2)/(2*width**2) )*height

## creating the fitter
# fit_parameters dictionary specifies the initial guess
fit_par = {"pos":(100,100), "width":10., "height":5.}
fitter = pll.Fitter(gaussian, xarg_name=["x","y"], fit_parameters=fit_par)
xs, ys = np.meshgrid(np.arange(img.shape[0]), np.arange(img.shape[1]), indexing="ij")
↪# building x and y coordinates for the image
```

(continues on next page)

(continued from previous page)

```
# fit_stderr is a dictionary containing the fit error for the corresponding parameters
fit_par, fit_func, fit_stderr = fitter.fit([xs,ys], img, return_stderr=True)
imshow(fit_func(xs, ys))  # plot fit result
```

The full module documentation is given at `pylablib.core.dataproc.fitting`.

1.3.2 Filtering and decimation

There are several functions present for filtering the data to smooth it or reduce its size. Most of them are thin wrapper around standard numpy or scipy method, but they provide more universal interface which work both with numpy arrays and pandas DataFrames:

- First are the decimation functions: `filters.decimate()` (and its special case `filters.binning_average()`), `filters.decimate_full()` and `filters.decimate_datasets()`. The first one splits the supplied trace into consecutive segments of *n* points and compresses them into a single value using the supplied method (e.g., `"mean"` will average them together, which is used for `filters.binning_average()`). The second one completely decimates the dataset along the given axis (which is essentially identical to using the standard numpy methods such as `np.mean` or `np.max`). The last one decimates several datasets together, which is similar to combining them into a large `(n+1)D` array and fully decimating along the given axis:

```
>> trace = np.arange(10)
>> pll.binning_average(trace, 3)  # average every block of 3 points to a single␣
↪value
array([1., 4., 7.])
>> pll.decimate(trace, 3, dec="max")
array([2, 5, 8])
>> pll.decimate_full(trace, "mean")  # same as np.mean(trace)
4.5
>> trace2 = np.arange(10)**2
>> pll.decimate_datasets([trace, trace2], "sum")  # same as np.sum([trace,␣
↪trace2],axis=0)
array([ 0,  2,  6, 12, 20, 30, 42, 56, 72, 90])
```

- Sliding decimation methods `filters.sliding_average()`, `filters.median_filter()` and `filters.sliding_filter()` are related, but use a sliding window of *n* points instead of complete decimation of *n* points together. They only work for 1D traces or 2D multi-column datasets. Note that `filters.sliding_filter()` is implemented through a simple Python loop, so it is fairly inefficient:

```
>> trace = np.arange(10)
>> pll.sliding_average(trace, 4)  # average points in 4-point window (by default␣
↪use "reflect" boundary conditions)
array([0.75, 1.5 , 2.5 , 3.5 , 4.5 , 5.5 , 6.5 , 7.5 , 8.25, 8.5 ])
>> pll.sliding_filter(trace, 4, "max")  # find maximum of points in 4-point window
array([2, 3, 4, 5, 6, 7, 8, 9, 9, 9])
```

- Next are convolution filters which operate by convolving the trace with a given kernel function. These involve `filters.gaussian_filter()` (and `filters.gaussian_filter_nd()`, which is simply a wrapper around `scipy.ndimage.gaussian_filter()`), and a more generic `filters.convolution_filter()`. Related are infinite impulse response (IIR) filter `filters.low_pass_filter()` and `filters.high_pass_filter()`, which mimic standard single-pole low-pass and high-pass filters. In principle, they can be modelled as a convolution with an exponential decay, but the implementation using the recursive filters is more efficient for large widths.

- Finally, there are Fourier filters, which Fourier-transform the trace, scale the transform values, and transform it back to the real domain. These involve the main function *filters.fourier_filter()*, which takes a generic frequency response function, as well as two specific response function generators *filters.fourier_filter_bandpass()* and *filters.fourier_filter_bandstop()*, both generating hard frequency cutoff filters.

1.3.3 Fourier transform

There is a couple of methods to work with Fourier transform. They are built around `numpy.fft.fft()`, but allow more convenient normalization (e.g., in units of power spectral density), and work better with pandas DataFrames. They also have an option to automatically trim the trace length to the nearest "good" size, which is a product of small primes. This can have fairly strong (up to a factor of several) effect on the transform runtime, while typically trimming off less than 1% of the data.

The main methods are *fourier.fourier_transform()* for the direct transform, *fourier.inverse_fourier_transform()* for the inverse transform, and *fourier.power_spectral_density()* for the power spectral density:

```
>> x = np.random.normal(size=10**5)  # normal distribution centered at 0 with a width␣
↪of 1
>> PSD = pll.power_spectral_density(x, dt=1E-3)  # by default, use density␣
↪normalization; assume time step of 1ms
>> df = PSD[1,0] - PSD[0,0]
>> df  # total span is 1kHz with 10**5 points, resulting in 0.01Hz step
0.01
>> np.sum(PSD[:,1]) * df  # integrated PSD is equal to the original trace RMS squared,
↪ which is about 1 for the normal distribution
1.005262206692361
>> np.mean(x**2)
1.005262206692361
```

1.3.4 Feature detection

There are several methods for simple feature detection:

- The peak detection, which is usually achieved by the combination of *feature.multi_scale_peakdet()* and *feature.find_peaks_cutoff()*. The first applies difference-of-Lorentzians or difference-of-Gaussians filter, which detects peaks of a particular width. The second finds peaks using a cutoff.
- Another way to find peaks is using *feature.find_local_extrema()*, which finds local minima or maxima in a sliding window of a given width.
- Switching between two states with a noisy trace can be detected using *feature.latching_trigger()*. It implements a more robust approach to find when the trace is above/below threshold by considering two thresholds: a higher "on" thresholds and a lower "off" threshold. It makes the on/off state "latch" to its current value and is robust to small trace fluctuations around the threshold, which would lead to rapid on/off switches in a single-threshold scheme.

1.3.5 Miscellaneous utilities

Additionally, there is a variety of small functions to simplify some data analyses and transforms:

- Checking trace properties: *dataproc.utils.is_ascending()*, *dataproc.utils.is_descending()*, *dataproc.utils.is_ordered()*, *dataproc.utils.is_linear()*.

- Sorting by a given column: *dataproc.utils.sort_by()*; work both on pandas and numpy arrays
- Filtering: *dataproc.utils.filter_by()* and *dataproc.utils.unique_slices()* (a simple analog of pandas `pandas.DataFrame.groupby()`, which works on numpy arrays)
- Binary search (both in ordered and unordered 1D arrays): *dataproc.utils.find_closest_arg()*, *dataproc.utils.find_closest_value()*, and *dataproc.utils.get_range_indices()*.
- Traces step analysis and unwrapping: *dataproc.utils.find_discrete_step()* tries to find a single number which divides all values within a reasonable precision, and *dataproc.utils.unwrap_mod_data()* "unwraps" modulo data (e.g., phase, which is defined mod 2pi) provided that the steps between two consecutive points are less than 1/2 of the module.
- Cutting the trace to the given range, or cutting out a given range: *dataproc.utils.cut_to_range()* and *dataproc.utils.cut_out_regions()*.
- Converting between 2-column "XY" and complex representations: *dataproc.utils.xy2c()* and *dataproc.utils.c2xy()*
- Scalar numerical utilities: *utils.numerical.limit_to_range()* (limit a value to lie in a given range, including option for no limits in one or both directions), *utils.numerical.gcd()* and *utils.numerical.gcd_approx()* (greatest common divisor or its approximate version for non-integer values)

1.4 Data storage

Complex data storage in pylablib centers around 2 main components: the multi-level dictionary for representing hierarchical data within the code, and file IO to (among other things) load and store it in a human-readable format.

1.4.1 Multi-level dictionary

dictionary.Dictionary is an expansion of the standard *dict* class which supports tree structures (nested dictionaries). The extensions include:

- handling multi-level paths and nested dictionaries, with several different indexing methods
- iteration over the immediate branches, or over the whole tree structure
- some additional methods: mapping, filtering, finding difference between two dictionaries
- combined with *pylablib.core.fileio* allows to save and load the content in a human-readable format.

Creating and indexing:

```
>>> d = pll.Dictionary()
>>> d['d/0/x'] = 5
>>> d
Dictionary('d/0/x': 5)
>>> d['d/0/x']  # string path indexing
5
>>> d['d']['0']['x']  # nested indexing
5
>>> d['d','0','x']  # multi-level path indexing
5
>>> d['d',0,'x']  # all path elements are converted into strings
5
>>> d['d/0']['x']  # indexing styles can be freely mixed
5
```

(continues on next page)

(continued from previous page)

```
>>> d['d','0/x']
5
>>> b = d['d']  # indexing a branch yields another Dictionary object
>>> b
Dictionary('0/x': 5)
>>> b['0/x'] = 10  # the branch shares the data with the main dictionary
>>> d
Dictionary('d/0/x': 10)
```

A dictionary can be build from a Python `dict`, which automatically normalizes paths and nested dictionaries:

```
>>> d = pll.Dictionary({ 'a':1, 'b/i':2, 'c':{'i':3, 'ii':4}, 'd/0/x':5 })
>>> d
Dictionary('b/i': 2
'c/i': 3
'c/ii': 4
'd/0/x': 5
'a': 1)
```

Note: There are several limitations on the dictionary structure (mostly they involve possible paths and keys):

- As mentioned above, the keys are converted into strings to get the path; therefore, different Python object can merge together (e.g., number `0` and string literal `'0'`). This also discourages use of some of the objects with "underdefined" (implementation dependent) representations, for example, floating point numbers.
- Since the `'/'` symbol is used to split different path entries, it can't be used inside a single-level key. It is possible to re-define this symbol on dictionary creation; however, it might lead to compatibility issues.
- Empty keys are not allowed. When building a path, they are automatically dropped, so `'a/b'`, `'a/b/'`, `'a///b//'` all correspond to the same path.
- One path can either correspond to a branch node, or a leaf node. In other words, one path can't be a prefix of other paths and also contain data: structures like `pll.Dictionary({ 'a':1, 'a/b':2})` are not allowed. To get around this, one can define a specific "data key" not used anywhere else, and store data in a node under that key (e.g., with the data key `'#'` the example before turns into a valid structure `pll.Dictionary({ 'a/#':1, 'a/b/#':2})`).

Thus, it is generally recommended to only use strings or non-negative integers as keys, and apply the same restrictions to them as to the Python variable names (with the addition of names starting with a digit).

1.4.2 File IO

`pylablib.core.fileio` contains several function for saving and loading data into different kinds of files: binary (*`loadfile.load_bin()`* and *`savefile.save_bin()`*), CSV (*`loadfile.load_csv()`* and *`savefile.save_csv()`*), or dictionary (*`loadfile.load_dict()`* and *`savefile.save_dict()`*).

Binary files

The first (binary files) closely corresponds to numpy `fromfile`. In addition, it also allows automatic conversion into pandas arrays, setting column names, and skipping some number of bytes from the start:

```
>> table = np.arange(6).reshape((3,2))
>> pll.save_bin(table, "table.dat", dtype="<f8)
>> pll.load_bin("table.dat", columns=["Column1", "Column2"], dtype="<f8)
   Column1  Column2
0      0.0      1.0
1      2.0      3.0
2      4.0      5.0
```

Furthermore, there is an option to save the binary data with a preamble dictionary file, which describes its structure (columns, dtype, etc.) This way, one does not have specify these parameter in the loading code:

```
>> table = pd.DataFrame({"C1":arange(3),"C2":arange(3)**2/3})
>> table
   C1        C2
0   0  0.000000
1   1  0.333333
2   2  1.333333
>> pll.save_bin_desc(table, "table.dat")
>> pll.load_bin_desc("table.dat")
    C1        C2
0  0.0  0.000000
1  1.0  0.333333
2  2.0  1.333333
>> np.fromfile("table_data.bin", "<f8").reshape((3, 2))  # the data is still stored␣
↪in the regular binary format
array([[0.        , 0.        ],
       [1.        , 0.33333333],
       [2.        , 1.33333333]])
```

Note that only homogeneous data (i.e., all columns having the same type) is currently supported. That's why the first column got converted from integers into reals.

CSV files

The functionality of the second one mimics pandas `read_csv`, but offers a bit more flexibility with more complicated values in columns, such as tuples or binary strings:

```
>> table = pd.DataFrame({ "C1":np.arange(3), "C2":[(i**2,i**3) for i in range(3)] })
>> table  # the second columns contains tuples
   C1      C2
0   0  (0, 0)
1   1  (1, 1)
2   2  (4, 8)
>> pll.save_csv(table, "table.csv")
>> pll.load_csv("table.csv", dtype="generic")  # need to specify generic values type,␣
↪which handle complicated cases, but is somewhat slower
   C1      C2
0   0  (0, 0)
1   1  (1, 1)
2   2  (4, 8)
```

In addition, its default settings are a bit different: the column separator is a whitespace, the column names are contained in the comment string (which removes occasional ambiguity), and the creation date string is appended by default. Hence, the content of the file created above is

```
# C1        C2
0   (0, 0)
1   (1, 1)
2   (4, 8)

# Saved on 2021/01/01 12:00:00
```

Note that currently it operates only with simple flat tables and does not support advanced pandas features such as index or multi-index. If these are required, you can use `savefile.save_csv_desc()` and `loadfile.load_csv_desc()`. Similarly to `savefile.save_bin_desc()` and `loadfile.load_bin_desc()`, it saves a dictionary containing additional description; however, the table is inlined by default, so only one file is generated:

```
>> table = pd.DataFrame({ "C1":np.arange(3), "C2":[(i**2,i**3) for i in range(3)] },␣
↪index=np.arange(3)+10)
>> table  # non-trivial index colum
    C1      C2
10   0   (0, 0)
11   1   (1, 1)
12   2   (4, 8)
>> pll.save_csv(table, "table.csv")
>> pll.load_csv("table.csv", dtype="generic")  # index is lost
    C1      C2
0     0   (0, 0)
1     1   (1, 1)
2     2   (4, 8)
>> pll.save_csv_desc(table, "table.dat")
>> pll.load_csv_desc("table.dat")  # index is preserved (also note that here dtype is
↪"generic" by default)
    C1      C2
10   0   (0, 0)
11   1   (1, 1)
12   2   (4, 8)
```

Dictionary files

Finally, dictionary saving and loading operates with *dictionary* objects. It is generally useful to load or save various heterogeneous settings or parameters, such as device parameters, data processing parameters, and GUI or device state. It supports most basic Python data types as values: standard scalar types (integers, reals, complex numbers, strings, booleans, `None`), containers (tuples, lists, dictionaries, sets, including nested ones), binary and raw string representation (e.g., `b"\x00"` or `r"m\n\o"`), short numpy arrays (represented as, e.g., `"array([1, 2, 3])"`), and inline tables (which are interpreted as pandas table by default). The only common data type not included is named tuples; they get automatically converted to regular tuples on saving.

The dictionary files have the `key value` line formats and typically use full paths (as opposed to, say, XML hierarchy), which makes them easier to inspect and parse without pylablib. For example, the dictionary from the previous section will be saved as

```
b/i 2
c/i 3
c/ii 4
d/0/x 5
a 1
```

With more complicated data types, it might look more like

```
process/points  array([1., 2., 3.])
process/default/frequency   10+2.j
# Lines starting with # are treated as comments
plot/position   [(0,0), (1,1), (2,3)]
plot/label  r"$\nu_0$"
# Keys do not have to be in any particular order
process/default/amplitude   5.
```

which results in a dictionary

```
Dictionary('plot/label': $\nu_0$
'plot/position': [(0, 0), (1, 1), (2, 3)]
'process/default/amplitude': 5.0
'process/default/frequency': (10+2j)
'process/points': [1. 2. 3.])
```

The format also supports hierarchy using `//branch` to mark a start of sub-branch and `///` to mark its end. For example, the dictionary above can be also saved as

```
//process
    # indentation is not required, but helps to see the structure
    points  array([1., 2., 3.])
    default/frequency   10+2.j
    default/amplitude   5.
///

//plot
    position   [(0,0), (1,1), (2,3)]
    label  r"$\nu_0$"
///
```

Finally, it is possible to specify inline tables using special comment lines. For example,

```
# The key without the value marks the path to the table within the dictionary
data/table
## Begin table
1   1.j
2   4.j
3   9.j
## End table
```

produces a dictionary containing pandas DataFrame:

```
Dictionary('data/table':
   0                   1
0  1  0.000000+1.000000j
1  2  0.000000+4.000000j
2  3  0.000000+9.000000j )
```

1.5 Various utilities

1.5.1 File system

There is a number of methods which are minor expansions of the built-in file utilities:

- Accessing and changing file times: *utils.files.get_file_creation_time()*, *utils.files.get_file_modification_time()*, *utils.files.touch()* (update the modification date).
- Generating new file names (e.g., for storing a new dataset): *utils.files.generate_indexed_filename()* and *utils.files.generate_prefixed_filename()*.
- Some path analysis methods: *utils.files.fullsplit()*, *utils.files.normalize_path()*, *utils.files.paths_equal()*, *utils.files.relative_path()*; a lot of these have also been implemented in `pathlib` module, and are kept for backwards compatibility.
- Checking if a string is a valid path: *utils.files.is_path_valid()*.
- File copying and moving, which also creates containing folders if necessary: *utils.files.copy_file()*, *utils.files.move_file()*.
- Folder creation and cleaning: *utils.files.ensure_dir()*, *utils.files.remove_dir()*, *utils.files.remove_dir_if_empty()*, *utils.files.clean_dir()*.
- Analyzing folder content: *utils.files.list_dir()*, *utils.files.list_dir_recursive()*, *utils.files.dir_empty()*, *utils.files.walk_dir()*. Compared to the built-in methods, allows for more complicated (e.g., regex) filters for listed files and folders, as well as for visited folders.
- Copying, moving, and comparing folders: *utils.files.copy_dir()*, *utils.files.move_dir()*, *utils.files.cmp_dirs()*; like methods above, allows for regex filters for files and folders.
- Retrying versions of most of the above methods: e.g., *utils.files.retry_move()* or *utils.files.retry_clean_dir()*. These functions try to copy/move/remove files or folders several times if errors arise, in case the files or folders are only temporarily blocked. Useful when, e.g., using network shares or some software which makes files or folders unavailable for a short period of time.
- Wrapping methods for working with zip files: *utils.files.zip_folder()*, *utils.files.zip_file()*, *utils.files.zip_multiple_files()*, *utils.files.unzip_folder()*, *utils.files.unzip_file()*.

1.5.2 Network

There is a simple wrapper class *utils.net.ClientSocket*, which simplifies some operations with the built-in `socket` module. In addition, it also implements a couple of higher-level ways to send the data: either fixed length (as in the usual socket), with the length prepended (in case the total length is initially unknown at the receiving end), or using a delimiter to mark the end of the message.

In addition, there are several methods for gaining local or remote host information (*utils.net.get_local_addr()*, *utils.net.get_all_local_addr()*, *utils.net.get_local_hostname()*, *utils.net.get_all_remote_addr()*, *utils.net.get_remote_hostname()*), receiving JSON-formatted values (*utils.net.recv_JSON()*), and listening on a given port (*utils.net.listen()*).

1.5.3 Strings

There are several string manipulation functions present:

- Powerful to/from string conversion. The main function are *utils.string.to_string()* and *utils.string.from_string()*, which can convert a large variety of values: simple scalar values (numbers, strings, bools, `None`), containers (lists, tuples, sets, dictionaries), escaped and byte strings (e.g., `b"\x00"`), complex types such as numpy arrays (represented as, e.g., `"array([0, 1, 2, 3, 4])"`). The latter version requires setting `use_classes=True` in *utils.string.to_string()*, which is not enabled by default to make the results more compatible with other parsers:

```
>> pll.to_string(np.arange(5))  # by default, use the standard str method, which␣
↪makes array look like a list
'[0, 1, 2, 3, 4]'
>> pll.from_string('[0, 1, 2, 3, 4]')  # gets converted back into a list
[0, 1, 2, 3, 4]
>> pll.to_string(np.arange(5), use_classes=True)  # use representation class
'array([0, 1, 2, 3, 4])'
>> pll.from_string('array([0, 1, 2, 3, 4])')  # get converted back into an array
array([0, 1, 2, 3, 4])
```

More complex data classes can be added using *utils.string.add_conversion_class()* and *utils.string.add_namedtuple_class()*:

```
>> NamedTuple = collections.namedtuple("NamedTuple", ["field1", "field2"])
>> nt = NamedTuple(1,2)
>> nt
NamedTuple(field1=1, field2=2)
>> pll.to_string(nt, use_classes=True)  # class is not registered, so use the␣
↪default tuple representation
'(1, 2)'
>> pll.add_namedtuple_class(NamedTuple)
>> pll.to_string(nt, use_classes=True)  # now the name marker is added
'NamedTuple(1, 2)'
>> pll.from_string('NamedTuple(1, 2)')
NamedTuple(field1=1, field2=2)
>> DifferentNamedTuple = collections.namedtuple("DifferentNamedTuple", ["field1",
↪"field2"])
>> pll.from_string('DifferentNamedTuple(1, 2)')  # note that if the class is not␣
↪registered, it can't be parsed, so the string is returned back
'DifferentNamedTuple(1, 2)'
```

Furthermore, there is a couple of auxiliary string functions to parse more complicated situations: *utils.string.escape_string()* and *utils.string.unescape_string()* for escaping and unescaping string with potentially confusing or unprintable characters (e.g., quotation marks, spaces, new lines); *utils.string.from_string_partial()*, *utils.string.from_row_string()*, *utils.string.extract_escaped_string()* to determine and extract the first value in a string which potentially has several values.

- Comparing and searching string: *utils.string.string_equal()* (compare string using different rules such as case sensitivity), *utils.string.find_list_string()*, *utils.string.find_dict_string()* (find string in a list or a dictionary using different comparison rules).
- Filtering strings: *utils.string.get_string_filter()*, *utils.string.sfglob()*, and *utils.string.sfregex()*. Creates filter functions which may include or exclude certain string patterns; these filter functions can be later used in, e.g., file-related methods such as *utils.files.list_dir()*.

1.5.4 Misc utilities

A variety of small useful methods and classes:

- Dictionary manipulation functions: *utils.general.any_item()* (get a random dict key-value pair), *utils.general.merge_dicts()* (merge several dictionaries together), *utils.general.filter_dict()* (filter dictionary according ot key or value), *utils.general.map_dict_keys()*, *utils.general.map_dict_values()*, *utils.general.to_dict()* (convert a dict or a list of pairs into a dictionary, using a default value for a non-pair list elements), *utils.general.invert_dict()* (turn keys into values and vice versa).

- List manipulation functions: *utils.general.flatten_list()* (flatten a nested list structure), *utils.general.partition_list()* (split a list into two lists according to a predicate), *utils.general.split_in_groups()* (split list into several groups according to a key function), *utils.general.sort_set_by_list()* (convert set into a list, whose values are sorted according to a second supplied list), *utils.general.compare_lists()* (compare two lists and return their intersection and differences).
- *utils.general.DummyResource*: a "dummy" resource class, which can be used in a `with` block but does nothing; can be used to, e.g., replace multi-threading resources such as locks to turn them off.
- Unique ID generators: *utils.general.UIDGenerator* and *utils.general.NamedUIDGenerator*, which generate unique names (based on a counter), with a thread-safe option (useful to create, e.g., unique data markers).
- Timekeeping: *utils.general.Countdown* for single shot and *utils.general.Timer* for repeating tasks. Simplifies dealing with operation timeouts: checking how much time is left (including options for infinite timeout), checking if timeout is passed, resetting, etc.
- Script restarting vua *utils.general.restart()* (thread-controller style applications can also use *thread.controller.restart_app()* for a more managed restart).
- *utils.general.StreamFileLogger*, which can be set up to log all outputs into a stream (e.g., `stdout`):

```
from pylablib import StreamFileLogger
import sys
sys.stderr = StreamLogger("logerr.txt", sys.stderr)  # replace stderr stream with␣
↪a logged version
# perform some tasks ...
sys.stderr = sys.stderr.stream  # revert back, if necessary
```

  With the code above, all output to `stderr` will be logged into `logerr.txt` to be analyzed later. It can also be set with `autoflush=True` to automatically flush the printed text, which helps with identifying crushing bugs, and it can be supplied with a lock to help separate printouts from different threads.

1.6 Change log

This is a list of changes between each version.

1.6.1 Version 1.x

Transitioning from version 0.x to version 1.x saw lots of interface changes which break backward compatibility. The previous version of the library can be either obtained on PyPi using `pip install "pylablib<1"`, or by using `legacy` module. Hence, instead of

```
import pylablib as pll
from pylablib.aux_libs.devices import Lakeshore
```

you can write

```
import pylablib.legacy as pll
from pylablib.legacy.aux_libs.devices import Lakeshore
```

1.2.1

- General
  - Added restarting methods for regular and threaded applications.
- Threading
  - Bugfixes in cameras and camera threads.
  - Bugfixes in streaming.

1.2.0

- General
  - Added `timing` context manager for simple code timing checks.
  - Improved RPyC wrapper logging and reliability.
  - Added Anaconda support.
  - Added minor network and file functions.
- Devices
  - Added Newport Picomotor 8742 motor controller, Toptica iBeam Smart laser, older version of Thorlabs FW motorized filter wheel.
  - Added camera frame output format (list or array).
  - Added `use_cavity` option to M2 Solstis laser.
  - Added method for auto-detecting associations between PhotonFocus cameras and frame grabbers.
  - Updated some generic classes (DCAM cameras, Thorlabs TLCamera cameras).
  - Updated SCPI failsafe operation, improved Thorlabs FW reliability.
  - Fixed several minor bugs.
- GUI
  - Rewritten GUI values handling to pass calls in a hierarchical manner. This makes the operation more predictable and overloading the behavior a bit easier.
  - Added out-of-range value action for combo boxes.
  - Fixed `ImagePlotter` incompatibility with the newer pyqtgraph versions, added separate x and y axis line cuts selection.
  - Minor layout handling bugfixes.
- Threading
  - Released advanced threading functionality: table/frame streaming, device threads, basic frame processing.
  - Task thread additions: delayed batch job stopping, context manager for task loop pausing.
  - Added argument-dependent call queue limit.
  - Improved threading speed and stability.

1.1.0

- General
  - Reorganized the core modules import structure: now `__init__.py` modules are mostly empty, and all the necessary imports are either exposed directly in `pylablib` (e.g., `pylablib.Fitter`), or should be accessed directly by the module (e.g. `pll.core.dataproc.fitting.Fitter`). Intermediate access (e.g., `pll.core.dataproc.Fitter`) is no longer supported.
  - File IO functions (e.g., `read_csv`) can now take file-like objects in addition to paths.
- Devices
  - Added Silicon Software frame grabbers interface and rearranged PhotonFocus code to include both IMAQ and SiliconSoftware frame grabbers.
  - Fixed various compatibility bugs arising for specific versions of Python or dependency modules: Kinesis error with specific pyft232 versions, some DLL-dependent devices errors with Python 3.8+, DLL types in 32-bit Python.
  - Addressed issue with occasional uc480 acquisition restarts, fixed M2 communication report errors.
- GUI and threading
  - Added container and layout management classes in addition to parameter tables for more consistent GUI structure organization.
  - Added `pylablib.widgets` module which combines all custom widgets for the ease of using in layout managers or custom applications.
  - Fixed support for `PySide2` Qt5 backed.
  - Renamed `setupUi` -> `setup` for all widgets, and changed the GUI setup organization for many of them (the functioning stayed the same).
  - Reorganized scheduling in `QTaskThread` to treat jobs, commands, and subscriptions more consistently.
  - Added basic data stream management.

1.0.0

There have been too many alterations to list here comprehensively. Below is the list of the largest changes.

- General
  - Removed built-in `DataTable` class (together with `core.datatable` subpackage) in favor of pandas.
  - Renamed file IO functions: instead of generic `load` and `save` methods there are now more specific *`loadfile.load_csv()`*, *`loadfile.load_dict()`*, etc.
  - Removed some legacy modules which are not used in the rest of the library.
  - Renamed or moved certain modules: `core.utils.rpyc` -> `core.utils.rpyc_utils`, `core.fileio.logfile` -> `core.fileio.table_stream`, `core.fileio.binio` -> `core.utils.binio` , `core.devio.backend` -> `core.devio.backencd_comm`, `core.devio.untis` -> `core.utils.units`, `core.dataproc.waveforms` -> `core.dataproc.utils`
- Devices
  - Some legacy devices have been removed, since without access to the hardware it is hard to maintain and expand them. These include most of Agilent devices (33502A amplifier, N9310A microwave generator, HP 8712B and HP 8722D network analyzers, HP 8168F laser), Rigol DSA1030A spectrum analyzer, Tektronix MDO3000 oscilloscope, Vaunix LabBrick generators, Zurich Instruments HF2 and UHF, Andor

Shamrock spectrographs (should be restored in future releases), NuPhoton NP2000 EDFA, PurePhotonics PPCL200 laser, Sirah Matisse laser (should be restored in future releases), Thorlabs PM100 power meter (should be restored in future releases), Lakeshore 370 resistance bridge (should be restored in future releases), MKS 900-series pressure gauges, and some custom devices (Arduino and Olimex AVR boards and Janis-related hardware).

  - The main devices package has been moved from `pylablib.aux_libs.devices` (which now refers to the legacy code) to `pylablib.devices`. Module organization has also changed slightly. To find the required modules and device class names, see the *devices list*.
  - Lots of devices' interface has varied slightly, to make the interface more uniform and compatible between different kinds of devices. The changes are usually fairly straightforward (e.g., `move_to` instead of `move`). In many cases the interface was also expanded to include additional available methods.
  - Several devices have been added, generalized, or restructured:
    * Combined Thorlabs KDC101 and K10CR1 into a single class *`pylablib.devices.Thorlabs.BasicKinesisDevice`*, which also accommodates similar kinds of devices.
    * Added Arcus Performax2EXStage device for 2-axis controller with a slightly different interface (*`pylablib.devices.Arcus.Performax2EXStage`*)
    * Added *several more AWGs* with similar interfaces
  - Simplified the way external DLLs are *handled*
  - Unified the *error handling*

- GUI and threading
  - Changed module structure
    * threading and GUI are now separate sub-packages `core.thread` and `core.gui`
    * all widgets are available simply through `pylablib.widgets` (simplifies integration with Qt Designer)
    * moved parameter tables widgets to the core library
  - Renamed some widgets to remove the `LV` prefix.
  - Interfaces changes in some of the classes: thread controllers, parameter tables, value tables. The changes are mostly cosmetics and involve names and parameters order. Most important changes:
    * thread controller methods: `subscribe` -> `subscribe_sync`, `sync_exec` -> `sync_exec_point`,
    * thread controller command/query shortcut: `.c` -> `.ca`, `.q` -> `.cs`, `.qi` -> `.csi`, `.qs` -> `.css`
    * thread controller variable access uses `.v` shortcut, i.e., instead of `ctl[name]` it is now `ctl.v[name]`
    * GUI value storage `ValuesTable`/`IndicatorValuesTable` are now combined and named as `GUIValues`
    * `ParamTable` and `GUIValues` uses `.h` shortcut to access value handlers, i.e., instead of `table[name]` it is now `table.h[name]`
    * `ParamTable`, `ImagePlotterCtl`, `TracePlotterCtl` constructor arguments: `display_table` -> `gui_values`, `display_table_root` -> `gui_values_root`
    * value-changed signal names in `ParamTable` and `GUIValues`: `changed_event` -> `get_value_changed_signal`

    * value-changed signal names in value handlers: `value_changed_signal` -> `get_value_changed_signal`
    * `ParamTable` methods: `lock` -> `set_enabled`, `add_button(checkable=True)` -> `add_toggle_button`
    * `NumEdit` and `NumLabel` methods: `set_number_format` -> `set_formatter`, `set_number_limit` -> `set_limiter` (the call signature also changed)
    * renamed signals to multicasts to avoid confusion with built-in Qt signals. Leads to `ThreadController.send_signal` -> `send_multicast`, `ThreadController.process_signal` -> `process_multicast`, `ThreadController` constructor argument `signal_pool` -> `multicast_pool`, class `SignalPool` -> `MulticastPool`, `QSignalThreadCallScheduler` -> QMulticastThreadCallScheduler.

1.6.2 Version 0.x

0.4.1

Interface changes

- Slightly changed representations of complex number in to-string conversions depending on the conversion rules (`"python"` vs `"text"`).

Additions

- Devices
  - Added Thorlabs K10CR1 rotational stage (`legacy.aux_libs.devices.Thorlabs.K10CR1`)
  - Added Andor Shamrock spectrographs (`legacy.aux_libs.devices.AndorShamrock`)
  - Expanded Agilent AWG class
  - Added more 32bit dlls
  - Added `list_resources` method to every backend class, which lists available connections for this backend (not available for every backend; so far only works in `legacy.core.devio.backed.VisaDeviceBackend`, `legacy.core.devio.backed.SerialDeviceBackend`, and `legacy.core.devio.backed.FT232BackendOpenError`.
- GUI and threading
  - Added `legacy.aux_libs.gui.helpers.TableAccumulatorThread.preprocess_data` method to pre-process incoming data before adding it to the table
  - Added `update_only_on_visible` argument to `legacy.aux_libs.gui.widgets.trace_plotter.TracePlotter.setupUi` method, and `legacy.aux_libs.gui.widgets.trace_plotter.TracePlotter.get_required_channels` method.

0.4.0

Interface changes

- Dictionary entries (`legacy.core.fileio.dict_entry`) system has been slightly redesigned: building entries from stored objects has been moved from `legacy.core.fileio.dict_entry.IDictionaryEntry.build_entry` class method to a dedicated function `legacy.core.fileio.dict_entry.build_entry`, and entry classes have been added.

- legacy.aux_libs.gui.helpers.StreamFormerThread architecture changes, so that it can accumulates several rows before adding them into the storage; this lead to replacement of legacy.aux_libs.gui.helpers.StreamFormerThread.prepare_new_row method by legacy.aux_libs.gui.helpers.StreamFormerThread.prepare_new_data.

Additions

- General
  - Added pandas support in a bunch of places: loading/saving tables and dictionaries; data processing routines in legacy.core.dataproc; conversion of legacy.core.dataproc.datatable.DataTable and legacy.core.utils.dictionary.Dictionary object to/from pandas dataframes.
  - Expanded string conversion to support more explicit variable classes. For example, a numpy array np.array([1,2,3]) can be converted into a string 'array([1, 2, 3])' instead of a more ambiguous string '[1, 2, 3]' (which can also be a list). This behavior is controlled by a new argument use_classes in string conversion functions (such as legacy.core.utils.string.to_string and legacy.core.utils.string.from_string) and an argument use_rep_classes in file saving (legacy.core.fileio.savefile.save)
  - Added general library parameters, which can be accessed via pylablib.par (works as a dictionary object). So far there's only one supported parameter: the default return type of the CSV file reading (can be "pandas" for pandas dataframe, "table" for legacy.core.dataproc.datatable.DataTable object, or "array" for raw numpy array).
- Devices
  - Added LaserQuantum Finesse device class (legacy.aux_libs.devices.devices.LaserQuantum)
  - NI DAQ now supports output of waveforms
  - Added legacy.aux_libs.devices.PCO_SC2.reset_api and legacy.aux_libs.devices.PCO_SC2.PCOSC2Camera.reboot methods for resetting API and cameras
  - Added legacy.aux_libs.devices.Thorlabs.list_kinesis_devices function to list connected Kinesis devices
  - Added serial communication methods for IMAQ cameras (legacy.aux_libs.devices.IMAQ.IMAQCamera)
- GUI and threading
  - Added line plotter (legacy.aux_libs.gui.widgets.line_plotter) and trace plotter (legacy.aux_libs.gui.widgets.trace_plotter) widgets
  - Added virtual elements to value tables and parameter tables
  - Added gui_thread_safe parameter to value tables and parameter tables. Enabling it make most common methods thread-safe (i.e., transparently called from the GUI thread)
  - Added a corresponding legacy.core.gui.qt.thread.controller.gui_thread_method wrapper to implement the change above
  - Added functional thread variables (legacy.core.gui.qt.thread.controller.QThreadController.set_func_variable)
- File saving / loading
  - Added notation for dictionary files to include nested structures ('prefix blocks'). This lets one avoid common path prefix in stored dictionary files. For example, a file

```
some/long/prefix/x  1
some/long/prefix/y  2
some/long/prefix/y  3
```

can be represented as

```
//some/long/prefix
    x   1
    y   2
    z   3
///
```

The meaningful elements are `//some/long/prefix` line denoting that following elements have the given prefix, and `///` line denoting that the prefix block is done (indentation is only added for clarity).

  - New dictionary entries: *dict_entry.ExternalNumpyDictionaryEntry* (external numpy array, can have arbitrary number of dimensions) and *dict_entry.ExpandedContainerDictionaryEntry* (turns lists, tuples and dicts into dictionary branches, so that their content can benefit from the automatic table inlining, dictionary entry classes, etc.).

- Data processing
  - `legacy.core.dataproc.fitting.Fitter` now takes default scale and limit as constructor arguments.
  - `legacy.core.dataproc.feature.multi_scale_peakdet` has new `norm_ratio` argument.
  - `legacy.core.dataproc.image.get_region` and `legacy.core.dataproc.image.get_region_sum` take `axis` argument.
- Miscellaneous
  - Functions introspection module now supports Python 3 - style functions, and added a new function `legacy.core.utils.functions.funcsig`
  - `legacy.core.utils.general.StreamFileLogger` supports multiple destination paths
  - New network function `legacy.core.utils.net.get_all_local_addr` (return list of all local addresses on all interfaces) and `legacy.core.utils.net.get_local_hostname`

1.7 pylablib

1.7.1 pylablib package

Subpackages

pylablib.core package

Subpackages

pylablib.core.dataproc package

Submodules

pylablib.core.dataproc.callable module

class pylablib.core.dataproc.callable.**ICallable**

Bases: `object`

Fit function generalization.

Has a set of mandatory argument with no default values and a set of parameters with default values (there may or may not be an explicit list of them).

All the arguments are passed explicitly by name. Passed value supersede default values. Extra arguments (not used in the calculations) are ignored.

Assumed (but not enforced) to be immutable: changes after creation can break the behavior.

Implements (possibly; depends on subclasses) call namelist binding boosting: if the function is to be called many times with the same parameter names list, one can first bind parameters list, and then call bound function with the corresponding arguments. This way, `callable(**p)` should be equivalent to `callable.bind(p.keys())(*p.values())`.

has_arg(*arg_name*)

Determine if the function has an argument *arg_name* (of all 3 categories)

filter_args_dict(*args*)

Filter argument names dictionary to leave only the arguments that are used

get_mandatory_args()

Return list of mandatory arguments (these are the ones without default values)

is_mandatory_arg(*arg_name*)

Check if the argument *arg_name* is mandatory

get_arg_default(*arg_name*)

Return default value of the argument *arg_name*.

Raise `KeyError` if the argument is not defined or `ValueError` if it has no default value.

bind(*arg_names*, ***bound_params*)

Bind function to a given parameters set, leaving *arg_names* as free parameters (in the given order)

class NamesBoundCall(*func*, *names*, *bound_params*)

Bases: `object`

bind_namelist(*arg_names*, ***bound_params*)

Bind namelist to boost subsequent calls.

Similar to `bind(arg_names)`, but bound function doesn't accept additional parameters and can be boosted.

class pylablib.core.dataproc.callable.**MultiplexedCallable**(*func*, *multiplex_by*, *join_method='stack'*)

Bases: *pylablib.core.dataproc.callable.ICallable*

Multiplex a single callable based on a single parameter.

If the function is called with this parameter as an iterable, then the underlying callable will be called for each value of the parameter separately, and the results will be joined into a single array (if return the values are scalar, they're joined in 1D array; otherwise, they're joined using *join_method*).

Parameters

- **func** (*callable*) – Function to be parallelized.
- **multiplex_by** (*str*) – Name of the argument to be multiplexed by.

- **join_method** (*str*) – Method for combining individual results together if they're non-scalars. Can be either 'list' (combine the results in a single list), 'stack' (combine using numpy.column_stack(), i.e., add dimension to the result), or 'concatenate' (concatenate the return values; the dimension of the result stays the same).

Multiplexing also makes use of call signatures for underlying function even if `__call__` is used.

Note that this operation is slow, and should be used only for high-dimensional multiplexing; for 1D case it's much better to just use numpy arrays as arguments and rely on numpy parallelizing.

has_arg (*arg_name*)
Determine if the function has an argument *arg_name* (of all 3 categories)

get_mandatory_args ()
Return list of mandatory arguments (these are the ones without default values)

get_arg_default (*arg_name*)
Return default value of the argument *arg_name*.

Raise KeyError if the argument is not defined or ValueError if it has no default value.

class NamesBoundCall (*func*, *names*, *bound_params*)
Bases: object

bind (*arg_names*, ***bound_params*)
Bind function to a given parameters set, leaving *arg_names* as free parameters (in the given order)

bind_namelist (*arg_names*, ***bound_params*)
Bind namelist to boost subsequent calls.

Similar to `bind(arg_names)`, but bound function doesn't accept additional parameters and can be boosted.

filter_args_dict (*args*)
Filter argument names dictionary to leave only the arguments that are used

is_mandatory_arg (*arg_name*)
Check if the argument *arg_name* is mandatory

class pylablib.core.dataproc.callable.**JoinedCallable** (*funcs*, *join_method='stack'*)
Bases: *pylablib.core.dataproc.callable.ICallable*

Join several callables sharing the same arguments list.

The results will be joined into a single array (if return the values are scalar, they're joined in 1D array; otherwise, they're joined using *join_method*).

Parameters

- **funcs** (*[callable]*) – List of functions to be joined together.
- **join_method** (*str*) – Method for combining individual results together if they're non-scalars. Can be either 'list' (combine the results in a single list), 'stack' (combine using numpy.column_stack(), i.e., add dimension to the result), or 'concatenate' (concatenate the return values; the dimension of the result stays the same).

has_arg (*arg_name*)
Determine if the function has an argument *arg_name* (of all 3 categories)

get_mandatory_args ()
Return list of mandatory arguments (these are the ones without default values)

get_arg_default (*arg_name*)
Return default value of the argument *arg_name*.

Raise `KeyError` if the argument is not defined or `ValueError` if it has no default value.

class NamesBoundCall(*func*, *names*, *bound_params*)
Bases: `object`

bind(*arg_names*, ***bound_params*)
Bind function to a given parameters set, leaving *arg_names* as free parameters (in the given order)

bind_namelist(*arg_names*, ***bound_params*)
Bind namelist to boost subsequent calls.

Similar to `bind(arg_names)`, but bound function doesn't accept additional parameters and can be boosted.

filter_args_dict(*args*)
Filter argument names dictionary to leave only the arguments that are used

is_mandatory_arg(*arg_name*)
Check if the argument *arg_name* is mandatory

class pylablib.core.dataproc.callable.**FunctionCallable**(*func*, *function_signature=None*, *defaults=None*, *alias=None*)
Bases: `pylablib.core.dataproc.callable.ICallable`

Callable based on a function or a method.

Parameters

- **func** – Function to be wrapped.
- **function_signature** – A `functions.FunctionSignature` object supplying information about function's argument names and default values, if they're different from what's extracted from its signature.
- **defaults** (`dict`) – A dictionary `{name: value}` of additional default parameters values. Override the defaults from the signature. All default values must be pass-able to the function as a parameter
- **alias** (`dict`) – A dictionary `{alias: original}` for renaming some of the original arguments. Original argument names can't be used if aliased (though, multi-aliasing can be used explicitly, e.g., `alias={'alias':'arg', 'arg':'arg'}`). A name can be blocked (its usage causes error) if it's aliased to None (`alias={'blocked_name':None}`).

Optional non-named arguments in the form `*args` are not supported, since all the arguments are passed to the function by keywords.

Optional named arguments in the form `**kwargs` are supported only if their default values are explicitly provided in defaults (otherwise it would be unclear whether argument should be added into `**kwargs` or ignored altogether).

has_arg(*arg_name*)
Determine if the function has an argument *arg_name* (of all 3 categories)

get_mandatory_args()
Return list of mandatory arguments (these are the ones without default values)

get_arg_default(*arg_name*)
Return default value of the argument *arg_name*.

Raise `KeyError` if the argument is not defined or `ValueError` if it has no default value.

class NamesBoundCall(*func*, *names*, *bound_params*)
Bases: `object`

bind(*arg_names*, ***bound_params*)
Bind function to a given parameters set, leaving *arg_names* as free parameters (in the given order)

bind_namelist(*arg_names*, ***bound_params*)
Bind namelist to boost subsequent calls.

Similar to `bind(arg_names)`, but bound function doesn't accept additional parameters and can be boosted.

filter_args_dict(*args*)
Filter argument names dictionary to leave only the arguments that are used

is_mandatory_arg(*arg_name*)
Check if the argument *arg_name* is mandatory

class pylablib.core.dataproc.callable.**MethodCallable**(*method*, *function_signature=None*, *defaults=None*, *alias=None*)
Bases: `pylablib.core.dataproc.callable.FunctionCallable`

Similar to `FunctionCallable`, but accepts class method instead of a function.

The only addition is that now object's attributes can also parameters to the function: all the parameters which are not explicitly mentioned in the method signature are assumed to be object's attributes.

The parameters are affected by alias, but NOT affected by defaults (since it's impossible to ensure that all object's attributes are kept constant, and it's impractical to reset them all to default values at every function call).

Parameters

- **method** – Method to be wrapped.
- **function_signature** – A `functions.FunctionSignature` object supplying information about function's argument names and default values, if they're different from what's extracted from its signature. If it's assumed that the first self argument is already excluded.
- **defaults** (`dict`) – A dictionary `{name: value}` of additional default parameters values. Override the defaults from the signature. All default values must be pass-able to the function as a parameter
- **alias** (`dict`) – A dictionary `{alias: original}` for renaming some of the original arguments. Original argument names can't be used if aliased (though, multi-aliasing can be used explicitly, e.g., `alias={'alias':'arg', 'arg':'arg'}`). A name can be blocked (its usage causes error) if it's aliased to None (`alias={'blocked_name':None}`).

This callable is implemented largely to be used with `TheoryCalculator` class (currently deprecated).

has_arg(*arg_name*)
Determine if the function has an argument *arg_name* (of all 3 categories)

get_arg_default(*arg_name*)
Return default value of the argument *arg_name*.

Raise `KeyError` if the argument is not defined or `ValueError` if it has no default value.

class NamesBoundCall(*func*, *names*, *bound_params*)
Bases: `object`

bind(*arg_names*, ***bound_params*)
Bind function to a given parameters set, leaving *arg_names* as free parameters (in the given order)

bind_namelist(*arg_names*, ***bound_params*)
Bind namelist to boost subsequent calls.

Similar to `bind(arg_names)`, but bound function doesn't accept additional parameters and can be boosted.

filter_args_dict(*args*)
Filter argument names dictionary to leave only the arguments that are used

get_mandatory_args()
Return list of mandatory arguments (these are the ones without default values)

is_mandatory_arg(*arg_name*)
Check if the argument *arg_name* is mandatory

`pylablib.core.dataproc.callable.`**to_callable**(*func*)
Convert a function to an `ICallable` instance.

If it's already `ICallable`, return unchanged. Otherwise, return `FunctionCallable` or `MethodCallable` depending on whether it's a function or a bound method.

pylablib.core.dataproc.feature module

Traces feature detection: peaks, baseline, local extrema.

class `pylablib.core.dataproc.feature.`**Baseline**
Bases: `pylablib.core.dataproc.feature.Baseline`

Baseline (background) for a trace.

position is the background level, and *width* is its noise width.

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

position

width

`pylablib.core.dataproc.feature.`**get_baseline_simple**(*trace*, *find_width=True*)
Get the baseline of the 1D trace.

If `find_width==True`, calculate its width as well.

`pylablib.core.dataproc.feature.`**subtract_baseline**(*trace*)
Subtract baseline from the trace (make its background zero).

class `pylablib.core.dataproc.feature.`**Peak**
Bases: `pylablib.core.dataproc.feature.Peak`

A trace peak.

kernel defines its shape (for, e.g., generation purposes).

count()
Return number of occurrences of value.

height

index()

Return first index of value.

Raises ValueError if the value is not present.

kernel

position

width

pylablib.core.dataproc.feature.**find_peaks_cutoff**(*trace*, *cutoff*, *min_width=0*, *kind='peak'*, *subtract_bl=True*)

Find peaks in the data using cutoff.

Parameters

- **trace** – 1D data array.
- **cutoff** (*float*) – Cutoff value for the peak finding.
- **min_width** (*int*) – Minimal uninterrupted width (in datapoints) of a peak. Any peaks this width are ignored.
- **kind** (*str*) – Peak kind. Can be `'peak'` (positive direction), `'dip'` (negative direction) or `'both'` (both directions).
- **subtract_bl** (*bool*) – If `True`, subtract baseline of the trace before checking cutoff.

Returns List of *Peak* objects.

pylablib.core.dataproc.feature.**rescale_peak**(*peak*, *xoff=0.0*, *xscale=1.0*, *yoff=0*, *yscale=1.0*)

Rescale peak's position, width and height.

xscale rescales position and width, *xoff* shifts position, *yscale* and *yoff* affect peak height.

pylablib.core.dataproc.feature.**peaks_sum_func**(*peaks*, *peak_func='lorentzian'*)

Create a function representing sum of *peaks*.

peak_func determines default peak kernel (used if `peak.kernel=="generic"`). Kernel is either a name string or a function taking 3 arguments `(x, width, height)`.

pylablib.core.dataproc.feature.**get_kernel**(*width*, *kernel_width=None*, *kernel='lorentzian'*)

Get a finite-sized kernel.

Return 1D array of length `2*kernel_width+1` containing the given kernel. By default, `kernel_width=int(width*3)`.

pylablib.core.dataproc.feature.**get_peakdet_kernel**(*peak_width*, *background_width*, *kernel_width=None*, *kernel='lorentzian'*)

Get a peak detection kernel.

Return 1D array of length `2*kernel_width+1` containing the kernel. The kernel is a sum of narrow positive peak (with the width *peak_width*) and a broad negative peak (with the width *background_width*); both widths are specified in datapoints (index). Each peak is normalized to have unit sum, i.e., the kernel has zero total sum. By default, `kernel_width=int(background_width*3)`.

pylablib.core.dataproc.feature.**multi_scale_peakdet**(*trace*, *widths*, *background_ratio*, *kind='peak'*, *norm_ratio=None*, *kernel='lorentzian'*)

Detect multiple peak widths using `get_peakdet_kernel()` kernel.

Parameters

- **trace** – 1D data array.
- **widths** (*[float]*) – Array of possible peak widths.
- **background_ratio** (*float*) – ratio of the *background_width* to the *peak_width* in `get_peakdet_kernel()`.
- **kind** (*str*) – Peak kind. Can be `'peak'` (positive direction) or `'dip'` (negative direction).
- **norm_ratio** (*float*) – if not `None`, defines the width of the "normalization region" (in units of the kernel width, same as for the background kernel); it is then used to calculate a local trace variance to normalize the peaks magnitude.
- **kernel** – Peak matching kernel.

Returns Filtered trace which shows peak 'affinity' at each point.

`pylablib.core.dataproc.feature.`**`find_local_extrema`**(*wf*, *region_width=3*, *kind='max'*, *min_distance=None*)

Find local extrema (minima or maxima) of 1D trace.

kind can be `"min"` or `"max"` and determines the kind of the extrema. Local minima (maxima) are defined as points which are smaller (greater) than all other points in the region of width *region_width* around it. *region_width* is always round up to an odd integer. *min_distance* defines the minimal distance between the extrema (`region_width//2` by default). If there are several extrema within *min_distance*, their positions are averaged together.

`pylablib.core.dataproc.feature.`**`latching_trigger`**(*wf*, *threshold_on*, *threshold_off*, *init_state='undef'*, *result_kind='separate'*)

Determine indices of rise and fall trigger events with hysteresis (latching) thresholds.

Return either two arrays `(rise_trig, fall_trig)` containing trigger indices (if `result_kind=="separate"`), or a single array of tuples `[(dir,pos)]`, where *dir* is the trigger direction (`+1` or `-1`) and *pos* is its index (if `result_kind=="joined"`). Triggers happen when a state switch from 'high' to 'low' (rising) or vice versa (falling). The state switches from 'low' to 'high' when the trace value goes above *threshold_on*, and from 'high' to 'low' when the trace value goes below *threshold_off*. For a stable hysteresis effect, *threshold_on* should be larger than *threshold_off*, which means that the trace values between these two thresholds can not change the state. *init_state* specifies the initial state: `"low"`, `"high"`, or `"undef"` (undefined state).

pylablib.core.dataproc.filters module

Routines for filtering arrays (mostly 1D data).

`pylablib.core.dataproc.filters.`**`convolve1d`**(*trace*, *kernel*, *mode='reflect'*, *cval=0.0*)

Convolution filter.

Convolves *trace* with the given *kernel* (1D array). *mode* and *cval* determine how the endpoints are handled. Simply a wrapper around the standard `scipy.ndimage.convolve1d()` that handles complex arguments.

`pylablib.core.dataproc.filters.`**`convolution_filter`**(*a*, *width*, *kernel='gaussian'*, *kernel_span='auto'*, *mode='reflect'*, *cval=0.0*, *kernel_height=None*)

Convolution filter.

Parameters

- **a** – array for filtering.

- **width** (*float*) – kernel width (second parameter to the kernel function).
- **kernel** – either a string defining the kernel function (see `specfunc.get_kernel_func()` for possible kernels), or a function taking 3 arguments `(pos, width, height)`, where *height* can be `None` (assumes normalization by area).
- **kernel_span** – the cutoff for the kernel function. Either an integer (number of points) or `'auto'` (autodetect for `"gaussian"`, `"rectangle"` and `"exp_decay"`, full trace width for all other kernels).
- **mode** (*str*) – convolution mode (see `scipy.ndimage.convolve()`).
- **cval** (*float*) – convolution fill value (see `scipy.ndimage.convolve()`).
- **kernel_height** – height parameter to be passed to the kernel function. `None` means normalization by area.

`pylablib.core.dataproc.filters.`**`gaussian_filter`**(*a*, *width*, *mode='reflect'*, *cval=0.0*)

Simple gaussian filter. Can handle complex data.

Equivalent to a convolution with a gaussian. Equivalent to `scipy.ndimage.gaussian_filter1d()`, uses `convolution_filter()`.

`pylablib.core.dataproc.filters.`**`gaussian_filter_nd`**(*a*, *width*, *mode='reflect'*, *cval=0.0*)

Simple gaussian filter. Can't handle complex data.

Equivalent to a convolution with a gaussian. Wrapper around `scipy.ndimage.gaussian_filter()`.

`pylablib.core.dataproc.filters.`**`low_pass_filter`**(*trace*, *t*, *mode='reflect'*, *cval=0.0*)

Simple single-pole low-pass filter.

t is the filter time constant, *mode* and *cval* are the trace expansion parameters (only from the left). Implemented as a recursive digital filter, so its performance doesn't depend strongly on *t*. Works only for 1D arrays.

`pylablib.core.dataproc.filters.`**`high_pass_filter`**(*trace*, *t*, *mode='reflect'*, *cval=0.0*)

Simple single-pole high-pass filter (equivalent to subtracting a low-pass filter).

t is the filter time constant, *mode* and *cval* are the trace expansion parameters (only from the left). Implemented as a recursive digital filter, so its performance doesn't depend strongly on *t*. Works only for 1D arrays.

`pylablib.core.dataproc.filters.`**`integrate`**(*trace*)

Calculate the integral of the trace.

Alias for `numpy.cumsum()`.

`pylablib.core.dataproc.filters.`**`differentiate`**(*trace*)

Calculate the differential of the trace.

Note that since the data dimensions are changed (length is reduced by 1), the index is not preserved for pandas DataFrames.

`pylablib.core.dataproc.filters.`**`sliding_average`**(*a*, *width*, *mode='reflect'*, *cval=0.0*)

Simple sliding average filter

Equivalent to convolution with a rectangle peak function.

`pylablib.core.dataproc.filters.`**`median_filter`**(*a*, *width*, *mode='reflect'*, *cval=0.0*)

Median filter.

Wrapper around `scipy.ndimage.median_filter()`.

`pylablib.core.dataproc.filters.`**`sliding_filter`**(*trace*, *n*, *dec='bin'*, *mode='reflect'*, *cval=0.0*)

Perform sliding filtering on the data.

Parameters

- **trace** – 1D array-like object.
- **n** (*int*) – bin width.
- **dec** (*str*) – decimation method. Can be - `'bin'` or `'mean'` - do a binning average; - `'sum'` - sum points; - `'min'` - leave min point; - `'max'` - leave max point; - `'median'` - leave median point (works as a median filter). - a function which takes a single 1D array and compresses it into a number
- **mode** (*str*) – Expansion mode. Can be `'constant'` (added values are determined by *cval*), `'nearest'` (added values are end values of the trace), `'reflect'` (reflect trace with respect to its endpoint) or `'wrap'` (wrap the values from the other size).
- **cval** (*float*) – If `mode=='constant'`, determines the expanded values.

`pylablib.core.dataproc.filters.`**`decimate`**(*a*, *n*, *dec='skip'*, *axis=0*, *mode='drop'*)

Decimate the data.

Note that since the data dimensions are changed, the index is not preserved for pandas DataFrames.

Parameters

- **a** – data array.
- **n** (*int*) – decimation factor.
- **dec** (*str*) – decimation method. Can be - `'skip'` - just leave every n'th point while completely omitting everything else; - `'bin'` or `'mean'` - do a binning average; - `'sum'` - sum points; - `'min'` - leave min point; - `'max'` - leave max point; - `'median'` - leave median point (works as a median filter). - a function which takes two arguments (nD numpy array and an axis) and compresses the array along the given axis
- **axis** (*int*) – axis along which to perform the decimation; can also be a tuple, in which case the same decimation is performed sequentially along several axes.
- **mode** (*str*) – determines what to do with the last bin if it's incomplete. Can be either `'drop'` (omit the last bin) or `'leave'` (keep it).

`pylablib.core.dataproc.filters.`**`binning_average`**(*a*, *width*, *axis=0*, *mode='drop'*)

Binning average filter.

Equivalent to *`decimate()`* with `dec=='bin'`.

`pylablib.core.dataproc.filters.`**`decimate_full`**(*a*, *dec='skip'*, *axis=0*)

Completely decimate the data along a given axis.

Parameters

- **a** – data array.
- **dec** (*str*) – decimation method. Can be - `'skip'` - just leave every n'th point while completely omitting everything else; - `'bin'` or `'mean'` - do a binning average; - `'sum'` - sum points; - `'min'` - leave min point; - `'max'` - leave max point; - `'median'` - leave median point (works as a median filter). - a function which takes two arguments (nD numpy array and an axis) and compresses the array along the given axis
- **axis** (*int*) – axis along which to perform the decimation; can also be a tuple, in which case the same decimation is performed along several axes.

`pylablib.core.dataproc.filters.`**`decimate_datasets`**(*arrs*, *dec='mean'*)

Decimate datasets with the same shape element-wise (works only for 1D or 2D arrays).

Note that the index data is taken from the first array in the list.

dec has the same values and meaning as in `decimate()`. The format of the output (numpy or pandas, and the name of columns in pandas DataFrame) is determined by the first array in the list.

`pylablib.core.dataproc.filters.`**`collect_into_bins`**(*values*, *distance*, *preserve_order=False*, *to_return='value'*)

Collect all values into bins separated at least by *distance*.

Return the extent of each bin. If `preserve_order==False`, values are sorted before splitting. If `to_return="value"`, the extent is given in values; if `to_return="index"`, it is given in indices (only useful if `preserve_order=True`, as otherwise the indices correspond to a sorted array). If *distance* is a tuple, then it denotes the minimal and the maximal separation between consecutive elements; otherwise, it is a single number denoting maximal absolute distance (i.e., it corresponds to a tuple `(-distance,distance)`).

`pylablib.core.dataproc.filters.`**`split_into_bins`**(*values*, *max_span*, *max_size=None*)

Split values into bins of the span at most *max_span* and number of elements at most *max_size*.

If *max_size* is `None`, it's assumed to be infinite. Return array of indices for each bin. Values are sorted before splitting.

`pylablib.core.dataproc.filters.`**`fourier_filter`**(*trace*, *response*, *dt=1*, *preserve_real=True*)

Apply filter to a trace in the frequency domain.

response is a (possibly) complex function with single 1D real numpy array as a frequency argument. *dt* specifies time step between consecutive points. Note that in case of a multi-column data the filter is applied column-wise; this is in contrast with the Fourier transform methods, which would assume the first column to be times.

If `preserve_real==True`, then the *response* for negative frequencies is automatically taken to be complex conjugate of the *response* for positive frequencies (so that the real trace stays real).

`pylablib.core.dataproc.filters.`**`fourier_make_response_real`**(*response*)

Turn a frequency filter function into a real one (in the time domain).

Done by reflecting and complex conjugating positive frequency part to negative frequencies. *response* is a function with a single argument (frequency), return value is a modified function.

`pylablib.core.dataproc.filters.`**`fourier_filter_bandpass`**(*pass_range_min*, *pass_range_max*)

Generate a bandpass filter function (hard cutoff).

The function is symmetric, so that it corresponds to a real response in time domain.

`pylablib.core.dataproc.filters.`**`fourier_filter_bandstop`**(*stop_range_min*, *stop_range_max*)

Generate a bandstop filter function (hard cutoff).

The function is symmetric, so that it corresponds to a real response in time domain.

pylablib.core.dataproc.fitting module

Universal function fitting interface.

`class` `pylablib.core.dataproc.fitting.`**`Fitter`**(*func*, *xarg_name=None*, *fit_parameters=None*, *fixed_parameters=None*, *scale=None*, *limits=None*, *weights=None*)

Bases: `object`

Fitter object.

Can handle variety of different functions, complex arguments or return values, array arguments.

Parameters

- **func** (*callable*) – Fit function. Can be anything callable (function, method, object with `__call__` method, etc.).
- **xarg_name** (*str or list*) – Name (or multiple names) for x arguments. These arguments are passed to *func* (as named arguments) when calling for fitting. Can be a string (single argument) or a list (arbitrary number of arguments, including zero).
- **fit_parameters** (*dict*) – Dictionary `{name: value}` of parameters to be fitted (*value* is the starting value for the fitting procedure). If *value* is `None`, try and get the default value from the *func*.
- **fixed_parameters** (*dict*) – Dictionary `{name: value}` of parameters to be fixed during the fitting procedure. If *value* is `None`, try and get the default value from the *func*.
- **scale** (*dict*) – Defines typical scale of fit parameters (used to normalize fit parameters supplied of `scipy.optimize.least_squares()`). Note: for complex parameters scale must also be a complex number, with re and im parts of the scale variable corresponding to the scale of the re and im part.
- **limits** (*dict*) – Boundaries for the fit parameters (missing entries are assumed to be unbound). Each boundary parameter is a tuple `(lower, upper)`. `lower` or `upper` can be `None`, `numpy.nan` or `numpy.inf` (with the appropriate sign), which implies no bounds in the given direction. Note: for compound data types (such as lists) the entries are still tuples of 2 elements, each of which is either `None` (no bound for any sub-element) or has the same structure as the full parameter. Note: for complex parameters limits must also be complex numbers (or `None`), with re and im parts of the limits variable corresponding to the limits of the re and im part.
- **weights** (*list or numpy.ndarray*) – Determines the weights of y-points. Can be either an array broadcastable to *y* (e.g., a scalar or an array with the same shape as *y*), in which case it's interpreted as list of individual point weights (which multiply residuals before they are squared). Or it can be an array with number of elements which is square of the number of elements in *y*, in which case it's interpreted as a weights matrix (which matrix-multiplies residuals before they are squared).

set_xarg_name(*xarg_name*)
Set names of x arguments.

Can be a string (single argument) or a list (arbitrary number of arguments, including zero).

use_xarg()
Return `True` if the function requires x arguments

set_fixed_parameters(*fixed_parameters*)
Change fixed parameters

update_fixed_parameters(*fixed_parameters*)
Update the dictionary of fixed parameters

del_fixed_parameters(*fixed_parameters*)
Remove fixed parameters

set_fit_parameters(*fit_parameters*)
Change fit parameters

update_fit_parameters(*fit_parameters*)
Update the dictionary of fit parameters

del_fit_parameters(*fit_parameters*)
Remove fit parameters

fit (*x=None*, *y=0*, *fit_parameters=None*, *fixed_parameters=None*, *scale='default'*, *limits='default'*, *weights=1.0*, *parscore=None*, *return_stderr=False*, *return_residual=False*, ***kwargs*)

Fit the data.

Parameters

- **x** – x arguments. If the function has single x argument, *x* is an array-like object; otherwise, *x* is a list of array-like objects (can be `None` if there are no x parameters).
- **y** – Target function values.
- **fit_parameters** (*dict*) – Adds to the default *fit_parameters* of the fitter (has priority on duplicate entries).
- **fixed_parameters** (*dict*) – Adds to the default *fixed_parameters* of the fitter (has priority on duplicate entries).
- **scale** (*dict*) – Defines typical scale of fit parameters (used to normalize fit parameters supplied of `scipy.optimize.least_squares()`). Note: for complex parameters scale must also be a complex number, with re and im parts of the scale variable corresponding to the scale of the re and im part. If value is `"default"`, use the value supplied on the fitter creation (by default, no specific scales).
- **limits** (*dict*) – Boundaries for the fit parameters (missing entries are assumed to be unbound). Each boundary parameter is a tuple `(lower, upper)`. `lower` or `upper` can be `None`, `numpy.nan` or `numpy.inf` (with the appropriate sign), which implies no bounds in the given direction. Note: for compound data types (such as lists) the entries are still tuples of 2 elements, each of which is either `None` (no bound for any sub-element) or has the same structure as the full parameter. Note: for complex parameters limits must also be complex numbers (or `None`), with re and im parts of the limits variable corresponding to the limits of the re and im part. If value is `"default"`, use the value supplied on the fitter creation (by default, no limits).
- **weights** (*list or numpy.ndarray*) – Determines the weights of y-points. Can be either an array broadcastable to *y* (e.g., a scalar or an array with the same shape as *y*), in which case it's interpreted as list of individual point weights (which multiply residuals before they are squared). Or it can be an array with number of elements which is square of the number of elements in *y*, in which case it's interpreted as a weights matrix (which matrix-multiplies residuals before they are squared). If value is `"default"`, use the value supplied on the fitter creation (by default, no weights)
- **parscore** (*callable*) – parameter score function, whose value is added to the mean-square error (sum of all residuals squared) after applying weights. Takes the same parameters as the fit function, only without the x-arguments, and return an array-like value. Can be used for, e.g., 'soft' fit parameter constraining.
- **return_stderr** (*bool*) – If `True`, append *stderr* to the output.
- **return_residual** – If not `False`, append *residual* to the output.
- ****kwargs** – arguments passed to `scipy.optimize.least_squares()` function.

Returns

(params, bound_func[, stderr][, residual]):

- *params*: a dictionary `{name: value}` of the parameters supplied to the function (both fit and fixed).
- *bound_func*: the fit function with all the parameters bound (i.e., it only requires x parameters).

- ***stderr*: a dictionary `{name: error}` of standard deviation for fit parameters to the return parameters.** If the fitting routine returns no residuals (usually for a bad or an under-constrained fit), all residuals are set to NaN.
- ***residual*: either a full array of residuals `func(x,**params)-y` (if `return_residual=='full'`),** a mean magnitude of the residuals `mean(abs(func(x,**params)-y)**2)` (if `return_residual==True` or `return_residual=='mean'`), or the total residuals including weights `mean(abs((func(x,**params)-y)*weights)**2)` (if `return_residual=='weighted'`).

Return type tuple

initial_guess(*fit_parameters=None*, *fixed_parameters=None*, *return_stderr=False*, *return_residual=False*)
Return the initial guess for the fitting.

Parameters

- **fit_parameters** (*dict*) – Overrides the default *fit_parameters* of the fitter.
- **fixed_parameters** (*dict*) – Overrides the default *fixed_parameters* of the fitter.
- **return_stderr** (*bool*) – If `True`, append *stderr* to the output.
- **return_residual** – If not `False`, append *residual* to the output.

Returns

`(params, bound_func)`.

- *params*: a dictionary `{name: value}` of the parameters supplied to the function (both fit and fixed).
- *bound_func*: the fit function with all the parameters bound (i.e., it only requires x parameters).
- ***stderr*: a dictionary `{name: error}` of standard deviation for fit parameters to the return parameters.** Always zero, added for better compatibility with `fit()`.
- ***residual*: either a full array of residuals `func(x,**params)-y` (if `return_residual=='full'`)** a mean magnitude of the residuals `mean(abs(func(x,**params)-y)**2)` (if `return_residual==True` or `return_residual=='mean'`). Always zero, added for better compatibility with `fit()`.

Return type tuple

`pylablib.core.dataproc.fitting.`**huge_error**(*x*, *factor=100.0*)

`pylablib.core.dataproc.fitting.`**get_best_fit**(*x*, *y*, *fits*)
Select the best (lowest residual) fit result.

x and *y* are the argument and the value of the bound fit function. *fits* is the list of fit results (tuples returned by `Fitter.fit()`).

pylablib.core.dataproc.fourier module

Routines for Fourier transform.

`pylablib.core.dataproc.fourier.`**get_prev_len**(*l*, *maxprime=7*)
Get the largest number less or equal to *l*, which is composed of prime factors up to *maxprime*.

So far, only *maxprime* of 2, 3, 5, 7 and 11 are supported. *maxprime* of 5 gives less than 15% length reduction (and less than 6% for lengths above 400). *maxprime* of 11 gives less than 8% length reduction (and less than 4% for lengths above 300).

`pylablib.core.dataproc.fourier.`**`truncate_trace`**(*trace*, *maxprime=7*)

Truncate trace length to the nearest smaller length which is composed of prime factors up to *maxprime*.

So far, only *maxprime* of 2, 3, 5, 7 and 11 are supported. *maxprime* of 5 gives less than 15% length reduction (and less than 6% for lengths above 400). *maxprime* of 11 gives less than 8% length reduction (and less than 4% for lengths above 300).

`pylablib.core.dataproc.fourier.`**`normalize_fourier_transform`**(*ft*, *normalization='none'*, *df=None*, *copy=False*)

Normalize the Fourier transform data.

ft is a 1D trace or a 2D array with 2 columns: frequency and complex amplitude. *normalization* can be `'none'` (standard numpy normalization), `'sum'` (the power sum is preserved: `sum(abs(ft)**2)==sum(abs(trace)**2)`), `'rms'` (the power sum is equal to the trace RMS power: `sum(abs(ft)**2)==mean(abs(trace)**2)`), `'density'` (power spectral density normalization, `sum(abs(ft[:,1])**2)*df==mean(abs(trace[:,1])**2)`), or `'dBc'` (same as `'density'`, but normalized by the mean of the trace) If `normalization=='density'`, then *df* can specify the frequency step between two consecutive bins; if *df* is `None`, it is extracted from the first two points of the frequency axis (or set to 1, if *ft* is a 1D trace)

`pylablib.core.dataproc.fourier.`**`apply_window`**(*trace_values*, *window='rectangle'*, *window_power_compensate=True*)

Apply FT window to the trace.

If `window_power_compensate==True`, multiply the data is multiplied by a compensating factor to preserve power in the spectrum.

`pylablib.core.dataproc.fourier.`**`fourier_transform`**(*trace*, *dt=None*, *truncate=False*, *normalization='none'*, *single_sided=False*, *window='rectangle'*, *window_power_compensate=True*, *raw=False*)

Calculate a fourier transform of the trace.

Parameters

- **`trace`** – Time trace to be transformed. It can be a 1D trace of values, a 2-column trace, or a 3-column trace. If *dt* is `None`, then the first column is assumed to be time (only support uniform time step), and the other columns are either the trace values (for a single data column) or real and imaginary parts of the trace (for two data columns). If *dt* is not `None`, then the time column is assumed to be missing, so the two columns are assumed to be the real and the imaginary parts.
- **`dt`** – if not `None`, can specify the time step between the consecutive samples, in which case it is assumed that the time column is missing from the trace; otherwise, try to get it from the time column of the trace if it exists, or set to 1 otherwise.
- **`truncate`** (*`bool or int`*) – Determines whether to truncate the trace to the nearest product of small primes (speeds up FFT algorithm); can be `False` (no truncation), an integer 2, 3, 5, 7, or 11 (truncate to a product of primes up to and including this number), or `True` (default prime factorization, currently set to 7)
- **`normalization`** (*`str`*) – Fourier transform normalization: - `'none'`: no

(i.e., default numpy) normalization; - `'sum'`: the norm of the data is conserved (`sum(abs(ft[:,1])**2)==sum(abs(trace[:,1])**2)`); - `'rms'`: sum of the PSD is equal to the RMS trace amplitude squared (`sum(abs(ft[:,1])**2)==mean(abs(trace[:,1])**2)`); - `'density'`: power spectral density normalization, in `x/rtHz` (`sum(abs(ft[:,1])**2)*df==mean(abs(trace[:,1])**2)`); - `'dBc'`: like `'density'`, but normalized to the mean trace value.

- **single_sided** (*bool*) – If `True`, only leave positive frequency side of the transform.
- **window** (*str*) – FT window. Can be `'rectangle'` (essentially, no window), `'hann'` or `'hamming'`.
- **window_power_compensate** (*bool*) – If `True`, the data is multiplied by a compensating factor to preserve power in the spectrum.
- **raw** (*bool*) – if `True`, return a simple 1D trace with the result.

Returns a two-column array of the same kind as the input, where the first column is frequency, and the second is complex FT data.

`pylablib.core.dataproc.fourier.`**`flip_fourier_transform`**(*ft*)

Flip the fourier transform (analogous to making frequencies negative and flipping the order).

`pylablib.core.dataproc.fourier.`**`inverse_fourier_transform`**(*ft, df=None, truncate=False, zero_loc=None, symmetric_time=False, raw=False*)

Calculate an inverse fourier transform of the trace.

Parameters

- **ft** – Fourier transform data to be inverted. It can be a 1D trace of values, a 2-column trace, or a 3-column trace. If *df* is `None`, then the first column is assumed to be frequency (only support uniform frequency step), and the other columns are either the trace values (for a single data column) or real and imaginary parts of the trace (for two data columns). If *df* is not `None`, then the frequency column is assumed to be missing, so the two columns are assumed to be the real and the imaginary parts.
- **df** – if not `None`, can specify the frequency step between the consecutive samples; otherwise, try to get it from the frequency column of the trace if it exists, or set to 1 otherwise.
- **truncate** (*bool or int*) – Determines whether to truncate the trace to the nearest product of small primes (speeds up FFT algorithm); can be `False` (no truncation), an integer 2, 3, 5, 7, or 11 (truncate to a product of primes up to and including this number), or `True` (default prime factorization, currently set to 7)
- **zero_loc** (*bool*) – Location of the zero frequency point. Can be `None` (the one with the value of f-axis closest to zero, or the first point if the frequency column is missing), `'center'` (mid-point), or an integer index.
- **symmetric_time** (*bool*) – If `True`, make time axis go from `(-0.5/df, 0.5/df)` rather than `(0, 1./df)`.
- **raw** (*bool*) – if `True`, return a simple 1D trace with the result.

Returns a two-column array, where the first column is frequency, and the second is the complex-valued trace data.

`pylablib.core.dataproc.fourier.`**`power_spectral_density`**(*trace*, *dt=None*, *truncate=False*, *normalization='density'*, *single_sided=False*, *window='rectangle'*, *window_power_compensate=True*, *raw=False*)

Calculate a power spectral density of the trace.

Parameters

- **`trace`** – Time trace to be transformed. It can be a 1D trace of values, a 2-column trace, or a 3-column trace. If *dt* is `None`, then the first column is assumed to be time (only support uniform time step), and the other columns are either the trace values (for a single data column) or real and imaginary parts of the trace (for two data columns). If *dt* is not `None`, then the time column is assumed to be missing, so the two columns are assumed to be the real and the imaginary parts.
- **`dt`** – if not `None`, can specify the time step between the consecutive samples; otherwise, try to get it from the time column of the trace if it exists, or set to 1 otherwise.
- **`truncate`** (*`bool or int`*) – Determines whether to truncate the trace to the nearest product of small primes (speeds up FFT algorithm); can be `False` (no truncation), an integer 2, 3, 5, 7, or 11 (truncate to a product of primes up to and including this number), or `True` (default prime factorization, currently set to 7)
- **`normalization`** (*`str`*) – Fourier transform normalization: - `'none'`: no (i.e., default numpy) normalization; - `'sum'`: the norm of the data is conserved (`sum(PSD[:,1])==sum(abs(trace[:,1])**2)`); - `'rms'`: sum of the PSD is equal to the RMS trace amplitude squared (`sum(PSD[:,1])==mean(abs(trace[:,1])**2)`); - `'density'`: power spectral density normalization, in `x/rtHz` (`sum(PSD[:,1])*df==mean(abs(trace[:,1])**2)`); - `'dBc'`: like `'density'`, but normalized to the mean trace value.
- **`single_sided`** (*`bool`*) – If `True`, only leave positive frequency side of the PSD.
- **`window`** (*`str`*) – FT window. Can be `'rectangle'` (essentially, no window), `'hann'` or `'hamming'`.
- **`window_power_compensate`** (*`bool`*) – If `True`, the data is multiplied by a compensating factor to preserve power in the spectrum.
- **`raw`** (*`bool`*) – if `True`, return a simple 1D trace with the result.

Returns a two-column array, where the first column is frequency, and the second is positive PSD.

`pylablib.core.dataproc.fourier.`**`get_real_part_ft`**(*ft*)

Get the fourier transform of the real part only from the fourier transform of a complex variable.

`pylablib.core.dataproc.fourier.`**`get_imag_part_ft`**(*ft*)

Get the fourier transform of the imaginary part only from the fourier transform of a complex variable.

`pylablib.core.dataproc.fourier.`**`get_correlations_ft`**(*ft_a*, *ft_b*, *zero_mean=True*, *normalization='none'*)

Calculate the correlation function of the two variables given their fourier transforms.

Parameters

- **`ft_a`** – first variable fourier transform
- **`ft_b`** – second variable fourier transform

- **zero_mean** (*bool*) – If `True`, the value corresponding to the zero frequency is set to zero (only fluctuations around means of `a` and `b` are calculated).
- **normalization** (*str*) – Can be `'whole'` (correlations are normalized by product of PSDs derived from *ft_a* and *ft_b*) or `'individual'` (normalization is done for each frequency individually, so that the absolute value is always 1).

pylablib.core.dataproc.iir_transform module

Digital recursive infinite impulse response filter.

Implemented using Numba library (JIT high-performance compilation) if possible.

`pylablib.core.dataproc.iir_transform.`**`iir_apply_complex`**(*trace*, *xcoeff*, *ycoeff*)
Apply digital, (possibly) recursive filter with coefficients *xcoeff* and *ycoeff* along the first axis.

Result is filtered signal *y* with `y[n]=sum_j x[n-j]*xcoeff[j] + sum_k y[n-k-1]*ycoeff[k]`.

pylablib.core.dataproc.image module

`pylablib.core.dataproc.image.`**`convert_shape_indexing`**(*shape*, *src*, *dst*)
Convert image indexing style.

shape is the source image shape (2-tuple), *src* and *dst* are current format and desired format. Formats can be `"rcb"` (first index is row, second is column, rows count from the bottom), `"rct"` (same, but rows count from the top). `"xyb"` (first index is column, second is row, rows count from the bottom), or `"xyt"` (same but rows count form the top). `"rc"` is interpreted as `"rct"`, `"xy"` as `"xyt"`

`pylablib.core.dataproc.image.`**`convert_image_indexing`**(*img*, *src*, *dst*, *axes=(0, 1)*)
Convert image indexing style.

img is the source image (ND numpy array with N>=2), *src* and *dst* are current format and desired format, *axes* specify correspondingly the row and the column axes (by default, the first two array axes). Formats can be `"rcb"` (first index is row, second is column, rows count from the bottom), `"rct"` (same, but rows count from the top). `"xyb"` (first index is column, second is row, rows count from the bottom), or `"xyt"` (same but rows count form the top). `"rc"` is interpreted as `"rct"`, `"xy"` as `"xyt"`

`class` `pylablib.core.dataproc.image.`**`ROI`**(*imin=0*, *imax=None*, *jmin=0*, *jmax=None*)
Bases: `object`

`copy`()

`center`(*shape=None*)

`size`(*shape=None*)

`area`(*shape=None*)

`tup`(*shape=None*)

`ispan`(*shape=None*)

`jspan`(*shape=None*)

`classmethod from_centersize`(*center*, *size*, *shape=None*)

`classmethod intersect`(**args*)

`limit`(*shape*)

`pylablib.core.dataproc.image.`**`get_region`**(*image*, *center*, *size*, *axis=(-2, -1)*)

Get part of the image with the given center and size (both are tuples `(i, j)`).

The region is automatically reduced if a part of it is outside of the image.

`pylablib.core.dataproc.image.`**`get_region_sum`**(*image*, *center*, *size*, *axis=(-2, -1)*)

Sum part of the image with the given center and size (both are tuples `(i, j)`).

The region is automatically reduced if a part of it is outside of the image. Return tuple `(sum, area)`, where area is the actual summer region are (in pixels).

pylablib.core.dataproc.interpolate module

`pylablib.core.dataproc.interpolate.`**`interpolate1D_func`**(*x*, *y*, *kind='linear'*, *axis=-1*, *copy=True*, *bounds_error=True*, *fill_values=nan*, *assume_sorted=False*)

1D interpolation.

Simply a wrapper around `scipy.interpolate.interp1d`.

Parameters

- **`x`** – 1D arrays of x coordinates for the points at which to find the values.
- **`y`** – array of values corresponding to x points (can have more than 1 dimension, in which case the output values are (N-1)-dimensional)
- **`kind`** – Interpolation method.
- **`axis`** – axis in y-data over which to interpolate.
- **`copy`** – if `True`, make internal copies of *x* and *y*.
- **`bounds_error`** – if `True`, raise error if interpolation function arguments are outside of *x* bounds.
- **`fill_values`** – values to fill the outside-bounds regions if `bounds_error==False`.
- **`assume_sorted`** – if `True`, assume that *data* is sorted.

Returns A 1D array with interpolated data.

`pylablib.core.dataproc.interpolate.`**`interpolate1D`**(*data*, *x*, *kind='linear'*, *bounds_error=True*, *fill_values=nan*, *assume_sorted=False*)

1D interpolation.

Parameters

- **`data`** – 2-column array [(x,y)], where `y` is a function of `x`.
- **`x`** – Arrays of x coordinates for the points at which to find the values.
- **`kind`** – Interpolation method.
- **`bounds_error`** – if `True`, raise error if *x* values are outside of *data* bounds.
- **`fill_values`** – values to fill the outside-bounds regions if `bounds_error==False`
- **`assume_sorted`** – if `True`, assume that *data* is sorted.

Returns A 1D array with interpolated data.

`pylablib.core.dataproc.interpolate.`**`interpolate2D`**`(`*data*, *x*, *y*, *method='linear'*, *fill_value=nan*`)`

Interpolate data in 2D.

Simply a wrapper around `scipy.interpolate.griddata()`.

Parameters

- **`data`** – 3-column array [(x,y,z)], where `z` is a function of `x` and `y`.
- **`x/y`** – Arrays of x and y coordinates for the points at which to find the values.
- **`method`** – Interpolation method.

Returns A 2D array with interpolated data.

`pylablib.core.dataproc.interpolate.`**`interpolateND`**`(`*data*, *xs*, *method='linear'*`)`

Interpolate data in N dimensions.

Simply a wrapper around `scipy.interpolate.griddata()`.

Parameters

- **`data`** – `(N+1)`-column array `[(x_1,..,x_N,y)]`, where `y` is a function of `x_1, ... ,x_N`.
- **`xs`** – `N`-tuple of arrays of coordinates for the points at which to find the values.
- **`method`** – Interpolation method.

Returns An ND array with interpolated data.

`pylablib.core.dataproc.interpolate.`**`regular_grid_from_scatter`**`(`*data*, *x_points*, *y_points*, *x_range=None*, *y_range=None*, *method='nearest'*`)`

Turn irregular scatter-points data into a regular 2D grid function.

Parameters

- **`data`** – 3-column array `[(x,y,z)]`, where `z` is a function of `x` and `y`.
- **`x_points/y_points`** – Number of points along x/y axes.
- **`x_range/y_range`** – If not `None`, a tuple specifying the desired range of the data (all points in *data* outside the range are excluded).
- **`method`** – Interpolation method (see `scipy.interpolate.griddata()` for options).

Returns A nested tuple `(data, (x_grid, y_grid))`, where all entries are 2D arrays (either with data or with gridpoint locations).

`pylablib.core.dataproc.interpolate.`**`interpolate_trace`**`(`*trace*, *step*, *rng=None*, *x_column=0*, *select_columns=None*, *kind='linear'*, *assume_sorted=False*`)`

Interpolate trace data over a regular grid with the given step.

rng specifies interpolation range (by default, whole data range). *x_column* specifies column index for x-data. *select_column* specifies which columns to interpolate and keep at the output (by default, all data). If `assume_sorted==True`, assume that x-data is sorted. *kind* specifies interpolation method.

`pylablib.core.dataproc.interpolate.`**`average_interpolate_1D`**(*data*, *step*, *rng=None*, *avg_kernel=1*, *min_weight=0*, *kind='linear'*)

1D interpolation combined with pre-averaging.

Parameters

- **data** – 2-column array [(x,y)], where `y` is a function of `x`.
- **step** – distance between the points in the interpolated data (all resulting x-coordinates are multiples of *step*).
- **rng** – if not `None`, specifies interpolation range (by default, whole data range).
- **avg_kernel** – kernel used for initial averaging. Can be either a 1D array, where each point corresponds to the relative bin weight, or an integer, which specifies simple rectangular kernel of the given width.
- **min_weight** – minimal accumulated weight in the bin to consider it 'valid' (if the bin is invalid, its accumulated value is ignored, and its value is obtained by the interpolation step). *min_weight* of 0 implies any non-zero weight; otherwise, weight `>=min_weight`.
- **kind** – Interpolation method.

Returns A 2-column array with the interpolated data.

pylablib.core.dataproc.specfunc module

Specific useful functions.

`pylablib.core.dataproc.specfunc.`**`gaussian_k`**(*x*, *sigma=1.0*, *height=None*)

Gaussian kernel function.

Normalized by the area if *height* is `None`, otherwise *height* is the value at 0.

`pylablib.core.dataproc.specfunc.`**`rectangle_k`**(*x*, *width=1.0*, *height=None*)

" Symmetric rectangle kernel function.

Normalized by the area if *height* is `None`, otherwise *height* is the value at 0.

`pylablib.core.dataproc.specfunc.`**`lorentzian_k`**(*x*, *gamma=1.0*, *height=None*)

Lorentzian kernel function

Normalized by the area if *height* is `None`, otherwise *height* is the value at 0.

`pylablib.core.dataproc.specfunc.`**`complex_lorentzian_k`**(*x*, *gamma=1.0*, *amplitude=1j*)

Complex Lorentzian kernel function.

`pylablib.core.dataproc.specfunc.`**`exp_decay_k`**(*x*, *width=1.0*, *height=None*, *mode='causal'*)

Exponential decay kernel function

Normalized by area if `height=None` (if possible), otherwise *height* is the value at 0.

Mode determines value for `x<0`:

- `'causal'` - it's 0 for `x<0`;
- `'step'` - it's constant for `x<=0`;
- `'continue'` - it's a continuous decaying exponent;
- `'mirror'` - function is symmetric: `exp(-|x|/width)`.

`pylablib.core.dataproc.specfunc.`**`get_kernel_func`**(*kernel*)
Get a kernel function by its name.

Available functions are: `'gaussian'`, `'rectangle'`, `'lorentzian'`, `'exp_decay'`, `'complex_lorentzian'`.

`pylablib.core.dataproc.specfunc.`**`rectangle_w`**(*x*, *N*, *ft_compensated=False*)
Rectangle FT window function

`pylablib.core.dataproc.specfunc.`**`gen_hamming_w`**(*x*, *N*, *alpha*, *beta*, *ft_compensated=False*)
Generalized Hamming FT window function.

If `ft_compensated==True`, multiply the window function by a compensating factor to preserve power in the spectrum.

`pylablib.core.dataproc.specfunc.`**`hann_w`**(*x*, *N*, *ft_compensated=False*)
Hann FT window function.

If `ft_compensated==True`, multiply the window function by a compensating factor to preserve power in the spectrum.

`pylablib.core.dataproc.specfunc.`**`hamming_w`**(*x*, *N*, *ft_compensated=False*)
Specific Hamming FT window function.

If `ft_compensated==True`, multiply the window function by a compensating factor to preserve power in the spectrum.

`pylablib.core.dataproc.specfunc.`**`get_window_func`**(*window*)
Get a window function by its name.

Available functions are: `'hamming'`, `'rectangle'`, `'hann'`.

`pylablib.core.dataproc.specfunc.`**`gen_hamming_w_ft`**(*f*, *t*, *alpha*, *beta*)
Get Fourier Transform of a generalized Hamming FT window function.

f is the argument, *t* is the total window size.

`pylablib.core.dataproc.specfunc.`**`rectangle_w_ft`**(*f*, *t*)
Get Fourier Transform of the rectangle FT window function.

f is the argument, *t* is the total window size.

`pylablib.core.dataproc.specfunc.`**`hann_w_ft`**(*f*, *t*)
Get Fourier Transform of the Hann FT window function.

f is the argument, *t* is the total window size.

`pylablib.core.dataproc.specfunc.`**`hamming_w_ft`**(*f*, *t*)
Get Fourier Transform of the specific Hamming FT window function.

f is the argument, *t* is the total window size.

`pylablib.core.dataproc.specfunc.`**`get_window_ft_func`**(*window*)
Get a Fourier Transform of a window function by its name.

Available functions are: `'hamming'`, `'rectangle'`, `'hann'`.

pylablib.core.dataproc.table_wrap module

Utilities for uniform treatment of pandas tables and numpy arrays for functions which can deal with them both.

`class` `pylablib.core.dataproc.table_wrap.`**`IGenWrapper`**(*container*)
Bases: `object`

The interface for a wrapper that gives a uniform access to basic methods of wrapped objects'.

get_type()
Get a string representing the wrapped object type

copy(*wrapped=False*)
Copy the object.

If `wrapped==True`, return a new wrapper containing the object copy; otherwise, just return the copy.

ndim()

shape()

class pylablib.core.dataproc.table_wrap.**I1DWrapper**(*container*)
Bases: *pylablib.core.dataproc.table_wrap.IGenWrapper*

A wrapper containing a 1D object (a 1D numpy array or a pandas Series object).

Provides a uniform access to basic methods of a wrapped object.

class Accessor(*wrapper*)
Bases: `object`

An accessor: creates a simple uniform interface to treat the wrapped object element-wise (get/set/iterate over elements).

Generated automatically for each table on creation, doesn't need to be created explicitly.

subcolumn(*idx*, *wrapped=False*)
Return a subcolumn at index *idx*.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

static from_array(*array*, *index=None*, *force_copy=False*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *array* (a 1D numpy array or a list).

If `force_copy==True`, make a copy of supplied array. If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

classmethod from_columns(*columns*, *column_names=None*, *index=None*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *columns* (a list of columns; only length-1 lists is supported).

column_names parameter is ignored. If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

array_replaced(*array*, *force_copy=False*, *preserve_index=False*, *wrapped=False*)
Return a copy of the column with the data replaced by *array*.

All of the parameters are the same as in *from_array()*.

get_index()
Get index of the given 1D trace, or `None` if none is available

get_type()
Get a string representing the wrapped object type

copy(*wrapped=False*)
Copy the object.

If `wrapped==True`, return a new wrapper containing the object copy; otherwise, just return the copy.

ndim()

shape()

class pylablib.core.dataproc.table_wrap.**Array1DWrapper**(*container*)

Bases: `pylablib.core.dataproc.table_wrap.I1DWrapper`

A wrapper for a 1D numpy array.

Provides a uniform access to basic methods of a wrapped object.

get_deleted(*idx*, *wrapped=False*)

Return a copy of the column with the data at index *idx* deleted.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

get_inserted(*idx*, *val*, *wrapped=False*)

Return a copy of the column with the data *val* added at index *idx*.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

insert(*idx*, *val*)

Add data *val* to index *idx*

get_appended(*val*, *wrapped=False*)

Return a copy of the column with the data *val* appended at the end.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

append(*val*)

Append data *val* to the end

subcolumn(*idx*, *wrapped=False*)

Return a subcolumn at index *idx*.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

static from_array(*array*, *index=None*, *force_copy=False*, *wrapped=False*)

Build a new object of the type corresponding to the wrapper from the supplied *array* (a 1D numpy array or a list).

If `force_copy==True`, make a copy of supplied array. If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

get_type()

Get a string representing the wrapped object type

copy(*wrapped=False*)

Copy the object.

If `wrapped==True`, return a new wrapper containing the object copy; otherwise, just return the copy.

class Accessor(*wrapper*)

Bases: `object`

An accessor: creates a simple uniform interface to treat the wrapped object element-wise (get/set/iterate over elements).

Generated automatically for each table on creation, doesn't need to be created explicitly.

array_replaced(*array*, *force_copy=False*, *preserve_index=False*, *wrapped=False*)

Return a copy of the column with the data replaced by *array*.

All of the parameters are the same as in `from_array()`.

classmethod from_columns(*columns*, *column_names=None*, *index=None*, *wrapped=False*)

Build a new object of the type corresponding to the wrapper from the supplied *columns* (a list of columns; only length-1 lists is supported).

column_names parameter is ignored. If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

get_index()
Get index of the given 1D trace, or `None` if none is available

ndim()

shape()

class pylablib.core.dataproc.table_wrap.**Series1DWrapper**(*container*)
Bases: *pylablib.core.dataproc.table_wrap.I1DWrapper*

A wrapper for a pandas Series object.

Provides a uniform access to basic methods of a wrapped object.

get_deleted(*idx*, *wrapped=False*)
Return a copy of the column with the data at index *idx* deleted.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

get_inserted(*idx*, *val*, *wrapped=False*)
Return a copy of the column with the data *val* added at index *idx*.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

get_appended(*val*, *wrapped=False*)
Return a copy of the column with the data *val* appended at the end.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

subcolumn(*idx*, *wrapped=False*)
Return a subcolumn at index *idx*.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

static from_array(*array*, *index=None*, *force_copy=False*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *array* (a 1D numpy array or a list).

If `force_copy==True`, make a copy of supplied array. If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

get_index()
Get index of the given 1D trace, or `None` if none is available

get_type()
Get a string representing the wrapped object type

copy(*wrapped=False*)
Copy the object.

If `wrapped==True`, return a new wrapper containing the object copy; otherwise, just return the copy.

class Accessor(*wrapper*)
Bases: `object`

An accessor: creates a simple uniform interface to treat the wrapped object element-wise (get/set/iterate over elements).

Generated automatically for each table on creation, doesn't need to be created explicitly.

array_replaced(*array*, *force_copy=False*, *preserve_index=False*, *wrapped=False*)
Return a copy of the column with the data replaced by *array*.

All of the parameters are the same as in *from_array()*.

classmethod from_columns(*columns*, *column_names=None*, *index=None*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *columns* (a list of columns; only length-1 lists is supported).

column_names parameter is ignored. If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

ndim()

shape()

class pylablib.core.dataproc.table_wrap.**I2DWrapper**(*container*, *r=None*, *c=None*, *t=None*)
Bases: *pylablib.core.dataproc.table_wrap.IGenWrapper*

A wrapper containing a 2D object (a 2D numpy array or a pandas DataFrame object).

Provides a uniform access to basic methods of a wrapped object.

classmethod from_columns(*columns*, *column_names=None*, *index=None*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *columns* (a list of columns).

column_names supplies names of the columns (only relevant for *DataFrame2DWrapper*). If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

columns_replaced(*columns*, *preserve_index=False*, *wrapped=False*)
Return copy of the object with the data replaced by *columns*.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

static from_array(*array*, *column_names=None*, *index=None*, *force_copy=False*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *array* (a list of rows or a 2D numpy array).

column_names supplies names of the columns (only relevant for *DataFrame2DWrapper*). If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

array_replaced(*array*, *preserve_index=None*, *force_copy=False*, *wrapped=False*)
Return a copy of the column with the data replaced by *array*.

All of the parameters are the same as in *from_array()*.

get_index()
Get index of the given 2D table, or `None` if none is available

get_type()
Get a string representing the wrapped object type

copy(*wrapped=False*)
Copy the object.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

column(*idx*, *wrapped=False*)
Get a column at index *idx*.

Return a 1D numpy array for a 2D numpy array object, and an Series object for a pandas DataFrame. If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

subtable(*idx*, *wrapped=False*)
Return a subtable at index *idx*.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

ndim()

shape()

class pylablib.core.dataproc.table_wrap.**Array2DWrapper**(*container*)

Bases: *pylablib.core.dataproc.table_wrap.I2DWrapper*

A wrapper for a 2D numpy array.

Provides a uniform access to basic methods of a wrapped object.

set_container(*cont*)

class RowAccessor(*wrapper*, *storage*)

Bases: object

A row accessor: creates a simple uniform interface to treat the wrapped object row-wise (append/insert/delete/iterate over rows).

Generated automatically for each table on creation, doesn't need to be created explicitly.

get_deleted(*idx*, *wrapped=False*)

Return a new table with the rows at *idx* deleted.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

get_inserted(*idx*, *val*, *wrapped=False*)

Return a new table with new rows given by *val* inserted at *idx*.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

insert(*idx*, *val*)

Insert new rows given by *val* at index *idx*.

get_appended(*val*, *wrapped=False*)

Return a new table with new rows given by *val* appended to the end of the table.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

append(*val*)

Insert new rows given by *val* to the end of the table

class ColumnAccessor(*wrapper*, *storage*)

Bases: object

A column accessor: creates a simple uniform interface to treat the wrapped object column-wise (append/insert/delete/iterate over columns).

Generated automatically for each table on creation, doesn't need to be created explicitly.

get_deleted(*idx*, *wrapped=False*)

Return a new table with the columns at *idx* deleted.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

get_inserted(*idx*, *val*, *wrapped=False*)

Return a new table with new columns given by *val* inserted at *idx*.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

insert(*idx*, *val*)

Insert new columns given by *val* at index *idx*.

get_appended(*val*, *wrapped=False*)

Return a new table with new columns given by *val* appended to the end of the table.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

append(*val*)
Insert new columns given by *val* to the end of the table

set_names(*names*)
Set column names (does nothing)

get_names()
Get column names (all names are `None`)

get_column_index(*idx*)
Get number index for a given column index

class TableAccessor(*storage*)
Bases: `object`

A table accessor: accessing the table data through this interface returns an object of the appropriate type (numpy array for numpy wrapped object, and a DataFrame for a pandas DataFrame wrapped object).

Generated automatically for each table on creation, doesn't need to be created explicitly.

subtable(*idx*, *wrapped=False*)
Return a subtable at index *idx* of the appropriate type (2D numpy array).

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

column(*idx*, *wrapped=False*)
Get a column at index *idx* as a 1D numpy array.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

classmethod from_columns(*columns*, *column_names=None*, *index=None*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *columns* (a list of columns).

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table. *column_names* parameter is ignored.

static from_array(*array*, *column_names=None*, *index=None*, *force_copy=False*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *array* (a list of rows or a 2D numpy array).

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table. *column_names* parameter is ignored.

get_type()
Get a string representing the wrapped object type

copy(*wrapped=False*)
Copy the object.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

array_replaced(*array*, *preserve_index=None*, *force_copy=False*, *wrapped=False*)
Return a copy of the column with the data replaced by *array*.

All of the parameters are the same as in `from_array()`.

columns_replaced(*columns*, *preserve_index=False*, *wrapped=False*)
Return copy of the object with the data replaced by *columns*.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

get_index()
Get index of the given 2D table, or `None` if none is available

ndim()

shape()

class pylablib.core.dataproc.table_wrap.**DataFrame2DWrapper**(*container*)

Bases: *pylablib.core.dataproc.table_wrap.I2DWrapper*

A wrapper for a pandas DataFrame object.

Provides a uniform access to basic methods of a wrapped object.

class RowAccessor(*wrapper*, *storage*)

Bases: object

A row accessor: creates a simple uniform interface to treat the wrapped object row-wise (append/insert/delete/iterate over rows).

Generated automatically for each table on creation, doesn't need to be created explicitly.

get_deleted(*idx*, *wrapped=False*)

Return a copy of the column with the data at index *idx* deleted.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

get_inserted(*idx*, *val*, *wrapped=False*)

Return a new table with new rows given by *val* inserted at *idx*.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

insert(*idx*, *val*)

Insert new rows given by *val* at index *idx*.

get_appended(*val*, *wrapped=False*)

Return a new table with new rows given by *val* appended to the end of the table.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

append(*val*)

Insert new rows given by *val* to the end of the table

class ColumnAccessor(*wrapper*, *storage*)

Bases: object

A column accessor: creates a simple uniform interface to treat the wrapped object column-wise (append/insert/delete/iterate over columns).

Generated automatically for each table on creation, doesn't need to be created explicitly.

get_deleted(*idx*, *wrapped=False*)

Return a new table with the columns at *idx* deleted.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

get_inserted(*idx*, *val*, *column_name=None*, *wrapped=False*)

Return a new table with new columns given by *val* inserted at *idx*.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

insert(*idx*, *val*, *column_name=None*)

Insert new columns given by *val* at index *idx*

get_appended(*val*, *column_name=None*, *wrapped=False*)

Return a new table with new columns given by *val* appended to the end of the table.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

append(*val*, *column_name=None*)
Insert new columns given by *val* to the end of the table

set_names(*names*)
Set column names

get_names()
Get column names

get_column_index(*idx*)
Get number index for a given column index

class TableAccessor(*storage*)
Bases: `object`

A table accessor: accessing the table data through this interface returns an object of the appropriate type (numpy array for numpy wrapped object, and a DataFrame for a pandas DataFrame wrapped object).

Generated automatically for each table on creation, doesn't need to be created explicitly.

subtable(*idx*, *wrapped=False*)
Return a subtable at index *idx* of the appropriate type (pandas DataFrame).

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

column(*idx*, *wrapped=False*)
Get a column at index *idx* as a pandas Series object.

If `wrapped==True`, return a new wrapper containing the column; otherwise, just return the column.

classmethod from_columns(*columns*, *column_names=None*, *index=None*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *columns* (a list of columns).

column_names supplies names of the columns (only relevant for `DataFrame2DWrapper`). If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

static from_array(*array*, *column_names=None*, *index=None*, *force_copy=False*, *wrapped=False*)
Build a new object of the type corresponding to the wrapper from the supplied *array* (a list of rows or a 2D numpy array).

column_names supplies names of the columns (only relevant for `DataFrame2DWrapper`). If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

get_index()
Get index of the given 2D table, or `None` if none is available

get_type()
Get a string representing the wrapped object type

copy(*wrapped=False*)
Copy the object. If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table

array_replaced(*array*, *preserve_index=None*, *force_copy=False*, *wrapped=False*)
Return a copy of the column with the data replaced by *array*.

All of the parameters are the same as in `from_array()`.

columns_replaced(*columns*, *preserve_index=False*, *wrapped=False*)
Return copy of the object with the data replaced by *columns*.

If `wrapped==True`, return a new wrapper containing the table; otherwise, just return the table.

ndim()

shape()

pylablib.core.dataproc.table_wrap.**wrap1d**(*container*)
Wrap a 1D container (a 1D numpy array or or a pandas Series) into an appropriate wrapper

pylablib.core.dataproc.table_wrap.**wrap2d**(*container*)
Wrap a 2D container (a 2D numpy array or a pandas DataFrame) into an appropriate wrapper

pylablib.core.dataproc.table_wrap.**wrap**(*container*)
Wrap container (a numpy array, a pandas Series or a pandas DataFrame) into an appropriate wrapper

pylablib.core.dataproc.utils module

Generic utilities for dealing with numerical arrays.

pylablib.core.dataproc.utils.**is_ascending**(*trace*)
Check the if the trace is ascending.

If it has more than 1 dimension, check all lines along 0'th axis.

pylablib.core.dataproc.utils.**is_descending**(*trace*)
Check if the trace is descending.

If it has more than 1 dimension, check all lines along 0'th axis.

pylablib.core.dataproc.utils.**is_ordered**(*trace*)
Check if the trace is ordered (ascending or descending).

If it has more than 1 dimension, check all lines along 0'th axis.

pylablib.core.dataproc.utils.**is_linear**(*trace*)
Check if the trace is linear (values go with a constant step).

If it has more than 1 dimension, check all lines along 0'th axis (with the same step for all).

pylablib.core.dataproc.utils.**get_x_column**(*t*, *x_column=None*, *idx_default=False*)
Get x column of the table.

x_column **can be**

- an array: return as is;
- `'#'`: return index array;
- `None`: equivalent to '#' for 1D data if `idx_default==False`, or to `0` otherwise;
- integer: return the column with this index.

pylablib.core.dataproc.utils.**get_y_column**(*t*, *y_column=None*)
Get y column of the table.

y_column **can be**

- an array: return as is;
- `'#'`: return index array;
- `None`: return *t* for 1D data, or the column `1` otherwise;
- integer: return the column with this index.

pylablib.core.dataproc.utils.**sort_by**(*t*, *x_column=None*, *reverse=False*, *stable=False*)
Sort a table using selected column as a key and preserving rows.

If `reverse==True`, sort in descending order. *x_column* values are described in `get_x_column()`. If `stable==True`, use stable sort (could be slower and uses more memory, but preserves the order of elements for the same key)

`pylablib.core.dataproc.utils.`**`filter_by`**(*t*, *columns=None*, *pred=None*, *exclude=False*)
Filter 1D or 2D array using a predicate.

If the data is 2D, *columns* contains indices of columns to be passed to the *pred* function. If `exclude==False`, drop all of the rows satisfying *pred* rather than keep them.

`pylablib.core.dataproc.utils.`**`unique_slices`**(*t*, *u_column*)
Split a table into subtables with different values in a given column.

Return a list of *t* subtables, each of which has a different (and equal among all rows in the subtable) value in *u_column*.

`pylablib.core.dataproc.utils.`**`merge`**(*ts*, *idx=None*, *as_array=True*)
Merge several tables column-wise.

If *idx* is not `None`, then it is a list of index columns (one column per table) used for merging. The rows that have the same value in the index columns are merged; if some values aren't contained in all the *ts*, the corresponding rows are omitted. If *idx* is `None`, just join the tables together (they must have the same number of rows).

If `as_array==True`, return a simple numpy array as a result; otherwise, return a pandas DataFrame if applicable (note that in this case all column names in all tables must be different to avoid conflicts)

class `pylablib.core.dataproc.utils.`**`Range`**(*start=None*, *stop=None*)
Bases: `object`

Single data range.

If *start* or *stop* are `None`, it's implied that they're at infinity (i.e., Range(None,None) is infinite). If the range object is `None`, it's implied that the range is empty

`start`

`stop`

`contains`(*x*)
Check if *x* is in the range

`intersect`(**rngs*)
Find an intersection of multiple ranges.

If the intersection is empty, return `None`.

`rescale`(*mult=1.0*, *shift=0.0*)

`tup`()

`pylablib.core.dataproc.utils.`**`find_closest_arg`**(*xs*, *x*, *approach='both'*, *ordered=False*)
Find the index of a value in *xs* that is closest to *x*.

approach can take values `'top'`, `'bottom'` or `'both'` and denotes from which side should array elements approach *x* (meaning that the found array element should be `>x`, `<x` or just the closest one). If there are no elements lying on the desired side of *x* (e.g. `approach=='top'` and all elements of *xs* are less than *x*), the function returns `None`. if `ordered==True`, then *xs* is assumed to be in ascending or descending order, and binary search is implemented (works only for 1D arrays). if there are recurring elements, return any of them.

`pylablib.core.dataproc.utils.`**`find_closest_value`**(*xs*, *x*, *approach='both'*, *ordered=False*)

`pylablib.core.dataproc.utils.`**`get_range_indices`**(*xs*, *xs_range*, *ordered=False*)
Find trace indices corresponding to the given range.

The range is defined as `xs_range[0]:xs_range[1]`, or infinite if `xs_range=None` (so the data is returned unchanged in that case). If `ordered==True`, then the function assumes that *xs* in ascending or descending order.

`pylablib.core.dataproc.utils.`**`cut_to_range`**(*t*, *xs_range*, *x_column=None*, *ordered=False*)

Cut the table to the given range based on *x_column*.

The range is defined as `xs_range[0]:xs_range[1]`, or infinite if `xs_range=None`. *x_column* is used to determine which column's values to use to check if the point is in range (see `get_x_column()`). If `ordered_x==True`, then the function assumes that *x_column* in ascending order.

`pylablib.core.dataproc.utils.`**`cut_out_regions`**(*t*, *regions*, *x_column=None*, *ordered=False*, *multi_pass=True*)

Cut the regions out of the *t* based on *x_column*.

x_column is used to determine which column's values to use to check if the point is in range (see `get_x_column()`). If `ordered_x==True`, then the function assumes that *x_column* in ascending order. If `multi_pass==False`, combine all indices before deleting the data in a single operation (works faster, but only for non-intersecting regions).

`pylablib.core.dataproc.utils.`**`find_discrete_step`**(*trace*, *min_fraction=1e-08*, *tolerance=1e-05*)

Try to find a minimal divisor of all steps in a 1D trace.

min_fraction is the minimal possible size of the divisor (relative to the minimal non-zero step size). *tolerance* is the tolerance of the division. Raise an `ArithmeticError` if no such value was found.

`pylablib.core.dataproc.utils.`**`unwrap_mod_data`**(*trace*, *wrap_range*)

Unwrap data given *wrap_range*.

Assume that every jump greater than `0.5*wrap_range` is not real and is due to value being restricted. Can be used to, e.g., unwrap the phase data.

`pylablib.core.dataproc.utils.`**`pad_trace`**(*trace*, *pad*, *mode='constant'*, *cval=0.0*)

Expand 1D trace or a multi-column table for different convolution techniques.

Wrapper around `numpy.pad()`, but can handle pandas dataframes or multi-column arrays. Note that the index data is not preserved.

Parameters

- **`trace`** – 1D array-like object.
- **`pad`** (*`int or tuple`*) – Expansion size. Can be an integer, if pad on both sides is equal, or a 2-tuple `(left, right)` for pads on opposite sides.
- **`mode`** (*`str`*) – Expansion mode. Takes the same values as `numpy.pad()`. Common values are `'constant'` (added values are determined by *cval*), `'edge'` (added values are end values of the trace), `'reflect'` (reflect trace with respect to its endpoint) or `'wrap'` (wrap the values from the other size).
- **`cval`** (*`float`*) – If `mode=='constant'`, determines the expanded values.

`pylablib.core.dataproc.utils.`**`xy2c`**(*t*)

Convert a trace or a table from xy representation to a single complex data.

t is a 2D array with either 2 columns (x and y) or 3 columns (index, x and y). Return 2D array with either 1 column (c) or 2 columns (index and c).

`pylablib.core.dataproc.utils.`**`c2xy`**(*t*)

Convert the a trace or a table from complex representation to a split x and y data.

t is either 1D array (c data) or a 2D array with either 1 column (c) or 2 columns (index and c). Return 2D array with either 2 column (x and y) or 3 columns (index, x and y).

Module contents

pylablib.core.devio package

Submodules

pylablib.core.devio.SCPI module

class pylablib.core.devio.SCPI.**SCPIDevice**(*conn*, *term_write=None*, *term_read=None*, *wait_callback=None*, *backend='auto'*, *backend_defaults=None*, *failsafe=None*, *timeout=None*, *backend_params=None*)

Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*

A base class for a device controlled with the usual SCPI syntax.

Implements two functions:

- deals with composing and parsing of standard SCPI commands and simplifying repetitive property access routines
- implements automatic re-sending and reconnecting on communication failures (fail-safe mode)

Parameters

- **conn** – Connection parameters (depend on the backend). Can also be an opened *comm_backend.IDeviceCommBackend* class for a custom backend.
- **term_write** (*str*) – Line terminator for writing operations.
- **wait_callback** (*callable*) – A function to be called periodically (every 300ms by default) while waiting for operations to complete.
- **backend** (*str*) – Connection backend (e.g., `'serial'` or `'visa'`).
- **backend_defaults** – if not `None`, specifies a dictionary `{backend: params}` with default connection parameters (depending on the backend), which are added to *conn*
- **failsafe** (*bool*) – If `True`, the device is working in a fail-safe mode: if an operation times out, attempt to repeat it several times before raising error. If `None`, use the class value *_default_failsafe* (`False` by default).
- **timeout** (*float*) – Default timeout (in seconds).

Error
alias of *pylablib.core.devio.base.DeviceError*

ReraiseError = None

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

sleep(*delay*)
Wait for *delay* seconds

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive *write()* operations are bundled together with ; delimiter. The actual write is performed at the *read()*/*ask()* operation or at the end of the block.

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

reset()
Reset the device (by default, "*RST" command)

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either 'sync' (perform *wait_sync()*), 'dev' (perform *wait_dev()*) or 'none' (do nothing).

static get_arg_type(*arg*)
Autodetect argument type

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with ",".
- **arg_type** (*str*) – Argument type. Can be 'raw' (in which case data is sent raw), 'string', 'int', 'float', 'bool', a format string (such as '{:.3f}') or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., arg_type='{0};{1}' with arg=[1,2] will produce a string '1;2'); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with ",".
- **unit** (*str*) – If not None, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple (false_value, true_value) of two strings to represent bool argument; by default, use ._bool_selector attribute.
- **wait_sync** – if True, append the sync command (specified as ._wait_sync_comm attribute, "*OPC?" by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if None, use the class default ._default_write_sync attribute (False by default).

- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in `read()`. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

apply_settings(*settings*)
Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

pylablib.core.devio.backend_logger module

class pylablib.core.devio.backend_logger.**BackendLogger**(*path*)
Bases: `object`

Backend logger.

Receives log requests from backends and stores them in a predefined file.

Parameters path – path to save the log

start(*header*)
Start logging section

stop()
Stop logging section

section(*header*)
Context manager for operations within a header

log(*operation*, *value*)
Log the operation

pylablib.core.devio.backend_logger.**load_logfile**(*path*)
Load backend log file.

Return a list of tuples `[(header, section)]`, where `header` is the header name, and `section` is the list `[(op, value)]` with operations (`"r"`, `"w"`, or `"e"`) nd corresponding values.

pylablib.core.devio.base module

exception `pylablib.core.devio.base.`**`DeviceError`**

Bases: `RuntimeError`

Generic device communication error

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

pylablib.core.devio.comm_backend module

Routines for defining a unified interface across multiple backends.

exception `pylablib.core.devio.comm_backend.`**`DeviceBackendError`**(*exc*)

Bases: *`pylablib.core.devio.base.DeviceError`*

Generic exception relaying a backend error

`args`

`with_traceback`()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

`pylablib.core.devio.comm_backend.`**`reraise`**(*func*)

Wrapper for a backend method which intercepts backend exceptions and re-emits them as a subclass of *`DeviceBackendError`* defined in the class

`pylablib.core.devio.comm_backend.`**`logerror`**(*func*)

Wrapper for a backend method which logs if any errors escaped

class `pylablib.core.devio.comm_backend.`**`IDeviceCommBackend`**(*conn*, *timeout=None*, *term_write=None*, *term_read=None*, *datatype='auto'*, *reraise_error=None*)

Bases: `object`

An abstract class for a device communication backend.

Connection is automatically opened on creation.

Parameters

- **conn** – Connection parameters (depend on the backend).
- **timeout** (*float*) – Default timeout (in seconds).
- **term_write** (*str*) – Line terminator for writing operations.
- **term_read** (*str*) – Line terminator for reading operations.
- **datatype** (*str*) – Type of the returned data; can be `"bytes"` (return `bytes` object), `"str"` (return `str` object), or `"auto"` (default Python result: `str` in Python 2 and `bytes` in Python 3)

- **reraise_error** – if not `None`, specifies an error to be re-raised on any backend exception (by default, use backend-specific error); should be a subclass of *DeviceBackendError*.

BackendError = None
Base class for the errors raised by the backend operations

Error
alias of *DeviceBackendError*

classmethod combine_conn(*conn1*, *conn2*)
Combined two connection parameters into a single dictionary (*conn1* overrides *conn2*)

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

setup_cooldown(***kwargs*)
Setup cooldown times for various operations.

The arguments are of the form `kind=value`, where `value` is the cooldown time (in seconds), and `kind` is the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`). `kind` can also be `default` (default value for all kind), or `all` (reset all cooldown values to this value). The cooldowns of the given kinds are usually called after the corresponding operation (it is necessary for some devices, otherwise the communication can freeze or crush). Default cooldown values are specified by `_default_operation_cooldown` class attribute dictionary.

cooldown(*kind='default'*)
Cooldown between the operations.

`kind` specifies the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`); `"default"` corresponds to the default cooldown (usually, specified as 0). Called automatically by various backend operations, so usually there is no need to call explicitly.

set_timeout(*timeout*)
Set operations timeout (in seconds)

get_timeout()
Get operations timeout (in seconds)

using_timeout(*timeout=None*)
Context manager for usage of a different timeout inside a block

readline(*remove_term=True*, *timeout=None*, *skip_empty=True*)
Read a single line from the device.

Parameters

- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.

- **timeout** – Operation timeout. If `None`, use the default device timeout.
- **skip_empty** (*bool*) – If `True`, ignore empty lines (works only for `remove_term==True`).

readlines(*lines_num*, *remove_term=True*, *timeout=None*, *skip_empty=True*)
Read multiple lines from the device.

Parameters are the same as in `readline()`.

read(*size=None*)
Read data from the device.

If *size* is not None, read *size* bytes (the standard timeout applies); otherwise, read all available data (return immediately).

flush_read()
Flush the device output (read all the available data; return the number of bytes read)

write(*data*, *flush=True*, *read_echo=False*, *read_echo_delay=0*, *read_echo_lines=1*)
Write data to the device.

If `flush==True`, flush the write buffer. If `read_echo==True`, wait for *read_echo_delay* seconds and then perform `readline()` (*read_echo_lines* times).

ask(*query*, *delay=0.0*, *read_all=False*)
Perform a write followed by a read, with *delay* in between.

If `read_all==True`, read all the available data; otherwise, read a single line.

static list_resources(*desc=False*)
List all available resources for this backend.

If `desc==False`, return list of connections (usually strings or tuples), which can be used to connect to the device. Otherwise, return a list of descriptions, which have more info, but can be backend-dependent.

Might not be implemented (depending on the backend), in which case returns `None`.

pylablib.core.devio.comm_backend.**remove_longest_term**(*msg*, *terms*)
Remove the longest terminator among *terms* from the end of the message.

exception pylablib.core.devio.comm_backend.**DeviceVisaError**(*exc*)
Bases: `pylablib.core.devio.comm_backend.DeviceBackendError`

Visa backend operation error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.core.devio.comm_backend.**VisaDeviceBackend**(*conn*, *timeout=10.0*, *term_write=None*, *term_read=None*, *do_lock=None*, *datatype='auto'*, *reraise_error=None*)
Bases: `pylablib.core.devio.comm_backend.IDeviceCommBackend`

NIVisa backend (via pyVISA).

Connection is automatically opened on creation.

Parameters

- **conn** (*str*) – Connection string.

- **timeout** (*float*) – Default timeout (in seconds).
- **term_write** (*str*) – Line terminator for writing operations; appended to the data
- **term_read** (*str*) – Line terminator for reading operations (specifies when *readline()* stops).
- **do_lock** (*bool*) – If `True`, employ locking operations; otherwise, locking function does nothing.
- **datatype** (*str*) – Type of the returned data; can be `"bytes"` (return *bytes* object), `"str"` (return *str* object), or `"auto"` (default Python result: *str* in Python 2 and *bytes* in Python 3)
- **reraise_error** – if not `None`, specifies an error to be re-raised on any backend exception (by default, use backend-specific error); should be a subclass of *DeviceBackendError*.

BackendError
alias of `builtins.object`

Error
alias of *DeviceVisaError*

static list_resources(*desc=False*)
List all available resources for this backend.

If `desc==False`, return list of connections (usually strings or tuples), which can be used to connect to the device. Otherwise, return a list of descriptions, which have more info, but can be backend-dependent.

Might not be implemented (depending on the backend), in which case returns `None`.

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes

unlock()
Unlock the access to the device from other threads/processes

locking(*timeout=None*)
Context manager for lock & unlock

set_timeout(*timeout*)
Set operations timeout (in seconds)

get_timeout()
Get operations timeout (in seconds)

readline(*remove_term=True*, *timeout=None*, *skip_empty=True*)
Read a single line from the device.

Parameters

- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.

- **skip_empty** (*bool*) – If `True`, ignore empty lines (works only for `remove_term==True`).

read(*size=None*)
Read data from the device.

If *size* is not None, read *size* bytes (the standard timeout applies); otherwise, read all available data (return immediately).

write(*data*, *flush=True*, *read_echo=False*, *read_echo_delay=0*, *read_echo_lines=1*)
Write data to the device.

If `flush==True`, flush the write buffer. If `read_echo==True`, wait for *read_echo_delay* seconds and then perform `readline()` (*read_echo_lines* times).

ask(*query*, *delay=0.0*, *read_all=False*)
Perform a write followed by a read, with *delay* in between.

If `read_all==True`, read all the available data; otherwise, read a single line.

classmethod combine_conn(*conn1*, *conn2*)
Combined two connection parameters into a single dictionary (*conn1* overrides *conn2*)

cooldown(*kind='default'*)
Cooldown between the operations.

`kind` specifies the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`); `"default"` corresponds to the default cooldown (usually, specified as 0). Called automatically by various backend operations, so usually there is no need to call explicitly.

flush_read()
Flush the device output (read all the available data; return the number of bytes read)

readlines(*lines_num*, *remove_term=True*, *timeout=None*, *skip_empty=True*)
Read multiple lines from the device.

Parameters are the same as in `readline()`.

setup_cooldown(***kwargs*)
Setup cooldown times for various operations.

The arguments are of the form `kind=value`, where `value` is the cooldown time (in seconds), and `kind` is the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`). `kind` can also be `default` (default value for all kind), or `all` (reset all cooldown values to this value). The cooldowns of the given kinds are usually called after the corresponding operation (it is necessary for some devices, otherwise the communication can freeze or crush). Default cooldown values are specified by `_default_operation_cooldown` class attribute dictionary.

using_timeout(*timeout=None*)
Context manager for usage of a different timeout inside a block

exception `pylablib.core.devio.comm_backend.`**DeviceSerialError**(*exc*)
Bases: `pylablib.core.devio.comm_backend.DeviceBackendError`

Serial backend operation error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.core.devio.comm_backend.**SerialDeviceBackend**(*conn*, *timeout=10.0*, *term_write=None*, *term_read=None*, *connect_on_operation=False*, *open_retry_times=3*, *no_dtrrts=False*, *datatype='auto'*, *reraise_error=None*)

Bases: *pylablib.core.devio.comm_backend.IDeviceCommBackend*

Serial backend (via pySerial).

Connection is automatically opened on creation.

Parameters

- **conn** – Connection parameters. Can be either a string (for a port), or a list/tuple `(port, baudrate, bytesize, parity, stopbits, xonxoff, rtscts, dsrdtr)` supplied to the serial connection (default is `('COM1',19200,8,'N',1,0,0,0)`), or a dict with the same parameters.
- **timeout** (*float*) – Default timeout (in seconds).
- **term_write** (*str*) – Line terminator for writing operations; appended to the data
- **term_read** (*str*) – List of possible single-char terminator for reading operations (specifies when `readline()` stops).
- **connect_on_operation** (*bool*) – If `True`, the connection is normally closed, and is opened only on the operations (normally two processes can't be simultaneously connected to the same device).
- **open_retry_times** (*int*) – Number of times the connection is attempted before giving up.
- **no_dtrrts** (*bool*) – If `True`, turn off DTR and RTS status lines before opening (e.g., turns off reset-on-connection for Arduino controllers).
- **datatype** (*str*) – Type of the returned data; can be `"bytes"` (return *bytes* object), `"str"` (return *str* object), or `"auto"` (default Python result: *str* in Python 2 and *bytes* in Python 3)
- **reraise_error** – if not `None`, specifies an error to be re-raised on any backend exception (by default, use backend-specific error); should be a subclass of `DeviceBackendError`.

BackendError
alias of `builtins.object`

Error
alias of `DeviceSerialError`

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

single_op()
Context manager for a single operation.

If `connect_on_operation==True` during creation, wrapping several command in *single_op* prevents the connection from being closed and reopened between the operations (only opened in the beginning and closed in the end).

set_timeout(*timeout*)
Set operations timeout (in seconds)

get_timeout()
Get operations timeout (in seconds)

readline(*remove_term=True*, *timeout=None*, *skip_empty=True*, *error_on_timeout=True*)
Read a single line from the device.

Parameters

- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.
- **skip_empty** (*bool*) – If `True`, ignore empty lines (works only for `remove_term==True`).
- **error_on_timeout** (*bool*) – If `False`, return an incomplete line instead of raising the error on timeout.

read(*size=None*)
Read data from the device.

If *size* is not None, read *size* bytes (usual timeout applies); otherwise, read all available data (return immediately).

read_multichar_term(*term*, *remove_term=True*, *timeout=None*, *error_on_timeout=True*)
Read a single line with multiple possible terminators.

Parameters

- **term** – Either a string (single multi-char terminator) or a list of strings (multiple terminators).
- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.
- **error_on_timeout** (*bool*) – If `False`, return an incomplete line instead of raising the error on timeout.

write(*data*, *flush=True*, *read_echo=False*, *read_echo_delay=0*, *read_echo_lines=1*)
Write data to the device.

If `flush==True`, flush the write buffer. If `read_echo==True`, wait for *read_echo_delay* seconds and then perform `readline()` (*read_echo_lines* times).

static list_resources(*desc=False*)
List all available resources for this backend.

If `desc==False`, return list of connections (usually strings or tuples), which can be used to connect to the device. Otherwise, return a list of descriptions, which have more info, but can be backend-dependent.

Might not be implemented (depending on the backend), in which case returns `None`.

ask(*query*, *delay=0.0*, *read_all=False*)
Perform a write followed by a read, with *delay* in between.

If read_all==True, read all the available data; otherwise, read a single line.

classmethod combine_conn(*conn1*, *conn2*)
Combined two connection parameters into a single dictionary (*conn1* overrides *conn2*)

cooldown(*kind='default'*)
Cooldown between the operations.

kind specifies the operation kind (common kinds are open, close, read, write, timeout, and flush); "default" corresponds to the default cooldown (usually, specified as 0). Called automatically by various backend operations, so usually there is no need to call explicitly.

flush_read()
Flush the device output (read all the available data; return the number of bytes read)

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

readlines(*lines_num*, *remove_term=True*, *timeout=None*, *skip_empty=True*)
Read multiple lines from the device.

Parameters are the same as in readline().

setup_cooldown(***kwargs*)
Setup cooldown times for various operations.

The arguments are of the form kind=value, where value is the cooldown time (in seconds), and kind is the operation kind (common kinds are open, close, read, write, timeout, and flush). kind can also be default (default value for all kind), or all (reset all cooldown values to this value). The cooldowns of the given kinds are usually called after the corresponding operation (it is necessary for some devices, otherwise the communication can freeze or crush). Default cooldown values are specified by _default_operation_cooldown class attribute dictionary.

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_timeout(*timeout=None*)
Context manager for usage of a different timeout inside a block

exception pylablib.core.devio.comm_backend.**DeviceFT232Error**(*exc*)
Bases: pylablib.core.devio.comm_backend.DeviceBackendError

FT232 backend operation error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.core.devio.comm_backend.**FT232DeviceBackend**(*conn*, *timeout=10.0*, *term_write=None*, *term_read=None*, *open_retry_times=3*, *datatype='auto'*, *reraise_error=None*)
Bases: pylablib.core.devio.comm_backend.IDeviceCommBackend

FT232 backend (via pyft232).

Connection is automatically opened on creation.

Parameters

- **conn** – Connection parameters. Can be either a string (for a port), or a list/tuple `(port, baudrate, bytesize, parity, stopbits, xonxoff, rtscts)` supplied to the serial connection (default is `('COM1',19200,8,'N',1,0,0,0)`), or a dict with the same parameters.
- **timeout** (*float*) – Default timeout (in seconds).
- **term_write** (*str*) – Line terminator for writing operations; appended to the data
- **term_read** (*str*) – List of possible single-char terminator for reading operations (specifies when `readline()` stops).
- **open_retry_times** (*int*) – Number of times the connection is attempted before giving up.
- **datatype** (*str*) – Type of the returned data; can be `"bytes"` (return *bytes* object), `"str"` (return *str* object), or `"auto"` (default Python result: *str* in Python 2 and *bytes* in Python 3)
- **reraise_error** – if not `None`, specifies an error to be re-raised on any backend exception (by default, use backend-specific error); should be a subclass of *DeviceBackendError*.

BackendError
alias of `builtins.object`

Error
alias of *DeviceFT232Error*

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

single_op()
Context manager for a single operation.

Does nothing.

set_timeout(*timeout*)
Set operations timeout (in seconds)

get_timeout()
Get operations timeout (in seconds)

readline(*remove_term=True*, *timeout=None*, *skip_empty=True*, *error_on_timeout=True*)
Read a single line from the device.

Parameters

- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.
- **skip_empty** (*bool*) – If `True`, ignore empty lines (works only for `remove_term==True`).
- **error_on_timeout** (*bool*) – If `False`, return an incomplete line instead of raising the error on timeout.

read(*size=None*)
Read data from the device.

If *size* is not None, read *size* bytes (usual timeout applies); otherwise, read all available data (return immediately).

read_multichar_term(*term*, *remove_term=True*, *timeout=None*, *error_on_timeout=True*)
Read a single line with multiple possible terminators.

Parameters

- **term** – Either a string (single multi-char terminator) or a list of strings (multiple terminators).
- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.
- **error_on_timeout** (*bool*) – If `False`, return an incomplete line instead of raising the error on timeout.

write(*data*, *flush=True*, *read_echo=False*, *read_echo_delay=0*, *read_echo_lines=1*)
Write data to the device.

If `flush==True`, flush the write buffer. If `read_echo==True`, wait for *read_echo_delay* seconds and then perform `readline()` (*read_echo_lines* times).

static list_resources(*desc=False*)
List all available resources for this backend.

If `desc==False`, return list of connections (usually strings or tuples), which can be used to connect to the device. Otherwise, return a list of descriptions, which have more info, but can be backend-dependent.

Might not be implemented (depending on the backend), in which case returns `None`.

ask(*query*, *delay=0.0*, *read_all=False*)
Perform a write followed by a read, with *delay* in between.

If `read_all==True`, read all the available data; otherwise, read a single line.

classmethod combine_conn(*conn1*, *conn2*)
Combined two connection parameters into a single dictionary (*conn1* overrides *conn2*)

cooldown(*kind='default'*)
Cooldown between the operations.

`kind` specifies the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`); `"default"` corresponds to the default cooldown (usually, specified as 0). Called automatically by various backend operations, so usually there is no need to call explicitly.

flush_read()
Flush the device output (read all the available data; return the number of bytes read)

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

readlines(*lines_num*, *remove_term=True*, *timeout=None*, *skip_empty=True*)
Read multiple lines from the device.

Parameters are the same as in `readline()`.

setup_cooldown(***kwargs*)

Setup cooldown times for various operations.

The arguments are of the form `kind=value`, where `value` is the cooldown time (in seconds), and `kind` is the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`). `kind` can also be `default` (default value for all kind), or `all` (reset all cooldown values to this value). The cooldowns of the given kinds are usually called after the corresponding operation (it is necessary for some devices, otherwise the communication can freeze or crush). Default cooldown values are specified by `_default_operation_cooldown` class attribute dictionary.

unlock()

Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_timeout(*timeout=None*)

Context manager for usage of a different timeout inside a block

exception pylablib.core.devio.comm_backend.**DeviceNetworkError**(*exc*)

Bases: *pylablib.core.devio.comm_backend.DeviceBackendError*

Network backend operation error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.core.devio.comm_backend.**NetworkDeviceBackend**(*conn*, *timeout=10.0*, *term_write=None*, *term_read=None*, *datatype='auto'*, *reraise_error=None*)

Bases: *pylablib.core.devio.comm_backend.IDeviceCommBackend*

Serial backend (via pySerial).

Connection is automatically opened on creation.

Parameters

- **conn** – Connection parameters. Can be either a string `"IP:port"` (e.g., `"127.0.0.1:80"`), or a tuple `(IP,port)`, where *IP* is a string and *port* is a number.
- **timeout** (*float*) – Default timeout (in seconds).
- **term_write** (*str*) – Line terminator for writing operations; appended to the data
- **term_read** (*str*) – List of possible single-char terminator for reading operations (specifies when *readline()* stops).
- **datatype** (*str*) – Type of the returned data; can be `"bytes"` (return *bytes* object), `"str"` (return *str* object), or `"auto"` (default Python result: *str* in Python 2 and *bytes* in Python 3)
- **reraise_error** – if not `None`, specifies an error to be re-raised on any backend exception (by default, use backend-specific error); should be a subclass of *DeviceBackendError*.

Note: If *term_read* is a string, its behavior is different from the VISA backend: instead of being a multi-char terminator it is assumed to be a set of single-char terminators. If multi-char terminator is required, *term_read* should be a single-element list instead of a string.

BackendError
alias of `builtins.OSError`

Error
alias of *DeviceNetworkError*

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

set_timeout(*timeout*)
Set operations timeout (in seconds)

get_timeout()
Get operations timeout (in seconds)

readline(*remove_term=True*, *timeout=None*, *skip_empty=True*)
Read a single line from the device.

Parameters

- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.
- **skip_empty** (*bool*) – If `True`, ignore empty lines (works only for `remove_term==True`).

read(*size=None*)
Read data from the device.

If *size* is not None, read *size* bytes (usual timeout applies); otherwise, read all available data (return immediately).

read_multichar_term(*term*, *remove_term=True*, *timeout=None*)
Read a single line with multiple possible terminators.

Parameters

- **term** – Either a string (single multi-char terminator) or a list of strings (multiple terminators).
- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.

write(*data*, *flush=True*, *read_echo=False*, *read_echo_delay=0*, *read_echo_lines=1*)
Write data to the device.

If `read_echo==True`, wait for *read_echo_delay* seconds and then perform *readline()* (*read_echo_lines* times). *flush* parameter is ignored.

ask(*query*, *delay=0.0*, *read_all=False*)
Perform a write followed by a read, with *delay* in between.

If `read_all==True`, read all the available data; otherwise, read a single line.

classmethod combine_conn(*conn1*, *conn2*)
Combined two connection parameters into a single dictionary (*conn1* overrides *conn2*)

cooldown(*kind='default'*)
Cooldown between the operations.

`kind` specifies the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`); `"default"` corresponds to the default cooldown (usually, specified as 0). Called automatically by various backend operations, so usually there is no need to call explicitly.

flush_read()
Flush the device output (read all the available data; return the number of bytes read)

static list_resources(*desc=False*)
List all available resources for this backend.

If `desc==False`, return list of connections (usually strings or tuples), which can be used to connect to the device. Otherwise, return a list of descriptions, which have more info, but can be backend-dependent.

Might not be implemented (depending on the backend), in which case returns `None`.

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

readlines(*lines_num*, *remove_term=True*, *timeout=None*, *skip_empty=True*)
Read multiple lines from the device.

Parameters are the same as in `readline()`.

setup_cooldown(***kwargs*)
Setup cooldown times for various operations.

The arguments are of the form `kind=value`, where `value` is the cooldown time (in seconds), and `kind` is the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`). `kind` can also be `default` (default value for all kind), or `all` (reset all cooldown values to this value). The cooldowns of the given kinds are usually called after the corresponding operation (it is necessary for some devices, otherwise the communication can freeze or crush). Default cooldown values are specified by `_default_operation_cooldown` class attribute dictionary.

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_timeout(*timeout=None*)
Context manager for usage of a different timeout inside a block

exception pylablib.core.devio.comm_backend.**DeviceUSBError**(*exc*)
Bases: `pylablib.core.devio.comm_backend.DeviceBackendError`

USB backend operation error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.core.devio.comm_backend.**PyUSBDeviceBackend**(*conn*, *timeout=10.0*, *term_write=None*, *term_read=None*, *check_read_size=True*, *datatype='auto'*, *reraise_error=None*)
Bases: `pylablib.core.devio.comm_backend.IDeviceCommBackend`

USB backend (via PyUSB package).

Connection is automatically opened on creation.

Parameters

- **conn** – Connection parameters. Can be either a string (for a port), or a list/tuple (vendorID, productID, index, endpoint_read, endpoint_write, backend) supplied to the connection (default is (0x0000,0x0000,0,0x00, 0x01,'libusb1'), which is invalid for most devices), or a dict with the same parameters. vendorID and productID specify device kind, index is an integer index (starting from zero) of the device among several identical (i.e., with the same ids) ones, and endpoint_read and endpoint_write specify connection endpoints for the specific device.
- **timeout** (*float*) – Default timeout (in seconds).
- **term_write** (*str*) – Line terminator for writing operations; appended to the data
- **term_read** (*str*) – List of possible single-char terminator for reading operations (specifies when *readline()* stops).
- **datatype** (*str*) – Type of the returned data; can be "bytes" (return *bytes* object), "str" (return *str* object), or "auto" (default Python result: *str* in Python 2 and *bytes* in Python 3)
- **reraise_error** – if not None, specifies an error to be re-raised on any backend exception (by default, use backend-specific error); should be a subclass of *DeviceBackendError*.

BackendError
alias of usb.core.USBError

Error
alias of *DeviceUSBError*

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

set_timeout(*timeout*)
Set operations timeout (in seconds)

get_timeout()
Get operations timeout (in seconds)

readline(*remove_term=True*, *timeout=None*, *skip_empty=True*, *error_on_timeout=True*)
Read a single line from the device.

Parameters

- **remove_term** (*bool*) – If True, remove terminal characters from the result.
- **timeout** – Operation timeout. If None, use the default device timeout.
- **skip_empty** (*bool*) – If True, ignore empty lines (works only for remove_term==True).
- **error_on_timeout** (*bool*) – If False, return an incomplete line instead of raising the error on timeout.

read(*size=None*, *max_read_size=65536*)
Read data from the device.

If *size* is not None, read *size* bytes (usual timeout applies); otherwise, read all available data (return immediately).

read_multichar_term(*term*, *remove_term=True*, *timeout=None*, *error_on_timeout=True*)
Read a single line with multiple possible terminators.

Parameters

- **term** – Either a string (single multi-char terminator) or a list of strings (multiple terminators).
- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.
- **error_on_timeout** (*bool*) – If `False`, return an incomplete line instead of raising the error on timeout.

write(*data*, *read_echo=False*, *read_echo_delay=0*, *read_echo_lines=1*)
Write data to the device.

If `read_echo==True`, wait for *read_echo_delay* seconds and then perform `readline()` (*read_echo_lines* times).

static list_resources(*desc=False*, ***kwargs*)
List all available resources for this backend.

If `desc==False`, return list of connections (usually strings or tuples), which can be used to connect to the device. Otherwise, return a list of descriptions, which have more info, but can be backend-dependent.

Might not be implemented (depending on the backend), in which case returns `None`.

ask(*query*, *delay=0.0*, *read_all=False*)
Perform a write followed by a read, with *delay* in between.

If `read_all==True`, read all the available data; otherwise, read a single line.

classmethod combine_conn(*conn1*, *conn2*)
Combined two connection parameters into a single dictionary (*conn1* overrides *conn2*)

cooldown(*kind='default'*)
Cooldown between the operations.

`kind` specifies the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`); `"default"` corresponds to the default cooldown (usually, specified as 0). Called automatically by various backend operations, so usually there is no need to call explicitly.

flush_read()
Flush the device output (read all the available data; return the number of bytes read)

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

readlines(*lines_num*, *remove_term=True*, *timeout=None*, *skip_empty=True*)
Read multiple lines from the device.

Parameters are the same as in `readline()`.

setup_cooldown(***kwargs*)
Setup cooldown times for various operations.

The arguments are of the form `kind=value`, where `value` is the cooldown time (in seconds), and `kind` is the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`). `kind` can also be `default` (default value for all kind), or `all` (reset all cooldown values to this value). The cooldowns of the given kinds are usually called after the corresponding operation (it is necessary for some devices, otherwise the communication can freeze or crush). Default cooldown values are specified by `_default_operation_cooldown` class attribute dictionary.

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_timeout(*timeout=None*)
Context manager for usage of a different timeout inside a block

exception `pylablib.core.devio.comm_backend.`**DeviceRecordedError**(*exc*)
Bases: *pylablib.core.devio.comm_backend.DeviceBackendError*

Recorded backend operation error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `pylablib.core.devio.comm_backend.`**RecordedDeviceBackend**(*conn, datatype='auto', reraise_error=None*)
Bases: *pylablib.core.devio.comm_backend.IDeviceCommBackend*

Recorded backend.

Connection is automatically opened on creation.

Parameters

- **conn** – connection parameters (recorded log path)
- **datatype** (*str*) – Type of the returned data; can be `"bytes"` (return *bytes* object), `"str"` (return *str* object), or `"auto"` (default Python result: *str* in Python 2 and *bytes* in Python 3)
- **reraise_error** – if not `None`, specifies an error to be re-raised on any backend exception (by default, use backend-specific error); should be a subclass of *DeviceBackendError*.

BackendError
alias of `builtins.OSError`

Error
alias of *DeviceRecordedError*

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

start(*header*)
Start recorded section

stop()
Stop logging section

section(*header*)

readline(*remove_term=True*, *timeout=None*, *skip_empty=True*)
Read a single line from the device.

Parameters

- **remove_term** (*bool*) – If `True`, remove terminal characters from the result.
- **timeout** – Operation timeout. If `None`, use the default device timeout.
- **skip_empty** (*bool*) – If `True`, ignore empty lines (works only for `remove_term==True`).

read(*size=None*)
Read data from the device.

If *size* is not None, read *size* bytes (usual timeout applies); otherwise, read all available data (return immediately).

write(*data*, *flush=True*, *read_echo=False*, *read_echo_delay=0*, *read_echo_lines=1*)
Write data to the device.

If `flush==True`, flush the write buffer. If `read_echo==True`, wait for *read_echo_delay* seconds and then perform `readline()` (*read_echo_lines* times).

ask(*query*, *delay=0.0*, *read_all=False*)
Perform a write followed by a read, with *delay* in between.

If `read_all==True`, read all the available data; otherwise, read a single line.

classmethod combine_conn(*conn1*, *conn2*)
Combined two connection parameters into a single dictionary (*conn1* overrides *conn2*)

cooldown(*kind='default'*)
Cooldown between the operations.

`kind` specifies the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`); `"default"` corresponds to the default cooldown (usually, specified as 0). Called automatically by various backend operations, so usually there is no need to call explicitly.

flush_read()
Flush the device output (read all the available data; return the number of bytes read)

get_timeout()
Get operations timeout (in seconds)

static list_resources(*desc=False*)
List all available resources for this backend.

If `desc==False`, return list of connections (usually strings or tuples), which can be used to connect to the device. Otherwise, return a list of descriptions, which have more info, but can be backend-dependent.

Might not be implemented (depending on the backend), in which case returns `None`.

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

readlines (*lines_num*, *remove_term=True*, *timeout=None*, *skip_empty=True*)
Read multiple lines from the device.

Parameters are the same as in `readline()`.

set_timeout (*timeout*)
Set operations timeout (in seconds)

setup_cooldown (***kwargs*)
Setup cooldown times for various operations.

The arguments are of the form `kind=value`, where `value` is the cooldown time (in seconds), and `kind` is the operation kind (common kinds are `open`, `close`, `read`, `write`, `timeout`, and `flush`). `kind` can also be `default` (default value for all kind), or `all` (reset all cooldown values to this value). The cooldowns of the given kinds are usually called after the corresponding operation (it is necessary for some devices, otherwise the communication can freeze or crush). Default cooldown values are specified by `_default_operation_cooldown` class attribute dictionary.

unlock ()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_timeout (*timeout=None*)
Context manager for usage of a different timeout inside a block

`pylablib.core.devio.comm_backend.`**autodetect_backend** (*conn*, *default='visa'*)
Try to determine the backend by the connection.

default specifies the default backend which is returned if the backend is unclear.

`pylablib.core.devio.comm_backend.`**new_backend** (*conn*, *backend='auto'*, *defaults=None*, ***kwargs*)
Build new backend with the supplied parameters.

Parameters

- **conn** – Connection parameters (depend on the backend). Can be simply connection parameters (tuple or dict) for the given backend (e.g., `"192.168.0.1"` or `("COM1", 19200)`), a tuple `(backend, conn)` which specifies both backend and connection (in which case it overrides the supplied backend), or an already opened backend (in which case it is returned as is)
- **backend** (`str`) – Backend type. Available backends are `'auto'` (try to autodetect based on the connection), `'visa'`, `'serial'`, `'ft232'`, `'network'`, and `"pyusb"`. Can also be directly a backend class (more appropriate for custom backends), or a tuple `('auto', backend)`, which is analogous to `'auto'`, but it returns the specified `backend` if the autodetection fails; by default, the fallback backend is `'visa'`, so `'auto'` is exactly the same as `('auto', 'visa')`.
- **defaults** – if not `None`, specifies a dictionary `{backend: params}` with default connection parameters (depending on the backend), which are added to the connection parameters
- ****kwargs** – parameters sent to the backend.

`pylablib.core.devio.comm_backend.`**backend_error** (*backend*, *conn=None*)
Return error class corresponding to the current backend.

Like `new_backend()`, allows setting `backend="auto"`, in which case *conn* is used to try and autodetect the backend kind (not completely reliable, should be avoided).

`pylablib.core.devio.comm_backend.`**list_backend_resources** (*backend=None*, *desc=False*)
List all resources for the given backend.

If *backend* is `None`, return dictionary `{backend: resources}` for all available backends. If `desc==False`, return list of connections (usually strings or tuples), which can be used to connect to the device. Otherwise, return a list of descriptions, which have more info, but can be backend-dependent.

class pylablib.core.devio.comm_backend.**ICommBackendWrapper**(*instr*)

Bases: *pylablib.core.devio.interface.IDevice*

A base class for an instrument using a communication backend.

Parameters **instr** – Backend (assumed to be already opened).

class NoParameterCaller(*device*, *kind*)

Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)

Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_device_variable(*key*)

Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)

Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)

Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)

Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

set_device_variable(*key*, *value*)

Set the value of a settings parameter

open()

Open the backend

close()

Close the backend

is_opened()

Check if the device is connected

lock(*timeout=None*)

Lock the access to the device from other threads/processes (isn't necessarily implemented)

unlock()

Unlock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

pylablib.core.devio.data_format module

Library for binary data encoding/decoding for device communication and dealing with different data format representations in different contexts (numpy, SCPI, etc.).

class pylablib.core.devio.data_format.**DataFormat**(*size*, *kind*, *byteorder*)
Bases: `object`

Describes data encoding for device communications.

Parameters

- **size** (*int*) – Size of a single element (in bytes).
- **kind** (*str*) – Kind of the element. Can be `'i'` (integer), `'u'` (unsigned integer), `'f'` (floating point) or `'ascii'` (text representation).
- **byteorder** (*str*) – Byte order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

flip_byteorder()
Flip byteorder of the description

is_ascii()
Check of the format is textual

static from_desc(*desc*, *str_type='numpy'*)
Build the format from the string description.

str_type is the description format. Can be `'numpy'` (numpy dtype description), `'struct'` (`struct` description) or `'SCPI'` (the standard SCPI description).

static from_desc_SCPI(*desc*, *border='norm'*)
Build the format from the string SCPI description.

border describes byte order (either `'norm'` or `'swap'`).

to_desc(*str_type='auto'*)
Build a description string of this format.

str_type can be `'auto'` (similar to `'numpy'`, but also accepts `'ascii'`), `'numpy'`, `'struct'` or `'SCPI'` (return tuple `(desc, border)`).

convert_from_str(*data*)
Convert the string data into an array

convert_to_str(*data*, *ascii_format='.5f'*)
Convert the array into a string data.

ascii_format is the `str.format()` string for textual representation.

pylablib.core.devio.interface module

class pylablib.core.devio.interface.**IDevice**
Bases: `object`

A base class for an instrument.

Contains some useful functions for dealing with device settings.

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

set_device_variable(*key*, *value*)
Set the value of a settings parameter

class pylablib.core.devio.interface.**IParameterClass**(*name*)
Bases: `object`

A generic parameter class.

Deals with converting device interface representation and the 'internal' representation (e.g., names used in SCPI commands or integer indices). Also responsible for validating the user-passed and device-returned parameters.

Needs to define to methods: `__call__` for converting user parameters ('alias') into the device parameters ('value') and *`i()`* for the opposite conversion. In addition, it provides *`using_device()`* context manager to temporarily change the `device` attribute, which can be used by some parameter classes for device-dependent conversions.

Parameters name – parameter class name; used to match method arguments with corresponding classes.

using_device (*device*)
Context manager for temporarily changing the `device` attribute to the given device instance

docstring ()
Get a parameter docstring

i (*value*, *device=None*)
Convert device parameter value into a corresponding use parameter value

If not `None`, *device* specifies the corresponding device instance for device-dependent conversion.

class pylablib.core.devio.interface.**ICheckingParameterClass** (*name*)
Bases: *pylablib.core.devio.interface.IParameterClass*

Parameter class which separately handles checking and conversion.

Specifies six methods: *check_value()*, *to_alias()* and `_value_error_str` for handling value-to-alias conversion, and *check_alias()*, *to_value()* and `_alias_error_str` for handling alias-to-value conversion.

check_alias (*alias*)
Check if the alias is valid

check_value (*value*)
Check if the device value is valid

to_value (*alias*)
Convert the alias into a device value

to_alias (*value*)
Convert the device value into an alias

i (*value*, *device=None*)
Convert device parameter value into a corresponding use parameter value

If not `None`, *device* specifies the corresponding device instance for device-dependent conversion.

docstring ()
Get a parameter docstring

using_device (*device*)
Context manager for temporarily changing the `device` attribute to the given device instance

class pylablib.core.devio.interface.**RangeParameterClass** (*name*, *minval=None*, *maxval=None*, *out_of_range='error'*)
Bases: *pylablib.core.devio.interface.ICheckingParameterClass*

Parameter class for numerical values constrained to a certain range.

Parameters

- **name** – parameter class name
- **minval** – minimal allowed value (inclusive); `None` means no lower limit
- **maxval** – maximal allowed value (inclusive); `None` means no upper limit
- **out_of_range** – action if an out-of-range value is supplied; can be either `"error"` (raise an error), or `"truncate"` (truncate to the nearest limit).

check_value (*value*)
Check if the device value is valid

check_alias (*alias*)
Check if the alias is valid

to_value(*alias*)
Convert the alias into a device value

docstring()
Get a parameter docstring

i(*value*, *device=None*)
Convert device parameter value into a corresponding use parameter value

If not `None`, *device* specifies the corresponding device instance for device-dependent conversion.

to_alias(*value*)
Convert the device value into an alias

using_device(*device*)
Context manager for temporarily changing the `device` attribute to the given device instance

class pylablib.core.devio.interface.**IEnumParameterClass**(*name*, *allowed_alias='device_values'*, *allowed_value='exact'*, *alias_case=None*, *value_case=None*, *match_prefix=False*)

Bases: *pylablib.core.devio.interface.ICheckingParameterClass*

Parameter class for a generic enum (i.e., predefined values) parameter.

Defines two methods for handling conversion:

- `_get_value_map` which returns a dictionary for converting device values into aliases,
- `_get_alias_map` which returns a dictionary for converting aliases into device values.

These methods need to be redefined in subclasses.

Parameters

- **name** – parameter class name
- **allowed_alias** – specifies a range of allowed aliases; can be `"exact"` (only exact map matches are allowed), `"device_value"` (exact map matches and raw device values are allowed), or `"all"` (all values are allowed); in the latter two cases the value not in the map are passed as is.
- **allowed_value** – specifies a range of allowed device values; can be `"exact"` (only exact map matches are allowed), or `"all"` (all values are allowed); in the latter case the value not in the map is passed as is.
- **alias_case** – default alias parameter case for string values; can be `None` (no case normalization), or `"lower"` or `"upper"` (any received or returned alias will be normalized into this case)
- **value_case** – default value parameter case for string values; can be `None` (no case normalization), or `"lower"` or `"upper"` (any received or returned device value will be normalized into this case)
- **match_prefix** – if `True`, then the keys in the value map (returned by `_get_value_map` method) are interpreted as prefixes, so in the value-to-alias conversion the converted value matches the map value if it just starts with it; in the case of ambiguity (several map values are prefixes for the same converted value), the exact match takes priority; useful for some SCPI devices, where the shorter version of the value can sometimes be returned.

check_value(*value*)
Check if the device value is valid

check_alias(*alias*)
Check if the alias is valid

to_value(*alias*)
Convert the alias into a device value

to_alias(*value*)
Convert the device value into an alias

docstring()
Get a parameter docstring

i(*value*, *device=None*)
Convert device parameter value into a corresponding use parameter value

If not `None`, *device* specifies the corresponding device instance for device-dependent conversion.

using_device(*device*)
Context manager for temporarily changing the `device` attribute to the given device instance

class pylablib.core.devio.interface.**EnumParameterClass**(*name*, *alias_map*, *value_map=None*, *allowed_alias='device_values'*, *allowed_value='exact'*, *alias_case=None*, *value_case=None*, *match_prefix=False*)

Bases: *pylablib.core.devio.interface.IEnumParameterClass*

Parameter class for a enum (i.e., predefined values) parameter with the specified mapping.

Parameters

- **name** – parameter class name
- **alias_map** – mapping of aliases to device values; can be a dictionary, or a list of `(alias,value)` tuples (in the latter case non-tuple values are also allowed, indicating that value is the same as the alias); the list representation is useful in cases where the same alias maps to more than one value, so the map inversion is impossible
- **value_map** – mapping of device values to aliases; can only be a dictionary or `None`, which means that the alias map is automatically inverted
- **allowed_alias** – specifies a range of allowed aliases; can be `"exact"` (only exact map matches are allowed), `"device_value"` (exact map matches and raw device values are allowed), or `"all"` (all values are allowed); in the latter two cases the value not in the map are passed as is.
- **allowed_value** – specifies a range of allowed device values; can be `"exact"` (only exact map matches are allowed), or `"all"` (all values are allowed); in the latter case the value not in the map is passed as is.
- **alias_case** – default alias parameter case for string values; can be `None` (no case normalization), or `"lower"` or `"upper"` (any received or returned alias will be normalized into this case)
- **value_case** – default value parameter case for string values; can be `None` (no case normalization), or `"lower"` or `"upper"` (any received or returned device value will be normalized into this case)

- **match_prefix** – if `True`, then the keys in the value map (or values in the alias map, if only it is provided) are assumed to br prefixes, so in the value-to-alias conversion the converted value matches the map value if it just starts with it; useful for some SCPI devices, where the shorter version of the value can sometimes be returned.

check_alias(*alias*)
Check if the alias is valid

check_value(*value*)
Check if the device value is valid

docstring()
Get a parameter docstring

i(*value*, *device=None*)
Convert device parameter value into a corresponding use parameter value

If not `None`, *device* specifies the corresponding device instance for device-dependent conversion.

to_alias(*value*)
Convert the device value into an alias

to_value(*alias*)
Convert the alias into a device value

using_device(*device*)
Context manager for temporarily changing the `device` attribute to the given device instance

class `pylablib.core.devio.interface.`**FunctionParameterClass**(*name*, *to_alias=None*, *to_value=None*, *check_value=None*, *check_alias=None*, *alias_err=None*, *value_err=None*)

Bases: `pylablib.core.devio.interface.ICheckingParameterClass`

Parameter class which uses supplied methods for checking, conversion, and generating error messages.

The arguments correspond to the parameter methods with the same names. When not supplied, checking methods always return `True`, conversion methods leave value intact, and error string methods generate the default error messages.

check_value(*value*)
Check if the device value is valid

check_alias(*alias*)
Check if the alias is valid

to_alias(*value*)
Convert the device value into an alias

to_value(*alias*)
Convert the alias into a device value

docstring()
Get a parameter docstring

i(*value*, *device=None*)
Convert device parameter value into a corresponding use parameter value

If not `None`, *device* specifies the corresponding device instance for device-dependent conversion.

using_device(*device*)
Context manager for temporarily changing the `device` attribute to the given device instance

class pylablib.core.devio.interface.**CombinedParameterClass**(*name*, *parameters*)
Bases: *pylablib.core.devio.interface.IParameterClass*

A multi-stage combined parameter class, which performs several conversion/check stages.

Parameters

- **name** – parameter class name
- **parameters** – list of parameters classes which are combined; the order is from the 'most alias' to the 'most device parameter', i.e., when converting an alias to a device parameter, it is first passed to the first class, then the second, etc. (the reverse is done when converting device values into aliases)

docstring()
Get a parameter docstring

i(*value*, *device=None*)
Convert device parameter value into a corresponding use parameter value

If not `None`, *device* specifies the corresponding device instance for device-dependent conversion.

using_device(*device*)
Context manager for temporarily changing the `device` attribute to the given device instance

class pylablib.core.devio.interface.**TRawParameterValue**(*value*)
Bases: `tuple`

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

value

pylablib.core.devio.interface.**pval**(*value*)
Mark that the value has already been treated by the parameter class

pylablib.core.devio.interface.**use_parameters**(**args*, ***kwargs*)
Wrapper to indicate that a device class method uses device parameter classes.

The corresponding parameters classes are automatically determined if the argument name matches the parameter class name. The parameters classes can also be defined explicitly using keywords arguments `arg=parameter` supplied to the wrapper, where `arg` is the argument, and `parameter` is either a parameter class instance, or a parameter class name (the more preferable way). In addition, an argument `_returns` can be used to define the parameter class for the return value; it can also be a list or a tuple of parameter classes, indicating that the returned value is also a list or a tuple.

Module contents

pylablib.core.fileio package

Submodules

pylablib.core.fileio.datafile module

class pylablib.core.fileio.datafile.**DataFile**(*data*, *filepath=None*, *filetype=None*, *creation_time=None*, *comments=None*, *props=None*)

Bases: `object`

Describes a single datafile.

Parameters

- **data** – the main content of the file (usually a numpy array, a pandas DataFrame or a *Dictionary*).
- **filepath** (*str*) – absolute path from which the file was read
- **filetype** (*str*) – a source type (e.g., `"csv"` or `"bin"`)
- **creation_time** (*datetime.datetime*) – File creation time
- **props** (*dict*) – all the metainfo about the file (extracted from comments, filename etc.)
- **comments** (*list*) – all the comments excluding the ones containing props

get(*name*, *default=None*)
Get a property from the dictionary. Use default value if it's not found

pylablib.core.fileio.dict_entry module

Classes for dealing with the *Dictionary* entries with special conversion rules when saved or loaded. Used to redefine how certain objects (e.g., tables) inside dictionaries are written into files and read from files.

pylablib.core.fileio.dict_entry.**is_dict_entry_branch**(*branch*)
Check if the dictionary branch contains a dictionary entry which needs to be specially converted.

class pylablib.core.fileio.dict_entry.**DictEntryBuilder**(*entry_cls*, *pred=None*, ***kwargs*)

Bases: `object`

Object for building dictionary entries from objects.

Parameters

- **entry_cls** – dictionary entry class
- **pred** – method used to check if an object can be turned into the corresponding entry; if `None`, use the default entry class checker (`entry_class.is_data_valid`)
- **kwargs** – keyword arguments passed to the entry constructor along with the data

is_data_valid(*data*)
Check if a data object can be wrapped by the current entry class

from_data(*data*)
Build a dictionary entry from the data

class pylablib.core.fileio.dict_entry.**DictEntryParser**(*entry_cls*, *pred=None*, ***kwargs*)

Bases: `object`

Object for building dictionary entries from dictionary branches.

Parameters

- **entry_cls** – dictionary entry class
- **pred** – method used to check if a dictionary branch can be turned into the corresponding entry; if `None`, use the default entry class checker (`entry_class.is_branch_valid`)
- **kwargs** – keyword arguments passed to the entry `from_dict` class method along with the branch

is_branch_valid(*branch*)
Check if a branch can be parsed by the current entry class

from_dict(*dict_ptr*, *loc*)
Build a dictionary entry from the branch and the file location

`pylablib.core.fileio.dict_entry.`**add_dict_entry_builder**(*builder*)
Add an entry builder to the global list of builders

`pylablib.core.fileio.dict_entry.`**add_dict_entry_parser**(*parser*)
Add an entry parser to the global list of parsers

`pylablib.core.fileio.dict_entry.`**add_dict_entry_class**(*cls*)
Add an entry class.

Automatically registers builder and parser, which take no additional arguments and use default class method to determine if an object/branch can be converted into an entry.

`pylablib.core.fileio.dict_entry.`**from_data**(*data*, *builders=None*)
Build a dictionary entry from the data.

builders can contain an additional list of builder to try before using the default ones.

`pylablib.core.fileio.dict_entry.`**from_dict**(*dict_ptr*, *loc*, *parsers=None*)
Build a dictionary entry from the dictionary branch and the file location.

parsers can contain an additional list of parsers to try before using the default ones.

class `pylablib.core.fileio.dict_entry.`**IDictionaryEntry**(*data*)
Bases: `object`

A generic *Dictionary* entry.

Contains data represented by the node, as well as the way to represent this data as a dictionary branch.

Parameters **data** – data to be wrapped

is_data_valid(*class method*)
check if a data object can be wrapped by the current entry class

is_branch_valid(*class method*)
check if a branch can be parsed by the current entry class

from_dict(*class method*)
create a dictionary entry of a given class from the dictionary branch

to_dict()
convert the entry to a dictionary branch

classmethod is_data_valid(*data*)
Check if a data object can be wrapped by the current entry class

classmethod is_branch_valid(*branch*)
Check if a branch can be parsed by the current entry class

classmethod from_dict(*dict_ptr*, *loc*)
Convert a dictionary branch to a specific *IDictionaryEntry* object.

Parameters

- **dict_ptr** (dictionary.DictionaryPointer) – Pointer to the dictionary location for the entry.
- **loc** – Location for the data to be loaded.

to_dict(*dict_ptr*, *loc*)
Convert data to a dictionary branch on saving.

Parameters

- **dict_ptr** (dictionary.DictionaryPointer) – Pointer to the dictionary location for the entry.
- **loc** – File location for the data to be saved.

pylablib.core.fileio.dict_entry.**parse_stored_table_data**(*desc=None*, *data=None*, *out_type='pandas'*)
Parse table data corresponding to the given description dictionary and data.

Parameters

- **desc** – description dictionary; can be `None`, if no description is given
- **data** – separately loaded data; can be `None`, if no data is given (in this case assume that it is stored in the description dictionary); can be a tuple `(column_data, column_names)` (such as the one returned by *parse_csv.read_table()*), or a an *InlineTable* object containing such tuple.
- **out_type** (*str*) – Output format of the data (`'array'` for numpy arrays or `'pandas'` for pandas DataFrame objects).

Returns tuple `(data, columns)`, where `data` is the data table in the specified format, and `columns` is the list of columns

class pylablib.core.fileio.dict_entry.**ITableDictionaryEntry**(*data*, *columns=None*)
Bases: *pylablib.core.fileio.dict_entry.IDictionaryEntry*

A generic table Dictionary entry.

Parameters

- **data** – Table data.
- **columns** (*list*) – If not `None`, list of column names (if `None` and data is a pandas DataFrame object, get column names from that).

classmethod is_data_valid(*data*)
Check if a data object can be wrapped by the current entry class

classmethod from_dict(*dict_ptr*, *loc*, *out_type='pandas'*)
Convert a dictionary branch to a specific DictionaryEntry object.

Parameters

- **dict_ptr** (dictionary.DictionaryPointer) – Pointer to the dictionary location for the entry.
- **loc** – Location for the data to be loaded.

- **out_type** (*str*) – Output format of the data ('array' for numpy arrays or 'pandas' for pandas DataFrame objects), used only if the dictionary doesn't provide the format.

classmethod is_branch_valid(*branch*)
Check if a branch can be parsed by the current entry class

to_dict(*dict_ptr, loc*)
Convert data to a dictionary branch on saving.

Parameters

- **dict_ptr** (dictionary.DictionaryPointer) – Pointer to the dictionary location for the entry.
- **loc** – File location for the data to be saved.

class pylablib.core.fileio.dict_entry.**InlineTableDictionaryEntry**(*data, columns=None*)
Bases: *pylablib.core.fileio.dict_entry.ITableDictionaryEntry*

An inlined table Dictionary entry.

Parameters

- **data** – Table data.
- **columns** (*list*) – If not None, a list of column names (if None and data is a pandas DataFrame object, get column names from that).

to_dict(*dict_ptr, loc*)
Convert the data to a dictionary branch and write the table to the file.

classmethod from_dict(*dict_ptr, loc, out_type='pandas'*)
Build an *InlineTableDictionaryEntry* object from the dictionary and read the inlined data.

Parameters

- **dict_ptr** (dictionary.DictionaryPointer) – Pointer to the dictionary location for the entry.
- **loc** – Location for the data to be loaded.
- **out_type** (*str*) – Output format of the data ('array' for numpy arrays or 'pandas' for pandas DataFrame objects).

classmethod is_branch_valid(*branch*)
Check if a branch can be parsed by the current entry class

classmethod is_data_valid(*data*)
Check if a data object can be wrapped by the current entry class

class pylablib.core.fileio.dict_entry.**IExternalTableDictionaryEntry**(*data, file_format, name, columns, force_name=True*)
Bases: *pylablib.core.fileio.dict_entry.ITableDictionaryEntry*

classmethod from_dict(*dict_ptr, loc, out_type='pandas'*)
Convert a dictionary branch to a specific DictionaryEntry object.

Parameters

- **dict_ptr** (`dictionary.DictionaryPointer`) – Pointer to the dictionary location for the entry.
- **loc** – Location for the data to be loaded.
- **out_type** (*str*) – Output format of the data (`'array'` for numpy arrays or `'pandas'` for pandas DataFrame objects), used only if the dictionary doesn't provide the format.

classmethod is_branch_valid(*branch*)
Check if a branch can be parsed by the current entry class

classmethod is_data_valid(*data*)
Check if a data object can be wrapped by the current entry class

to_dict(*dict_ptr*, *loc*)
Convert data to a dictionary branch on saving.

Parameters

- **dict_ptr** (`dictionary.DictionaryPointer`) – Pointer to the dictionary location for the entry.
- **loc** – File location for the data to be saved.

class pylablib.core.fileio.dict_entry.**ExternalTextTableDictionaryEntry**(*data=None*, *file_format='csv'*, *name=''*, *columns=None*, *force_name=True*)

Bases: *pylablib.core.fileio.dict_entry.IExternalTableDictionaryEntry*

An external text table Dictionary entry.

Parameters

- **data** – Table data.
- **file_format** (*str*) – Output file format.
- **name** (*str*) – Name template for the external file (default is the full path connected with `"_"` symbol).
- **columns** (*list*) – If not `None`, a list of column names (if `None` and data is a pandas DataFrame object, get column names from that).
- **force_name** (*bool*) – If `False` and the target file already exists, generate a new unique name; otherwise, overwrite the file.

to_dict(*dict_ptr*, *loc*)
Convert the data to a dictionary branch and save the table to an external file.

classmethod from_dict(*dict_ptr*, *loc*, *out_type='pandas'*)
Build an *ExternalTextTableDictionaryEntry* object from the dictionary and load the external data.

Parameters

- **dict_ptr** (`dictionary.DictionaryPointer`) – Pointer to the dictionary location for the entry.
- **loc** – Location for the data to be loaded.
- **out_type** (*str*) – Output format of the data (`'array'` for numpy arrays or `'pandas'` for pandas DataFrame objects).

classmethod is_branch_valid(*branch*)
Check if a branch can be parsed by the current entry class

classmethod is_data_valid(*data*)
Check if a data object can be wrapped by the current entry class

class pylablib.core.fileio.dict_entry.**ExternalBinTableDictionaryEntry**(*data=None, file_format='bin', name='', columns=None, force_name=True*)

Bases: *pylablib.core.fileio.dict_entry.IExternalTableDictionaryEntry*

An external binary table Dictionary entry.

Parameters

- **data** – Table data.
- **file_format** (*str*) – Output file format.
- **name** (*str*) – Name template for the external file (default is the full path connected with "_" symbol).
- **columns** (*list*) – If not None, a list of column names (if None and data is a pandas DataFrame object, get column names from that).
- **force_name** (*bool*) – If False and the target file already exists, generate a new unique name; otherwise, overwrite the file.

to_dict(*dict_ptr, loc*)
Convert the data to a dictionary branch and save the table to an external file.

classmethod from_dict(*dict_ptr, loc, out_type='pandas'*)
Build an *ExternalBinTableDictionaryEntry* object from the dictionary and load the external data.

Parameters

- **dict_ptr** (dictionary.DictionaryPointer) – Pointer to the dictionary location for the entry.
- **loc** – Location for the data to be loaded.
- **out_type** (*str*) – Output format of the data ('array' for numpy arrays or 'pandas' for pandas DataFrame objects).

classmethod is_branch_valid(*branch*)
Check if a branch can be parsed by the current entry class

classmethod is_data_valid(*data*)
Check if a data object can be wrapped by the current entry class

pylablib.core.fileio.dict_entry.**table_entry_builder**(*table_format='inline'*)
Make an entry builder for tables depending on the table format.

Parameters table_format (*str*) – Default format for table (numpy arrays or pandas DataFrames) entries. Can be 'inline' (table is written inside the file), 'csv' (external CSV file) or 'bin' (external binary file).

class pylablib.core.fileio.dict_entry.**IExternalFileDictionaryEntry**(*data, name='', force_name=True*)

Bases: *pylablib.core.fileio.dict_entry.IDictionaryEntry*

Generic dictionary entry for data in an external file.

Parameters

- **data** – Stored data.
- **name** (*str*) – Name template for the external file (default is the full path connected with "_" symbol).
- **force_name** (*bool*) – If False and the target file already exists, generate a new unique name; otherwise, overwrite the file.

file_format = None

static add_file_format (*subclass*)
Register an *IExternalFileDictionaryEntry* as a possible stored file format.

Used to automatically invoke a correct loader when loading the dictionary file. Only needs to be done once after the subclass declaration.

to_dict (*dict_ptr, loc*)
Convert the data to a dictionary branch and save the data to an external file

classmethod from_dict (*dict_ptr, loc*)
Build an *IExternalFileDictionaryEntry* object from the dictionary and load the external data.

Parameters

- **dict_ptr** (dictionary.DictionaryPointer) – Pointer to the dictionary location for the entry.
- **loc** – Location for the data to be loaded.

get_preamble ()
Generate preamble (dictionary with supplementary data which allows to load the data from the file)

save_file (*loc_file*)
Save stored data into the given location.

Virtual method, should be overloaded in subclasses

classmethod load_file (*loc_file, preamble*)
Load stored data from the given location, using the supplied preamble.

Virtual method, should be overloaded in subclasses

classmethod is_branch_valid (*branch*)
Check if a branch can be parsed by the current entry class

classmethod is_data_valid (*data*)
Check if a data object can be wrapped by the current entry class

class pylablib.core.fileio.dict_entry.**ExternalNumpyDictionaryEntry** (*data, name='', force_name=True, dtype=None*)

Bases: *pylablib.core.fileio.dict_entry.IExternalFileDictionaryEntry*

A dictionary entry which stores the numpy array data into an external file in binary format.

Parameters

- **data** – Numpy array data.
- **name** (*str*) – Name template for the external file (default is the full path connected with "_" symbol).

- **force_name** (*bool*) – If False and the target file already exists, generate a new unique name; otherwise, overwrite the file.
- **dtype** – numpy dtype to load/save the data (by default, dtype of the supplied data).

file_format = 'numpy'

get_preamble()
Generate preamble (dictionary with supplementary data which allows to load the data from the file)

save_file(*loc_file*)
Save stored data into the given location

classmethod **load_file**(*loc_file*, *preamble*)
Load stored data from the given location, using the supplied preamble

static **add_file_format**(*subclass*)
Register an *IExternalFileDictionaryEntry* as a possible stored file format.

Used to automatically invoke a correct loader when loading the dictionary file. Only needs to be done once after the subclass declaration.

classmethod **from_dict**(*dict_ptr*, *loc*)
Build an *IExternalFileDictionaryEntry* object from the dictionary and load the external data.

Parameters

- **dict_ptr** (dictionary.DictionaryPointer) – Pointer to the dictionary location for the entry.
- **loc** – Location for the data to be loaded.

classmethod **is_branch_valid**(*branch*)
Check if a branch can be parsed by the current entry class

classmethod **is_data_valid**(*data*)
Check if a data object can be wrapped by the current entry class

to_dict(*dict_ptr*, *loc*)
Convert the data to a dictionary branch and save the data to an external file

class pylablib.core.fileio.dict_entry.**ExpandedContainerDictionaryEntry**(*data*)
Bases: *pylablib.core.fileio.dict_entry.IDictionaryEntry*

A dictionary entry which expands containers (lists, tuples, dictionaries) into subdictionaries.

Useful when the data in the containers is complex, so writing it into one line (as is default for lists and tuples) wouldn't work.

Parameters **data** – Container data.

to_dict(*dict_ptr*, *loc*)
Convert the stored container to a dictionary branch

classmethod **from_dict**(*dict_ptr*, *loc*)
Build an *ExpandedContainerDictionaryEntry* object from the dictionary

classmethod **is_branch_valid**(*branch*)
Check if a branch can be parsed by the current entry class

classmethod **is_data_valid**(*data*)
Check if a data object can be wrapped by the current entry class

pylablib.core.fileio.loadfile module

Utilities for reading data files.

class pylablib.core.fileio.loadfile.**IInputFileFormat**

Bases: object

Generic class for an input file format.

Based on *file_format* or autodetection, calls one of its subclasses to read the file.

Defines a single static method

static detect_file_format(*location_file*)

read(*location_file*)

Read a file at a given location

class pylablib.core.fileio.loadfile.**ITextInputFileFormat**

Bases: *pylablib.core.fileio.loadfile.IInputFileFormat*

Generic class for a text input file format.

Based on *file_format* or autodetection, calls one of its subclasses to read the file.

static detect_file_format(*location_file*)

read(*location_file*)

Read a file at a given location

class pylablib.core.fileio.loadfile.**CSVTableInputFileFormat**(*out_type='default', dtype='numeric', columns=None, delimiters=None, empty_entry_substitute=None, ignore_corrupted_lines=True, skip_lines=0*)

Bases: *pylablib.core.fileio.loadfile.ITextInputFileFormat*

Class for CSV input file format.

Parameters

- **out_type** (*str*) – type of the result: 'array' for numpy array, 'pandas' for pandas DataFrame, or 'default' (determined by the library default; 'pandas' by default)
- **dtype** – dtype of entries; can be either a single type, or a list of types (one per column). Possible dtypes are: 'int', 'float', 'complex', 'numeric' (tries to coerce to minimal possible numeric type, raises error if data can't be converted to *complex*), 'generic' (accept arbitrary types, including lists, dictionaries, escaped strings, etc.), 'raw' (keep raw string).
- **columns** – either a number if columns, or a list of columns names.
- **delimiters** (*str*) – Regex string which recognizes entries delimiters (by default r"\s*,\s*|\s+", i.e., commas and whitespaces).
- **empty_entry_substitute** – Substitute for empty table entries. If None, all empty table entries are skipped.
- **ignore_corrupted_lines** (*bool*) – If True, skip corrupted (e.g., non-numeric for numeric dtype, or with too few entries) lines; otherwise, raise ValueError.

- **skip_lines** (*int*) – Number of lines to skip from the beginning of the file.

read(*location_file*)
Read a file at a given location

static detect_file_format(*location_file*)

class pylablib.core.fileio.loadfile.**DictionaryInputFileFormat**(*case_normalization=None, inline_dtype='generic', inline_out_type='default', entry_format='value', allow_duplicate_keys=False, skip_lines=0*)

Bases: *pylablib.core.fileio.loadfile.ITextInputFileFormat*

Class for Dictionary input file format.

Parameters

- **location_file** – Location of the data.
- **case_normalization** (*str*) – If None, the dictionary paths are case-sensitive; otherwise, defines the way the entries are normalized ('lower' or 'upper').
- **inline_dtype** (*str*) – dtype for inlined tables.
- **inline_out_type** (*str*) – type of the result of the inline table: 'array' for numpy array, 'pandas' for pandas DataFrame, 'raw' for raw *InlineTable* data containing tuple (column_data, column_names), or 'default' (determined by the library default; 'pandas' by default).
- **entry_format** (*str*) – Determines the way for dealing with *dict_entry.IDictionaryEntry* objects (objects transformed into dictionary branches with special recognition rules). Can be 'branch' (don't attempt to recognize those object, leave dictionary as in the file), 'dict_entry' (recognize and leave as *dict_entry.IDictionaryEntry* objects) or 'value' (recognize and keep the value).
- **allow_duplicate_keys** (*bool*) – if False and the same key is mentioned twice in the file, raise and error
- **skip_lines** (*int*) – Number of lines to skip from the beginning of the file.

read(*location_file*)
Read a file at a given location

static detect_file_format(*location_file*)

class pylablib.core.fileio.loadfile.**BinaryTableInputFileFormatter**(*out_type='default', dtype='<f8', columns=None, packing='flatten', preamble=None, skip_bytes=0*)

Bases: *pylablib.core.fileio.loadfile.IInputFileFormat*

Class for binary input file format.

Parameters

- **location_file** – Location of the data.
- **out_type** (*str*) – type of the result: `'array'` for numpy array, `'pandas'` for pandas DataFrame, or `'default'` (determined by the library default; `'pandas'` by default)
- **dtype** – `numpy.dtype` describing the data.
- **columns** – either number if columns, or a list of columns names.
- **packing** (*str*) – The way the 2D array is packed. Can be either `'flatten'` (data is stored row-wise) or `'transposed'` (data is stored column-wise).
- **preamble** (*dict*) – If not `None`, defines binary file parameters that supersede the parameters supplied to the function. The defined parameters are `'dtype'`, `'packing'`, `'ncols'` (number of columns) and `'nrows'` (number of rows).
- **skip_bytes** (*int*) – Number of bytes to skip from the beginning of the file.

read(*location_file*)

Read a file at a given location

static detect_file_format(*location_file*)

`pylablib.core.fileio.loadfile.`**build_file_format**(*location_file*, *file_format='generic'*, ***kwargs*)

Create file format (*IInputFileFormat* instance) for given parameters and file locations.

If `file_format` is already an instance of *IInputFileFormat*, return unchanged. If `file_format` is generic (e.g., `"generic"` or `"test"`), attempt to autodetect it from the file. `**kwargs` are passed to the file format constructor.

`pylablib.core.fileio.loadfile.`**load_csv**(*path=None*, *out_type='default'*, *dtype='numeric'*, *columns=None*, *delimiters=None*, *empty_entry_substitute=None*, *ignore_corrupted_lines=True*, *skip_lines=0*, *loc='file'*, *return_file=False*)

Load data table from a CSV/table file.

Parameters

- **path** (*str*) – path to the file of a file-like object
- **out_type** (*str*) – type of the result: `'array'` for numpy array, `'pandas'` for pandas DataFrame, or `'default'` (determined by the library default; `'pandas'` by default)
- **dtype** – dtype of entries; can be either a single type, or a list of types (one per column). Possible dtypes are: `'int'`, `'float'`, `'complex'`, `'numeric'` (tries to coerce to minimal possible numeric type, raises error if data can't be converted to *complex*), `'generic'` (accept arbitrary types, including lists, dictionaries, escaped strings, etc.), `'raw'` (keep raw string).
- **columns** – either a number if columns, or a list of columns names
- **delimiters** (*str*) – regex string which recognizes entries delimiters (by default `r"\s*,\s*|\s+"`, i.e., commas and whitespaces)
- **empty_entry_substitute** – substitute for empty table entries. If `None`, all empty table entries are skipped

- **ignore_corrupted_lines** (*bool*) – if True, skip corrupted (e.g., non-numeric for numeric dtype, or with too few entries) lines; otherwise, raise ValueError
- **skip_lines** (*int*) – number of lines to skip from the beginning of the file
- **loc** (*str*) – location type ("file" means the usual file location; see *location.get_location()* for details)
- **return_file** (*bool*) – if True, return *DataFile* object (contains some metainfo); otherwise, return just the file data

pylablib.core.fileio.loadfile.**load_csv_desc**(*path=None*, *loc='file'*, *return_file=False*)
Load data from the extended CSV table file.

Analogous to *load_dict()*, but doesn't allow any additional parameters (which don't matter in this case).

Parameters

- **path** (*str*) – path to the file of a file-like object
- **loc** (*str*) – location type ("file" means the usual file location; see *location.get_location()* for details)
- **return_file** (*bool*) – if True, return *DataFile* object (contains some metainfo); otherwise, return just the file data

pylablib.core.fileio.loadfile.**load_bin**(*path=None*, *out_type='default'*, *dtype='<f8'*, *columns=None*, *packing='flatten'*, *preamble=None*, *skip_bytes=0*, *loc='file'*, *return_file=False*)
Load data from the binary file.

Parameters

- **path** (*str*) – path to the file of a file-like object
- **out_type** (*str*) – type of the result: 'array' for numpy array, 'pandas' for pandas DataFrame, or 'default' (determined by the library default; 'pandas' by default)
- **dtype** – numpy.dtype describing the data.
- **columns** – either number if columns, or a list of columns names.
- **packing** (*str*) – The way the 2D array is packed. Can be either 'flatten' (data is stored row-wise) or 'transposed' (data is stored column-wise).
- **preamble** (*dict*) – If not None, defines binary file parameters that supersede the parameters supplied to the function. The defined parameters are 'dtype', 'packing', 'ncols' (number of columns) and 'nrows' (number of rows).
- **skip_bytes** (*int*) – Number of bytes to skip from the beginning of the file.
- **loc** (*str*) – location type ("file" means the usual file location; see *location.get_location()* for details)
- **return_file** (*bool*) – if True, return *DataFile* object (contains some metainfo); otherwise, return just the file data

pylablib.core.fileio.loadfile.**load_bin_desc**(*path=None*, *loc='file'*, *return_file=False*)
Load data from the binary file with a description.

Analogous to *load_dict()*, but doesn't allow any additional parameters (which don't matter in this case).

Parameters

- **path** (*str*) – path to the file of a file-like object

- **loc** (*str*) – location type ("file" means the usual file location; see *location.get_location()* for details)
- **return_file** (*bool*) – if True, return *DataFile* object (contains some metainfo); otherwise, return just the file data

pylablib.core.fileio.loadfile.**load_dict**(*path=None*, *case_normalization=None*, *inline_dtype='generic'*, *entry_format='value'*, *inline_out_type='default'*, *skip_lines=0*, *allow_duplicate_keys=False*, *loc='file'*, *return_file=False*)

Load data from the dictionary file.

Parameters

- **path** (*str*) – path to the file of a file-like object
- **case_normalization** (*str*) – If None, the dictionary paths are case-sensitive; otherwise, defines the way the entries are normalized ('lower' or 'upper').
- **inline_dtype** (*str*) – dtype for inlined tables.
- **inline_out_type** (*str*) – type of the result of the inline table: 'array' for numpy array, 'pandas' for pandas DataFrame, 'raw' for raw *InlineTable* data containing tuple (column_data, column_names), or 'default' (determined by the library default; 'pandas' by default).
- **entry_format** (*str*) – Determines the way for dealing with *dict_entry.IDictionaryEntry* objects (objects transformed into dictionary branches with special recognition rules). Can be 'branch' (don't attempt to recognize those object, leave dictionary as in the file), 'dict_entry' (recognize and leave as *dict_entry.IDictionaryEntry* objects) or 'value' (recognize and keep the value).
- **allow_duplicate_keys** (*bool*) – if False and the same key is mentioned twice in the file, raise and error
- **skip_lines** (*int*) – Number of lines to skip from the beginning of the file.
- **loc** (*str*) – location type ("file" means the usual file location; see *location.get_location()* for details)
- **return_file** (*bool*) – if True, return *DataFile* object (contains some metainfo); otherwise, return just the file data

pylablib.core.fileio.loadfile.**load_generic**(*path=None*, *file_format=None*, *loc='file'*, *return_file=False*, ***kwargs*)

Load data from the file.

Parameters

- **path** (*str*) – path to the file of a file-like object
- **file_format** (*str*) – input file format; if None, attempt to auto-detect file format (same as 'generic'); can also be an *IInputFileFormat* instance for specific reading method
- **loc** (*str*) – location type ("file" means the usual file location; see *location.get_location()* for details)
- **return_file** (*bool*) – if True, return *DataFile* object (contains some metainfo); otherwise, return just the file data

****kwargs** are passed to the file formatter used to read the data (see *CSVTableInputFileFormat*, *DictionaryInputFileFormat* and *BinaryTableInputFileFormatter* for the possible arguments). The default format names are:

- `'generic'`: Generic file format. Attempt to autodetect, raise `IOError` if unsuccessful;
- `'txt'`: Generic text file. Attempt to autodetect, raise `IOError` if unsuccessful
- `'csv'`: CSV file, corresponds to *CSVTableInputFileFormat*;
- `'dict'`: Dictionary file, corresponds to *DictionaryInputFileFormat*;
- `'bin'`: Binary file, corresponds to *BinaryTableInputFileFormatter*

pylablib.core.fileio.loadfile_utils module

Miscellaneous utilities for reading data files.

`pylablib.core.fileio.loadfile_utils.`**`is_unprintable_character`**(*chn*)

`pylablib.core.fileio.loadfile_utils.`**`detect_binary_file`**(*stream*)
Check if the opened file is binary

`pylablib.core.fileio.loadfile_utils.`**`test_row_type`**(*line*)
Try to determine whether the line is a comment line, a numerical data row, a dictionary row or an unrecognized row.

Doesn't distinguish with a great accuracy; useful only for trying to guess file format.

`pylablib.core.fileio.loadfile_utils.`**`detect_textfile_type`**(*stream*)
Try to autodetect text file type: dictionary or table

`pylablib.core.fileio.loadfile_utils.`**`test_savetime_comment`**(*line*)
Test if the comment resembles a savetime line

`pylablib.core.fileio.loadfile_utils.`**`find_savetime_comment`**(*comments*)
Try to find savetime comment

`pylablib.core.fileio.loadfile_utils.`**`test_columns_line`**(*line*, *cols_num*)
Test if the line looks like a list of columns for a given columns number

`pylablib.core.fileio.loadfile_utils.`**`find_columns_lines`**(*corrupted*, *comments*, *cols_num*)
Try to find a column line (for a given columns number) among the comment and corrupted lines

class `pylablib.core.fileio.loadfile_utils.`**`InlineTable`**(*table*)
Bases: `object`

Simple marker class that denotes that the wrapped numpy 2D array should be written inline

`pylablib.core.fileio.loadfile_utils.`**`parse_dict_line`**(*line*)
Parse stripped dictionary file line

`pylablib.core.fileio.loadfile_utils.`**`read_dict_and_comments`**(*f*, *case_normalization=None*, *inline_dtype='generic'*, *allow_duplicate_keys=False*)
Load dictionary entries and comments from the file stream.

Parameters

- **f** – file stream
- **case_normalization** – case normalization for the returned dictionary; None means that it's case sensitive, "upper" and "lower" determine how they are normalized
- **inline_dtype** – dtype for inline tables; by default, use the most generic type (can include Python objects such as lists or strings)
- **allow_duplicate_keys** – if False and the same key is listed twice, raise and error

Return tuple (data, comment_lines), where data is a dictionary with parsed entries (tables are still represented as 'raw', i.e., as a tuple of columns list and column names list), and comment_lines is a list of comment lines

pylablib.core.fileio.location module

Classes for describing a generic file location.

class pylablib.core.fileio.location.**LocationName**(*path=None*, *ext=None*)

Bases: object

File name inside a location.

Parameters

- **path** – Path inside the location. Gets normalized according to the Dictionary rules (not case-sensitive; '/' and '\' are the delimiters).
- **ext** (*str*) – Name extension (None is default).

get_path(*default_path=''*, *sep='/'*)

Get the string path.

If the object's *path* is None, use *default_path* instead. If *sep* is not None, use it to join the path entries; otherwise, return the path in a list form.

get_ext(*default_ext=''*)

Get the extension.

If the object's *ext* is None, use *default_ext* instead.

to_string(*default_path=''*, *default_ext=''*, *path_sep='/'*, *ext_sep='|'*, *add_empty_ext=True*)

Convert the path to a string representation.

Parameters

- **default_path** (*str*) – Use it as path if the object's *path* is None.
- **default_ext** (*str*) – Use it as path if the object's *ext* is None.
- **path_sep** (*str*) – Use it to join the path entries.
- **ext_sep** (*str*) – Use it to join path and extension.
- **add_empty_ext** (*str*) – If False and the extension is empty, don't add *ext_sep* in the end.

to_path(*default_path=''*, *default_ext=''*, *ext_sep='|'*, *add_empty_ext=True*)

Convert the path to a list representation.

Extension is added with *ext_sep* to the last entry in the path.

Parameters

- **default_path** (*str*) – Use it as path if the object's *path* is None.

- **default_ext** (*str*) – Use it as path if the object's *ext* is `None`.
- **ext_sep** (*str*) – Use it to join path and extension.
- **add_empty_ext** (*str*) – If `False` and the extension is empty, don't add *ext_sep* in the end.

static from_string(*expr*, *ext_sep='|'*)
Create a *LocationName* object from a string representation.

ext_sep defines extension separator; the path separators are `'/'` and `'\'`. Empty path or extension translate into `None`.

static from_object(*obj*)
Create a *LocationName* object from an object.

obj can be a *LocationName* (return unchanged), tuple or list (use as construct arguments), string (treat as a string representation) or `None` (return empty name).

copy()

class pylablib.core.fileio.location.**LocationFile**(*loc*, *name=None*)
Bases: `object`

A file at a location.

Combines information about the location and the name within this location. Can be opened for reading or writing.

Parameters

- **loc** – File location.
- **name** – File's name inside the location.

loc
File location.

name
File's name inside the location.

opened
Whether the file is currently opened.

open(*mode='read'*, *data_type='text'*)
Open the file.

Parameters

- **mode** (*str*) – Opening mode. Can be `'read'`, `'write'` or `'append'`, as well as standard abbreviation (e.g., `"r"` or `"wb"`).
- **data_type** (*str*) – Either `'text'` or `'binary'`; if *mode* is an abbreviation, this parameter is ignored (i.e., `open("r","binary")` still opens file as text).

close()
Close the file

class pylablib.core.fileio.location.**IDataLocation**
Bases: `object`

Generic location.

is_free(*name=None*)
Check if the name is unoccupied

generate_new_name (*prefix_name*, *idx=0*)
Generate a new name inside the location using the given prefix and starting index.

If *idx* is `None`, check just the *prefix_name* first before starting to append indices.

open (*name=None*, *mode='read'*, *data_type='text'*)
Open a location file.

Parameters

- **name** – File name inside the location (`None` means 'default' location),
- **mode** (*str*) – Opening mode. Can be `'read'`, `'write'` or `'append'`, as well as standard abbreviation (e.g., `"r"` or `"wb"`).
- **data_type** (*str*) – Either `'text'` or `'binary'`; if *mode* is an abbreviation, this parameter is ignored (i.e., `open("r","binary")` still opens file as text).

close (*name*)
Close a location file.

list_opened_files ()
Get a dictionary `{string_name: location_file}` of all files opened in this location

class pylablib.core.fileio.location.**OpenedFileLocation** (*f*, *open_error=False*, *check_mode=False*, *check_data_type=True*)
Bases: `object`

File location which corresponds to an already opened file.

is_free (*name=None*)

generate_new_name (*prefix_name*, *idx=0*)

open (*name=None*, *mode='read'*, *data_type='text'*)

close (*name*)

list_opened_files ()

class pylablib.core.fileio.location.**IFileSystemDataLocation**
Bases: *pylablib.core.fileio.location.IDataLocation*

A generic filesystem data location.

A single file name describes a single file in the filesystem.

get_filesystem_path (*name=None*, *path_type='absolute'*)
Get the filesystem path corresponding to a given name.

path_type can be `'absolute'` (return absolute path), `'relative'` (return relative path; level depends on the location) or `'name'` (only return path inside the location).

is_free (*name=None*)
Check if the name is unoccupied

open (*name=None*, *mode='read'*, *data_type='text'*)
Open a location file.

Parameters

- **name** – File name inside the location (`None` means 'default' location),
- **mode** (*str*) – Opening mode. Can be `'read'`, `'write'` or `'append'`, as well as standard abbreviation (e.g., `"r"` or `"wb"`).

- **data_type** (*str*) – Either 'text' or 'binary'; if *mode* is an abbreviation, this parameter is ignored (i.e., open("r","binary") still opens file as text).

close(*name*)
Close a location file

list_opened_files()
Get a dictionary {string_name: location_file} of all files opened in this location

generate_new_name(*prefix_name*, *idx=0*)
Generate a new name inside the location using the given prefix and starting index.

If *idx* is None, check just the *prefix_name* first before starting to append indices.

class pylablib.core.fileio.location.**SingleFileSystemDataLocation**(*file_path*)
Bases: *pylablib.core.fileio.location.IFileSystemDataLocation*

A location describing a single file.

Any use of a non-default name raises ValueError.

Parameters **file_path** (*str*) – The path to the file.

get_filesystem_path(*name=None*, *path_type='absolute'*)
Get the filesystem path corresponding to a given name.

path_type can be 'absolute' (return absolute path), 'relative' (return relative path; level depends on the location) or 'name' (only return path inside the location).

close(*name*)
Close a location file

generate_new_name(*prefix_name*, *idx=0*)
Generate a new name inside the location using the given prefix and starting index.

If *idx* is None, check just the *prefix_name* first before starting to append indices.

is_free(*name=None*)
Check if the name is unoccupied

list_opened_files()
Get a dictionary {string_name: location_file} of all files opened in this location

open(*name=None*, *mode='read'*, *data_type='text'*)
Open a location file.

Parameters

- **name** – File name inside the location (None means 'default' location),
- **mode** (*str*) – Opening mode. Can be 'read', 'write' or 'append', as well as standard abbreviation (e.g., "r" or "wb").
- **data_type** (*str*) – Either 'text' or 'binary'; if *mode* is an abbreviation, this parameter is ignored (i.e., open("r","binary") still opens file as text).

class pylablib.core.fileio.location.**PrefixedFileSystemDataLocation**(*file_path*, *prefix_template='{0}_{1}'*)
Bases: *pylablib.core.fileio.location.IFileSystemDataLocation*

A location describing a set of prefixed files.

Parameters

- **file_path** (*str*) – A master path. Its name is used as a prefix, and its extension is used as a default.
- **prefix_template** (*str*) – A `str.format()` string for generating prefixed files. Has two arguments: the first is the master name, the second is the sub_location.

Multi-level paths translate into nested folders (the top level folder is combined from the *file_path* prefix and the first path entry).

get_filesystem_path (*name=None*, *path_type='absolute'*)
Get the filesystem path corresponding to a given name.

path_type can be `'absolute'` (return absolute path), `'relative'` (return relative path; level depends on the location) or `'name'` (only return path inside the location).

close (*name*)
Close a location file

generate_new_name (*prefix_name*, *idx=0*)
Generate a new name inside the location using the given prefix and starting index.

If *idx* is `None`, check just the *prefix_name* first before starting to append indices.

is_free (*name=None*)
Check if the name is unoccupied

list_opened_files ()
Get a dictionary `{string_name: location_file}` of all files opened in this location

open (*name=None*, *mode='read'*, *data_type='text'*)
Open a location file.

Parameters

- **name** – File name inside the location (`None` means 'default' location),
- **mode** (*str*) – Opening mode. Can be `'read'`, `'write'` or `'append'`, as well as standard abbreviation (e.g., `"r"` or `"wb"`).
- **data_type** (*str*) – Either `'text'` or `'binary'`; if *mode* is an abbreviation, this parameter is ignored (i.e., `open("r","binary")` still opens file as text).

class pylablib.core.fileio.location.**FolderFileSystemDataLocation** (*folder_path*, *default_name='content'*, *default_ext=''*)

Bases: *pylablib.core.fileio.location.IFileSystemDataLocation*

A location describing a single folder.

Parameters

- **folder_path** (*str*) – A path to the folder. Can also have one or two `'|'` symbols in the end (e.g., `'folder|file|dat'`), which separate default name and extension (overrides *default_name* and *default_ext* parameters)
- **default_name** (*str*) – The default file name.
- **default_ext** (*str*) – The default file extension.

Multi-level paths translate into nested subfolders.

get_filesystem_path (*name=None*, *path_type='absolute'*)
Get the filesystem path corresponding to a given name.

path_type can be `'absolute'` (return absolute path), `'relative'` (return relative path; level depends on the location) or `'name'` (only return path inside the location).

close (*name*)
Close a location file

generate_new_name (*prefix_name*, *idx=0*)
Generate a new name inside the location using the given prefix and starting index.

If *idx* is `None`, check just the *prefix_name* first before starting to append indices.

is_free (*name=None*)
Check if the name is unoccupied

list_opened_files ()
Get a dictionary `{string_name: location_file}` of all files opened in this location

open (*name=None*, *mode='read'*, *data_type='text'*)
Open a location file.

Parameters

- **name** – File name inside the location (`None` means 'default' location),
- **mode** (*str*) – Opening mode. Can be `'read'`, `'write'` or `'append'`, as well as standard abbreviation (e.g., `"r"` or `"wb"`).
- **data_type** (*str*) – Either `'text'` or `'binary'`; if *mode* is an abbreviation, this parameter is ignored (i.e., `open("r","binary")` still opens file as text).

`pylablib.core.fileio.location.`**get_location** (*path*, *loc*, **args*, ***kwargs*)
Build a location.

If *path* or *loc* are instances of *IDataLocation*, return them unchanged. If *loc* is a string, it describes location kind:

- `'single_file'`: *SingleFileSystemDataLocation* with the given *path*.
- `'file'` or `'prefixed_file'`: *PrefixedFileSystemDataLocation* with the given *path* as a master path.
- `'folder'`: *FolderFileSystemDataLocation* with the given folder *path*.

Any additional arguments are relayed to the constructors.

pylablib.core.fileio.parse_csv module

Utilities for parsing CSV files.

class `pylablib.core.fileio.parse_csv.`**ChunksAccumulator** (*dtype='numeric'*, *ignore_corrupted_lines=True*, *trim_rows=False*)
Bases: `object`

Class for accumulating data chunks into a single array.

Parameters

- **dtype** – dtype of entries; can be either a single type, or a list of types (one per column). Possible dtypes are: `'int'`, `'float'`, `'complex'`, `'numeric'` (tries to coerce to minimal possible numeric type, raises error if data can't be converted to complex), `'generic'` (accept arbitrary types, including lists, dictionaries, escaped strings, etc.), `'raw'` (keep raw string).

- **ignore_corrupted_lines** – if `True`, skip corrupted (e.g., non-numeric for numeric dtype, or with too few entries) lines; otherwise, raise `ValueError`.
- **trim_rows** – if `True` and the row length is larger than expected, drop extra entries; otherwise, treat the row as corrupted

corrupted_number()

convert_columns(*raw_columns*)
Convert raw columns into appropriate data structure (numpy array for numeric dtypes, list for generic and raw).

add_columns(*columns*)
Append columns (lists or numpy arrays) to the existing data.

add_chunk(*chunk*)
Add a chunk (2D list) to the pre-existing data.

pylablib.core.fileio.parse_csv.**read_columns**(*f*, *dtype*, *delimiters='\\s*, \\s*|\\s+'*, *empty_entry_substitute=None*, *ignore_corrupted_lines=True*, *trim_rows=False*, *stop_comment=None*)
Load columns from the file stream *f*.

Parameters

- **dtype** – dtype of entries; can be either a single type, or a list of types (one per column). Possible dtypes are: `'int'`, `'float'`, `'complex'`, `'numeric'` (tries to coerce to minimal possible numeric type, raises error if data can't be converted to complex), `'generic'` (accept arbitrary types, including lists, dictionaries, escaped strings, etc.), `'raw'` (keep raw string).
- **delimiters** (*str*) – Regex string which recognizes delimiters (by default `r"\s*,\s*|\s+"`, i.e., commas and whitespaces).
- **empty_entry_substitute** – Substitute for empty table entries. If `None`, all empty table entries are skipped.
- **ignore_corrupted_lines** – If `True`, skip corrupted (e.g., non-numeric for numeric dtype, or with too few entries) lines; otherwise, raise `ValueError`.
- **trim_rows** – if `True` and the row length is larger than expected, drop extra entries; otherwise, treat the row as corrupted
- **stop_comment** (*str*) – Regex string for the stopping comment. If not `None`. the function will stop if comment satisfying *stop_comment* regex is encountered.

Returns

`(columns, comments, corrupted_lines).`

columns is a list of columns with data.

comments is a list of comment strings.

corrupted_lines is a dict `{'size':list, 'type':list}` of corrupted lines (already split into entries), based on the corruption type (`'size'` means too small size, `'type'` means it couldn't be converted using provided dtype).

Return type tuple

pylablib.core.fileio.parse_csv.**columns_to_table**(*data*, *columns=None*, *dtype='numeric'*, *out_type='columns'*)

Convert *data* (columns list) into a table.

Parameters

- **columns** – either number if columns, or a list of columns names.
- **out_type** (*str*) – type of the result: `'array'` for numpy array, `'pandas'` for pandas DataFrame, `'columns'` for tuple `(data, columns)`

pylablib.core.fileio.parse_csv.**read_table**(*f*, *dtype='numeric'*, *columns=None*, *out_type='columns'*, *delimiters='\\s*, \\s*|\\s+'*, *empty_entry_substitute=None*, *ignore_corrupted_lines=True*, *trim_rows=False*, *stop_comment=None*)

Load table from the file stream *f*.

Arguments are the same as in `read_columns()` and `columns_to_table()`.

Returns

`(table, comments, corrupted_lines)`.

table is a table of the format *out_type*.

corrupted_lines is a dict `{'size':list, 'type':list}` of corrupted lines (already split into entries), based on the corruption type (`'size'` means too small size, `'type'` means it couldn't be converted using provided dtype).

comments is a list of comment strings.

Return type tuple

pylablib.core.fileio.savefile module

Utilities for writing data files.

class pylablib.core.fileio.savefile.**IOutputFileFormat**(*format_name*)

Bases: `object`

Generic class for an output file format.

Parameters **format_name** (*str*) – The name of the format (to be defined in subclasses).

write_file(*location_file*, *to_save*)

write_data(*location_file*, *data*)

write(*location_file*, *data*)

class pylablib.core.fileio.savefile.**ITextOutputFileFormat**(*format_name*, *save_props=True*, *save_comments=True*, *save_time=True*, *new_time=True*)

Bases: `pylablib.core.fileio.savefile.IOutputFileFormat`

Generic class for a text output file format.

Parameters

- **format_name** (*str*) – The name of the format (to be defined in subclasses).

- **save_props** (*bool*) – If True and saving *datafile.DataFile* object, save its props metainfo.
- **save_comments** (*bool*) – If True and saving *datafile.DataFile* object, save its comments metainfo.
- **save_time** (*bool*) – If True, append the file creation time in the end.
- **new_time** (*bool*) – If saving *datafile.DataFile* object, determines if the time should be updated to the current time.

make_comment_line (*comment*)

make_prop_line (*name*, *value*)

make_savetime_line (*time*)

static write_line (*stream*, *line*)

write_comments (*stream*, *comments*)

write_props (*stream*, *props*)

write_savetime (*stream*, *time*)

write_file (*location_file*, *to_save*)

write (*location_file*, *data*)

write_data (*location_file*, *data*)

class pylablib.core.fileio.savefile.**CSVTableOutputFileFormat** (*delimiters='t'*, *value_formats=None*, *use_rep_classes=False*, *save_columns=True*, *save_props=True*, *save_comments=True*, *save_time=True*)

Bases: *pylablib.core.fileio.savefile.ITextOutputFileFormat*

Class for CSV output file format.

Parameters

- **delimiters** (*str*) – Used to separate entries in a row.
- **value_formats** (*str*) – If not None, defines value formats to be passed to *utils.string.to_string()* function.
- **use_rep_classes** (*bool*) – If True, use representation classes for Dictionary entries (e.g., numpy arrays will be represented as "array([1, 2, 3])" instead of just "[1, 2, 3]"); This improves storage fidelity, but makes result harder to parse (e.g., by external string parsers).
- **save_columns** (*bool*) – If True, save column names as a comment line in the beginning of the file.
- **save_props** (*bool*) – If True and saving *datafile.DataFile* object, save its props metainfo.
- **save_comments** (*bool*) – If True and saving *datafile.DataFile* object, save its comments metainfo.
- **save_time** (*bool*) – If True, append the file creation time in the end.

get_table_line (*line*)

get_columns_line (*columns*)

write_data (*location_file*, *data*)
Write data to a CSV file.

Parameters

- **location_file** – Location of the destination.
- **data** – Data to be saved. Can be a pandas DataFrame or an arbitrary 2D array (numpy array, 2D list, etc.); if the data is not DataFrame or numpy 2D array, it gets converted into a DataFrame using the standard constructor (i.e., 2D list is interpreted as a list of rows)

make_comment_line (*comment*)

make_prop_line (*name*, *value*)

make_savetime_line (*time*)

write (*location_file*, *data*)

write_comments (*stream*, *comments*)

write_file (*location_file*, *to_save*)

static write_line (*stream*, *line*)

write_props (*stream*, *props*)

write_savetime (*stream*, *time*)

class pylablib.core.fileio.savefile.**DictionaryOutputFileFormat** (*param_formats=None*, *use_rep_classes=False*, *table_format='inline'*, *inline_delimiters='t'*, *inline_formats=None*, *save_props=True*, *save_comments=True*, *save_time=True*)

Bases: `pylablib.core.fileio.savefile.ITextOutputFileFormat`

Class for Dictionary output file format.

Parameters

- **param_formats** (*str*) – If not `None`, defines value formats to be passed to `utils.string.to_string()` function when writing Dictionary entries.
- **use_rep_classes** (*bool*) – If `True`, use representation classes for Dictionary entries (e.g., numpy arrays will be represented as `"array([1, 2, 3])"` instead of just `"[1, 2, 3]"`); This improves storage fidelity, but makes result harder to parse (e.g., by external string parsers).
- **table_format** (*str*) – Default format for table (numpy arrays or pandas DataFrames) entries. Can be `'inline'` (table is written inside the file), `'csv'` (external CSV file) or `'bin'` (external binary file).
- **inline_delimiters** (*str*) – Used to separate entries in a row for inline tables.
- **inline_formats** (*str*) – If not `None`, defines value formats to be passed to `utils.string.to_string()` function when writing inline tables.

- **save_props** (*bool*) – If True and saving *datafile.DataFile* object, save its props metainfo.
- **save_comments** (*bool*) – If True and saving *datafile.DataFile* object, save its comments metainfo.
- **save_time** (*bool*) – If True, append the file creation time in the end.

get_dictionary_line (*path, value*)

write_data (*loc_file, data*)
Write data to a Dictionary file.

Parameters

- **location_file** – Location of the destination.
- **data** – Data to be saved. Should be object of class *Dictionary*.

make_comment_line (*comment*)

make_prop_line (*name, value*)

make_savetime_line (*time*)

write (*location_file, data*)

write_comments (*stream, comments*)

write_file (*location_file, to_save*)

static write_line (*stream, line*)

write_props (*stream, props*)

write_savetime (*stream, time*)

class pylablib.core.fileio.savefile.**IBinaryOutputFileFormat** (*format_name*)
Bases: *pylablib.core.fileio.savefile.IOutputFileFormat*

get_preamble (*loc_file, data*)

write (*location_file, data*)

write_data (*location_file, data*)

write_file (*location_file, to_save*)

class pylablib.core.fileio.savefile.**TableBinaryOutputFileFormat** (*dtype=None, transposed=False*)
Bases: *pylablib.core.fileio.savefile.IBinaryOutputFileFormat*

Class for binary output file format.

Parameters

- **dtype** – a string with numpy dtype (e.g., "<f8") used to save the data. By default, use little-endian ("<") variant kind of the supplied data array dtype
- **transposed** (*bool*) – If False, write the data row-wise; otherwise, write it column-wise.

get_dtype (*table*)

get_preamble (*loc_file, data*)
Generate a preamble (dictionary describing the file format).

The parameters are `'dtype'`, `'packing'` (`'transposed'` or `'flatten'`, depending on the *transposed* attribute), `'ncol'` (number of columns) and `'nrows'` (number of rows).

write_data (*location_file*, *data*)

Write data to a binary file.

Parameters

- **location_file** – Location of the destination.
- **data** – Data to be saved. Can be a pandas DataFrame or an arbitrary 2D array (numpy array, 2D list, etc.) Converted to numpy array before saving.

write_file (*location_file*, *to_save*)

write (*location_file*, *data*)

`pylablib.core.fileio.savefile.`**get_output_format** (*data*, *output_format*, ***kwargs*)

`pylablib.core.fileio.savefile.`**save_csv** (*data*, *path*, *delimiters='\t'*, *value_formats=None*, *use_rep_classes=False*, *save_columns=True*, *save_props=True*, *save_comments=True*, *save_time=True*, *loc='file'*)

Save data to a CSV file.

Parameters

- **data** – Data to be saved (2D numpy array, pandas DataFrame, or a `datafile.DataFile` object containing this data).
- **path** (*str*) – Path to the file or a file-like object.
- **delimiters** (*str*) – Used to separate entries in a row.
- **value_formats** (*str*) – If not `None`, defines value formats to be passed to `utils.string.to_string()` function.
- **use_rep_classes** (*bool*) – If `True`, use representation classes for Dictionary entries (e.g., numpy arrays will be represented as `"array([1, 2, 3])"` instead of just `"[1, 2, 3]"`); This improves storage fidelity, but makes result harder to parse (e.g., by external string parsers).
- **save_columns** (*bool*) – If `True`, save column names as a comment line in the beginning of the file.
- **save_props** (*bool*) – If `True` and saving `datafile.DataFile` object, save its props metainfo.
- **save_comments** (*bool*) – If `True` and saving `datafile.DataFile` object, save its comments metainfo.
- **save_time** (*bool*) – If `True`, append the file creation time in the end.
- **loc** (*str*) – Location type.

`pylablib.core.fileio.savefile.`**save_csv_desc** (*data*, *path*, *loc='file'*)

Save data table to a dictionary file with an inlined table.

Compared to `save_csv()`, supports more pandas features (index, column multi-index), but can only be directly read by pylablib.

Parameters

- **data** – Data to be saved (2D numpy array, pandas DataFrame, or a `datafile.DataFile` object containing this data).

- **path** (*str*) – Path to the file or a file-like object.
- **loc** (*str*) – Location type.

`pylablib.core.fileio.savefile.`**`save_bin`**(*data, path, dtype=None, transposed=False, loc='file'*)

Save data to a binary file.

Parameters

- **data** – Data to be saved (2D numpy array, pandas DataFrame, or a `datafile.DataFile` object containing this data).
- **path** (*str*) – Path to the file or a file-like object.
- **dtype** – `numpy.dtype` describing the data. By default, use little-endian (`"<"`) variant kind of the supplied data array dtype.
- **transposed** (*bool*) – If `False`, write the data row-wise; otherwise, write it column-wise.
- **loc** (*str*) – Location type.

`pylablib.core.fileio.savefile.`**`save_bin_desc`**(*data, path, loc='file'*)

Save data to a binary file with an additional description file, which contains all of the data related to loading (shape, dtype, columns, etc.)

Parameters

- **data** – Data to be saved (2D numpy array, pandas DataFrame, or a `datafile.DataFile` object containing this data).
- **path** (*str*) – Path to the file or a file-like object.
- **loc** (*str*) – Location type.

`pylablib.core.fileio.savefile.`**`save_dict`**(*data, path, param_formats=None, use_rep_classes=False, table_format='inline', inline_delimiters='\t', inline_formats=None, save_props=True, save_comments=True, save_time=True, loc='file'*)

Save dictionary to a text file.

Parameters

- **data** – Data to be saved.
- **path** (*str*) – Path to the file or a file-like object.
- **param_formats** (*str*) – If not `None`, defines value formats to be passed to `utils.string.to_string()` function when writing Dictionary entries.
- **use_rep_classes** (*bool*) – If `True`, use representation classes for Dictionary entries (e.g., numpy arrays will be represented as `"array([1, 2, 3])"` instead of just `"[1, 2, 3]"`); This improves storage fidelity, but makes result harder to parse (e.g., by external string parsers).
- **table_format** (*str*) – Default format for table (numpy arrays or pandas DataFrames) entries. Can be `'inline'` (table is written inside the file), `'csv'` (external CSV file) or `'bin'` (external binary file).
- **inline_delimiters** (*str*) – Used to separate entries in a row for inline tables.
- **inline_formats** (*str*) – If not `None`, defines value formats to be passed to `utils.string.to_string()` function when writing inline tables.

- **save_props** (*bool*) – If True and saving *datafile.DataFile* object, save its props metainfo.
- **save_comments** (*bool*) – If True and saving *datafile.DataFile* object, save its comments metainfo.
- **save_time** (*bool*) – If True, append the file creation time in the end.
- **loc** (*str*) – Location type.

pylablib.core.fileio.savefile.**save_generic**(*data, path, output_format=None, loc='file', **kwargs*)

Save data to a file.

Parameters

- **data** – Data to be saved.
- **path** (*str*) – Path to the file or a file-like object.
- **output_format** (*str*) – Output file format. Can be either None (defaults to 'csv' for table data and 'dict' for Dictionary data), a string with one of the default format names, or an already prepared *IOutputFileFormat* object.
- **loc** (*str*) – Location type.

***kwargs* are passed to the file formatter constructor (see *CSVTableOutputFileFormat*, *DictionaryOutputFileFormat* and *TableBinaryOutputFileFormat* for the possible arguments). The default format names are:

- 'csv': CSV file, corresponds to *CSVTableOutputFileFormat* and *save_csv()*;
- 'csv': CSV file with an additional dictionary containing format description, corresponds to *DictionaryOutputFileFormat* and *save_csv_desc()*;
- 'bin': Binary file, corresponds to *TableBinaryOutputFileFormat* and *save_bin()*;
- 'bin_desc': Binary file with an additional dictionary containing format description, corresponds to *DictionaryOutputFileFormat* and *save_bin_desc()*;
- 'dict': Dictionary file, corresponds to *DictionaryOutputFileFormat* and *save_dict()*

pylablib.core.fileio.table_stream module

class pylablib.core.fileio.table_stream.**TableStreamFile**(*path, columns=None, delimiter='t', fmt=None, add_timestamp=False, header_prepend='# '*)

Bases: object

Expanding table file.

Can define column names and formats for different columns, and repeatedly write data into the same file. Useful for, e.g., continuous log files.

Parameters

- **path** (*str*) – Path to the destination file.
- **columns** (*list*) – If not None, it's a list of column names to be added as a header on creation.
- **delimiter** (*str*) – Values delimiter.

- **fmt** (*str*) – If not None, it's a list of format strings for the line entries (e.g., ".3f"); instead of format string one can also be None, which means using the standard *to_string()* conversion function
- **add_timestamp** (*bool*) – If True, add the UNIX timestamp in the beginning of each line (columns and format are expanded accordingly)
- **header_prepend** – the string to prepend to the header line; by default, a comment symbol, which is best compatibly with *loadfile.load_csv()* function

write_text_lines (*lines*)
Write several text lines into the file.

Create the file if it doesn't exist (in which case the header is automatically added).

Parameters **lines** (*[str]*) – List of lines to write.

write_row (*row*)
Write a single data row into the file.

Create the file if it doesn't exist (in which case the header is automatically added).

Parameters **data** (*list or numpy.ndarray*) – Data row to be added.

write_multiple_rows (*rows*)
Write a multiple data lines into the file.

Create the file if it doesn't exist (in which case the header is automatically added).

Parameters **rows** (*[list or numpy.ndarray]*) – Data rows to be added.

Module contents

pylablib.core.gui package

Subpackages

pylablib.core.gui.widgets package

Submodules

pylablib.core.gui.widgets.button module

class pylablib.core.gui.widgets.button.**ToggleButton** (*parent*)
Bases: object

Expanded toggle button.

Maintains internally stored consistent value (which can be, e.g., accessed from different threads). Allows setting different captions of pressed/unpressed, and uses those to represent values.

set_value_labels (*labels*)
Set a list of values corresponding to combo box indices.

Can be either a list of values, whose length must be equal to the number of options, or None (don't change the button label on toggle).

value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
Signal emitted when value is changed

get_value()
Get current numerical value

set_value(*value*, *notify_value_change=True*)
Set current value.

If `notify_value_change==True`, emit the *value_changed* signal; otherwise, change value silently.

repr_value(*value*)
Return representation of *value* as a combo box text

pylablib.core.gui.widgets.combo_box module

class pylablib.core.gui.widgets.combo_box.**ComboBox**(*parent*)
Bases: `object`

Expanded combo box.

Maintains internally stored consistent value (which can be, e.g., accessed from different threads). Allows setting values which are reported via `value_changed` signal instead of simple indices.

set_out_of_range(*action='error'*)
Set behavior when out-of-range value is applied.

Can be `"error"` (raise error), `"reset"` (reset to no-value position), or `"ignore"` (keep current value).

index_to_value(*idx*)
Turn numerical index into value

value_to_index(*value*)
Turn value into a numerical index

set_index_values(*index_values*, *value=None*)
Set a list of values corresponding to combo box indices.

Can be either a list of values, whose length must be equal to the number of options, or `None` (simply use indices). Note: if the number of combo box options changed (e.g., using `addItem` or `insertItem` methods), the index values need to be manually updated; otherwise, the errors might arise if the index is larger than the number of values.

set_options(*options*, *index_values=None*, *value=None*)
Set new set of options.

If *index_values* is not `None`, set these as the new index values; otherwise, index values are reset.

value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
Signal emitted when value is changed

get_value()
Get current numerical value

set_value(*value*, *notify_value_change=True*)
Set current value.

If `notify_value_change==True`, emit the *value_changed* signal; otherwise, change value silently.

repr_value(*value*)
Return representation of *value* as a combo box text

pylablib.core.gui.widgets.container module

class pylablib.core.gui.widgets.container.**TTimer**(*name*, *period*, *timer*)

Bases: `tuple`

count()

Return number of occurrences of value.

index()

Return first index of value.

Raises ValueError if the value is not present.

name

period

timer

class pylablib.core.gui.widgets.container.**TTimerEvent**(*start*, *loop*, *stop*, *timer*)

Bases: `tuple`

count()

Return number of occurrences of value.

index()

Return first index of value.

Raises ValueError if the value is not present.

loop

start

stop

timer

class pylablib.core.gui.widgets.container.**TChild**(*name*, *widget*, *gui_values_path*)

Bases: `tuple`

count()

Return number of occurrences of value.

gui_values_path

index()

Return first index of value.

Raises ValueError if the value is not present.

name

widget

class pylablib.core.gui.widgets.container.**IQContainer**(**args*, *name=None*, ***kwargs*)

Bases: `object`

Basic controller object which combines and controls several other widget.

Can either corresponds to a widget (e.g., a frame or a group box), or simply be an organizing entity.

Parameters **name** – entity name (used by default when adding this object to a values table)

Abstract mix-in class, which needs to be added to a class inheriting from `QObject`. Alternatively, one can directly use *QContainer*, which already inherits from `QObject`.

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

setup_name(*name*)
Set the object's name

setup(*name=None*)
Setup the container by initializing its GUI values and setting the `ctl` attribute

add_timer(*name*, *period*, *autostart=True*)
Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

start_timer(*name*)
Start the timer with the given name (also called automatically on `start()` method)

stop_timer(*name*)
Stop the timer with the given name (also called automatically on `stop()` method)

is_timer_running(*name*)
Check if the timer with the given name is running

add_timer_event(*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)
Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)
Add child's values to the container's table.

If *widget* is a container and `path==""` or ends in `"/*"` (e.g., `"subpath/*"`), use its `setup_gui_values` to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if `add_change_event==True`, changing of the widget's value emits the container's `contained_value_changed` event

add_child(*name*, *widget*, *gui_values_path=True*, *add_change_event=True*)
Add a contained child widget.

If *gui_values_path* is `False` or `None`, do not add it to the GUI values table; if it is `True`, add it under the same root (`path==""`) if it's a container, and under *name* if it's not; otherwise, `gui_values_path` specifies the path under which the widget values are stored. if `add_change_event==True`, changing of the widget's value emits the container's `contained_value_changed` event

get_child(*name*)
Get the child widget with the given name

remove_child(*name*)
Remove child from the container and clear it

add_virtual_element(*name*, *value=None*, *multivalued=False*, *add_indicator=True*)
Add a virtual value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get. If `add_indicator==True`, add default indicator handler as well.

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If `add_indicator==True`, add default (stored value) indicator handler as well.

start()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

stop()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

is_running()
Check if the container is running (started and not yet stopped)

clear()
Clear the container.

Stop all timers and widgets, and call `clear` methods of all contained widgets, remove all widgets from the values table, remove all widgets from the table.

get_handler(*name*)
Get value handler of a widget with the given name

get_widget(*name*)
Get a widget corresponding to a value with the given name

get_value(*name=None*)
Get value of a widget with the given name (`None` means all values)

get_all_values()
Get values of all widget in the container

set_value(*name*, *value*)
Set value of a widget with the given name (`None` means all values)

set_all_values(*value*)
Set values of all widgets in the container

get_value_changed_signal(*name*)
Get a value-changed signal for a widget with the given name

get_indicator(*name=None*)
Get indicator value for a widget with the given name (`None` means all indicators)

get_all_indicators()
Get indicator values of all widget in the container

set_indicator(*name*, *value*, *ignore_missing=False*)
Set indicator value for a widget or a branch with the given name

set_all_indicators(*value*, *ignore_missing=True*)

update_indicators()
Update all indicators to represent current values

class pylablib.core.gui.widgets.container.**QContainer**(**args*, *name=None*, ***kwargs*)
Bases: *pylablib.core.gui.widgets.container.IQContainer*, object

Basic controller object which combines and controls several other widget.

Can either corresponds to a widget (e.g., a frame or a group box), or simply be an organizing entity.

Parameters name – entity name (used by default when adding this object to a values table)

Simply a combination of *IQContainer* and QObject.

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

add_child(*name*, *widget*, *gui_values_path=True*, *add_change_event=True*)
Add a contained child widget.

If *gui_values_path* is False or None, do not add it to the GUI values table; if it is True, add it under the same root (path=="") if it's a container, and under *name* if it's not; otherwise, gui_values_path specifies the path under which the widget values are stored. if add_change_event==True, changing of the widget's value emits the container's contained_value_changed event

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)
Add child's values to the container's table.

If *widget* is a container and path=="" or ends in "/*" (e.g., "subpath/*"), use its setup_gui_values to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if add_change_event==True, changing of the widget's value emits the container's contained_value_changed event

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If add_indicator==True, add default (stored value) indicator handler as well.

add_timer(*name*, *period*, *autostart=True*)
Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If autostart==True and the container has been started (by calling *start()* method), start the timer as well.

add_timer_event(*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)
Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If autostart==True and the container has been started (by calling *start()* method), start the timer as well.

add_virtual_element(*name*, *value=None*, *multivalued=False*, *add_indicator=True*)
Add a virtual value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get. If add_indicator==True, add default indicator handler as well.

clear()
Clear the container.

Stop all timers and widgets, and call `clear` methods of all contained widgets, remove all widgets from the values table, remove all widgets from the table.

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

get_all_indicators()
Get indicator values of all widget in the container

get_all_values()
Get values of all widget in the container

get_child(*name*)
Get the child widget with the given name

get_handler(*name*)
Get value handler of a widget with the given name

get_indicator(*name=None*)
Get indicator value for a widget with the given name (`None` means all indicators)

get_value(*name=None*)
Get value of a widget with the given name (`None` means all values)

get_value_changed_signal(*name*)
Get a value-changed signal for a widget with the given name

get_widget(*name*)
Get a widget corresponding to a value with the given name

is_running()
Check if the container is running (started and not yet stopped)

is_timer_running(*name*)
Check if the timer with the given name is running

remove_child(*name*)
Remove child from the container and clear it

set_all_indicators(*value*, *ignore_missing=True*)

set_all_values(*value*)
Set values of all widgets in the container

set_indicator(*name*, *value*, *ignore_missing=False*)
Set indicator value for a widget or a branch with the given name

set_value(*name*, *value*)
Set value of a widget with the given name (`None` means all values)

setup(*name=None*)
Setup the container by initializing its GUI values and setting the `ctl` attribute

setup_name(*name*)
Set the object's name

start()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

start_timer(*name*)
Start the timer with the given name (also called automatically on `start()` method)

stop()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

stop_timer(*name*)
Stop the timer with the given name (also called automatically on *stop()* method)

update_indicators()
Update all indicators to represent current values

class pylablib.core.gui.widgets.container.**IQWidgetContainer**(**args*, ***kwargs*)
Bases: *pylablib.core.gui.widgets.layout_manager.IQLayoutManagedWidget*, *pylablib.core.gui.widgets.container.IQContainer*

Generic widget container.

Combines *IQContainer* management of GUI values and timers with *IQLayoutManagedWidget* management of the contained widget's layout.

Typically, adding widget adds them both to the container values and to the layout; however, this can be skipped by either using *QLayoutManagedWidget.add_to_layout()* (only add to the layout), or specifying `location="skip"` in *add_child()* (only add to the container).

Abstract mix-in class, which needs to be added to a class inheriting from `QWidget`. Alternatively, one can directly use *QWidgetContainer*, which already inherits from `QWidget`.

setup(*layout='vbox'*, *no_margins=False*, *name=None*)
Setup the layout.

Parameters

- **layout** – layout kind; can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).
- **no_margins** – if `True`, set all layout margins to zero (useful when the widget is in the middle of layout hierarchy)

add_child(*name*, *widget*, *location=None*, *gui_values_path=True*)
Add a contained child widget.

name specifies the child storage name; if `name==False`, only add the widget to they layout, but not to the container. *location* specifies the layout location to which the widget is added; if `location=="skip"`, skip adding it to the layout (can be manually added later). Note that if the widget is added to the layout, it will be completely deleted when `clear` or `remove_child` methods are called; otherwise, simply its `clear` method will be called, and its GUI values will be deleted.

If *gui_values_path* is `False` or `None`, do not add it to the GUI values table; if it is `True`, add it under the same root (`path==""`) if it's a container, and under *name* if it's not; otherwise, `gui_values_path` specifies the path under which the widget values are stored.

remove_child(*name*)
Remove widget from the container and the layout, clear it, and remove it

add_frame(*name*, *layout='vbox'*, *location=None*, *gui_values_path=True*, *no_margins=True*)
Add a new frame container to the layout.

layout specifies the layout (`"vbox"`, `"hbox"`, or `"grid"`) of the new frame, and *location* specifies its location within the container layout. If `no_margins==True`, the frame will have no inner layout margins. The other parameters are the same as in *add_child()* method.

add_group_box(*name*, *caption*, *layout='vbox'*, *location=None*, *gui_values_path=True*, *no_margins=True*)
Add a new group box container with the given *caption* to the layout.

layout specifies the layout ("vbox", "hbox", or "grid") of the new frame, and *location* specifies its location within the container layout. If no_margins==True, the frame will have no inner layout margins. The other parameters are the same as in add_child() method.

clear()

Clear the container.

All the timers are stopped, all the contained widgets are cleared and removed.

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)

Add child's values to the container's table.

If *widget* is a container and path=="" or ends in "/*" (e.g., "subpath/*"), use its setup_gui_values to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if add_change_event==True, changing of the widget's value emits the container's contained_value_changed event

add_decoration_label(*text*, *location='next'*)

Add a decoration text label with the given text

add_padding(*kind='auto'*, *location='next'*, *stretch=0*)

Add a padding (expandable spacer) of the given kind to the given location.

kind can be "vertical", "horizontal", "auto" (vertical for grid and vbox layouts, horizontal for hbox), or "both" (stretches in both directions). If *stretch* is not None, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); can also be a tuple with two stretches along vertical and horizontal directions.

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)

Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If add_indicator==True, add default (stored value) indicator handler as well.

add_spacer(*height=0*, *width=0*, *stretch_height=False*, *stretch_width=False*, *stretch=0*, *location='next'*)

Add a spacer with the given width and height to the given location.

If stretch_height==True or stretch_width==True, the widget will stretch in these directions; otherwise, the widget size is fixed. If *stretch* is not None, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); if *kind=="both"*', it can also be a tuple with two stretches along vertical and horizontal directions.

add_sublayout(*name*, *kind='grid'*, *location=None*)

Add a sublayout to the given location.

name specifies the sublayout name, which can be used to refer to it in specifying locations later. *kind* can be "grid", "vbox" (vertical single-column box), or "hbox" (horizontal single-row box).

add_timer(*name*, *period*, *autostart=True*)

Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If autostart==True and the container has been started (by calling start() method), start the timer as well.

add_timer_event(*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)

Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_to_layout(*element*, *location=None*, *kind='widget'*)
Add an existing *element* to the layout at the given *location*.

kind can be `"widget"` for widgets, `"layout"` for other layouts, or `"item"` for layout items (spacers).

add_virtual_element(*name*, *value=None*, *multivalued=False*, *add_indicator=True*)
Add a virtual value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get. If `add_indicator==True`, add default indicator handler as well.

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

get_all_indicators()
Get indicator values of all widget in the container

get_all_values()
Get values of all widget in the container

get_child(*name*)
Get the child widget with the given name

get_handler(*name*)
Get value handler of a widget with the given name

get_indicator(*name=None*)
Get indicator value for a widget with the given name (`None` means all indicators)

get_sublayout(*name=None*)
Get the previously added sublayout

get_sublayout_kind(*name=None*)
Get the kind of the previously added sublayout

get_value(*name=None*)
Get value of a widget with the given name (`None` means all values)

get_value_changed_signal(*name*)
Get a value-changed signal for a widget with the given name

get_widget(*name*)
Get a widget corresponding to a value with the given name

insert_column(*col*, *sublayout=None*, *stretch=0*)
Insert a new column at the given location in the grid layout

insert_row(*row*, *sublayout=None*, *stretch=0*)
Insert a new row at the given location in the grid layout

is_running()
Check if the container is running (started and not yet stopped)

is_timer_running(*name*)
Check if the timer with the given name is running

remove_layout_element(*element*)
Remove a previously added layout element

set_all_indicators(*value*, *ignore_missing=True*)

set_all_values(*value*)
Set values of all widgets in the container

set_column_stretch(**args*, *layout=None*)
Set column stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all columns.

set_indicator(*name*, *value*, *ignore_missing=False*)
Set indicator value for a widget or a branch with the given name

set_row_stretch(**args*, *layout=None*)
Set row stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all rows.

set_value(*name*, *value*)
Set value of a widget with the given name (`None` means all values)

setup_name(*name*)
Set the object's name

start()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

start_timer(*name*)
Start the timer with the given name (also called automatically on `start()` method)

stop()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

stop_timer(*name*)
Stop the timer with the given name (also called automatically on `stop()` method)

update_indicators()
Update all indicators to represent current values

using_layout(*name*)
Use a different sublayout as default inside the `with` block

using_new_sublayout(*name*, *kind='grid'*, *location=None*)
Create a different sublayout and use it as default inside the `with` block.

kind can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

class pylablib.core.gui.widgets.container.**QWidgetContainer**(**args*, ***kwargs*)
Bases: `pylablib.core.gui.widgets.container.IQWidgetContainer`, `object`

Generic widget container.

Combines `IQContainer` management of GUI values and timers with `IQLayoutManagedWidget` management of the contained widget's layout.

Typically, adding widget adds them both to the container values and to the layout; however, this can be skipped by either using `QLayoutManagedWidget.add_to_layout()` (only add to the layout), or specifying `location="skip"` in `add_child()` (only add to the container).

Simply a combination of `IQWidgetContainer` and `QWidget`.

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

add_child(*name*, *widget*, *location=None*, *gui_values_path=True*)
Add a contained child widget.

name specifies the child storage name; if `name==False`, only add the widget to they layout, but not to the container. *location* specifies the layout location to which the widget is added; if `location=="skip"`, skip adding it to the layout (can be manually added later). Note that if the widget is added to the layout, it will be completely deleted when `clear` or `remove_child` methods are called; otherwise, simply its `clear` method will be called, and its GUI values will be deleted.

If *gui_values_path* is `False` or `None`, do not add it to the GUI values table; if it is `True`, add it under the same root (`path==""`) if it's a container, and under *name* if it's not; otherwise, `gui_values_path` specifies the path under which the widget values are stored.

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)
Add child's values to the container's table.

If *widget* is a container and `path==""` or ends in `"/*"` (e.g., `"subpath/*"`), use its `setup_gui_values` to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if `add_change_event==True`, changing of the widget's value emits the container's `contained_value_changed` event

add_decoration_label(*text*, *location='next'*)
Add a decoration text label with the given text

add_frame(*name*, *layout='vbox'*, *location=None*, *gui_values_path=True*, *no_margins=True*)
Add a new frame container to the layout.

layout specifies the layout (`"vbox"`, `"hbox"`, or `"grid"`) of the new frame, and *location* specifies its location within the container layout. If `no_margins==True`, the frame will have no inner layout margins. The other parameters are the same as in `add_child()` method.

add_group_box(*name*, *caption*, *layout='vbox'*, *location=None*, *gui_values_path=True*, *no_margins=True*)
Add a new group box container with the given *caption* to the layout.

layout specifies the layout (`"vbox"`, `"hbox"`, or `"grid"`) of the new frame, and *location* specifies its location within the container layout. If `no_margins==True`, the frame will have no inner layout margins. The other parameters are the same as in `add_child()` method.

add_padding(*kind='auto'*, *location='next'*, *stretch=0*)
Add a padding (expandable spacer) of the given kind to the given location.

kind can be `"vertical"`, `"horizontal"`, `"auto"` (vertical for `grid` and `vbox` layouts, horizontal for `hbox`), or `"both"` (stretches in both directions). If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); can also be a tuple with two stretches along vertical and horizontal directions.

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If `add_indicator==True`, add default (stored value) indicator handler as well.

add_spacer(*height=0*, *width=0*, *stretch_height=False*, *stretch_width=False*, *stretch=0*, *location='next'*)
Add a spacer with the given width and height to the given location.

If `stretch_height==True` or `stretch_width==True`, the widget will stretch in these directions; otherwise, the widget size is fixed. If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); if *kind=="both"*‘, it can also be a tuple with two stretches along vertical and horizontal directions.

add_sublayout (*name*, *kind='grid'*, *location=None*)
Add a sublayout to the given location.

name specifies the sublayout name, which can be used to refer to it in specifying locations later. *kind* can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

add_timer (*name*, *period*, *autostart=True*)
Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_timer_event (*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)
Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_to_layout (*element*, *location=None*, *kind='widget'*)
Add an existing *element* to the layout at the given *location*.

kind can be `"widget"` for widgets, `"layout"` for other layouts, or `"item"` for layout items (spacers).

add_virtual_element (*name*, *value=None*, *multivalued=False*, *add_indicator=True*)
Add a virtual value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get. If `add_indicator==True`, add default indicator handler as well.

clear ()
Clear the container.

All the timers are stopped, all the contained widgets are cleared and removed.

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

get_all_indicators ()
Get indicator values of all widget in the container

get_all_values ()
Get values of all widget in the container

get_child (*name*)
Get the child widget with the given name

get_handler (*name*)
Get value handler of a widget with the given name

get_indicator (*name=None*)
Get indicator value for a widget with the given name (`None` means all indicators)

get_sublayout (*name=None*)
Get the previously added sublayout

get_sublayout_kind (*name=None*)
Get the kind of the previously added sublayout

get_value(*name=None*)
Get value of a widget with the given name (None means all values)

get_value_changed_signal(*name*)
Get a value-changed signal for a widget with the given name

get_widget(*name*)
Get a widget corresponding to a value with the given name

insert_column(*col*, *sublayout=None*, *stretch=0*)
Insert a new column at the given location in the grid layout

insert_row(*row*, *sublayout=None*, *stretch=0*)
Insert a new row at the given location in the grid layout

is_running()
Check if the container is running (started and not yet stopped)

is_timer_running(*name*)
Check if the timer with the given name is running

remove_child(*name*)
Remove widget from the container and the layout, clear it, and remove it

remove_layout_element(*element*)
Remove a previously added layout element

set_all_indicators(*value*, *ignore_missing=True*)

set_all_values(*value*)
Set values of all widgets in the container

set_column_stretch(**args*, *layout=None*)
Set column stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all columns.

set_indicator(*name*, *value*, *ignore_missing=False*)
Set indicator value for a widget or a branch with the given name

set_row_stretch(**args*, *layout=None*)
Set row stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all rows.

set_value(*name*, *value*)
Set value of a widget with the given name (None means all values)

setup(*layout='vbox'*, *no_margins=False*, *name=None*)
Setup the layout.

Parameters

- **layout** – layout kind; can be "grid", "vbox" (vertical single-column box), or "hbox" (horizontal single-row box).
- **no_margins** – if True, set all layout margins to zero (useful when the widget is in the middle of layout hierarchy)

setup_name(*name*)
Set the object's name

start()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

start_timer(*name*)
Start the timer with the given name (also called automatically on *start()* method)

stop()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

stop_timer(*name*)
Stop the timer with the given name (also called automatically on *stop()* method)

update_indicators()
Update all indicators to represent current values

using_layout(*name*)
Use a different sublayout as default inside the `with` block

using_new_sublayout(*name*, *kind='grid'*, *location=None*)
Create a different sublayout and use it as default inside the `with` block.

kind can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

class pylablib.core.gui.widgets.container.**QFrameContainer**(**args*, ***kwargs*)
Bases: *pylablib.core.gui.widgets.container.IQWidgetContainer*, `object`

An extension of *IQWidgetContainer* for a `QFrame` Qt base class

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

add_child(*name*, *widget*, *location=None*, *gui_values_path=True*)
Add a contained child widget.

name specifies the child storage name; if `name==False`, only add the widget to they layout, but not to the container. *location* specifies the layout location to which the widget is added; if `location=="skip"`, skip adding it to the layout (can be manually added later). Note that if the widget is added to the layout, it will be completely deleted when `clear` or `remove_child` methods are called; otherwise, simply its `clear` method will be called, and its GUI values will be deleted.

If *gui_values_path* is `False` or `None`, do not add it to the GUI values table; if it is `True`, add it under the same root (`path==""`) if it's a container, and under *name* if it's not; otherwise, `gui_values_path` specifies the path under which the widget values are stored.

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)
Add child's values to the container's table.

If *widget* is a container and `path==""` or ends in `"/*"` (e.g., `"subpath/*"`), use its `setup_gui_values` to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if `add_change_event==True`, changing of the widget's value emits the container's `contained_value_changed` event

add_decoration_label(*text*, *location='next'*)
Add a decoration text label with the given text

add_frame(*name*, *layout='vbox'*, *location=None*, *gui_values_path=True*, *no_margins=True*)
Add a new frame container to the layout.

layout specifies the layout (`"vbox"`, `"hbox"`, or `"grid"`) of the new frame, and *location* specifies its location within the container layout. If `no_margins==True`, the frame will have no inner layout margins. The other parameters are the same as in *add_child()* method.

add_group_box(*name*, *caption*, *layout='vbox'*, *location=None*, *gui_values_path=True*, *no_margins=True*)
Add a new group box container with the given *caption* to the layout.

layout specifies the layout ("vbox", "hbox", or "grid") of the new frame, and *location* specifies its location within the container layout. If no_margins==True, the frame will have no inner layout margins. The other parameters are the same as in add_child() method.

add_padding(*kind='auto'*, *location='next'*, *stretch=0*)
Add a padding (expandable spacer) of the given kind to the given location.

kind can be "vertical", "horizontal", "auto" (vertical for grid and vbox layouts, horizontal for hbox), or "both" (stretches in both directions). If *stretch* is not None, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); can also be a tuple with two stretches along vertical and horizontal directions.

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If add_indicator==True, add default (stored value) indicator handler as well.

add_spacer(*height=0*, *width=0*, *stretch_height=False*, *stretch_width=False*, *stretch=0*, *location='next'*)
Add a spacer with the given width and height to the given location.

If stretch_height==True or stretch_width==True, the widget will stretch in these directions; otherwise, the widget size is fixed. If *stretch* is not None, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); if *kind=="both"*‘, it can also be a tuple with two stretches along vertical and horizontal directions.

add_sublayout(*name*, *kind='grid'*, *location=None*)
Add a sublayout to the given location.

name specifies the sublayout name, which can be used to refer to it in specifying locations later. *kind* can be "grid", "vbox" (vertical single-column box), or "hbox" (horizontal single-row box).

add_timer(*name*, *period*, *autostart=True*)
Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If autostart==True and the container has been started (by calling start() method), start the timer as well.

add_timer_event(*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)
Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If autostart==True and the container has been started (by calling start() method), start the timer as well.

add_to_layout(*element*, *location=None*, *kind='widget'*)
Add an existing *element* to the layout at the given *location*.

kind can be "widget" for widgets, "layout" for other layouts, or "item" for layout items (spacers).

add_virtual_element(*name*, *value=None*, *multivalued=False*, *add_indicator=True*)
Add a virtual value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get. If `add_indicator==True`, add default indicator handler as well.

clear()
Clear the container.

All the timers are stopped, all the contained widgets are cleared and removed.

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

get_all_indicators()
Get indicator values of all widget in the container

get_all_values()
Get values of all widget in the container

get_child(*name*)
Get the child widget with the given name

get_handler(*name*)
Get value handler of a widget with the given name

get_indicator(*name=None*)
Get indicator value for a widget with the given name (`None` means all indicators)

get_sublayout(*name=None*)
Get the previously added sublayout

get_sublayout_kind(*name=None*)
Get the kind of the previously added sublayout

get_value(*name=None*)
Get value of a widget with the given name (`None` means all values)

get_value_changed_signal(*name*)
Get a value-changed signal for a widget with the given name

get_widget(*name*)
Get a widget corresponding to a value with the given name

insert_column(*col*, *sublayout=None*, *stretch=0*)
Insert a new column at the given location in the grid layout

insert_row(*row*, *sublayout=None*, *stretch=0*)
Insert a new row at the given location in the grid layout

is_running()
Check if the container is running (started and not yet stopped)

is_timer_running(*name*)
Check if the timer with the given name is running

remove_child(*name*)
Remove widget from the container and the layout, clear it, and remove it

remove_layout_element(*element*)
Remove a previously added layout element

set_all_indicators(*value*, *ignore_missing=True*)

set_all_values(*value*)
Set values of all widgets in the container

set_column_stretch(**args*, *layout=None*)
Set column stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all columns.

set_indicator(*name*, *value*, *ignore_missing=False*)
Set indicator value for a widget or a branch with the given name

set_row_stretch(**args*, *layout=None*)
Set row stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all rows.

set_value(*name*, *value*)
Set value of a widget with the given name (`None` means all values)

setup(*layout='vbox'*, *no_margins=False*, *name=None*)
Setup the layout.

Parameters

- **layout** – layout kind; can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).
- **no_margins** – if `True`, set all layout margins to zero (useful when the widget is in the middle of layout hierarchy)

setup_name(*name*)
Set the object's name

start()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

start_timer(*name*)
Start the timer with the given name (also called automatically on `start()` method)

stop()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

stop_timer(*name*)
Stop the timer with the given name (also called automatically on `stop()` method)

update_indicators()
Update all indicators to represent current values

using_layout(*name*)
Use a different sublayout as default inside the `with` block

using_new_sublayout(*name*, *kind='grid'*, *location=None*)
Create a different sublayout and use it as default inside the `with` block.

kind can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

class pylablib.core.gui.widgets.container.**QGroupBoxContainer**(**args*, ***kwargs*)
Bases: `pylablib.core.gui.widgets.container.IQWidgetContainer`, `object`

An extension of `IQWidgetContainer` for a `QGroupBox` Qt base class

setup(*caption=None*, *layout='vbox'*, *no_margins=False*, *name=None*)
Setup the layout.

Parameters

- **layout** – layout kind; can be "grid", "vbox" (vertical single-column box), or "hbox" (horizontal single-row box).
- **no_margins** – if True, set all layout margins to zero (useful when the widget is in the middle of layout hierarchy)

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

add_child(*name*, *widget*, *location=None*, *gui_values_path=True*)
Add a contained child widget.

name specifies the child storage name; if name==False, only add the widget to they layout, but not to the container. *location* specifies the layout location to which the widget is added; if location=="skip", skip adding it to the layout (can be manually added later). Note that if the widget is added to the layout, it will be completely deleted when clear or remove_child methods are called; otherwise, simply its clear method will be called, and its GUI values will be deleted.

If *gui_values_path* is False or None, do not add it to the GUI values table; if it is True, add it under the same root (path=="") if it's a container, and under *name* if it's not; otherwise, gui_values_path specifies the path under which the widget values are stored.

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)
Add child's values to the container's table.

If *widget* is a container and path=="" or ends in "/*" (e.g., "subpath/*"), use its setup_gui_values to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if add_change_event==True, changing of the widget's value emits the container's contained_value_changed event

add_decoration_label(*text*, *location='next'*)
Add a decoration text label with the given text

add_frame(*name*, *layout='vbox'*, *location=None*, *gui_values_path=True*, *no_margins=True*)
Add a new frame container to the layout.

layout specifies the layout ("vbox", "hbox", or "grid") of the new frame, and *location* specifies its location within the container layout. If no_margins==True, the frame will have no inner layout margins. The other parameters are the same as in add_child() method.

add_group_box(*name*, *caption*, *layout='vbox'*, *location=None*, *gui_values_path=True*, *no_margins=True*)
Add a new group box container with the given *caption* to the layout.

layout specifies the layout ("vbox", "hbox", or "grid") of the new frame, and *location* specifies its location within the container layout. If no_margins==True, the frame will have no inner layout margins. The other parameters are the same as in add_child() method.

add_padding(*kind='auto'*, *location='next'*, *stretch=0*)
Add a padding (expandable spacer) of the given kind to the given location.

kind can be "vertical", "horizontal", "auto" (vertical for grid and vbox layouts, horizontal for hbox), or "both" (stretches in both directions). If *stretch* is not None, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); can also be a tuple with two stretches along vertical and horizontal directions.

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If add_indicator==True, add default (stored value) indicator handler as well.

add_spacer(*height=0*, *width=0*, *stretch_height=False*, *stretch_width=False*, *stretch=0*, *location='next'*)
Add a spacer with the given width and height to the given location.

If `stretch_height==True` or `stretch_width==True`, the widget will stretch in these directions; otherwise, the widget size is fixed. If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); if *kind=="both"‘*, it can also be a tuple with two stretches along vertical and horizontal directions.

add_sublayout(*name*, *kind='grid'*, *location=None*)
Add a sublayout to the given location.

name specifies the sublayout name, which can be used to refer to it in specifying locations later. *kind* can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

add_timer(*name*, *period*, *autostart=True*)
Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_timer_event(*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)
Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_to_layout(*element*, *location=None*, *kind='widget'*)
Add an existing *element* to the layout at the given *location*.

kind can be `"widget"` for widgets, `"layout"` for other layouts, or `"item"` for layout items (spacers).

add_virtual_element(*name*, *value=None*, *multivalued=False*, *add_indicator=True*)
Add a virtual value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get. If `add_indicator==True`, add default indicator handler as well.

clear()
Clear the container.

All the timers are stopped, all the contained widgets are cleared and removed.

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

get_all_indicators()
Get indicator values of all widget in the container

get_all_values()
Get values of all widget in the container

get_child(*name*)
Get the child widget with the given name

get_handler(*name*)
Get value handler of a widget with the given name

get_indicator(*name=None*)
Get indicator value for a widget with the given name (`None` means all indicators)

get_sublayout(*name=None*)
Get the previously added sublayout

get_sublayout_kind(*name=None*)
Get the kind of the previously added sublayout

get_value(*name=None*)
Get value of a widget with the given name (`None` means all values)

get_value_changed_signal(*name*)
Get a value-changed signal for a widget with the given name

get_widget(*name*)
Get a widget corresponding to a value with the given name

insert_column(*col*, *sublayout=None*, *stretch=0*)
Insert a new column at the given location in the grid layout

insert_row(*row*, *sublayout=None*, *stretch=0*)
Insert a new row at the given location in the grid layout

is_running()
Check if the container is running (started and not yet stopped)

is_timer_running(*name*)
Check if the timer with the given name is running

remove_child(*name*)
Remove widget from the container and the layout, clear it, and remove it

remove_layout_element(*element*)
Remove a previously added layout element

set_all_indicators(*value*, *ignore_missing=True*)

set_all_values(*value*)
Set values of all widgets in the container

set_column_stretch(**args*, *layout=None*)
Set column stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all columns.

set_indicator(*name*, *value*, *ignore_missing=False*)
Set indicator value for a widget or a branch with the given name

set_row_stretch(**args*, *layout=None*)
Set row stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all rows.

set_value(*name*, *value*)
Set value of a widget with the given name (`None` means all values)

setup_name(*name*)
Set the object's name

start()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

start_timer(*name*)
Start the timer with the given name (also called automatically on `start()` method)

stop()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

stop_timer(*name*)
Stop the timer with the given name (also called automatically on `stop()` method)

update_indicators()
Update all indicators to represent current values

using_layout(*name*)
Use a different sublayout as default inside the `with` block

using_new_sublayout(*name*, *kind='grid'*, *location=None*)
Create a different sublayout and use it as default inside the `with` block.

kind can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

class pylablib.core.gui.widgets.container.**QTabContainer**(**args*, ***kwargs*)
Bases: `pylablib.core.gui.widgets.container.IQContainer`, `object`

Container which manages tab widget.

Does not have its own layout, but can add or remove tabs, which are represented as `QFrameContainer` widgets.

add_tab(*name*, *caption*, *index=None*, *widget=None*, *layout='vbox'*, *gui_values_path=True*, *no_margins=True*)
Add a new tab container with the given *caption* to the widget.

index specifies the new tab's index (`None` means adding to the end, negative values count from the end). If *widget* is `None`, create a new frame widget using the given *layout* (`"vbox"`, `"hbox"`, or `"grid"`) and *no_margins* (specifies whether the frame has inner margins) arguments; otherwise, use the supplied widget. The other parameters are the same as in `add_child()` method.

remove_tab(*name*)
Remove a tab with the given name.

Clear it, remove its GUI values, and delete it and all contained widgets.

clear()
Clear the container.

Stop all timers and widgets, and call `clear` methods of all contained widgets, remove all widgets from the values table, remove all widgets from the table.

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

add_child(*name*, *widget*, *gui_values_path=True*, *add_change_event=True*)
Add a contained child widget.

If *gui_values_path* is `False` or `None`, do not add it to the GUI values table; if it is `True`, add it under the same root (`path==""`) if it's a container, and under *name* if it's not; otherwise, `gui_values_path` specifies the path under which the widget values are stored. if `add_change_event==True`, changing of the widget's value emits the container's `contained_value_changed` event

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)
Add child's values to the container's table.

If *widget* is a container and `path==""` or ends in `"/*"` (e.g., `"subpath/*"`), use its `setup_gui_values` to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if `add_change_event==True`, changing of the widget's value emits the container's `contained_value_changed` event

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If `add_indicator==True`, add default (stored value) indicator handler as well.

add_timer(*name*, *period*, *autostart=True*)
Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_timer_event(*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)
Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_virtual_element(*name*, *value=None*, *multivalued=False*, *add_indicator=True*)
Add a virtual value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get. If `add_indicator==True`, add default indicator handler as well.

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

get_all_indicators()
Get indicator values of all widget in the container

get_all_values()
Get values of all widget in the container

get_child(*name*)
Get the child widget with the given name

get_handler(*name*)
Get value handler of a widget with the given name

get_indicator(*name=None*)
Get indicator value for a widget with the given name (`None` means all indicators)

get_value(*name=None*)
Get value of a widget with the given name (`None` means all values)

get_value_changed_signal(*name*)
Get a value-changed signal for a widget with the given name

get_widget(*name*)
Get a widget corresponding to a value with the given name

is_running()
Check if the container is running (started and not yet stopped)

is_timer_running(*name*)
Check if the timer with the given name is running

remove_child(*name*)
Remove child from the container and clear it

set_all_indicators(*value*, *ignore_missing=True*)

set_all_values(*value*)
Set values of all widgets in the container

set_indicator(*name*, *value*, *ignore_missing=False*)
Set indicator value for a widget or a branch with the given name

set_value(*name*, *value*)
Set value of a widget with the given name (`None` means all values)

setup(*name=None*)
Setup the container by initializing its GUI values and setting the `ctl` attribute

setup_name(*name*)
Set the object's name

start()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

start_timer(*name*)
Start the timer with the given name (also called automatically on `start()` method)

stop()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

stop_timer(*name*)
Stop the timer with the given name (also called automatically on `stop()` method)

update_indicators()
Update all indicators to represent current values

pylablib.core.gui.widgets.edit module

class pylablib.core.gui.widgets.edit.**TextEdit**(*parent*, *value=None*)
Bases: `object`

Expanded text edit.

Maintains internally stored consistent value (which can be, e.g., accessed from different threads).

keyPressEvent(*event*)

value_entered = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
Signal emitted when value is entered (regardless of whether it stayed the same)

value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
Signal emitted when value is changed

get_value()
Get current numerical value

show_value(*interrupt_edit=False*)
Display currently stored numerical value

If `interrupt_edit==True` and the edit is currently being modified by the user, don't update the display.

set_value(*value*, *notify_value_change=True*, *interrupt_edit=False*)
Set current numerical value.

If `notify_value_change==True`, emit the *value_changed* signal; otherwise, change value silently. If `interrupt_edit==True` and the edit is currently being modified by the user, don't update the display (but still update the internally stored value).

class `pylablib.core.gui.widgets.edit.`**NumEdit**(*parent*, *value=None*, *limiter=None*, *formatter=None*, *custom_steps=None*)
Bases: `object`

Labview-style numerical edit.

Maintains internally stored consistent value (which can be, e.g., accessed from different threads). Supports different number representations, metric prefixes (in input or output), keyboard shortcuts (up/down for changing number, escape for cancelling).

Parameters

- **parent** – parent widget
- **value** – initial value (`None` means no value is set)
- **limiter** – number limiter (for details, see *set_limiter()*)
- **formatter** – number formatter (for details, see *set_formatter()*)
- **custom_steps** – if not `None`, can specify custom fixed value steps when up/down keys are pressed with a modifier key (Control, Alt, or Shift) specifies a dictionary `{'ctrl':ctrl_step, 'alt':alt_step, 'shift':shift_step}` with the corresponding steps (missing elements mean that the modifier key is ignored)

keyPressEvent(*event*)

set_limiter(*limiter*, *new_value=None*)
Change current numerical limiter.

Limiter can be a callable object which takes a single value and either returns a limited value, or raises *limiter.LimitError* if it should be ignored; or it can be a tuple `(lower, upper, action, value_type)`, where `lower` and `upper` are the limits (`None` means no limits), `action` defines out-of-limit action (either `"ignore"` to ignore entered value, or `"coerce"` to truncate to the nearest limit), and `value_type` can be `None` (keep value as is), `"float"` (cast value to float), `"int"` (cast value to int). If the tuple is shorter, the missing parts are filled by default values `(None, None, "ignore", None)`.

set_formatter(*formatter*)
Change current numerical formatter.

Formatter can be a callable object turning value into a string, a string (`"float"`, `"int"`, or a format string, e.g., `".5f"`), or a tuple starting with `"float"` which contains arguments to the *formatter.FloatFormatter*.

set_float_formatter(*output_format='auto'*, *digits=9*, *add_trailing_zeros=True*, *leading_zeros=0*, *explicit_sign=False*)
Set up float formatter.

Has the same functionality as *set_formatter()* (i.e., `set_float_formatter(*args)` is equivalent to `set_formatter(("float",)+args)`), but explicitly lists the arguments.

Parameters

- **output_format** (*str*) – can be `"auto"` (use standard Python conversion), `"SI"` (use SI prefixes if possible), or `"sci"` (scientific "E" notation).
- **digits** (*int*) – if `add_trailing_zeros==False`, determines the number of significant digits; otherwise, determines precision (number of digits after decimal point).
- **add_trailing_zeros** (*bool*) – if `True`, always show fixed number of digits after the decimal point, with zero padding if necessary.
- **leading_zeros** (*bool*) – determines the minimal size of the integer part (before the decimal point) of the number; pads with zeros if necessary.
- **explicit_sign** (*bool*) – if `True`, always add explicit plus sign.

set_custom_steps(*custom_steps=None*)
Specify custom fixed value steps when up/down keys are pressed with a modifier key (Control, Alt, or Shift).

custom_steps is a dictionary `{'ctrl':ctrl_step, 'alt':alt_step, 'shift':shift_step}` with the corresponding steps (missing elements mean that the modifier key is ignored).

get_cursor_order()
Get a decimal order of the text cursor

set_cursor_order(*order*)
Move text cursor to a given decimal order

repr_value(*value*)
Return representation of *value* according to the current numerical format

value_entered = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
Signal emitted when value is entered (regardless of whether it stayed the same)

value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
Signal emitted when value is changed

get_value()
Get current numerical value

show_value(*interrupt_edit=False*, *preserve_cursor_order=True*)
Display currently stored numerical value

If `interrupt_edit==False` and the edit is currently being modified by the user, don't update the display. If `preserve_cursor_order==True` and the display value is being edited, keep the decimal order of the cursor position after change.

set_value(*value*, *notify_value_change=True*, *interrupt_edit=False*, *preserve_cursor_order=True*)
Set and display current numerical value.

If `notify_value_change==True`, emit the `value_changed` signal; otherwise, change value silently. If `interrupt_edit==False` and the edit is currently being modified by the user, don't update the display (but still update the internally stored value). If `preserve_cursor_order==True` and the display value is being edited, keep the decimal order of the cursor position after change.

pylablib.core.gui.widgets.label module

class pylablib.core.gui.widgets.label.**TextLabel**(*parent*, *value=None*)

Bases: object

Labview-style text label.

The main difference from the standard QLabel is the changed event.

value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

Signal emitted when value is changed

get_value()

Get current numerical value

set_value(*value*)

Set and display current text value

class pylablib.core.gui.widgets.label.**NumLabel**(*parent*, *value=None*, *limiter=None*, *formatter=None*, *allow_text=True*)

Bases: object

Labview-style numerical label.

Supports different number representations and metric prefixes.

Parameters

- **parent** – parent widget
- **value** – initial value (None means no value is set)
- **limiter** – number limiter (for details, see *set_limiter()*)
- **formatter** – number formatter (for details, see *set_formatter()*)
- **allow_text** – if True, can also take text values (which are displayed as is); otherwise, raise an error.

set_limiter(*limiter*, *new_value=None*)

Change current numerical limiter.

Limiter can be a callable object which takes a single value and either returns a limited value, or raises *limiter.LimitError* if it should be ignored; or it can be a tuple (lower, upper, action, value_type), where lower and upper are the limits (None means no limits), action defines out-of-limit action (either "ignore" to ignore entered value, or "coerce" to truncate to the nearest limit), and value_type can be None (keep value as is), "float" (cast value to float), "int" (cast value to int). If the tuple is shorter, the missing parts are filled by default values (None, None, "ignore", None).

set_formatter(*formatter*)

Change current numerical formatter.

Formatter can be a callable object turning value into a string, a string ("float", "int", or a format string, e.g., ".5f"), or a tuple starting with "float" which contains arguments to the *formatter.FloatFormatter*.

set_float_formatter(*output_format='auto'*, *digits=9*, *add_trailing_zeros=True*, *leading_zeros=0*, *explicit_sign=False*)

Set up float formatter.

Has the same functionality as *set_formatter()* (i.e., set_float_formatter(*args) is equivalent to set_formatter(("float",)+args)), but explicitly lists the arguments.

Parameters

- **output_format** (*str*) – can be "auto" (use standard Python conversion), "SI" (use SI prefixes if possible), or "sci" (scientific "E" notation).
- **digits** (*int*) – if add_trailing_zeros==False, determines the number of significant digits; otherwise, determines precision (number of digits after decimal point).
- **add_trailing_zeros** (*bool*) – if True, always show fixed number of digits after the decimal point, with zero padding if necessary.
- **leading_zeros** (*bool*) – determines the minimal size of the integer part (before the decimal point) of the number; pads with zeros if necessary.
- **explicit_sign** (*bool*) – if True, always add explicit plus sign.

repr_value (*value*)
Return representation of *value* according to the current numerical format

value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
Signal emitted when value is changed

get_value ()
Get current numerical value

set_value (*value*)
Set and display current numerical value

pylablib.core.gui.widgets.layout_manager module

class pylablib.core.gui.widgets.layout_manager.**IQLayoutManagedWidget** (**args*, ***kwargs*)

Bases: object

GUI widget which can manage layouts.

Typically, first it is set up using *setup()* method to specify the master layout kind; afterwards, widgets and sublayout can be added using *add_to_layout()*. In addition, it can directly add named sublayouts using *add_sublayout()* method.

Abstract mix-in class, which needs to be added to a class inheriting from QWidget. Alternatively, one can directly use *QLayoutManagedWidget*, which already inherits from QWidget.

setup (*layout='grid'*, *no_margins=False*)
Setup the layout.

Parameters

- **layout** – layout kind; can be "grid", "vbox" (vertical single-column box), or "hbox" (horizontal single-row box).
- **no_margins** – if True, set all layout margins to zero (useful when the widget is in the middle of layout hierarchy)

using_layout (*name*)
Use a different sublayout as default inside the with block

add_to_layout (*element*, *location=None*, *kind='widget'*)
Add an existing *element* to the layout at the given *location*.

kind can be "widget" for widgets, "layout" for other layouts, or "item" for layout items (spacers).

remove_layout_element(*element*)
Remove a previously added layout element

add_sublayout(*name*, *kind='grid'*, *location=None*)
Add a sublayout to the given location.

name specifies the sublayout name, which can be used to refer to it in specifying locations later. *kind* can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

using_new_sublayout(*name*, *kind='grid'*, *location=None*)
Create a different sublayout and use it as default inside the `with` block.

kind can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

get_sublayout(*name=None*)
Get the previously added sublayout

get_sublayout_kind(*name=None*)
Get the kind of the previously added sublayout

add_spacer(*height=0*, *width=0*, *stretch_height=False*, *stretch_width=False*, *stretch=0*, *location='next'*)
Add a spacer with the given width and height to the given location.

If `stretch_height==True` or `stretch_width==True`, the widget will stretch in these directions; otherwise, the widget size is fixed. If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); if *kind=="both"*`, it can also be a tuple with two stretches along vertical and horizontal directions.

add_padding(*kind='auto'*, *location='next'*, *stretch=0*)
Add a padding (expandable spacer) of the given kind to the given location.

kind can be `"vertical"`, `"horizontal"`, `"auto"` (vertical for `grid` and `vbox` layouts, horizontal for `hbox`), or `"both"` (stretches in both directions). If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); can also be a tuple with two stretches along vertical and horizontal directions.

set_row_stretch(**args*, *layout=None*)
Set row stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all rows.

set_column_stretch(**args*, *layout=None*)
Set column stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all columns.

add_decoration_label(*text*, *location='next'*)
Add a decoration text label with the given text

insert_row(*row*, *sublayout=None*, *stretch=0*)
Insert a new row at the given location in the grid layout

insert_column(*col*, *sublayout=None*, *stretch=0*)
Insert a new column at the given location in the grid layout

clear()
Clear the layout and remove all the added elements

class pylablib.core.gui.widgets.layout_manager.**QLayoutManagedWidget**(**args*, ***kwargs*)
Bases: *pylablib.core.gui.widgets.layout_manager.IQLayoutManagedWidget*, `object`

GUI widget which can manage layouts.

Typically, first it is set up using `setup()` method to specify the master layout kind; afterwards, widgets and sublayout can be added using `add_to_layout()`. In addition, it can directly add named sublayouts using `add_sublayout()` method.

Simply a combination of `IQLayoutManagedWidget` and `QWidget`.

add_decoration_label(*text*, *location='next'*)
Add a decoration text label with the given text

add_padding(*kind='auto'*, *location='next'*, *stretch=0*)
Add a padding (expandable spacer) of the given kind to the given location.

kind can be `"vertical"`, `"horizontal"`, `"auto"` (vertical for `grid` and `vbox` layouts, horizontal for `hbox`), or `"both"` (stretches in both directions). If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); can also be a tuple with two stretches along vertical and horizontal directions.

add_spacer(*height=0*, *width=0*, *stretch_height=False*, *stretch_width=False*, *stretch=0*, *location='next'*)
Add a spacer with the given width and height to the given location.

If `stretch_height==True` or `stretch_width==True`, the widget will stretch in these directions; otherwise, the widget size is fixed. If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); if *kind=="both"‘*, it can also be a tuple with two stretches along vertical and horizontal directions.

add_sublayout(*name*, *kind='grid'*, *location=None*)
Add a sublayout to the given location.

name specifies the sublayout name, which can be used to refer to it in specifying locations later. *kind* can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

add_to_layout(*element*, *location=None*, *kind='widget'*)
Add an existing *element* to the layout at the given *location*.

kind can be `"widget"` for widgets, `"layout"` for other layouts, or `"item"` for layout items (spacers).

clear()
Clear the layout and remove all the added elements

get_sublayout(*name=None*)
Get the previously added sublayout

get_sublayout_kind(*name=None*)
Get the kind of the previously added sublayout

insert_column(*col*, *sublayout=None*, *stretch=0*)
Insert a new column at the given location in the grid layout

insert_row(*row*, *sublayout=None*, *stretch=0*)
Insert a new row at the given location in the grid layout

remove_layout_element(*element*)
Remove a previously added layout element

set_column_stretch(**args*, *layout=None*)
Set column stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all columns.

set_row_stretch(**args*, *layout=None*)
Set row stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all rows.

setup (*layout='grid'*, *no_margins=False*)
Setup the layout.

Parameters

- **layout** – layout kind; can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).
- **no_margins** – if `True`, set all layout margins to zero (useful when the widget is in the middle of layout hierarchy)

using_layout (*name*)
Use a different sublayout as default inside the `with` block

using_new_sublayout (*name*, *kind='grid'*, *location=None*)
Create a different sublayout and use it as default inside the `with` block.

kind can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

pylablib.core.gui.widgets.param_table module

class pylablib.core.gui.widgets.param_table.**ParamTable** (*parent=None*, *name=None*)
Bases: *pylablib.core.gui.widgets.container.QWidgetContainer*

GUI parameter table.

Simplifies creating code-generated controls and displays table layouts.

Has methods for adding various kinds of controls (labels, edit boxes, combo boxes, check boxes), automatically creates values table for easy settings/getting. By default supports 2-column (label-control) and 3-column (label-control-indicator) layout, depending on the parameters given to `setup()`.

Similar to `GUIValues`, has three container-like accessor: `.h` for getting the value handler (i.e., `self.get_handler(name)` is equivalent to `self.h[name]`), `.w` for getting the underlying widget (i.e., `self.get_widget(name)` is equivalent to `self.w[name]`), `.v` for settings/getting values using the default getting method (equivalent to `.wv` if `cache_values=False` in `setup()`, and to `.cv` otherwise), `.wv` for settings/getting current current widget values without caching (i.e., `self.get_value(name)` is equivalent to `self.v[name]`, and `self.set_value(name, value)` is equivalent to `self.v[name]=value`), `.cv` for settings/getting values using cached value's table for getting (i.e., `self.current_values[name]` is equivalent to `self.cv[name]`, and `self.set_value(name, value)` is equivalent to `self.cv[name]=value`), (i.e., `self.get_value(name)` is equivalent to `self.v[name]`, and `self.set_value(name, value)` is equivalent to `self.v[name]=value`), `.i` for settings/getting indicator values (i.e., `self.get_indicator(name)` is equivalent to `self.i[name]`, and `self.set_indicator(name, value)` is equivalent to `self.i[name]=value`) `.vs` for getting the value changed Qt signal (i.e., `self.get_value_changed_signal(name)` is equivalent to `self.s[name]`),

Like most widgets, requires calling `setup()` to set up before usage.

Parameters **parent** – parent widget

setup (*name=None*, *add_indicator=True*, *gui_thread_safe=False*, *cache_values=False*, *change_focused_control=False*)
Setup the table.

Parameters

- **name** (*str*) – table widget name
- **add_indicator** (*bool*) – if `True`, add indicators for all added widgets by default.

- **gui_thread_safe** (*bool*) – if `True`, all value-access and indicator-access calls (`get/set_value`, `get/set_all_values`, `get/set_indicator`, and `update_indicators`) are automatically called in the GUI thread.
- **cache_values** (*bool*) – if `True` or `"update_one"`, store a dictionary with all the current values and update it every time a GUI value is changed; provides a thread-safe way to check current parameters without lag (unlike `get_value()` or `get_all_values()` with `gui_thread_safe==True`, which re-route calls to a GUI thread and may cause up to 100ms delay) can also be set to `"update_all"`, in which case change of any value will cause value update of all variables; otherwise, change of a value will only cause update of that same value (might potentially miss some value updates for custom controls).
- **change_focused_control** (*bool*) – if `False` and `set_value()` method is called while the widget has user focus, ignore the value; note that `set_all_values()` will still set the widget value.

add_sublayout (*name*, *kind='grid'*, *location=('next'*, *0*, *1*, *'end')*)
Add a sublayout to the given location.

name specifies the sublayout name, which can be used to refer to it in specifying locations later. *kind* can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

using_new_sublayout (*name*, *kind='grid'*, *location=('next'*, *0*, *1*, *'end')*)
Create a different sublayout and use it as default inside the `with` block.

kind can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

pad_borders (*kind='both'*, *stretch=0*)
Add expandable paddings on the bottom and/or right border.

kind can be `"bottom"`, `"right"`, `"both"`, or `"none"` (do nothing). Note that if more elements are added, they will be placed after the padding, so the table will be padded in the middle.

add_frame (*name*, *layout='vbox'*, *location=('next'*, *0*, *1*, *'end')*, *gui_values_path=True*, *no_margins=True*)
Add a new frame container to the layout.

layout specifies the layout (`"vbox"`, `"hbox"`, or `"grid"`) of the new frame, and *location* specifies its location within the container layout. If `no_margins==True`, the frame will have no inner layout margins. The other parameters are the same as in `add_child()` method.

add_group_box (*name*, *caption*, *layout='vbox'*, *location=('next'*, *0*, *1*, *'end')*, *gui_values_path=True*, *no_margins=True*)
Add a new group box container with the given *caption* to the layout.

layout specifies the layout (`"vbox"`, `"hbox"`, or `"grid"`) of the new frame, and *location* specifies its location within the container layout. If `no_margins==True`, the frame will have no inner layout margins. The other parameters are the same as in `add_child()` method.

class ParamRow (*widget*, *label*, *indicator*, *value_handler*, *indicator_handler*)
Bases: `tuple`

count ()
Return number of occurrences of value.

index ()
Return first index of value.

Raises ValueError if the value is not present.

indicator

indicator_handler

label

value_handler

widget

add_simple_widget(*name*, *widget*, *label=None*, *value_handler=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*)

Add a 'simple' (single-spaced, single-valued) widget to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **widget** – widget to add
- **label** (*str*) – if not `None`, specifies label to put in front of the widget in the layout
- **value_handler** – value handler of the widget; by default, use auto-detected value handler (works for many simple built-in or custom widgets)
- **add_indicator** – if `True`, add an indicator label in the third column and a corresponding indicator handler in the built-in values table; by default, use the default value supplied to `setup()`
- **location** (*tuple*) – tuple `(row, column)` specifying location of the widget (or widget label, if it is specified); by default, add to a new row in the end and into the first column can also be a string `"skip"`, which means that the widget is added to some other location manually later (this option only works if `label=None`, and doesn't add any indicator)
- **tooltip** – widget tooltip (mouseover text)
- **add_change_event** (*bool*) – if `True`, changing of the widget's value emits the table's `contained_value_changed` event

Return the widget's value handler

add_custom_widget(*name*, *widget*, *value_handler=None*, *indicator_handler=None*, *location=None*, *tooltip=None*, *add_change_event=True*)

Add a 'custom' (multi-spaced, possibly complex-valued) widget to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **widget** – widget to add
- **value_handler** – value handler of the widget; by default, use auto-detected value handler (works for many simple built-in or custom widgets)
- **indicator_handler** – indicator handler of the widget; by default, use auto-detected indicator handler (use `set/get_indicator` methods if present, or no indicator otherwise)
- **location** (*tuple*) – tuple `(row, column, rowspan, colspan)` specifying location of the widget; by default, add to a new row in the end and into the first column, span one row and all table columns can also be a string `"skip"`, which means that the widget is added to some other location manually later
- **add_change_event** (*bool*) – if `True`, changing of the widget's value emits the table's `contained_value_changed` event

Return the widget's value handler

remove_widget(*name*)
Remove the widget and, if applicable, its indicator and label

add_virtual_element(*name*, *value=None*, *add_indicator=None*)
Add a virtual table element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get.

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If `add_indicator==True`, add default (stored value) indicator handler as well.

add_button(*name*, *caption*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a button to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **caption** (*str*) – text on the button
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_toggle_button(*name*, *caption*, *value=False*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a toggle button to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **caption** (*str or list*) – text on the button; can be a single string, or a list of two strings which specifies the caption for off and on states
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_check_box(*name*, *caption*, *value=False*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a checkbox to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **caption** (*str*) – text on the checkbox
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_text_label(*name*, *value=''*, *label=None*, *location=None*, *tooltip=None*, *add_change_event=False*, *virtual=False*)
Add a text label to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_num_label(*name*, *value=0*, *limiter=None*, *formatter=None*, *label=None*, *tooltip=None*, *location=None*, *add_change_event=False*, *virtual=False*)
Add a numerical label to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*float*) – specifies initial value
- **limiter** (*tuple*) – tuple `(upper_limit, lower_limit, action, value_type)` specifying value limits; see `limiter.as_limiter()` for details
- **formatter** (*tuple*) – either `"int"` (for integer values), or tuple specifying floating value format; see `formatter.as_formatter()` for details
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_text_edit(*name*, *value=''*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a text edit to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_num_edit(*name*, *value=None*, *limiter=None*, *formatter=None*, *custom_steps=None*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a numerical edit to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **limiter** (*tuple*) – tuple `(upper_limit, lower_limit, action, value_type)` specifying value limits; see `NumEdit.set_limiter()` for details

- **formatter** (*tuple*) – either "int" (for integer values), or tuple specifying floating value format; see NumEdit.set_formatter() for details
- **custom_steps** – if not None, can specify custom fixed value steps when up/down keys are pressed with a modifier key (Control, Alt, or Shift) specifies a dictionary {'ctrl':ctrl_step, 'alt':alt_step, 'shift':shift_step} with the corresponding steps (missing elements mean that the modifier key is ignored)
- **virtual** (*bool*) – if True, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as add_simple_widget().

add_progress_bar(*name*, *value=None*, *label=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)

Add a progress bar to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if True, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as add_simple_widget().

add_combo_box(*name*, *value=None*, *options=None*, *index_values=None*, *out_of_range='reset'*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)

Add a combo box to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **options** (*list*) – list of string specifying box options
- **index_values** (*list*) – list of values corresponding to box options; if supplies, these number are used when setting/getting values or sending signals.
- **out_of_range** (*str*) – behavior when out-of-range value is applied; c an be "error" (raise error), "reset" (reset to no-value position), or "ignore" (keep current value).
- **virtual** (*bool*) – if True, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as add_simple_widget().

set_enabled(*names=None*, *enabled=True*, *include_indicator=True*, *include_label=True*)

Enable or disable widgets with the given names (by default, all widgets)

set_visible(*names=None*, *visible=True*, *include_indicator=True*, *include_label=True*)

Show or hide widgets with the given names (by default, all widgets)

get_value(*name=None*)

get_all_values()

set_value(*name*, *value*, *force=False*)
Set value of a widget with the given name.

If `force==True`, force widget value (e.g., ignoring restriction on not changing values of focused widgets)

set_all_values(*value*, *force=False*)
Set values of all widgets in the table.

If `force==True`, force widget value (e.g., ignoring restriction on not changing values of focused widgets)

get_widget(*name*)
Get a widget corresponding to a value with the given name

get_child(*name*)
Get the child widget with the given name

remove_child(*name*)
Remove widget from the container and the layout, clear it, and remove it

get_indicator(*name=None*)

get_all_indicators()

set_indicator(*name*, *value*, *ignore_missing=False*)

set_all_indicators(*value*, *ignore_missing=True*)

update_indicators()

clear(*disconnect=False*)
Clear the table (remove all widgets)

If `disconnect==True`, also disconnect all slots connected to the `contained_value_changed` signal.

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

add_child(*name*, *widget*, *location=None*, *gui_values_path=True*)
Add a contained child widget.

name specifies the child storage name; if `name==False`, only add the widget to they layout, but not to the container. *location* specifies the layout location to which the widget is added; if `location=="skip"`, skip adding it to the layout (can be manually added later). Note that if the widget is added to the layout, it will be completely deleted when `clear` or `remove_child` methods are called; otherwise, simply its `clear` method will be called, and its GUI values will be deleted.

If *gui_values_path* is `False` or `None`, do not add it to the GUI values table; if it is `True`, add it under the same root (`path==""`) if it's a container, and under *name* if it's not; otherwise, `gui_values_path` specifies the path under which the widget values are stored.

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)
Add child's values to the container's table.

If *widget* is a container and `path==""` or ends in `"/*"` (e.g., `"subpath/*"`), use its `setup_gui_values` to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if `add_change_event==True`, changing of the widget's value emits the container's `contained_value_changed` event

add_decoration_label(*text*, *location='next'*)
Add a decoration text label with the given text

add_padding(*kind='auto'*, *location='next'*, *stretch=0*)
Add a padding (expandable spacer) of the given kind to the given location.

kind can be "vertical", "horizontal", "auto" (vertical for grid and vbox layouts, horizontal for hbox), or "both" (stretches in both directions). If *stretch* is not None, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); can also be a tuple with two stretches along vertical and horizontal directions.

add_spacer(*height=0*, *width=0*, *stretch_height=False*, *stretch_width=False*, *stretch=0*, *location='next'*)
Add a spacer with the given width and height to the given location.

If stretch_height==True or stretch_width==True, the widget will stretch in these directions; otherwise, the widget size is fixed. If *stretch* is not None, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); if *kind=="both"*', it can also be a tuple with two stretches along vertical and horizontal directions.

add_timer(*name*, *period*, *autostart=True*)
Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If autostart==True and the container has been started (by calling start() method), start the timer as well.

add_timer_event(*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)
Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If autostart==True and the container has been started (by calling start() method), start the timer as well.

add_to_layout(*element*, *location=None*, *kind='widget'*)
Add an existing *element* to the layout at the given *location*.

kind can be "widget" for widgets, "layout" for other layouts, or "item" for layout items (spacers).

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

get_handler(*name*)
Get value handler of a widget with the given name

get_sublayout(*name=None*)
Get the previously added sublayout

get_sublayout_kind(*name=None*)
Get the kind of the previously added sublayout

get_value_changed_signal(*name*)
Get a value-changed signal for a widget with the given name

insert_column(*col*, *sublayout=None*, *stretch=0*)
Insert a new column at the given location in the grid layout

insert_row(*row*, *sublayout=None*, *stretch=0*)
Insert a new row at the given location in the grid layout

is_running()
Check if the container is running (started and not yet stopped)

is_timer_running (*name*)
Check if the timer with the given name is running

remove_layout_element (*element*)
Remove a previously added layout element

set_column_stretch (**args, layout=None*)
Set column stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all columns.

set_row_stretch (**args, layout=None*)
Set row stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all rows.

setup_name (*name*)
Set the object's name

start ()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

start_timer (*name*)
Start the timer with the given name (also called automatically on `start()` method)

stop ()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

stop_timer (*name*)
Stop the timer with the given name (also called automatically on `stop()` method)

using_layout (*name*)
Use a different sublayout as default inside the `with` block

class pylablib.core.gui.widgets.param_table.**StatusTable** (*parent=None, name=None*)
Bases: `pylablib.core.gui.widgets.param_table.ParamTable`

Expansion of `ParamTable` which adds status lines, which automatically subscribe to signals and update values.

add_status_line (*name, label=None, srcs=None, tags=None, filt=None, fmt=None*)
Add a status line to the table:

Parameters

- **name** (`str`) – widget name (used to reference its value in the values table)
- **label** (`str`) – if not `None`, specifies label to put in front of the status line
- **srcs** (`list`) – status signal sources
- **tags** (`list`) – status signal tags
- **filt** (`list`) – filter function for the signals
- **fmt** – if not `None`, specifies a function which takes 3 arguments (signal source, tag, and value) and generates a status line text.

class ParamRow (*widget, label, indicator, value_handler, indicator_handler*)
Bases: `tuple`

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

indicator

indicator_handler

label

value_handler

widget

TimerUIDGenerator = <pylablib.core.utils.general.NamedUIDGenerator object>

add_button(*name*, *caption*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a button to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **caption** (*str*) – text on the button
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as *add_simple_widget()*.

add_check_box(*name*, *caption*, *value=False*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a checkbox to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **caption** (*str*) – text on the checkbox
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as *add_simple_widget()*.

add_child(*name*, *widget*, *location=None*, *gui_values_path=True*)
Add a contained child widget.

name specifies the child storage name; if `name==False`, only add the widget to they layout, but not to the container. *location* specifies the layout location to which the widget is added; if `location=="skip"`, skip adding it to the layout (can be manually added later). Note that if the widget is added to the layout, it will be completely deleted when `clear` or `remove_child` methods are called; otherwise, simply its `clear` method will be called, and its GUI values will be deleted.

If *gui_values_path* is `False` or `None`, do not add it to the GUI values table; if it is `True`, add it under the same root (`path==""`) if it's a container, and under *name* if it's not; otherwise, `gui_values_path` specifies the path under which the widget values are stored.

add_child_values(*name*, *widget*, *path*, *add_change_event=True*)
Add child's values to the container's table.

If *widget* is a container and `path==""` or ends in `"/*"` (e.g., `"subpath/*"`), use its `setup_gui_values` to make it share the same GUI values; otherwise, simply add it to the GUI values under the given path. if `add_change_event==True`, changing of the widget's value emits the container's `contained_value_changed` event

add_combo_box(*name*, *value=None*, *options=None*, *index_values=None*, *out_of_range='reset'*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a combo box to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **options** (*list*) – list of string specifying box options
- **index_values** (*list*) – list of values corresponding to box options; if supplies, these number are used when setting/getting values or sending signals.
- **out_of_range** (*str*) – behavior when out-of-range value is applied; c an be `"error"` (raise error), `"reset"` (reset to no-value position), or `"ignore"` (keep current value).
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_custom_widget(*name*, *widget*, *value_handler=None*, *indicator_handler=None*, *location=None*, *tooltip=None*, *add_change_event=True*)
Add a 'custom' (multi-spaced, possibly complex-valued) widget to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **widget** – widget to add
- **value_handler** – value handler of the widget; by default, use auto-detected value handler (works for many simple built-in or custom widgets)
- **indicator_handler** – indicator handler of the widget; by default, use auto-detected indicator handler (use `set/get_indicator` methods if present, or no indicator otherwise)
- **location** (*tuple*) – tuple `(row, column, rowspan, colspan)` specifying location of the widget; by default, add to a new row in the end and into the first column, span one row and all table columns can also be a string `"skip"`, which means that the widget is added to some other location manually later
- **add_change_event** (*bool*) – if `True`, changing of the widget's value emits the table's `contained_value_changed` event

Return the widget's value handler

add_decoration_label(*text*, *location='next'*)
Add a decoration text label with the given text

add_frame(*name*, *layout='vbox'*, *location=('next', 0, 1, 'end')*, *gui_values_path=True*, *no_margins=True*)
Add a new frame container to the layout.

layout specifies the layout (`"vbox"`, `"hbox"`, or `"grid"`) of the new frame, and *location* specifies its location within the container layout. If `no_margins==True`, the frame will have no inner layout margins. The other parameters are the same as in `add_child()` method.

add_group_box(*name*, *caption*, *layout='vbox'*, *location=('next', 0, 1, 'end')*, *gui_values_path=True*, *no_margins=True*)
Add a new group box container with the given *caption* to the layout.

layout specifies the layout (`"vbox"`, `"hbox"`, or `"grid"`) of the new frame, and *location* specifies its location within the container layout. If `no_margins==True`, the frame will have no inner layout margins. The other parameters are the same as in `add_child()` method.

add_num_edit(*name*, *value=None*, *limiter=None*, *formatter=None*, *custom_steps=None*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a numerical edit to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **limiter** (*tuple*) – tuple `(upper_limit, lower_limit, action, value_type)` specifying value limits; see `NumEdit.set_limiter()` for details
- **formatter** (*tuple*) – either `"int"` (for integer values), or tuple specifying floating value format; see `NumEdit.set_formatter()` for details
- **custom_steps** – if not `None`, can specify custom fixed value steps when up/down keys are pressed with a modifier key (Control, Alt, or Shift) specifies a dictionary `{'ctrl':ctrl_step, 'alt':alt_step, 'shift':shift_step}` with the corresponding steps (missing elements mean that the modifier key is ignored)
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_num_label(*name*, *value=0*, *limiter=None*, *formatter=None*, *label=None*, *tooltip=None*, *location=None*, *add_change_event=False*, *virtual=False*)
Add a numerical label to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*float*) – specifies initial value
- **limiter** (*tuple*) – tuple `(upper_limit, lower_limit, action, value_type)` specifying value limits; see `limiter.as_limiter()` for details
- **formatter** (*tuple*) – either `"int"` (for integer values), or tuple specifying floating value format; see `formatter.as_formatter()` for details
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_padding(*kind='auto'*, *location='next'*, *stretch=0*)
Add a padding (expandable spacer) of the given kind to the given location.

kind can be `"vertical"`, `"horizontal"`, `"auto"` (vertical for `grid` and `vbox` layouts, horizontal for `hbox`), or `"both"` (stretches in both directions). If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); can also be a tuple with two stretches along vertical and horizontal directions.

add_progress_bar(*name*, *value=None*, *label=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a progress bar to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as *add_simple_widget()*.

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If `add_indicator==True`, add default (stored value) indicator handler as well.

add_simple_widget(*name*, *widget*, *label=None*, *value_handler=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*)
Add a 'simple' (single-spaced, single-valued) widget to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **widget** – widget to add
- **label** (*str*) – if not `None`, specifies label to put in front of the widget in the layout
- **value_handler** – value handler of the widget; by default, use auto-detected value handler (works for many simple built-in or custom widgets)
- **add_indicator** – if `True`, add an indicator label in the third column and a corresponding indicator handler in the built-in values table; by default, use the default value supplied to *setup()*
- **location** (*tuple*) – tuple `(row, column)` specifying location of the widget (or widget label, if it is specified); by default, add to a new row in the end and into the first column can also be a string `"skip"`, which means that the widget is added to some other location manually later (this option only works if `label=None`, and doesn't add any indicator)
- **tooltip** – widget tooltip (mouseover text)
- **add_change_event** (*bool*) – if `True`, changing of the widget's value emits the table's `contained_value_changed` event

Return the widget's value handler

add_spacer(*height=0*, *width=0*, *stretch_height=False*, *stretch_width=False*, *stretch=0*, *location='next'*)
Add a spacer with the given width and height to the given location.

If `stretch_height==True` or `stretch_width==True`, the widget will stretch in these directions; otherwise, the widget size is fixed. If *stretch* is not `None`, it specifies stretch of the spacer the corresponding direction (applied to the upper row and leftmost column for multi-cell spacer); if *kind=="both"‘*, it can also be a tuple with two stretches along vertical and horizontal directions.

add_sublayout (*name*, *kind='grid'*, *location=('next', 0, 1, 'end')*)
Add a sublayout to the given location.

name specifies the sublayout name, which can be used to refer to it in specifying locations later. *kind* can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

add_text_edit (*name*, *value=''*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)
Add a text edit to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_text_label (*name*, *value=''*, *label=None*, *location=None*, *tooltip=None*, *add_change_event=False*, *virtual=False*)
Add a text label to the table.

Parameters

- **name** (*str*) – widget name (used to reference its value in the values table)
- **value** (*bool*) – specifies initial value
- **virtual** (*bool*) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_timer (*name*, *period*, *autostart=True*)
Add a periodic timer with the given *name* and *period*.

Rarely needs to be called explicitly (one is created automatically if timer event is created). If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_timer_event (*name*, *loop=None*, *start=None*, *stop=None*, *period=None*, *timer=None*, *autostart=True*)
Add timer event with the given *name*.

Add an event which should be called periodically (e.g., a GUI update). Internally implemented through Qt timers. *loop*, *start* and *stop* are the functions called, correspondingly, on timer (periodically), when timer is start, and when it's finished. One can either specify the timer by name (*timer* parameter), or create a new one with the given *period*. If `autostart==True` and the container has been started (by calling `start()` method), start the timer as well.

add_to_layout (*element*, *location=None*, *kind='widget'*)
Add an existing *element* to the layout at the given *location*.

kind can be `"widget"` for widgets, `"layout"` for other layouts, or `"item"` for layout items (spacers).

add_toggle_button(*name*, *caption*, *value=False*, *label=None*, *add_indicator=None*, *location=None*, *tooltip=None*, *add_change_event=True*, *virtual=False*)

Add a toggle button to the table.

Parameters

- **name** (`str`) – widget name (used to reference its value in the values table)
- **caption** (`str or list`) – text on the button; can be a single string, or a list of two strings which specifies the caption for off and on states
- **value** (`bool`) – specifies initial value
- **virtual** (`bool`) – if `True`, the widget is not added, and a virtual handler is added instead

Rest of the arguments and the return value are the same as `add_simple_widget()`.

add_virtual_element(*name*, *value=None*, *add_indicator=None*)

Add a virtual table element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get.

clear(*disconnect=False*)

Clear the table (remove all widgets)

If `disconnect==True`, also disconnect all slots connected to the `contained_value_changed` signal.

contained_value_changed = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

get_all_indicators()

get_all_values()

get_child(*name*)

Get the child widget with the given name

get_handler(*name*)

Get value handler of a widget with the given name

get_indicator(*name=None*)

get_sublayout(*name=None*)

Get the previously added sublayout

get_sublayout_kind(*name=None*)

Get the kind of the previously added sublayout

get_value(*name=None*)

get_value_changed_signal(*name*)

Get a value-changed signal for a widget with the given name

get_widget(*name*)

Get a widget corresponding to a value with the given name

insert_column(*col*, *sublayout=None*, *stretch=0*)

Insert a new column at the given location in the grid layout

insert_row(*row*, *sublayout=None*, *stretch=0*)

Insert a new row at the given location in the grid layout

is_running()
Check if the container is running (started and not yet stopped)

is_timer_running(*name*)
Check if the timer with the given name is running

pad_borders(*kind='both'*, *stretch=0*)
Add expandable paddings on the bottom and/or right border.

kind can be `"bottom"`, `"right"`, `"both"`, or `"none"` (do nothing). Note that if more elements are added, they will be placed after the padding, so the table will be padded in the middle.

remove_child(*name*)
Remove widget from the container and the layout, clear it, and remove it

remove_layout_element(*element*)
Remove a previously added layout element

remove_widget(*name*)
Remove the widget and, if applicable, its indicator and label

set_all_indicators(*value*, *ignore_missing=True*)

set_all_values(*value*, *force=False*)
Set values of all widgets in the table.

If `force==True`, force widget value (e.g., ignoring restriction on not changing values of focused widgets)

set_column_stretch(**args*, *layout=None*)
Set column stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all columns.

set_enabled(*names=None*, *enabled=True*, *include_indicator=True*, *include_label=True*)
Enable or disable widgets with the given names (by default, all widgets)

set_indicator(*name*, *value*, *ignore_missing=False*)

set_row_stretch(**args*, *layout=None*)
Set row stretch for a given layout.

Takes either two arguments `index` and `stretch`, or a single list of stretches for all rows.

set_value(*name*, *value*, *force=False*)
Set value of a widget with the given name.

If `force==True`, force widget value (e.g., ignoring restriction on not changing values of focused widgets)

set_visible(*names=None*, *visible=True*, *include_indicator=True*, *include_label=True*)
Show or hide widgets with the given names (by default, all widgets)

setup(*name=None*, *add_indicator=True*, *gui_thread_safe=False*, *cache_values=False*, *change_focused_control=False*)
Setup the table.

Parameters

- **name** (*str*) – table widget name
- **add_indicator** (*bool*) – if `True`, add indicators for all added widgets by default.
- **gui_thread_safe** (*bool*) – if `True`, all value-access and indicator-access calls (`get/set_value`, `get/set_all_values`, `get/set_indicator`, and `update_indicators`) are automatically called in the GUI thread.

- **cache_values** (*bool*) – if `True` or `"update_one"`, store a dictionary with all the current values and update it every time a GUI value is changed; provides a thread-safe way to check current parameters without lag (unlike `get_value()` or `get_all_values()` with `gui_thread_safe==True`, which re-route calls to a GUI thread and may cause up to 100ms delay) can also be set to `"update_all"`, in which case change of any value will cause value update of all variables; otherwise, change of a value will only cause update of that same value (might potentially miss some value updates for custom controls).
- **change_focused_control** (*bool*) – if `False` and `set_value()` method is called while the widget has user focus, ignore the value; note that `set_all_values()` will still set the widget value.

setup_name (*name*)
Set the object's name

start ()
Start the container.

Starts all the internal timers, and calls `start` method for all the contained widgets.

start_timer (*name*)
Start the timer with the given name (also called automatically on `start()` method)

stop ()
Stop the container.

Stops all the internal timers, and calls `stop` method for all the contained widgets.

stop_timer (*name*)
Stop the timer with the given name (also called automatically on `stop()` method)

update_indicators ()

using_layout (*name*)
Use a different sublayout as default inside the `with` block

using_new_sublayout (*name, kind='grid', location=('next', 0, 1, 'end')*)
Create a different sublayout and use it as default inside the `with` block.

kind can be `"grid"`, `"vbox"` (vertical single-column box), or `"hbox"` (horizontal single-row box).

Module contents

Submodules

pylablib.core.gui.formatter module

`pylablib.core.gui.formatter.`**parse_float** (*s*)
Parse string as a float, with metric prefixes recognition.

Return tuple `(sign, integer, dot, fractional, exponent, prefix)`, where each entry has structure `(begin, end, text)`. Return `None` if string is unrecognizable.

`pylablib.core.gui.formatter.`**pos_to_order** (*s, pos*)
For a given string representation of a float and position in the string, get the decimal order for this position.

Return `None` if string is un-parsable or position is out of range (not in mantissa section of the number).

`pylablib.core.gui.formatter.`**`order_to_pos`**(*s*, *order*)

For a given string representation of float and decimal order, get the position in the string corresponding to this order.

If order is out of range for a given representation, truncates to most/least significant digit position. Return `None` if string is un-parsable.

`pylablib.core.gui.formatter.`**`str_to_float`**(*s*)

Return float value of a string, with metric prefixes recognition.

Raise `ValueError` if string is unrecognizable.

`pylablib.core.gui.formatter.`**`is_integer`**(*n*, *tolerance=0.0*)

Check if *n* is less than *tolerance* away from the nearest integer.

`pylablib.core.gui.formatter.`**`float_to_str_SI`**(*n*, *digits=3*, *trailing_zeros=False*)

Represent float using SI metric prefixes.

For orders `>=27` and `<-24` use usual scientific notation with order being multiple of 3. If `trailing_zeros==True`, then digits define precision, rather than number significant digits

`class` `pylablib.core.gui.formatter.`**`FloatFormatter`**(*output_format='auto'*, *digits=3*, *add_trailing_zeros=True*, *leading_zeros=0*, *explicit_sign=False*)

Bases: `object`

Floating point number formatter.

Callable object with takes a number as an argument and returns is string representation.

Parameters

- **`output_format`** (*`str`*) – can be `"auto"` (use standard Python conversion), `"SI"` (use SI prefixes if possible), or `"sci"` (scientific "E" notation).
- **`digits`** (*`int`*) – if `add_trailing_zeros==False`, determines the number of significant digits; otherwise, determines precision (number of digits after decimal point).
- **`add_trailing_zeros`** (*`bool`*) – if `True`, always show fixed number of digits after the decimal point, with zero padding if necessary.
- **`leading_zeros`** (*`bool`*) – determines the minimal size of the integer part (before the decimal point) of the number; pads with zeros if necessary.
- **`explicit_sign`** (*`bool`*) – if `True`, always add explicit plus sign.

`class` `pylablib.core.gui.formatter.`**`IntegerFormatter`**

Bases: `object`

Simple integer number formatter.

Callable object with takes a number as an argument and returns is string representation.

For more flexibility (e.g., adding leading zeros) it is possible to use *`FloatFormatter`* with `digits=0` and `add_trailing_zeros=True`.

`class` `pylablib.core.gui.formatter.`**`FmtStringFormatter`**(*fmt*)

Bases: `object`

Formatter based on format string.

Callable object with takes a number as an argument and returns is string representation.

`pylablib.core.gui.formatter.`**`as_formatter`**(*formatter*)

Turn an object into a formatter.

Can be a callable object turning value into a string, a string ("float", "int", or a format string, e.g., ".5f"), or a tuple starting with "float" which contains arguments to the *FloatFormatter*.

pylablib.core.gui.limiter module

exception pylablib.core.gui.limiter.**LimitError**(*value*, *lower_limit=None*, *upper_limit=None*)

Bases: ArithmeticError

Error raised when the value is out of limits and can't be coerced

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.core.gui.limiter.**NumberLimit**(*lower_limit=None*, *upper_limit=None*, *action='coerce'*, *value_type=None*)

Bases: object

Number limiter, which checks validity of user inputs.

Callable object with takes a number as an argument and either returns its coerced version (or the number itself, if it is within limits), or raises *LimitError* if it should be ignored.

Parameters

- **lower_limit** – lower limit (inclusive), or None if there is no limit.
- **upper_limit** – upper limit (inclusive), or None if there is no limit.
- **action** (*str*) – action taken if the number is out of limits; either "coerce" (return the closest valid value), or "ignore" (raise *LimitError*).
- **value_type** (*str*) – determines value type coercion; can be None (do nothing, only check limits), "float" (cast to float), or "int" (cast to integer).

cast(*value*)

pylablib.core.gui.limiter.**filter_limiter**(*pred*)

Turn a predicate into a limiter.

Returns a function that raises *LimitError* if the predicate is false.

pylablib.core.gui.limiter.**as_limiter**(*limiter*)

Turn an object into a limiter.

Limiter can be a callable object which takes a single value and either returns a limited value, or raises *LimitError* if it should be ignored; or it can be a tuple (lower, upper, action, value_type), where lower and upper are the limits (None means no limits), action defines out-of-limit action (either "ignore" to ignore entered value, or "coerce" to truncate to the nearest limit), and value_type can be None (keep value as is), "float" (cast value to float), "int" (cast value to int). If the tuple is shorter, the missing parts are filled by default values (None, None, "ignore", None).

pylablib.core.gui.utils module

pylablib.core.gui.utils.**find_layout_element**(*layout*, *element*)

Find a layout element.

Can be a widget, a sublayout, or a layout element Return item index within the layout. If layout is empty or item is not present, return None

`pylablib.core.gui.utils.`**`delete_layout_item`**(*layout*, *idx*)
Remove and item with the given index (completely delete it)

`pylablib.core.gui.utils.`**`clean_layout`**(*layout*, *delete_layout=False*)
Delete all items from the layout.

If `delete_layout==True`, delete the layout as well.

`pylablib.core.gui.utils.`**`is_layout_row_empty`**(*layout*, *row*)
Check if the given row in a grid layout is empty

`pylablib.core.gui.utils.`**`get_last_filled_row`**(*layout*, *start_row=0*)
Find the last non-empty row in a grid layout after *start_row* (inclusive).

If all rows after (and including) *start_row* are empty, return `None` .

`pylablib.core.gui.utils.`**`get_first_empty_row`**(*layout*, *start_row=0*)
Find the first completely empty row in a grid layout after *start_row* (inclusive)

`pylablib.core.gui.utils.`**`insert_layout_row`**(*layout*, *row*, *stretch=0*, *compress=False*)
Insert row in a grid layout at a given index.

Any multi-column item spanning over the row (i.e., starting at least one row before *row* and ending at least one row after *row*) gets stretched. Anything else either stays in place (if it's above *row*), or gets moved one row down. *stretch* determines the stretch factor of the new row. If `compress==True`, try to find an empty row below the inserted position and shit it to the new row's place; otherwise, add a completely new row.

`pylablib.core.gui.utils.`**`is_layout_column_empty`**(*layout*, *col*)
Check if the given column in a grid layout is empty

`pylablib.core.gui.utils.`**`get_last_filled_column`**(*layout*, *start_col=0*)
Find the last non-empty column in a grid layout after *start_col* (inclusive).

If all rows after (and including) *start_col* are empty, return `None` .

`pylablib.core.gui.utils.`**`get_first_empty_column`**(*layout*, *start_col=0*)
Find the first completely empty column in a grid layout after *start_col* (inclusive)

`pylablib.core.gui.utils.`**`insert_layout_column`**(*layout*, *col*, *stretch=0*, *compress=False*)
Insert column in a grid layout at a given index.

Any multi-row item spanning over the column (i.e., starting at least one column before *col* and ending at least one column after *col*) gets stretched. Anything else either stays in place (if it's above *col*), or gets moved one column to the right. *stretch* determines the stretch factor of the new column. If `compress==True`, try to find an empty column below the inserted position and shit it to the new column's place; otherwise, add a completely new column.

`pylablib.core.gui.utils.`**`compress_grid_layout`**(*layout*)
Find all empty rows in a grid layout and shift them to the bottom

pylablib.core.gui.value_handling module

Uniform representation of values from different widgets: numerical and text edits and labels, combo and check boxes, buttons.

`pylablib.core.gui.value_handling.`**`build_children_tree`**(*root*, *types_include*, *is_atomic=None*, *is_excluded=None*, *self_node='#'*)

`pylablib.core.gui.value_handling.`**`has_methods`**`(`*widget*, *methods_sets*`)`
Chick if the widget has methods from given set.

methods_sets is a list of method sets. The function returns `True` if the widget has at least one method from each of the sets.

`pylablib.core.gui.value_handling.`**`get_method_kind`**`(`*method*, *add_args=0*`)`
Determine whether the method takes name as its argument

add_args specifies number of additional required arguments. Return `"named"` is the method has at least `add_args+1` arguments, and the first one is called `"name"`. Otherwise, return `"simple"`.

exception `pylablib.core.gui.value_handling.`**`NoParameterError`**
Bases: `KeyError`

Error raised by some handlers to indicate that the parameter is missing

`args`

`with_traceback``()`
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `pylablib.core.gui.value_handling.`**`IValueHandler`**`(`*widget*`)`
Bases: `object`

Generic handler of a widget value.

Has method to get and set the value (or all values, if the widget has internal value structure), representing values as strings, and value changed signal.

Parameters **`widget`** – handled widget.

`get_value``(`*name=None*`)`
Get widget value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

`set_value``(`*value*, *name=None*`)`
Set widget value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

`repr_value``(`*value*, *name=None*`)`
Return textual representation of the value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

`get_handler``(`*name=None*`)`
Get handler of a contained widget (or same widget, if `name==None`)

`get_value_changed_signal``()`
Get the Qt signal emitted when the value is changed

`connect_value_changed_handler``(`*handler*, *only_signal=True*`)`
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *`get_value_changed_signal()`* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *`get_value_changed_signal()`* directly.

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters focus – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

class pylablib.core.gui.value_handling.**VirtualValueHandler**(*value=None*, *multivalued=False*)
Bases: *pylablib.core.gui.value_handling.IValueHandler*

Virtual value handler (to simulate controls which are not present in the GUI).

Parameters

- **value** – initial value
- **multivalued** (*bool*) – if `True`, the internal value is assumed to be complex, so it is forced to be a *Dictionary* every time it is set.

get_value(*name=None*)
Get widget value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

set_value(*value*, *name=None*)
Set widget value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters focus – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value_changed_signal()
Get the Qt signal emitted when the value is changed

repr_value(*value*, *name=None*)
Return textual representation of the value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

class pylablib.core.gui.value_handling.**PropertyValueHandler**(*getter=None*, *setter=None*, *default_name=None*)
Bases: *pylablib.core.gui.value_handling.IValueHandler*

Virtual value handler which uses custom getter/setter methods to simulate a value.

If getter or setter are not supplied but are called, they raise *NoParameterError*; this means that they are ignored in *GUIValues.get_all_values()* and *GUIValues.set_all_values()* methods, but raise an error when access directly (e.g., using *GUIValues.get_value()*).

Parameters

- **getter** – value getter method; takes 0 or 1 (name) arguments and returns the value
- **setter** – value setter method; takes 1 (value) or 2 (name and value) arguments and sets the value
- **default_name** (*str*) – default name to be supplied to `getter` and `setter` methods if they require a name argument

get_value(*name=None*)
Get widget value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

set_value(*value*, *name=None*)
Set widget value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters **focus** – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value_changed_signal()
Get the Qt signal emitted when the value is changed

repr_value(*value*, *name=None*)
Return textual representation of the value.

If `name` is not `None`, it specifies the name of the value parameter inside the widget (for complex widgets).

class pylablib.core.gui.value_handling.**StandardValueHandler**(*widget*, *default_name=None*)
Bases: *pylablib.core.gui.value_handling.IValueHandler*

Standard value handler, typically used for custom widgets.

To implement getting and setting values, looks for `get/set_value` and `get/set_all_values` methods for the widget and uses them accordingly. To implement value representing, looks for `repr_value` method (if not defined, use simple string conversion). To implement value change signal, looks for `value_changed` widget signal.

Parameters

- **widget** – handled widget
- **default_name** (*str*) – default name to be supplied to get/set_value and get/set_all_values methods if they require a name argument.

get_value(*name=None*)
Get widget value.

If name is not None, it specifies the name of the value parameter inside the widget (for complex widgets).

set_value(*value*, *name=None*)
Set widget value.

If name is not None, it specifies the name of the value parameter inside the widget (for complex widgets).

repr_value(*value*, *name=None*)
Return textual representation of the value.

If name is not None, it specifies the name of the value parameter inside the widget (for complex widgets).

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if name==None)

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters **focus** – if False, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)
Connect value changed signal.

If only_signal==True, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if only_signal==False, it also works for some objects (e.g., QLabel) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_value_changed_signal()
Get the Qt signal emitted when the value is changed

class pylablib.core.gui.value_handling.**ISingleValueHandler**(*widget*)
Bases: *pylablib.core.gui.value_handling.IValueHandler*

Base class for single-value widget handler, typically used for built-in Qt widgets.

Defines new functions get/set_single_value which don't take a name argument; raises an error if the name is supplied to any of the standard functions.

Parameters **widget** – handled widget

get_single_value()
Get the widget value

get_value(*name=None*)
Get widget value.

If name is not None raise an error.

set_single_value(*value*)
Set the widget value

set_value(*value*, *name=None*)
Set widget value.

If `name` is not `None` raise an error.

repr_single_value(*value*)
Represent the widget value as a string

repr_value(*value*, *name=None*)
Return textual representation of the value.

If `name` is not `None` raise an error.

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters **focus** – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to `get_value_changed_signal()` signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use `get_value_changed_signal()` directly.

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value_changed_signal()
Get the Qt signal emitted when the value is changed

class pylablib.core.gui.value_handling.**LineEditValueHandler**(*widget*)
Bases: `pylablib.core.gui.value_handling.ISingleValueHandler`

Value handler for `QLineEdit` widget

get_single_value()
Get the widget value

set_single_value(*value*)
Set the widget value

get_value_changed_signal()
Get the Qt signal emitted when the value is changed

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters **focus** – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to `get_value_changed_signal()` signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value(*name=None*)
Get widget value.

If `name` is not `None` raise an error.

repr_single_value(*value*)
Represent the widget value as a string

repr_value(*value, name=None*)
Return textual representation of the value.

If `name` is not `None` raise an error.

set_value(*value, name=None*)
Set widget value.

If `name` is not `None` raise an error.

class pylablib.core.gui.value_handling.**LabelValueHandler**(*widget*)
Bases: *pylablib.core.gui.value_handling.ISingleValueHandler*

Value handler for `QLabel` widget

get_single_value()
Get the widget value

set_single_value(*value*)
Set the widget value

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters **focus** – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler, only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value(*name=None*)
Get widget value.

If `name` is not `None` raise an error.

get_value_changed_signal()
Get the Qt signal emitted when the value is changed

repr_single_value (*value*)
Represent the widget value as a string

repr_value (*value*, *name=None*)
Return textual representation of the value.

If `name` is not `None` raise an error.

set_value (*value*, *name=None*)
Set widget value.

If `name` is not `None` raise an error.

class pylablib.core.gui.value_handling.**IBoolValueHandler** (*widget*, *labels=('Off', 'On')*)
Bases: *pylablib.core.gui.value_handling.ISingleValueHandler*

Generic value handler for widgets with boolean values

repr_single_value (*value*)
Represent the widget value as a string

can_set_value (*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters focus – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler (*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler (*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_single_value ()
Get the widget value

get_value (*name=None*)
Get widget value.

If `name` is not `None` raise an error.

get_value_changed_signal ()
Get the Qt signal emitted when the value is changed

repr_value (*value*, *name=None*)
Return textual representation of the value.

If `name` is not `None` raise an error.

set_single_value (*value*)
Set the widget value

set_value (*value*, *name=None*)
Set widget value.

If `name` is not `None` raise an error.

class pylablib.core.gui.value_handling.**CheckboxValueHandler**(*widget*, *labels=('Off', 'On')*)

Bases: *pylablib.core.gui.value_handling.IBoolValueHandler*

Value handler for `QCheckBox` widget

get_single_value()

Get the widget value

set_single_value(*value*)

Set the widget value

get_value_changed_signal()

Get the Qt signal emitted when the value is changed

can_set_value(*allow_focus=True*)

Check if setting value from the code is allowed.

Parameters **focus** – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)

Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler(*name=None*)

Get handler of a contained widget (or same widget, if `name==None`)

get_value(*name=None*)

Get widget value.

If `name` is not `None` raise an error.

repr_single_value(*value*)

Represent the widget value as a string

repr_value(*value*, *name=None*)

Return textual representation of the value.

If `name` is not `None` raise an error.

set_value(*value*, *name=None*)

Set widget value.

If `name` is not `None` raise an error.

class pylablib.core.gui.value_handling.**PushButtonValueHandler**(*widget*, *labels=('Off', 'On')*)

Bases: *pylablib.core.gui.value_handling.IBoolValueHandler*

Value handler for `QPushButton` widget

get_single_value()

Get the widget value

set_single_value (*value*)
Set the widget value

get_value_changed_signal ()
Get the Qt signal emitted when the value is changed

repr_single_value (*value*)
Represent the widget value as a string

can_set_value (*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters **focus** – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler (*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler (*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value (*name=None*)
Get widget value.

If `name` is not `None` raise an error.

repr_value (*value*, *name=None*)
Return textual representation of the value.

If `name` is not `None` raise an error.

set_value (*value*, *name=None*)
Set widget value.

If `name` is not `None` raise an error.

class pylablib.core.gui.value_handling.**ToolButtonValueHandler** (*widget*, *labels=('Off', 'On')*)
Bases: *pylablib.core.gui.value_handling.IBoolValueHandler*

Value handler for `QToolButton` widget

get_single_value ()
Get the widget value

set_single_value (*value*)
Set the widget value

get_value_changed_signal ()
Get the Qt signal emitted when the value is changed

repr_single_value (*value*)
Represent the widget value as a string

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters focus – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value(*name=None*)
Get widget value.

If `name` is not `None` raise an error.

repr_value(*value*, *name=None*)
Return textual representation of the value.

If `name` is not `None` raise an error.

set_value(*value*, *name=None*)
Set widget value.

If `name` is not `None` raise an error.

class pylablib.core.gui.value_handling.**ComboBoxValueHandler**(*widget*)
Bases: *pylablib.core.gui.value_handling.ISingleValueHandler*

Value handler for `QComboBox` widget

get_single_value()
Get the widget value

set_single_value(*value*)
Set the widget value

get_value_changed_signal()
Get the Qt signal emitted when the value is changed

repr_single_value(*value*)
Represent the widget value as a string

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters focus – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value(*name=None*)
Get widget value.

If `name` is not `None` raise an error.

repr_value(*value*, *name=None*)
Return textual representation of the value.

If `name` is not `None` raise an error.

set_value(*value*, *name=None*)
Set widget value.

If `name` is not `None` raise an error.

class pylablib.core.gui.value_handling.**ProgressBarValueHandler**(*widget*)
Bases: *pylablib.core.gui.value_handling.ISingleValueHandler*

Value handler for `QProgressBar` widget

get_single_value()
Get the widget value

set_single_value(*value*)
Set the widget value

can_set_value(*allow_focus=True*)
Check if setting value from the code is allowed.

Parameters **focus** – if `False`, indicates that settings of focused widgets isn't allowed, with some exceptions (buttons, check boxes, combo boxes)

connect_value_changed_handler(*handler*, *only_signal=True*)
Connect value changed signal.

If `only_signal==True`, equivalent to connecting a handler function to *get_value_changed_signal()* signal; however, if `only_signal==False`, it also works for some objects (e.g., `QLabel`) don't have built-in on-changed signals by calling the handler explicitly every time the value is changed.

Note that the connection is always direct (i.e., it doesn't deal with message queues and different threads, but rather directly calls the handler function). If you need to connect a handler to a signal using some other connection method, you can use *get_value_changed_signal()* directly.

get_handler(*name=None*)
Get handler of a contained widget (or same widget, if `name==None`)

get_value(*name=None*)
Get widget value.

If `name` is not `None` raise an error.

get_value_changed_signal()
Get the Qt signal emitted when the value is changed

repr_single_value(*value*)
Represent the widget value as a string

repr_value(*value*, *name=None*)
Return textual representation of the value.

If name is not None raise an error.

set_value(*value*, *name=None*)
Set widget value.

If name is not None raise an error.

pylablib.core.gui.value_handling.**is_handled_widget**(*widget*)
Check if the widget can be handles by *StandardValueHandler*

pylablib.core.gui.value_handling.**create_value_handler**(*widget*)
Autodetect value handler for the given widget

class pylablib.core.gui.value_handling.**IIndicatorHandler**
Bases: object

Generic handler of an indicator.

Has methods to get and set the indicator value.

get_value(*name=None*)
Get indicator value.

If name is not None, it specifies the name of the indicator parameter inside the widget (for complex widgets).

set_value(*value*, *name=None*)
Set indicator value.

If name is not None, it specifies the name of the indicator parameter inside the widget (for complex widgets).

pylablib.core.gui.value_handling.**VirtualIndicatorHandler**
alias of *pylablib.core.gui.value_handling.VirtualValueHandler*

class pylablib.core.gui.value_handling.**StandardIndicatorHandler**(*widget*, *default_name=None*)
Bases: *pylablib.core.gui.value_handling.IIndicatorHandler*

Default indicator handler, typically used for custom widgets.

To implement getting and setting values, looks for get/set_indicator and get/set_all_indicators methods for the widget and uses them accordingly.

Parameters

- **widget** – handled widget
- **default_name** (*str*) – default name to be supplied to get/set_indicator methods if they require a name argument.

get_value(*name=None*)
Get indicator value.

If name is not None, it specifies the name of the indicator parameter inside the widget (for complex widgets).

set_value(*value*, *name=None*)
Set indicator value.

If `name` is not `None`, it specifies the name of the indicator parameter inside the widget (for complex widgets).

class `pylablib.core.gui.value_handling.`**LabelIndicatorHandler**(*label*, *formatter=None*, *repr_value_name=None*)

Bases: *pylablib.core.gui.value_handling.IIndicatorHandler*

Indicator handler which uses a label to show the value.

Can takes optional widget or widget handler which converts values into strings using its `repr_value` method (by default, use the standard string conversion).

Parameters

- **label** – widget or value handler used to represent the value (takes string values)
- **formatter** – specifies a way to turn values into string representation; can be a widget handler or a widget (its `repr_func` method is used to represent its value), a function (it takes either a single value argument or two arguments `name` and `value` and returns string value), or `None` (use simple string conversion)
- **repr_value_name** (*str*) – default name to be supplied to *repr_value* if it requires a name argument and name is not supplied

get_value(*name=None*)

Get indicator value.

If `name` is not `None`, it specifies the name of the indicator parameter inside the widget (for complex widgets).

repr_value(*value*, *name=None*)

Represent a value with a given name

set_value(*value*, *name=None*)

Set indicator value.

If `name` is not `None`, it specifies the name of the indicator parameter inside the widget (for complex widgets).

`pylablib.core.gui.value_handling.`**create_indicator_handler**(*widget*, *label=None*, *require_setter=False*)

Autodetect indicator handler for the given widget and optional indicator label

exception `pylablib.core.gui.value_handling.`**MissingGUIHandlerError**

Bases: `KeyError`

Missing GUI handler

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `pylablib.core.gui.value_handling.`**GUIValues**(*gui_thread_safe=True*)

Bases: `object`

A collection of values which can be used to manipulate many value handlers at once and represent them as a hierarchical structure.

Has four container-like accessor: `.h` for getting/adding/removing the value handler (i.e., `self.get_handler(name)` is equivalent to `self.h[name]`, and `self.add_handler(name, handler)` is equivalent to `self.h[name]=handler`, and `self.remove_handler(name)` is equivalent to `del self.h[name]`), `.w` for getting the underlying widget (i.e., `self.get_widget(name)` is equivalent to self.w[name]), .v for settings/getting values (i.e., self.get_value(name) is equivalent to self.v[name], and self.set_value(name, value) is equivalent to self.v[name]=value), .i for settings/getting indicator values (i.e., self.get_indicator(name) is equivalent to self.i[name], and self.set_indicator(name, value) is equivalent to self.i[name]=value) .vs for getting the value changed Qt signal (i.e., self.get_value_changed_signal(name) is equivalent to self.s[name]),

Parameters **gui_thread_safe** (*bool*) – if True, all value-access and indicator-access calls (get/set_value, get/set_all_values, get/set_indicator, get/set_all_indicators, and update_indicators) are automatically called in the GUI thread.

add_handler(*name*, *handler*)
Add a value handler under a given name

remove_handler(*name*, *remove_indicator=True*, *disconnect=False*)
Remove the value handler with a given name.

If remove_indicator==True, also try to remove the indicator widget. If disconnect==True, also disconnect all slots connected to the value_changed signal. Unlike most methods (e.g., *get_value()* or *get_handler()*), does not recursively query the children, so it only works if the handler is contained in this table.

get_handler(*name*)
Get the value handler with the given name

add_widget(*name*, *widget*, *add_indicator=True*)
Add a widget under a given name (value handler type is auto-detected)

get_widget(*name*)
Get the widget corresponding to the handler under the given name

add_nested(*name*, *gui_values*, *add_indicator=True*)
Add a nested *GUIValues* under a given name

add_virtual_element(*name*, *value=None*, *multivalued=False*, *add_indicator=True*)
Add a virtual value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view (its value can be set or read, it has on-change events, it can have indicator). The element value is simply stored on set and retrieved on get. If add_indicator==True, add default indicator handler as well.

add_property_element(*name*, *getter=None*, *setter=None*, *add_indicator=True*)
Add a property value element.

Doesn't correspond to any actual widget, but behaves very similarly from the application point of view; each time the value is set or get, the corresponding setter and getter methods are called. If add_indicator==True, add default (stored value) indicator handler as well.

add_all_children(*root*, *root_name=None*, *types_include=None*, *types_exclude=()*, *names_exclude=None*)
Add a widget and all its children to the values set.

The result is organized as a tree using parent-child relations (note that it implies that only children widgets correspond to tree nodes, i.e., only their values can be get/set).

Parameters

- **root** – root widget
- **root_name** – path to the sub-branch where the values will be placed

- **types_include** – if not `None`, specifies list of widget classes (e.g., `QCheckBox`) to include
- **types_include** – specifies list of widget classes to exclude
- **names_exclude** – if not `None`, specifies list of widget names to exclude

class IndicatorsSet(*ind*)
Bases: `tuple`

count()
Return number of occurrences of value.

ind

index()
Return first index of value.

Raises ValueError if the value is not present.

add_indicator_handler(*name*, *handler*, *ind_name='__default__'*)
Add indicator handler with a given name.

ind_name can distinguish different sub-indicators with the same name, if the same value has multiple indicators.

remove_indicator_handler(*name*, *ind_name=None*)
Remove indicator handler with a given name.

ind_name can distinguish different sub-indicators with the same name, if the same value has multiple indicators. By default, remove all indicators with this name

add_widget_indicator(*name*, *widget*, *label=None*, *ind_name='__default__'*)
Add widget-based indicator with a given name.

If `label` is `None`, use widget's `get/set_indicator` or `get/set_all_indicators` functions to indicate the value. Otherwise, use the given label to indicate the value (*label* is used to show the value, *widget* is used to represent it). *ind_name* can distinguish different sub-indicators with the same name, if the same value has multiple indicators.

add_label_indicator(*name*, *label*, *formatter=None*, *ind_name='__default__'*)
Add label-based indicator with a given name.

formatter specifies a way to turn values into string representation; can be a widget handler or a widget (its `repr_func` method is used to represent its value), a function (it takes either a single value argument or two arguments `name` and `value` and returns string value), or `None` (use simple string conversion) *ind_name* can distinguish different sub-indicators with the same name, if the same value has multiple indicators.

get_value(*name=None*)
Get a value or a set of values in a subtree under a given name (all values by default).

Automatically handles complex widgets and sub-names. If *name* refers to a branch, return a `Dictionary` object containing tree structure of the names. If supplied, *include* and *exclude* are containers specifying included and excluded names (relative to the root); by default, include everything and exclude nothing.

get_all_values(*root=None*)
Get all values in the given sub-branch.

Same as `get_value()`, but returns an empty dictionary if the *name* is missing.

set_value(*name*, *value*)
Set value under a given name.

Automatically handles complex widgets and sub-names

set_all_values(*value*, *root=None*)

get_indicator(*name=None*, *ind_name='__default__'*)
Get indicator value with a given name.

ind_name can distinguish different sub-indicators with the same name, if the same value has multiple indicators. If supplied, *include* and *exclude* are containers specifying included and excluded names (relative to the root); by default, include everything and exclude nothing.

get_all_indicators(*root=None*, *ind_name='__default__'*)
Get all indicator values in the given sub-branch.

Same as `get_indicator()`, but returns an empty dictionary if the *root* is missing.

set_indicator(*name*, *value*, *ind_name=None*, *ignore_missing=False*)
Set indicator value with a given name.

ind_name can distinguish different sub-indicators with the same name, if the same value has multiple indicators. By default, set all sub-indicators to the given value. If supplied, *include* and *exclude* are containers specifying included and excluded names (relative to the root); by default, include everything and exclude nothing. If `ignore_missing==True` and the given indicator and sub-indicator names are missing, raise an error; otherwise, do nothing.

set_all_indicators(*value*, *root=''*, *ind_name=None*, *ignore_missing=True*)

update_indicators(*root=''*)
Update all indicators in a subtree with the given root (all values by default) to represent current values.

If supplied, *include* and *exclude* are containers specifying included and excluded names (relative to the root); by default, include everything and exclude nothing.

repr_value(*name*, *value*)
Get a textual representation of a value under a given name.

Automatically handles complex widgets and sub-names.

get_value_changed_signal(*name*)
Get changed events for a value under a given name

update_value(*name=None*)
Send update signal for a handler with a given name or list of names.

Emit a value changed signal with the current value to notify the subscribed slots. If *name* is `None`, emit for all values in the table.

`pylablib.core.gui.value_handling.`**get_gui_values**(*gui_values=None*, *gui_values_path=''*)
Get new or existing `GUIValues` object and the sub-branch path inside it based on the supplied arguments.

If *gui_values* is `None` or `"new"`, create a new object and set empty root path. If *gui_values* itself has `gui_values` attribute, get this attribute, and prepend object's `gui_values_path` attribute to the given path. Otherwise, assume that *gui_values* is `GUIValues` object, and use the supplied root.

`pylablib.core.gui.value_handling.`**virtual_gui_values**(***kwargs*)
Create a gui values set with all virtual values.

`kwargs` define element names and default values.

Module contents

pylablib.core.thread package

Submodules

pylablib.core.thread.callsync module

class pylablib.core.thread.callsync.**QCallResultSynchronizer**(*skippable=True*)

Bases: *pylablib.core.thread.synchronizing.QThreadNotifier*

get_progress()

Get the progress of the call execution.

Can be "waiting" (call is not done executing), "done" (call done successfully), "fail" (call failed, probably due to thread being stopped), "skip" (call was skipped), or "exception" (call raised an exception).

skipped()

Check if the call was skipped

failed()

Check if the call failed

get_value_sync(*timeout=None*, *default=None*, *error_on_fail=True*, *error_on_skip=True*, *pass_exception=True*)

Wait (with the given *timeout*) for the value passed by the notifier

If error_on_fail==True and the controlled thread notifies of a fail (usually, if it's stopped before it executed the call), raise *threadprop.NoControllerThreadError*; otherwise, return *default*. If error_on_skip==True and the call was skipped (e.g., due to full call queue), raise *threadprop.SkippedCallError*; otherwise, return *default*. If pass_exception==True and the returned value represents exception, re-raise it in the caller thread; otherwise, return *default*.

done_notify()

Check if notifying is done

done_wait()

Check if waiting is done

get_value()

Get the value passed by the notifier (doesn't check if it has been passed already)

notify(**args*, ***kwargs*)

Notify the waiting process.

Can only be called once per notifier lifetime. If the notifier allows skipping, and this method is called before *wait()*, return immediately.

notifying_state()

success_wait()

Check if waiting is done successfully

wait(**args*, ***kwargs*)

Wait for the notification.

Can only be called once per notifier lifetime. If the notifier allows skipping, and this method is called after *notify()*, return immediately.

waiting()

Check if waiting is in progress

waiting_state()

class pylablib.core.thread.callsync.**QDummyResultSynchronizer**

Bases: object

Dummy result synchronizer for call which don't require result synchronization (e.g., multicasts)

notify(*value*)

class pylablib.core.thread.callsync.**QDirectResultSynchronizer**(*value*)

Bases: object

Result "synchronizer" for direct calls.

Behaves as a regular result synchronizer with an already executed call.

get_progress()
Get the progress of the call execution (always return "done")

skipped()
Check if the call was skipped (always return False)

failed()
Check if the call failed (always return False)

get_value()
Return stored value

get_value_sync(*timeout=None, default=None, error_on_fail=True, error_on_skip=True, pass_exception=True*)
Return stored value.

Parameters are only for compatibility with *QCallResultSynchronizer*.

wait(**args, **kwargs*)
Do nothing (present only for compatibility with *QCallResultSynchronizer*)

notify(**args, **kwargs*)
Do nothing (present only for compatibility with *QCallResultSynchronizer*)

waiting()
Check if waiting is in progress (always return False)

done_wait()
Check if waiting is done (always return True)

success_wait()
Check if waiting is done successfully (always return True)

done_notify()
Check if notifying is done (always return True)

waiting_state()

notifying_state()

class pylablib.core.thread.callsync.**QScheduledCall**(*func, args=None, kwargs=None, silent=False, result_synchronizer=None*)

Bases: object

Object representing a scheduled remote call.

Can be executed, skipped, or failed in the target thread, in which case it notifies the result synchronizer (if supplied).

Parameters

- **func** – callable to be invoked in the destination thread
- **args** – arguments to be passed to *func*
- **kwargs** – keyword arguments to be passed to *func*
- **silent** – if `True`, silence the exception in the execution thread and simply pass it to the caller thread; otherwise, the exception is raised in both threads
- **result_synchronizer** – result synchronizer object; can be `None` (create new `QCallResultSynchronizer`), `"async"` (no result synchronization), or a `QCallResultSynchronizer` object.

class Callback(*func*, *pass_result*, *call_on_exception*, *call_on_unschedule*)
Bases: `tuple`

call_on_exception

call_on_unschedule

count()
Return number of occurrences of value.

func

index()
Return first index of value.

Raises ValueError if the value is not present.

pass_result

execute(*silent=None*)
Execute the call and notify the result synchronizer (invoked by the destination thread)

add_callback(*callback*, *pass_result=True*, *call_on_exception=False*, *call_on_unschedule=False*, *front=False*)
Set the callback to be executed after the main call is done.

If `pass_result==True`, pass function result to the callback (or `None` if call failed); otherwise, pass no arguments. If `call_on_exception==True`, call it even if the original call raised an exception. If `call_on_unschedule==True`, call it for any call unscheduling event, including using `skip()` or `fail()` methods (this effectively ignores *call_on_exception*, since the callback is called regardless of the exception). If `front==True`, add the callback in the front of the line (executes first).

fail()
Notify that the call is failed (invoked by the destination thread)

skip()
Notify that the call is skipped (invoked by the destination thread)

class pylablib.core.thread.callsync.**TDefaultCallInfo**(*call_time*)
Bases: `tuple`

call_time

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

class pylablib.core.thread.callsync.**QScheduler**(*call_info_argname=None*)
Bases: object

Generic call scheduler.

Two methods are used by the external scheduling routines: *build_call()* to create a *QScheduledCall* with appropriate parameters, and *schedule()*, which takes a call and schedules it. The *schedule()* method should return True if the scheduling was successful (at least, for now), and False otherwise.

Parameters **call_info_argname** – if not None, supplies a name of a keyword argument via which call info (generated by *build_call_info()*) is passed on function call

build_call_info()
Build call info tuple which can be passed to scheduled calls

build_call(*func*, *args=None*, *kwargs=None*, *callback=None*, *pass_result=True*, *callback_on_exception=True*, *sync_result=True*)
Build *QScheduledCall* for subsequent scheduling.

Parameters

- **func** – function to be called
- **args** – arguments to be passed to *func*
- **kwargs** – keyword arguments to be passed to *func*
- **callback** – optional callback to be called when *func* is done
- **pass_result** (*bool*) – if True, pass *func* result as a single argument to the callback; otherwise, give no arguments
- **callback_on_exception** (*bool*) – if True, execute the callback on call fail or skip (if it requires an argument, None is supplied); otherwise, only execute it if the call was successful
- **sync_result** – if True, the call has a default result synchronizer; otherwise, no synchronization is made.

schedule(*call*)
Schedule the call

clear()
Clear the scheduler

class pylablib.core.thread.callsync.**QDirectCallScheduler**(*call_info_argname=None*)
Bases: *pylablib.core.thread.callsync.QScheduler*

Simplest call scheduler: directly executes the calls on scheduling in the scheduling thread.

Parameters **call_info_argname** – if not None, supplies a name of a keyword argument via which call info (generated by *QScheduler.build_call_info()*) is passed on function call

build_call(*func*, *args=None*, *kwargs=None*, *callback=None*, *pass_result=True*, *callback_on_exception=True*, *sync_result=False*)
Build *QScheduledCall* for subsequent scheduling.

Parameters

- **func** – function to be called
- **args** – arguments to be passed to *func*
- **kwargs** – keyword arguments to be passed to *func*

- **callback** – optional callback to be called when *func* is done
- **pass_result** (*bool*) – if `True`, pass *func* result as a single argument to the callback; otherwise, give no arguments
- **callback_on_exception** (*bool*) – if `True`, execute the callback on call fail or skip (if it requires an argument, `None` is supplied); otherwise, only execute it if the call was successful
- **sync_result** – if `True`, the call has a default result synchronizer; otherwise, no synchronization is made.

schedule(*call*)
Schedule the call

build_call_info()
Build call info tuple which can be passed to scheduled calls

clear()
Clear the scheduler

class pylablib.core.thread.callsync.**QQueueScheduler**(*on_full_queue='skip_current'*, *call_info_argname=None*)
Bases: *pylablib.core.thread.callsync.QScheduler*

Call scheduler with a builtin call queue.

Supports placing the calls and retrieving them (from the destination thread). Has ability to skip some calls if, e.g., the queue is too full. Whether the call should be skipped is determined by *can_schedule()* (should be overloaded in subclasses). Used as a default command scheduler.

Parameters

- **on_full_queue** – action to be taken if the call can't be scheduled (i.e., *can_schedule()* returns `False`); can be `"skip_current"` (skip the call which is being scheduled), `"skip_newest"` (skip the most recent call; place the current) `"skip_oldest"` (skip the oldest call in the queue; place the current), `"call_current"` (execute the call which is being scheduled immediately in the caller thread), `"call_newest"` (execute the most recent call immediately in the caller thread), `"call_oldest"` (execute the oldest call in the queue immediately in the caller thread), or `"wait"` (wait until the call can be scheduled, which is checked after every call removal from the queue; place the call)
- **call_info_argname** – if not `None`, supplies a name of a keyword argument via which call info (generated by *QScheduler.build_call_info()*) is passed on function call

Methods to overload:

- *can_schedule()*: check if the call can be scheduled
- *call_added()*: called when a new call has been added to the queue
- *call_popped()*: called when a call has been removed from the queue (either for execution, or for skipping)

can_schedule(*call*)
Check if the call can be scheduled

call_added(*call*)
Called whenever *call* has been added to the queue

call_popped(*call*, *idx*)
Called whenever *call* has been removed from the queue

idx determines the call position within the queue.

schedule(*call*)
Schedule a call

pop_call()
Pop the call from the queue head.

If the queue is empty, return `None`

unschedule(*call*)
Unschedule a given call.

Designed for joint queue operation, so the call is not notified (assume that it has been already notified elsewhere).

has_calls()
Check if there are queued calls

clear(*close=True*)
Clear the call queue.

If `close==True`, mark the queue as closed (any attempt to schedule more calls fails automatically) and fail all calls in the queue; otherwise, skip all calls currently in the queue.

build_call(*func*, *args=None*, *kwargs=None*, *callback=None*, *pass_result=True*, *callback_on_exception=True*, *sync_result=True*)
Build `QScheduledCall` for subsequent scheduling.

Parameters

- **func** – function to be called
- **args** – arguments to be passed to *func*
- **kwargs** – keyword arguments to be passed to *func*
- **callback** – optional callback to be called when *func* is done
- **pass_result** (*bool*) – if `True`, pass *func* result as a single argument to the callback; otherwise, give no arguments
- **callback_on_exception** (*bool*) – if `True`, execute the callback on call fail or skip (if it requires an argument, `None` is supplied); otherwise, only execute it if the call was successful
- **sync_result** – if `True`, the call has a default result synchronizer; otherwise, no synchronization is made.

build_call_info()
Build call info tuple which can be passed to scheduled calls

class pylablib.core.thread.callsync.**QQueueLengthLimitScheduler**(*max_len=1*, *on_full_queue='skip_current'*, *call_info_argname=None*)
Bases: `pylablib.core.thread.callsync.QQueueScheduler`

Queued call scheduler with a length limit.

Parameters

- **max_len** – maximal queue length; non-positive values are interpreted as no limit can also be a tuple `(arg_name, max_len)`, in which case the length is calculated separately for every value of the parameter `arg_name` supplied to the method
- **on_full_queue** – action to be taken if the call can't be scheduled (the queue is full); can be `"skip_current"` (skip the call which is being scheduled), `"skip_newest"` (skip the most recent call; place the current) `"skip_oldest"` (skip the oldest call in the queue; place the current), `"call_current"` (execute the call which is being scheduled immediately in the caller thread), `"call_newest"` (execute the most recent call immediately in the caller thread), `"call_oldest"` (execute the oldest call in the queue immediately in the caller thread), or `"wait"` (wait until the call can be scheduled, which is checked after every call removal from the queue; place the call)
- **call_info_argname** – if not `None`, supplies a name of a keyword argument via which call info (generated by `QScheduler.build_call_info()`) is passed on function call

change_max_len(*max_len*)
Change maximal length of the call queue (doesn't affect already scheduled calls)

get_current_len()
Get current number of calls in the queue

call_added(*call*)
Called whenever *call* has been added to the queue

call_popped(*call*, *idx*)
Called whenever *call* has been removed from the queue

idx determines the call position within the queue.

can_schedule(*call*)
Check if the call can be scheduled

build_call(*func*, *args=None*, *kwargs=None*, *callback=None*, *pass_result=True*, *callback_on_exception=True*, *sync_result=True*)
Build `QScheduledCall` for subsequent scheduling.

Parameters

- **func** – function to be called
- **args** – arguments to be passed to *func*
- **kwargs** – keyword arguments to be passed to *func*
- **callback** – optional callback to be called when *func* is done
- **pass_result** (*bool*) – if `True`, pass *func* result as a single argument to the callback; otherwise, give no arguments
- **callback_on_exception** (*bool*) – if `True`, execute the callback on call fail or skip (if it requires an argument, `None` is supplied); otherwise, only execute it if the call was successful
- **sync_result** – if `True`, the call has a default result synchronizer; otherwise, no synchronization is made.

build_call_info()
Build call info tuple which can be passed to scheduled calls

clear(*close=True*)
Clear the call queue.

If `close==True`, mark the queue as closed (any attempt to schedule more calls fails automatically) and fail all calls in the queue; otherwise, skip all calls currently in the queue.

has_calls()
Check if there are queued calls

pop_call()
Pop the call from the queue head.

If the queue is empty, return `None`

schedule(*call*)
Schedule a call

unschedule(*call*)
Unschedule a given call.

Designed for joint queue operation, so the call is not notified (assume that it has been already notified elsewhere).

class pylablib.core.thread.callsync.**QQueueSizeLimitScheduler**(*max_size=1*, *size_calc=None*, *on_full_queue='skip_current'*, *call_info_argname=None*)

Bases: *pylablib.core.thread.callsync.QQueueScheduler*

Queued call scheduler with a generic size limit; similar to *QQueueLengthLimitScheduler*, but more flexible and can implement more restrictions (e.g., queue length and arguments RAM size).

Parameters

- **max_size** – maximal total size of the arguments; can be either a single number, or a tuple (if several different size metrics are involved); non-positive values are interpreted as no limit
- **size_calc** – function that takes a single argument (call to be placed) and returns its size; can be either a single number, or a tuple (if several different size metrics are involved); by default, simply returns 1, which makes the scheduler behavior identical to *QQueueLengthLimitScheduler*
- **on_full_queue** – action to be taken if the call can't be scheduled (the queue is full); can be `"skip_current"` (skip the call which is being scheduled), `"skip_newest"` (skip the most recent call; place the current) `"skip_oldest"` (skip the oldest call in the queue; place the current), `"call_current"` (execute the call which is being scheduled immediately in the caller thread), `"call_newest"` (execute the most recent call immediately in the caller thread), `"call_oldest"` (execute the oldest call in the queue immediately in the caller thread), or `"wait"` (wait until the call can be scheduled, which is checked after every call removal from the queue; place the call)
- **call_info_argname** – if not `None`, supplies a name of a keyword argument via which call info (generated by *QScheduler.build_call_info()*) is passed on function call

change_max_size(*max_size*)
Change size restrictions

get_current_size()
Get current size metrics

call_added(*call*)
Called whenever *call* has been added to the queue

call_popped(*call*, *idx*)
Called whenever *call* has been removed from the queue

idx determines the call position within the queue.

can_schedule(*call*)
Check if the call can be scheduled

build_call(*func*, *args=None*, *kwargs=None*, *callback=None*, *pass_result=True*, *callback_on_exception=True*, *sync_result=True*)
Build *QScheduledCall* for subsequent scheduling.

Parameters

- **func** – function to be called
- **args** – arguments to be passed to *func*
- **kwargs** – keyword arguments to be passed to *func*
- **callback** – optional callback to be called when *func* is done
- **pass_result** (*bool*) – if `True`, pass *func* result as a single argument to the callback; otherwise, give no arguments
- **callback_on_exception** (*bool*) – if `True`, execute the callback on call fail or skip (if it requires an argument, `None` is supplied); otherwise, only execute it if the call was successful
- **sync_result** – if `True`, the call has a default result synchronizer; otherwise, no synchronization is made.

build_call_info()
Build call info tuple which can be passed to scheduled calls

clear(*close=True*)
Clear the call queue.

If `close==True`, mark the queue as closed (any attempt to schedule more calls fails automatically) and fail all calls in the queue; otherwise, skip all calls currently in the queue.

has_calls()
Check if there are queued calls

pop_call()
Pop the call from the queue head.

If the queue is empty, return `None`

schedule(*call*)
Schedule a call

unschedule(*call*)
Unschedule a given call.

Designed for joint queue operation, so the call is not notified (assume that it has been already notified elsewhere).

`pylablib.core.thread.callsync.`**schedule_multiple_queues**(*call*, *queues*)
Schedule the call simultaneously in several queues.

Go through queues in the given order and schedule call in every one of them. If one of the schedules failed or the call has been executed there, unschedule it from all the previous queues and return `False`; otherwise, return `True`.

class pylablib.core.thread.callsync.**QMultiQueueScheduler**(*schedulers*, *notifiers*)

Bases: object

Wrapper around *schedule_multiple_queues()* which acts as a single scheduler.

Support additional notifiers, which are called if the scheduling is successful (e.g., to notify and wake up the destination thread).

build_call(**args*, ***kwargs*)

schedule(*call*)

class pylablib.core.thread.callsync.**QThreadCallScheduler**(*thread=None*, *tag=None*, *priority=0*, *interrupt=True*, *call_info_argname=None*)

Bases: *pylablib.core.thread.callsync.QScheduler*

Call scheduler via thread calls (*QThreadController.call_in_thread_callback()*)

Parameters

- **thread** – destination thread (by default, thread which creates the scheduler)
- **tag** – if supplied, send the call in a message with the given tag; otherwise, use the interrupt call (generally, higher priority method).
- **priority** – message priority (only when *tag* is not None)
- **interrupt** – whether the call is an interrupt (call inside any loop, e.g., during waiting or sleeping), or it should be called in the main event loop
- **call_info_argname** – if not None, supplies a name of a keyword argument via which call info (generated by *QScheduler.build_call_info()*) is passed on function call

schedule(*call*)

Schedule the call

build_call(*func*, *args=None*, *kwargs=None*, *callback=None*, *pass_result=True*, *callback_on_exception=True*, *sync_result=True*)

Build *QScheduledCall* for subsequent scheduling.

Parameters

- **func** – function to be called
- **args** – arguments to be passed to *func*
- **kwargs** – keyword arguments to be passed to *func*
- **callback** – optional callback to be called when *func* is done
- **pass_result** (*bool*) – if True, pass *func* result as a single argument to the callback; otherwise, give no arguments
- **callback_on_exception** (*bool*) – if True, execute the callback on call fail or skip (if it requires an argument, None is supplied); otherwise, only execute it if the call was successful
- **sync_result** – if True, the call has a default result synchronizer; otherwise, no synchronization is made.

build_call_info()

Build call info tuple which can be passed to scheduled calls

clear()
Clear the scheduler

class pylablib.core.thread.callsync.**QMulticastThreadCallScheduler**(*thread=None, limit_queue=1, tag=None, priority=0, interrupt=True, call_info_argname=None*)

Bases: *pylablib.core.thread.callsync.QThreadCallScheduler*

Extended call scheduler via thread calls, which can limit number of queued calls.

Parameters

- **thread** – destination thread (by default, thread which creates the scheduler)
- **limit_queue** – call queue limit (non-positive numbers are interpreted as no limit)
- **tag** – if supplied, send the call in a message with the given tag; otherwise, use the interrupt call (generally, higher priority method).
- **priority** – message priority (only when *tag* is not None)
- **interrupt** – whether the call is an interrupt (call inside any loop, e.g., during waiting or sleeping), or it should be called in the main event loop
- **call_info_argname** – if not None, supplies a name of a keyword argument via which call info (generated by *QScheduler.build_call_info()*) is passed on function call

schedule(*call*)
Schedule the call

build_call(*func, args=None, kwargs=None, callback=None, pass_result=True, callback_on_exception=True, sync_result=True*)
Build *QScheduledCall* for subsequent scheduling.

Parameters

- **func** – function to be called
- **args** – arguments to be passed to *func*
- **kwargs** – keyword arguments to be passed to *func*
- **callback** – optional callback to be called when *func* is done
- **pass_result** (*bool*) – if True, pass *func* result as a single argument to the callback; otherwise, give no arguments
- **callback_on_exception** (*bool*) – if True, execute the callback on call fail or skip (if it requires an argument, None is supplied); otherwise, only execute it if the call was successful
- **sync_result** – if True, the call has a default result synchronizer; otherwise, no synchronization is made.

build_call_info()
Build call info tuple which can be passed to scheduled calls

clear()
Clear the scheduler

pylablib.core.thread.controller module

pylablib.core.thread.controller.**exint**(*error_msg_template='{}:'*)
Context that intercepts exceptions and stops the execution in a controlled manner (quitting the main thread)

pylablib.core.thread.controller.**exsafe**(*func*)
Decorator that intercepts exceptions raised by *func* and stops the execution in a controlled manner (quitting the main thread)

pylablib.core.thread.controller.**exsafeSlot**(**slargs*, ***slkwargs*)
Wrapper around Qt slot which intercepts exceptions and stops the execution in a controlled manner

pylablib.core.thread.controller.**toploopSlot**(**slargs*, ***slkwargs*)
Wrapper around Qt slot which intercepts exceptions and stops the execution in a controlled manner

class pylablib.core.thread.controller.**QThreadControllerThread**(*controller*)
Bases: `object`

finalized = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

run()

quit_sync()

pylablib.core.thread.controller.**remote_call**(*func*)
Decorator that turns a controller method into a remote call (call from a different thread is passed synchronously)

pylablib.core.thread.controller.**call_in_thread**(*thread_name*, *interrupt=True*, *pass_exception=True*, *silent=False*)
Decorator that turns any function into a remote call in a thread with a given name (call from a different thread is passed synchronously)

pylablib.core.thread.controller.**call_in_gui_thread**(*func=None*, *pass_exception=True*, *silent=False*)
Decorator that turns any function into a remote call in a GUI thread (call from a different thread is passed synchronously)

pylablib.core.thread.controller.**gui_thread_method**(*func*)
Decorator for an object's method that checks if the object's `gui_thread_safe` attribute is true, in which case the call is routed to the GUI thread

class pylablib.core.thread.controller.**QThreadController**(*name=None*, *kind='loop'*, *multicast_pool=None*)
Bases: `object`

Generic Qt thread controller.

Responsible for all inter-thread synchronization. There is one controller per thread, and

Parameters

- **name** (`str`) – thread name (by default, generate a new unique name); this name can be used to obtain thread controller via `get_controller()`
- **kind** (`str`) – thread kind; can be `"loop"` (thread is running in the Qt message loop; behavior is implemented in `process_message()` and remote calls), `"run"` (thread executes `run()` method and quits after it is complete), or `"main"` (can only be created in the main GUI thread)
- **multicast_pool** – `MulticastPool` for this thread (by default, use the default common pool)

Methods to overload:

- `on_start()`: executed on the thread startup (between synchronization points `"start"` and `"run"`)
- `on_finish()`: executed on thread cleanup (attempts to execute in any case, including exceptions)
- `run()`: executed once per thread; thread is stopped afterwards (only if `kind=="run"`)
- **`process_message()`: function that takes 2 arguments (tag and value) of the message and processes it; retur**
  (in which case it is stored and can be recovered via `wait_for_message()`/`pop_message()`); by default, always return `False`
- **`process_interrupt()`: function that tales 2 arguments (tag and value) of the interrupt message (message**
  by default, assumes that any value with tag `"execute"` is a function and executes it

Signals:

- `started`: emitted on thread start (after `on_start()` is executed)
- `finished`: emitted on thread finish (before `on_finish()` is executed)

started = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
This signal is emitted after the thread has started (after the setup code has been executed, before its lifetime state is changed)

finished = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>
This signal is emitted before the thread has finished (before the cleanup code has been executed, after its lifetime state is changed)

wait_for_message(*tag*, *timeout=None*, *top_loop=False*)
Wait for a single message with a given tag.

Return value of a received message with this tag. If timeout is passed, raise `threadprop.TimeoutThreadError`. If `top_loop==True`, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). Local call method.

new_messages_number(*tag*)
Get the number of queued messages with a given tag.

Local call method.

pop_message(*tag*)
Pop the latest message with the given tag.

Select the message with the highest priority, and among those the oldest one. If no messages are available, raise `threadprop.NoMessageThreadError`. Local call method.

wait_for_sync(*tag*, *uid*, *timeout=None*)
Wait for synchronization signal with the given tag and UID.

This method is rarely invoked directly, and is usually used by synchronizers code. If timeout is passed, raise `threadprop.TimeoutThreadError`. Local call method.

wait_for_any_message(*timeout=None*, *top_loop=False*)
Wait for any message (including synchronization messages or pokes).

If timeout is passed, raise `threadprop.TimeoutThreadError`. If `top_loop==True`, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). Local call method.

wait_until(*check*, *timeout=None*, *top_loop=False*)
Wait until a given condition is true.

Condition is given by the *check* function, which is called after every new received message and should return `True` if the condition is met. If `top_loop==True`, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). If timeout is passed, raise *threadprop.TimeoutThreadError*. Local call method.

check_messages (*top_loop=False*)
Receive new messages.

Runs the underlying message loop to process newly received message and signals (and place them in corresponding queues if necessary). This method is rarely invoked, and only should be used periodically during long computations to not 'freeze' the thread. If `top_loop==True`, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). Local call method.

sleep (*timeout*, *wake_on_message=False*, *top_loop=False*)
Sleep for a given time (in seconds).

Unlike `time.sleep()`, constantly checks the event loop for new messages (e.g., if stop or interrupt commands are issued). In addition, if `wake_on_message==True`, wake up if any message has been received; it this case. return `True` if the wait has been completed, and `False` if it has been interrupted by a message. If `top_loop==True`, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). If `timeout is None`, wait forever (usually, until the application is closed, or some interrupt message raises and error). Local call method.

no_stopping ()
Context manager, which temporarily suspends stop requests (*InterruptExceptionStop* exceptions).

If the stop request has been made within this block, raise the exception on exit. Note that *stop()* method and, correspondingly, *stop_controller()* still work, when called from the controlled thread.

process_interrupt (*tag*, *value*)
Process a new interrupt.

If the function returns `False`, the interrupt is put in the corresponding queue. Otherwise, the the message is interrupt to be already, and it gets 'absorbed'. Local call method, called automatically.

process_message (*tag*, *value*)
Process a new message.

If the function returns `False`, the message is put in the corresponding queue. Otherwise, the the message is considered to be already, and it gets 'absorbed'. Local call method, called automatically.

on_start ()
Method invoked on the start of the thread.

Local call method, called automatically.

on_finish ()
Method invoked in the end of the thread.

Called regardless of the stopping reason (normal finishing, exception, application finishing). Local call method, called automatically.

run ()
Method called to run the main thread code (only for `"run"` thread kind).

Local call method, called automatically.

subscribe_sync (*callback*, *srcs='any'*, *tags=None*, *dsts='any'*, *filt=None*, *subscription_priority=0*, *limit_queue=None*, *call_interrupt=True*, *add_call_info=False*, *sid=None*)
Subscribe a synchronous callback to a multicast.

If a multicast is sent, *callback* is called from the *dest_controller* thread (by default, thread which is calling this function) via the thread call mechanism (*QThreadController.*

`call_in_thread_callback()`). In Qt, analogous to making a signal connection with a queued call. By default, the subscribed destination is the thread's name. Local call method.

Parameters

- **callback** – callback function, which takes 3 arguments: source, tag, and value.
- **srcs** (*str or [str]*) – multicast source name or list of source names to filter the subscription; can be `"any"` (any source) or `"all"` (only multicasts specifically having `"all"` as a source).
- **tags** – multicast tag or list of tags to filter the subscription (any tag by default); can also contain Unix shell style pattern (`"*"` matches everything, `"?"` matches one symbol, etc.)
- **dsts** (*str or [str]*) – multicast destination name or list of destination names to filter the subscription; can be `"any"` (any destination) or `"all"` (only source specifically having `"all"` as a destination).
- **filt** (*callable*) – additional filter function which takes 4 arguments: source, destination, tag, and value, and checks whether multicast passes the requirements.
- **limit_queue** (*int*) – limits the maximal number of scheduled calls (if the multicast is sent while at least *limit_queue* callbacks are already in queue to be executed, ignore it) 0 or negative value means no limit (not recommended, as it can increase the queue indefinitely if the multicast rate is high enough)
- **call_interrupt** – whether the call is an interrupt (call inside any loop, e.g., during waiting or sleeping), or it should be called in the main event loop
- **subscription_priority** (*int*) – subscription priority (higher priority subscribers are called first).
- **sid** (*int*) – subscription ID (by default, generate a new unique id and return it).

subscribe_direct(*callback*, *srcs='any'*, *tags=None*, *dsts='any'*, *filt=None*, *subscription_priority=0*, *scheduler=None*, *sid=None*)

Subscribe asynchronous callback to a multicast.

If a multicast is sent, *callback* is called from the sending thread (not subscribed thread). Therefore, should be used with care. In Qt, analogous to making a signal connection with a direct call. By default, the subscribed destination is the thread's name. Local call method.

Parameters

- **callback** – callback function, which takes 3 arguments: source, tag, and value.
- **srcs** (*str or [str]*) – multicast source name or list of source names to filter the subscription; can be `"any"` (any source) or `"all"` (only multicasts specifically having `"all"` as a source).
- **tags** – multicast tag or list of tags to filter the subscription (any tag by default); can also contain Unix shell style pattern (`"*"` matches everything, `"?"` matches one symbol, etc.)
- **dsts** (*str or [str]*) – multicast destination name or list of destination names to filter the subscription; can be `"any"` (any destination) or `"all"` (only source specifically having `"all"` as a destination).
- **filt** (*callable*) – additional filter function which takes 4 arguments: source, destination, tag, and value, and checks whether multicast passes the requirements.
- **subscription_priority** (*int*) – subscription priority (higher priority subscribers are called first).

- **scheduler** – if defined, multicast call gets scheduled using this scheduler instead of being called directly (which is the default behavior)
- **sid** (*int*) – subscription ID (by default, generate a new unique id and return it).

unsubscribe (*sid*)
Unsubscribe from a subscription with a given ID.

Note that multicasts which are already emitted but not processed will remain in the queue; if they need to be ignored, it should be handled explicitly. Local call method.

send_multicast (*dst='any'*, *tag=None*, *value=None*, *src=None*)
Send a multicast to the multicast pool.

By default, the multicast source is the thread's name. Local call method.

Parameters

- **dst** (*str*) – multicast destination; can be a name, `"all"` (will pass all subscribers' destination filters), or `"any"` (will only be passed to subscribers specifically subscribed to multicast with `"any"` destination).
- **tag** (*str*) – multicast tag.
- **value** – multicast value.
- **src** (*str*) – multicast source; can be `None` (current thread name), a specific name, `"all"` (will pass all subscribers' source filters), or `"any"` (will only be passed to subscribers specifically subscribed to multicast with `"any"` source).

set_variable (*name*, *value*, *update=False*, *notify=False*, *notify_tag='changed/*'*, *simple=False*)
Set thread variable.

Can be called in any thread (controlled or external). If `notify==True`, send an multicast with the given *notify_tag* (where `"*"` symbol is replaced by the variable name). If `update==True` and the value is a dictionary, update the branch rather than overwrite it. If `simple==True`, assume that the result is a single atomic variable, in which case the lock is not used; note that in this case the threads waiting on this variable (or branches containing it) will not be notified. Local call method.

delete_variable (*name*, *missing_error=False*)
Delete thread variable.

If `missing_error==False` and no variable exists, do nothing; otherwise, raise and error. Local call method.

set_func_variable (*name*, *func*, *use_lock=True*)
Set a 'function' variable.

Acts as a thread variable to the external user, but instead of reading a stored value, it executed a function instead. Note, that the function is executed in the caller thread (i.e., the thread which tries to access the variable), so use of synchronization methods (commands, signals, locks) is highly advised.

If `use_lock==True`, then the function call will be wrapped into the usual variable lock, i.e., it won't run concurrently with other variable access. Local call method.

add_thread_method (*name*, *method*, *interrupt=True*)
Add a thread method.

Adds a named method to the thread, which can be called later using `call_thread_method()`. This method will be called in this thread.

Useful for GUI thread to set up some global access methods, which other threads can safely use. For `QTaskThread` threads it's a better idea to set up a command instead. Local call method.

delete_thread_method(*name*)
Delete a thread method.

Local call method.

call_thread_method(*name*, **args*, ***kwargs*)
Call a thread method.

Method needs to be set up beforehand using `add_thread_method()`. It is always executed in the current thread. Local call method.

send_message(*tag*, *value*, *priority=0*)
Send a message to the thread with a given tag, value and priority.

External call method.

send_interrupt(*tag*, *value*, *priority=0*)
Send an interrupt message to the thread with a given tag, value and priority.

External call method.

send_sync(*tag*, *uid*)
Send a synchronization signal with the given tag and UID.

This method is rarely invoked directly, and is usually used by synchronizers code (e.g., `QThreadNotifier`). External call method.

get_variable(*name*, *default=None*, *copy_branch=True*, *missing_error=False*, *simple=False*)
Get thread variable.

If `missing_error==False` and no variable exists, return *default*; otherwise, raise and error. If `copy_branch==True` and the variable is a `Dictionary` branch, return its copy to ensure that it stays unaffected on possible further variable assignments. If `simple==True`, assume that the result is a single atomic variable, in which case the lock is not used; this only works with actual variables and not function variables. Universal call method.

sync_variable(*name*, *pred*, *timeout=None*)
Wait until thread variable with the given *name* satisfies the condition given by *pred*.

pred is a function which takes one argument (variable value) and returns whether the condition is satisfied. It is executed in the caller thread. External call method.

start()
Start the thread.

External call method.

request_stop()
Request thread stop (send a stop command).

External call method.

stop(*code=0*, *sync=False*)
Stop the thread.

If called from the thread, stop immediately by raising a `threadprop.InterruptExceptionStop` exception. Otherwise, schedule thread stop. If the thread kind is `"main"`, stop the whole application with the given exit code. Otherwise, stop the thread. If `sync==True` and the thread is not main or current, wait until it is completely stopped. Universal call method.

sync_stop()
Wait until the controller and the thread are stopped.

External call method.

poke()
Send a dummy message to the thread.

A cheap way to notify the thread that something happened (useful for, e.g., making thread leave *wait_for_any_message()* method). External call method.

running()
Check if the thread is running

finishing()
Check if the thread is finishing

notify_exec_point(*point*)
Mark the given execution point as passed.

Automatically invoked points include "start" (thread starting), "run" (thread setup and ready to run), "cleanup" (thread stopping is invoked, starting to clean up) and "stop" (thread finished). Can be extended for arbitrary points. Local call method.

fail_exec_point(*point*)
Mark the given execution point as failed.

Automatically invoked for "run" (thread setup and ready to run) if the startup raised an error before the thread properly started ("start", "cleanup", and "stop" are notified in any case) Can be extended for arbitrary points. Local call method.

get_exec_counter(*point*)
Get the counter (number of notifications) for the given point.

See *sync_exec_point()* for details. External call.

sync_exec_point(*point*, *timeout=None*, *counter=1*)
Wait for the given execution point.

Automatically invoked points include "start" (thread starting), "run" (thread setup and ready to run), "cleanup" (thread stopping is invoked, starting to clean up) and "stop" (thread finished). If timeout is passed, raise *threadprop.TimeoutThreadError*. *counter* specifies the minimal number of pre-requisite *notify_exec_point()* calls to finish the waiting (by default, a single call is enough). Return actual number of notifier calls up to date. External call method.

add_stop_notifier(*func*, *call_if_stopped=True*)
Add stop notifier: a function which is called when the thread is about to be stopped (left the main message loop).

The supplied function is called in the controlled thread close to its shutdown, so it should be short, non-blocking, and thread-safe. If the thread is already stopped and call_if_stopped==True, call *func* immediately (from the caller's thread). Return True if the thread is still running and the notifier is added, and False otherwise. Local call method.

remove_stop_notifier(*func*)
Remove the stop notifier from this controller.

Return True if the notifier was in this thread and is now removed, and False otherwise. Local call method.

is_in_controlled()
Check if the thread executing this code is controlled by this controller

call_in_thread_callback(*func*, *args=None*, *kwargs=None*, *callback=None*, *tag=None*, *priority=0*, *interrupt=True*)
Call a function in this thread with the given arguments.

If *callback* is supplied, call it with the result as a single argument (call happens in the controller thread). If *tag* is supplied, send the call in a message with the given tag; otherwise, use the interrupt call (generally, higher priority method). If `interrupt==True`, method can be called inside any control loop (either main loop, or during waiting); otherwise, only call it in the top loop. Universal call method.

call_in_thread_sync(*func*, *args=None*, *kwargs=None*, *sync=True*, *callback=None*, *timeout=None*, *default_result=None*, *pass_exception=True*, *silent=False*, *tag=None*, *priority=0*, *interrupt=True*, *error_on_stopped=True*, *same_thread_shortcut=True*)

Call a function in this thread with the given arguments.

If `sync==True`, calling thread is blocked until the controlled thread executes the function, and the function result is returned (in essence, the fact that the function executes in a different thread is transparent). Otherwise, exit call immediately, and return a synchronizer object (*QCallResultSynchronizer*), which can be used to check if the call is done (method *is_done*) and obtain the result (method *QCallResultSynchronizer.get_value_sync()*). If *callback* is not `None`, call it after the function is successfully executed (from the target thread), with a single parameter being function result. If `pass_exception==True` and *func* raises an exception, re-raise it in the caller thread (applies only if `sync==True`). If `silent==True` and *func* raises an exception, silence it in the execution thread and only re-raise it in the caller thread; note that if `pass_exception==False` and `silent==True`, the exception is ignored in both threads. If *tag* is supplied, send the call in a message with the given tag and priority; otherwise, use the interrupt call (generally, higher priority method). If `interrupt==True`, method can be called inside any control loop (either main loop, or during waiting); otherwise, only call it in the top loop. If `error_on_stopped==True` and the controlled thread is stopped before it executed the call, raise *threadprop.NoControllerThreadError*; otherwise, return *default_result*. If `same_thread_shortcut==True` (default), the call is synchronous, and the caller thread is the same as the controlled thread, call the function directly. Universal call method.

class pylablib.core.thread.controller.**QTaskThread**(*name=None*, *args=None*, *kwargs=None*, *multicast_pool=None*)

Bases: *pylablib.core.thread.controller.QThreadController*

Thread which allows to set up and run jobs and batch jobs with a certain time period, and execute commands in the meantime.

Parameters

- **name** (*str*) – thread name (by default, generate a new unique name)
- **args** – args supplied to *setup_task()* method
- **kwargs** – keyword args supplied to *setup_task()* method
- **multicast_pool** – *MulticastPool* for this thread (by default, use the default common pool)

ca

asynchronous command accessor, which makes calls more function-like; `ctl.ca.comm(*args, **kwarg)` is equivalent to `ctl.call_command("comm",args,kwargs,sync=False)`

cad

asynchronous command accessor returning a result synchronizer, which makes calls more function-like; `ctl.cad.comm(*args,**kwarg)` is equivalent to `ctl.call_command("comm",args, kwargs,sync="delayed")`

cs

synchronous command accessor, which makes calls more function-like; `ctl.cs.comm(*args, **kwarg)` is equivalent to `ctl.call_command("comm",args,kwargs,sync=True)`

css

synchronous command accessor which is made 'exception-safe' via *exsafe()* wrapper (i.e., safe to directly connect to slots) `ctl.css.comm(*args,**kwarg)` is equivalent to `with exint(): ctl.call_command("comm",args,kwargs,sync=True)`

csi

synchronous command accessor which ignores and silences any exceptions (including missing /stopped controller) useful for sending queries during thread finalizing / application shutdown, when it's not guaranteed that the command recipient is running (commands already ignore any errors, unless their results are specifically requested); useful for synchronous commands in finalizing functions, where other threads might already be stopped

m

method accessor; directly calls the method corresponding to the command; `ctl.m.comm(*args, **kwarg)` is equivalent to `ctl.call_command("comm",*args,**kwargs)`, which is often also equivalent to `ctl.comm(*args,**kwargs)`; for most practical purposes it's the same as directly invoking the class method, but it makes intent more explicit (as command methods are usually not called directly from other threads), and it doesn't invoke warning about calling method instead of command from another thread.

Methods to overload:

- *setup_task()*: executed on the thread startup (between synchronization points `"start"` and `"run"`)
- *finalize_task()*: executed on thread cleanup (attempts to execute in any case, including exceptions)

class TBatchJob(*job*, *cleanup*, *min_run_time*, *priority*)

Bases: `tuple`

cleanup

count()

Return number of occurrences of value.

index()

Return first index of value.

Raises ValueError if the value is not present.

job

min_run_time

priority

class TCommand(*command*, *scheduler*, *priority*)

Bases: `tuple`

command

count()

Return number of occurrences of value.

index()

Return first index of value.

Raises ValueError if the value is not present.

priority

scheduler

class Job(*job*, *period*, *queue*, *jobs_order*)
Bases: `object`

A single job loop.

Deals with scheduling, time counting, pausing, and cleanup.

Parameters

- **job** – job function
- **period** – job period
- **queue** – thread controller's scheduling queue, to which the job must be added
- **jobs_order** – thread controller's job queue which determines the jobs scheduling order

schedule()
Schedule the job

mark_unscheduled()
Mark the job as unscheduled.

Called automatically on job completion.

unschedule()
Manually unschedule the job (e.g., when paused or removed)

clear()
Clear the job and remove it from the jobs list

change_period(*period*)
Change the job period

pause(*paused=True*, *unschedule=True*)
Pause or resume the job.

If pausing and `unschedule==True`, remove already scheduled job from the queue.

time_left(*t=None*)
Get the amount of time left till the next call, or `None` if the job is paused

add_job(*name*, *job*, *period*, *initial_call=True*, *priority=-10*)
Add a recurrent *job* which is called every *period* seconds.

The job starts running automatically when the main thread loop start executing. If `initial_call==True`, call *job* once immediately after adding. *priority* specifies the call priority in the scheduling queue; by default, it is lower than the command and multicasts (0). Local call method.

change_job_period(*name*, *period*)
Change the period of the job *name*.

Local call method.

remove_job(*name*)
Remove the job *name* from the job list.

Local call method.

add_batch_job(*name*, *job*, *cleanup=None*, *min_runtime=0*, *priority=-10*)
Add a batch *job* which is executed once, but with continuations.

After this call the job is just created, but is not running. To start it, call `start_batch_job()`. If specified, *cleanup* is a finalizing function which is called both when the job terminates normally, and when it is forcibly stopped (including thread termination). *min_runtime* specifies minimal expected runtime of a job; if a job executes faster than this time, it is repeated again unless at least *min_runtime* seconds passed; useful for high-throughput jobs, as it reduces overhead from the job scheduling mechanism (repeating within *min_runtime* time window is fast)

Unlike the usual recurrent jobs, here *job* is a generator (usually defined by a function with `yield` statement). When the job is running, the generator is periodically called until it raises `StopIteration` exception, which signifies that the job is finished. From generator function point of view, after the job is started, the function is executed normally, but every time `yield` statement is encountered, the execution is suspended for *period* seconds (specified in `start_batch_job()`). *priority* specifies the call priority in the scheduling queue; by default, it is lower than the command and multicasts (0). Local call method.

change_batch_job_parameters(*name*, *job='keep'*, *cleanup='keep'*, *min_runtime='keep'*, *priority='keep'*, *stop=False*, *restart=False*)

Change parameters (main body, cleanup function, and minimal runtime) of the batch job.

The parameters are the same as for `add_batch_job()`. If any of them are `"keep"`, don't change them. If `stop==True`, stop the job before changing the parameters; otherwise the job is continued with the previous parameters (including cleanup) until it is stopped and restarted. If `restart==True`, restart the job after changing the parameters. Local call method.

remove_batch_job(*name*)

Remove the batch job *name*, stopping it if necessary.

Local call method.

start_batch_job(*name*, *period*, **args*, *start_immediate=True*, ***kwargs*)

Start the batch job with the given name.

period specifies suspension period. Optional arguments are passed to the job and the cleanup functions. If `start_immediate==True`, start the job (i.e., run the first iteration) immediately during the call; otherwise, start it only when it is scheduled, after the currently running call is complete. Local call method.

is_batch_job_running(*name*)

Check if a given batch job running.

Local call method.

stop_batch_job(*name*, *stop_immediate=True*, *error_on_stopped=False*)

Stop a given batch job.

If `error_on_stopped==True` and the job is not currently running, raise an error. Otherwise, do nothing. If `stop_immediate==True`, stop the job (i.e., unschedule it and run the cleanup code) immediately during the call; otherwise, stop it when its next iteration is called. Local call method.

restart_batch_job(*name*, *start_immediate=True*, *error_on_stopped=False*)

Restart the running batch job with its current arguments.

If `error_on_stopped==True` and the job is not currently running, raise an error. Otherwise, do nothing. Local call method.

run_as_batch_job(*job*, *period*, *cleanup=None*, *name=None*, *priority=-10*, *start_immediate=True*, *args=None*, *kwargs=None*)

Create a temporarily batch job and immediately run it.

If *name* is `None`, generate a new unique name. The job is removed after it is complete (i.e., after cleanup). Note that this implies, that it can not be restarted using `restart_batch_job()`, as it will be removed after the stopping before the restart. All the parameters are the same as for `add_batch_job()` and `start_batch_job()`. Return the batch job name (either supplied or newly generated).

run()

Method called to run the main thread code (only for `"run"` thread kind).

Local call method, called automatically.

on_start()

Method invoked on the start of the thread.

Local call method, called automatically.

on_finish()

Method invoked in the end of the thread.

Called regardless of the stopping reason (normal finishing, exception, application finishing). Local call method, called automatically.

setup_task(**args*, ***kwargs*)

Setup the thread (called before the main task loop).

Local call method, called automatically.

finalize_task()

Finalize the thread (always called on thread termination, regardless of the reason).

Local call method, called automatically.

update_status(*kind*, *status*, *text=None*, *notify=True*)

Update status represented in thread variables.

kind is the status kind and *status* is its value. Status variable name is `"status/"+kind`. If `text is not None`, it specifies new status text stored in `"status/"+kind+"_text"`. If `notify==True`, send an multicast about the status change. Local call method.

add_command(*name*, *command=None*, *scheduler=None*, *limit_queue=None*, *on_full_queue='skip_current'*, *priority=0*)

Add a new command to the command set.

Return scheduler, which can be used for adding another command (if the same queue should be used for several commands). Local call method.

Parameters

- **name** – command name
- **command** – command function; if `None`, look for the method with the given *name*.
- **scheduler** – a command scheduler; by default, it is a *QQueueLengthLimitScheduler*, which maintains a call queue with the given length limit and full queue behavior; can also be a name of a different command, with which it will share a single queue with the same limitations; if supplied, *limit_queue* and *on_full_queue* parameters are ignored
- **limit_queue** – command call queue limit; `None` means no limit
- **on_full_queue** – action to be taken if the call can't be scheduled (the queue is full); can be `"skip_current"` (skip the call which is being scheduled), `"skip_newest"` (skip the most recent call; place the current) `"skip_oldest"` (skip the oldest call in the queue; place the current), `"call_current"` (execute the call which is being scheduled immediately in the caller thread), `"call_newest"` (execute the most recent call immediately in the caller thread), `"call_oldest"` (execute the oldest call in the queue immediately in the caller thread), or `"wait"` (wait until the call can be scheduled, which is checked after every call removal from the queue; place the call)

- **priority** – command priority; higher-priority multicasts and commands are always executed before the lower-priority ones.

add_direct_call_command(*name*, *command=None*, *error_on_async=True*)
Add a direct method call which appears as a command.

Unlike regular commands, the call is executed directly in the caller thread (i.e., it is identical to the direct method call). Useful for lightweight and/or lock-wrapped methods, which can be called in a thread-safe way, but which still use command interface for consistency. Note that this kind of commands doesn't have the same level of synchronization as regular commands (e.g., it can be executed during execution of another command, or commsync multicast method). Local call method.

Parameters

- **name** – command name
- **command** – command function; if `None`, look for the method with the given *name*.
- **error_on_async** – if `True` and the command is called asynchronously, raise an error; otherwise, substitute for a synchronous call

subscribe_commsync(*callback*, *srcs='any'*, *tags=None*, *dsts='any'*, *filt=None*, *subscription_priority=0*, *scheduler=None*, *limit_queue=None*, *on_full_queue='skip_current'*, *priority=0*, *add_call_info=False*, *sid=None*)
Subscribe a callback to a multicast which is synchronized with commands and jobs execution.

Unlike the standard `QThreadController.subscribe_sync()` method, the subscribed callback will only be executed between jobs or commands, not during one of these. Local call method.

Parameters

- **callback** – callback function, which takes 3 arguments: source, tag, and value.
- **srcs** (*str or [str]*) – multicast source name or list of source names to filter the subscription; can be `"any"` (any source) or `"all"` (only multicasts specifically having `"all"` as a source).
- **tags** – multicast tag or list of tags to filter the subscription (any tag by default); can also contain Unix shell style pattern (`"*"` matches everything, `"?"` matches one symbol, etc.)
- **dsts** (*str or [str]*) – multicast destination name or list of destination names to filter the subscription; can be `"any"` (any destination) or `"all"` (only source specifically having `"all"` as a destination).
- **filt** (*callable*) – additional filter function which takes 4 arguments: source, destination, tag, and value, and checks whether multicast passes the requirements.
- **subscription_priority** (*int*) – subscription priority (higher priority subscribers are called first).
- **scheduler** – if defined, multicast call gets scheduled using this scheduler; by default, create a new call queue scheduler with the given *limit_queue*, *on_full_queue* and *add_call_info* arguments.
- **limit_queue** (*int*) – limits the maximal number of scheduled calls (if the multicast is sent while at least *limit_queue* callbacks are already in queue to be executed, ignore it) 0 or negative value means no limit (not recommended, as it can increase the queue indefinitely if the multicast rate is high enough)
- **on_full_queue** – action to be taken if the call can't be scheduled (the queue is full); can be `"skip_current"` (skip the call which is being scheduled), `"skip_newest"` (skip the most recent call; place the current) `"skip_oldest"`

(skip the oldest call in the queue; place the current), "call_current" (execute the call which is being scheduled immediately in the caller thread), "call_newest" (execute the most recent call immediately in the caller thread), "call_oldest" (execute the oldest call in the queue immediately in the caller thread), or "wait" (wait until the call can be scheduled, which is checked after every call removal from the queue; place the call)

- **add_call_info** (*bool*) – if True, add a fourth argument containing a call information (tuple with a single element, a timestamps of the call).
- **sid** (*int*) – subscription ID (by default, generate a new unique id and return it).

call_command_direct (*name*, *args=None*, *kwargs=None*)
Invoke a command directly and immediately in the current thread.

Universal call method.

call_command (*name*, *args=None*, *kwargs=None*, *sync=False*, *callback=None*, *timeout=None*, *ignore_errors=False*)
Invoke command call with the given name and arguments

If *callback* is not None, call it after the command is successfully executed (from the target thread), with a single parameter being the command result. If sync==True, pause caller thread execution (for at most *timeout* seconds) until the command has been executed by the target thread, and then return the command result. If sync=="delayed", return *QCallResultSynchronizer* object which can be used to wait for and read the command result; otherwise, return None. In the sync==True case, if ignore_errors==True, ignore all possible problems with the call (controller stopped, call raised an exception, call was skipped) and return None instead; otherwise, these problems raise exceptions in the caller thread. Universal call method.

call_in_thread_commsync (*func*, *args=None*, *kwargs=None*, *sync=True*, *timeout=None*, *priority=0*, *ignore_errors=False*, *same_thread_shortcut=True*)
Call a function in this thread such that it is synchronous with other commands, and jobs.

Mostly equivalent to calling a command, only the command function is supplied instead of its name, and the advanced scheduling (maximal schedule size, sharing with different commands, etc.) is not used. *args* and *kwargs* specify the function arguments. If sync==True, pause caller thread execution (for at most *timeout* seconds) until the command has been executed by the target thread, and then return the command result. If sync=="delayed", return *QCallResultSynchronizer* object which can be used to wait for and read the command result; otherwise, return None. *priority* sets the call priority (by default, the same as the standard commands). In the sync==True case, if ignore_errors==True, ignore all possible problems with the call (controller stopped, call raised an exception, call was skipped) and return None instead; otherwise, these problems raise exceptions in the caller thread. If same_thread_shortcut==True (default) and the caller thread is the same as the controlled thread, call the function directly. Universal call method.

comm_paused ()
Context manager, which allows to temporarily pause all calls (commands, jobs, etc.)

class CommandAccess (*parent*, *sync*, *direct=False*, *timeout=None*, *safe=False*, *ignore_errors=False*)
Bases: object

Accessor object designed to simplify command syntax.

Automatically created by the thread, so doesn't need to be invoked externally.

add_stop_notifier (*func*, *call_if_stopped=True*)
Add stop notifier: a function which is called when the thread is about to be stopped (left the main message loop).

The supplied function is called in the controlled thread close to its shutdown, so it should be short, non-blocking, and thread-safe. If the thread is already stopped and `call_if_stopped==True`, call *func* immediately (from the caller's thread). Return `True` if the thread is still running and the notifier is added, and `False` otherwise. Local call method.

add_thread_method(*name*, *method*, *interrupt=True*)

Add a thread method.

Adds a named method to the thread, which can be called later using `call_thread_method()`. This method will be called in this thread.

Useful for GUI thread to set up some global access methods, which other threads can safely use. For `QTaskThread` threads it's a better idea to set up a command instead. Local call method.

call_in_thread_callback(*func*, *args=None*, *kwargs=None*, *callback=None*, *tag=None*, *priority=0*, *interrupt=True*)

Call a function in this thread with the given arguments.

If *callback* is supplied, call it with the result as a single argument (call happens in the controller thread). If *tag* is supplied, send the call in a message with the given tag; otherwise, use the interrupt call (generally, higher priority method). If `interrupt==True`, method can be called inside any control loop (either main loop, or during waiting); otherwise, only call it in the top loop. Universal call method.

call_in_thread_sync(*func*, *args=None*, *kwargs=None*, *sync=True*, *callback=None*, *timeout=None*, *default_result=None*, *pass_exception=True*, *silent=False*, *tag=None*, *priority=0*, *interrupt=True*, *error_on_stopped=True*, *same_thread_shortcut=True*)

Call a function in this thread with the given arguments.

If `sync==True`, calling thread is blocked until the controlled thread executes the function, and the function result is returned (in essence, the fact that the function executes in a different thread is transparent). Otherwise, exit call immediately, and return a synchronizer object (`QCallResultSynchronizer`), which can be used to check if the call is done (method *is_done*) and obtain the result (method `QCallResultSynchronizer.get_value_sync()`). If *callback* is not `None`, call it after the function is successfully executed (from the target thread), with a single parameter being function result. If `pass_exception==True` and *func* raises an exception, re-raise it in the caller thread (applies only if `sync==True`). If `silent==True` and *func* raises an exception, silence it in the execution thread and only re-raise it in the caller thread; note that if `pass_exception==False` and `silent==True`, the exception is ignored in both threads. If *tag* is supplied, send the call in a message with the given tag and priority; otherwise, use the interrupt call (generally, higher priority method). If `interrupt==True`, method can be called inside any control loop (either main loop, or during waiting); otherwise, only call it in the top loop. If `error_on_stopped==True` and the controlled thread is stopped before it executed the call, raise `threadprop.NoControllerThreadError`; otherwise, return *default_result*. If `same_thread_shortcut==True` (default), the call is synchronous, and the caller thread is the same as the controlled thread, call the function directly. Universal call method.

call_thread_method(*name*, **args*, ***kwargs*)

Call a thread method.

Method needs to be set up beforehand using `add_thread_method()`. It is always executed in the current thread. Local call method.

check_messages(*top_loop=False*)

Receive new messages.

Runs the underlying message loop to process newly received message and signals (and place them in corresponding queues if necessary). This method is rarely invoked, and only should be used periodically during long computations to not 'freeze' the thread. If `top_loop==True`, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). Local call method.

delete_thread_method(*name*)
Delete a thread method.

Local call method.

delete_variable(*name*, *missing_error=False*)
Delete thread variable.

If `missing_error==False` and no variable exists, do nothing; otherwise, raise and error. Local call method.

fail_exec_point(*point*)
Mark the given execution point as failed.

Automatically invoked for `"run"` (thread setup and ready to run) if the startup raised an error before the thread properly started (`"start"`, `"cleanup"`, and `"stop"` are notified in any case) Can be extended for arbitrary points. Local call method.

finished = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

finishing()
Check if the thread is finishing

get_exec_counter(*point*)
Get the counter (number of notifications) for the given point.

See `sync_exec_point()` for details. External call.

get_variable(*name*, *default=None*, *copy_branch=True*, *missing_error=False*, *simple=False*)
Get thread variable.

If `missing_error==False` and no variable exists, return *default*; otherwise, raise and error. If `copy_branch==True` and the variable is a `Dictionary` branch, return its copy to ensure that it stays unaffected on possible further variable assignments. If `simple==True`, assume that the result is a single atomic variable, in which case the lock is not used; this only works with actual variables and not function variables. Universal call method.

is_in_controlled()
Check if the thread executing this code is controlled by this controller

new_messages_number(*tag*)
Get the number of queued messages with a given tag.

Local call method.

no_stopping()
Context manager, which temporarily suspends stop requests (`InterruptExceptionStop` exceptions).

If the stop request has been made within this block, raise the exception on exit. Note that `stop()` method and, correspondingly, `stop_controller()` still work, when called from the controlled thread.

notify_exec_point(*point*)
Mark the given execution point as passed.

Automatically invoked points include `"start"` (thread starting), `"run"` (thread setup and ready to run), `"cleanup"` (thread stopping is invoked, starting to clean up) and `"stop"` (thread finished). Can be extended for arbitrary points. Local call method.

poke()
Send a dummy message to the thread.

A cheap way to notify the thread that something happened (useful for, e.g., making thread leave `wait_for_any_message()` method). External call method.

pop_message(*tag*)
Pop the latest message with the given tag.

Select the message with the highest priority, and among those the oldest one. If no messages are available, raise `threadprop.NoMessageThreadError`. Local call method.

process_interrupt(*tag*, *value*)
Process a new interrupt.

If the function returns `False`, the interrupt is put in the corresponding queue. Otherwise, the the message is interrupt to be already, and it gets 'absorbed'. Local call method, called automatically.

process_message(*tag*, *value*)
Process a new message.

If the function returns `False`, the message is put in the corresponding queue. Otherwise, the the message is considered to be already, and it gets 'absorbed'. Local call method, called automatically.

remove_stop_notifier(*func*)
Remove the stop notifier from this controller.

Return `True` if the notifier was in this thread and is now removed, and `False` otherwise. Local call method.

request_stop()
Request thread stop (send a stop command).

External call method.

running()
Check if the thread is running

send_interrupt(*tag*, *value*, *priority=0*)
Send an interrupt message to the thread with a given tag, value and priority.

External call method.

send_message(*tag*, *value*, *priority=0*)
Send a message to the thread with a given tag, value and priority.

External call method.

send_multicast(*dst='any'*, *tag=None*, *value=None*, *src=None*)
Send a multicast to the multicast pool.

By default, the multicast source is the thread's name. Local call method.

Parameters

- **dst** (`str`) – multicast destination; can be a name, `"all"` (will pass all subscribers' destination filters), or `"any"` (will only be passed to subscribers specifically subscribed to multicast with `"any"` destination).
- **tag** (`str`) – multicast tag.
- **value** – multicast value.
- **src** (`str`) – multicast source; can be `None` (current thread name), a specific name, `"all"` (will pass all subscribers' source filters), or `"any"` (will only be passed to subscribers specifically subscribed to multicast with `"any"` source).

send_sync(*tag*, *uid*)
Send a synchronization signal with the given tag and UID.

This method is rarely invoked directly, and is usually used by synchronizers code (e.g., *QThreadNotifier*). External call method.

set_func_variable(*name*, *func*, *use_lock=True*)
Set a 'function' variable.

Acts as a thread variable to the external user, but instead of reading a stored value, it executed a function instead. Note, that the function is executed in the caller thread (i.e., the thread which tries to access the variable), so use of synchronization methods (commands, signals, locks) is highly advised.

If `use_lock==True`, then the function call will be wrapped into the usual variable lock, i.e., it won't run concurrently with other variable access. Local call method.

set_variable(*name*, *value*, *update=False*, *notify=False*, *notify_tag='changed/*'*, *simple=False*)
Set thread variable.

Can be called in any thread (controlled or external). If `notify==True`, send an multicast with the given *notify_tag* (where `"*"` symbol is replaced by the variable name). If `update==True` and the value is a dictionary, update the branch rather than overwrite it. If `simple==True`, assume that the result is a single atomic variable, in which case the lock is not used; note that in this case the threads waiting on this variable (or branches containing it) will not be notified. Local call method.

sleep(*timeout*, *wake_on_message=False*, *top_loop=False*)
Sleep for a given time (in seconds).

Unlike `time.sleep()`, constantly checks the event loop for new messages (e.g., if stop or interrupt commands are issued). In addition, if `wake_on_message==True`, wake up if any message has been received; it this case. return `True` if the wait has been completed, and `False` if it has been interrupted by a message. If `top_loop==True`, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). If `timeout is None`, wait forever (usually, until the application is closed, or some interrupt message raises and error). Local call method.

start()
Start the thread.

External call method.

started = <Mock name='mock.QtCore.pyqtSignal()' id='140318724726672'>

stop(*code=0*, *sync=False*)
Stop the thread.

If called from the thread, stop immediately by raising a *threadprop.InterruptExceptionStop* exception. Otherwise, schedule thread stop. If the thread kind is `"main"`, stop the whole application with the given exit code. Otherwise, stop the thread. If `sync==True` and the thread is not main or current, wait until it is completely stopped. Universal call method.

subscribe_direct(*callback*, *srcs='any'*, *tags=None*, *dsts='any'*, *filt=None*, *subscription_priority=0*, *scheduler=None*, *sid=None*)
Subscribe asynchronous callback to a multicast.

If a multicast is sent, *callback* is called from the sending thread (not subscribed thread). Therefore, should be used with care. In Qt, analogous to making a signal connection with a direct call. By default, the subscribed destination is the thread's name. Local call method.

Parameters

- **callback** – callback function, which takes 3 arguments: source, tag, and value.
- **srcs** (*str or [str]*) – multicast source name or list of source names to filter the subscription; can be `"any"` (any source) or `"all"` (only multicasts specifically having `"all"` as a source).

- **tags** – multicast tag or list of tags to filter the subscription (any tag by default); can also contain Unix shell style pattern (`"*"` matches everything, `"?"` matches one symbol, etc.)
- **dsts** (*str or [str]*) – multicast destination name or list of destination names to filter the subscription; can be `"any"` (any destination) or `"all"` (only source specifically having `"all"` as a destination).
- **filt** (*callable*) – additional filter function which takes 4 arguments: source, destination, tag, and value, and checks whether multicast passes the requirements.
- **subscription_priority** (*int*) – subscription priority (higher priority subscribers are called first).
- **scheduler** – if defined, multicast call gets scheduled using this scheduler instead of being called directly (which is the default behavior)
- **sid** (*int*) – subscription ID (by default, generate a new unique id and return it).

subscribe_sync(*callback*, *srcs='any'*, *tags=None*, *dsts='any'*, *filt=None*, *subscription_priority=0*, *limit_queue=None*, *call_interrupt=True*, *add_call_info=False*, *sid=None*)

Subscribe a synchronous callback to a multicast.

If a multicast is sent, *callback* is called from the *dest_controller* thread (by default, thread which is calling this function) via the thread call mechanism (`QThreadController.call_in_thread_callback()`). In Qt, analogous to making a signal connection with a queued call. By default, the subscribed destination is the thread's name. Local call method.

Parameters

- **callback** – callback function, which takes 3 arguments: source, tag, and value.
- **srcs** (*str or [str]*) – multicast source name or list of source names to filter the subscription; can be `"any"` (any source) or `"all"` (only multicasts specifically having `"all"` as a source).
- **tags** – multicast tag or list of tags to filter the subscription (any tag by default); can also contain Unix shell style pattern (`"*"` matches everything, `"?"` matches one symbol, etc.)
- **dsts** (*str or [str]*) – multicast destination name or list of destination names to filter the subscription; can be `"any"` (any destination) or `"all"` (only source specifically having `"all"` as a destination).
- **filt** (*callable*) – additional filter function which takes 4 arguments: source, destination, tag, and value, and checks whether multicast passes the requirements.
- **limit_queue** (*int*) – limits the maximal number of scheduled calls (if the multicast is sent while at least *limit_queue* callbacks are already in queue to be executed, ignore it) 0 or negative value means no limit (not recommended, as it can increase the queue indefinitely if the multicast rate is high enough)
- **call_interrupt** – whether the call is an interrupt (call inside any loop, e.g., during waiting or sleeping), or it should be called in the main event loop
- **subscription_priority** (*int*) – subscription priority (higher priority subscribers are called first).
- **sid** (*int*) – subscription ID (by default, generate a new unique id and return it).

sync_exec_point(*point*, *timeout=None*, *counter=1*)

Wait for the given execution point.

Automatically invoked points include "start" (thread starting), "run" (thread setup and ready to run), "cleanup" (thread stopping is invoked, starting to clean up) and "stop" (thread finished). If timeout is passed, raise *threadprop.TimeoutThreadError*. *counter* specifies the minimal number of pre-requisite *notify_exec_point()* calls to finish the waiting (by default, a single call is enough). Return actual number of notifier calls up to date. External call method.

sync_stop()

Wait until the controller and the thread are stopped.

External call method.

sync_variable(*name*, *pred*, *timeout=None*)

Wait until thread variable with the given *name* satisfies the condition given by *pred*.

pred is a function which takes one argument (variable value) and returns whether the condition is satisfied. It is executed in the caller thread. External call method.

unsubscribe(*sid*)

Unsubscribe from a subscription with a given ID.

Note that multicasts which are already emitted but not processed will remain in the queue; if they need to be ignored, it should be handled explicitly. Local call method.

wait_for_any_message(*timeout=None*, *top_loop=False*)

Wait for any message (including synchronization messages or pokes).

If timeout is passed, raise *threadprop.TimeoutThreadError*. If top_loop==True, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). Local call method.

wait_for_message(*tag*, *timeout=None*, *top_loop=False*)

Wait for a single message with a given tag.

Return value of a received message with this tag. If timeout is passed, raise *threadprop.TimeoutThreadError*. If top_loop==True, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). Local call method.

wait_for_sync(*tag*, *uid*, *timeout=None*)

Wait for synchronization signal with the given tag and UID.

This method is rarely invoked directly, and is usually used by synchronizers code. If timeout is passed, raise *threadprop.TimeoutThreadError*. Local call method.

wait_until(*check*, *timeout=None*, *top_loop=False*)

Wait until a given condition is true.

Condition is given by the *check* function, which is called after every new received message and should return True if the condition is met. If top_loop==True, treat the waiting as the top message loop (i.e., any top loop message or signal can be executed here). If timeout is passed, raise *threadprop.TimeoutThreadError*. Local call method.

pylablib.core.thread.controller.**get_controller**(*name=None*, *sync=True*, *timeout=None*, *sync_point=None*)

Find a controller with a given name.

If *name* is not supplied, yield current controller instead. If the controller is not present and sync==True, wait (with the given timeout) until the controller is running; otherwise, raise error if the controller is not running. If *sync_point* is not None, synchronize to the thread *sync_point* point (by default, "run", i.e., after the setup is done) before returning.

pylablib.core.thread.controller.**sync_controller**(*name*, *sync_point='run'*, *timeout=None*)

Find a controller with a given name and synchronize to the given point.

If the controller is not present and `sync==True`, wait (with the given timeout) until the controller is running; otherwise, raise error if the controller is not running. Analogous to `get_controller(name, sync=True, timeout=timeout, sync_point=sync_point)`.

`pylablib.core.thread.controller.`**`get_gui_controller`**`(`*sync=False*, *timeout=None*, *create_if_missing=True*`)`

Get GUI thread controller.

If the controller is not present and `sync==True`, wait (with the given timeout) until the controller is running. If the controller is still not present and `create_if_missing==True`, initialize the standard GUI controller.

`pylablib.core.thread.controller.`**`stop_controller`**`(`*name=None*, *code=0*, *sync=True*, *require_controller=False*`)`

Stop a controller with a given name (current controller by default).

code specifies controller exit code (only applies to the main thread controller). If `require_controller==True` and the controller is not present, raise and error; otherwise, do nothing. If `sync==True`, wait until the controller is stopped.

`pylablib.core.thread.controller.`**`stop_all_controllers`**`(`*sync=True*, *concurrent=True*, *stop_self=True*`)`

Stop all running threads.

If `sync==True`, wait until the all of the controller are stopped. If `sync==True` and `concurrent==True` stop threads in concurrent manner (first issue stop messages to all of them, then wait until all are stopped). If `sync==True` and `concurrent==False` stop threads in consecutive manner (wait for each thread to stop before stopping the next one). If `stop_self==True` stop current thread after stopping all other threads.

`pylablib.core.thread.controller.`**`stop_app`**`(`*code=0*, *sync=False*`)`

Initialize stopping the application.

Do this either by stopping the GUI controller (if it exists), or by stopping all controllers. If *sync* is `True` and the thread is not the main one, wait at this point until the process is stopped during the app shutdown; otherwise, the execution will continue as normal, and the thread will be stopped at a later time during the app shutdown.

`pylablib.core.thread.controller.`**`restart_app`**`(`*code=0*, *sync=False*`)`

Restart the application.

Equivalent to `stop_app()` followed by the scrip restart. If *sync* is `True` and the thread is not the main one, wait at this point until the process is stopped during the app shutdown; otherwise, the execution will continue as normal, and the thread will be stopped at a later time during the app shutdown.

pylablib.core.thread.multicast_pool module

`class` `pylablib.core.thread.multicast_pool.`**`TMulticast`**`(`*src*, *tag*, *value*`)`

Bases: `tuple`

`count``()`

Return number of occurrences of value.

`index``()`

Return first index of value.

Raises ValueError if the value is not present.

`src`

`tag`

`value`

class pylablib.core.thread.multicast_pool.**MulticastPool**

Bases: object

Multicast dispatcher (somewhat similar in functionality to Qt signals).

Manages dispatching multicasts between sources and destinations (callback functions). Each multicast has defined source, destination (both can also be "all" or "any", see methods descriptions for details), tag and value. Any thread can send a multicast or subscribe for a multicast with given filters (source, destination, tag, additional filters). If a multicast is emitted, it is checked against filters for all subscribers, and the passing ones are then called.

subscribe_direct(*callback*, *srcs='any'*, *dsts='any'*, *tags=None*, *filt=None*, *priority=0*, *scheduler=None*, *sid=None*)

Subscribe an asynchronous callback to a multicast.

If a multicast is sent, *callback* is called from the sending thread (not subscribed thread). Therefore, should be used with care. In Qt, analogous to making a signal connection with a direct call.

Parameters

- **callback** – callback function, which takes 3 arguments: source, tag, and value.
- **srcs** (*str or [str]*) – multicast source name or list of source names to filter the subscription; can be "any" (any source) or "all" (only multicasts specifically having "all" as a source).
- **dsts** (*str or [str]*) – multicast destination name or list of destination names to filter the subscription; can be "any" (any destination) or "all" (only source specifically having "all" as a destination).
- **tags** – multicast tag or list of tags to filter the subscription (any tag by default); can also contain Unix shell style pattern ("*" matches everything, "?" matches one symbol, etc.)
- **filt** (*callable*) – additional filter function which takes 4 arguments: source, destination, tag, and value, and checks whether multicast passes the requirements.
- **priority** (*int*) – subscription priority (higher priority subscribers are called first).
- **scheduler** – if defined, multicast call gets scheduled using this scheduler instead of being called directly (which is the default behavior)
- **sid** (*int*) – subscription ID (by default, generate a new unique name).

Returns subscription ID, which can be used to unsubscribe later.

subscribe_sync(*callback*, *srcs='any'*, *dsts='any'*, *tags=None*, *filt=None*, *priority=0*, *limit_queue=None*, *dest_controller=None*, *call_tag=None*, *call_interrupt=True*, *add_call_info=False*, *sid=None*)

Subscribe a synchronous callback to a multicast.

If a multicast is sent, *callback* is called from the *dest_controller* thread (by default, thread which is calling this function) via the thread call mechanism (*QThreadController.call_in_thread_callback()*). In Qt, analogous to making a signal connection with a queued call.

Parameters

- **callback** – callback function, which takes 3 arguments: source, tag, and value.
- **srcs** (*str or [str]*) – multicast source name or list of source names to filter the subscription; can be "any" (any source) or "all" (only multicasts specifically having "all" as a source).

- **dsts** (*str or [str]*) – multicast destination name or list of destination names to filter the subscription; can be "any" (any destination) or "all" (only source specifically having "all" as a destination).
- **tags** – multicast tag or list of tags to filter the subscription (any tag by default); can also contain Unix shell style pattern ("*" matches everything, "?" matches one symbol, etc.)
- **filt** (*callable*) – additional filter function which takes 4 arguments: source, destination, tag, and value, and checks whether multicast passes the requirements.
- **priority** (*int*) – subscription priority (higher priority subscribers are called first).
- **limit_queue** (*int*) – limits the maximal number of scheduled calls (if the multicast is sent while at least *limit_queue* callbacks are already in queue to be executed, ignore it) 0 or negative value means no limit (not recommended, as it can increase the queue indefinitely if the multicast rate is high enough)
- **call_tag** (*str or None*) – tag used for the synchronized call; by default, use the interrupt call (which is the default of `call_in_thread`).
- **call_interrupt** – whether the call is an interrupt (call inside any loop, e.g., during waiting or sleeping), or it should be called in the main event loop
- **add_call_info** (*bool*) – if `True`, add a fourth argument containing a call information (tuple with a single element, a timestamps of the call).
- **sid** (*int*) – subscription ID (by default, generate a new unique name).

Returns subscription ID, which can be used to unsubscribe later.

unsubscribe(*sid*)
Unsubscribe from a subscription with a given ID

send(*src*, *dst='any'*, *tag=None*, *value=None*)
Send a multicast.

Parameters

- **src** (*str*) – multicast source; can be a name, "all" (will pass all subscribers' source filters), or "any" (will only be passed to subscribers specifically subscribed to multicasts with "any" source).
- **dst** (*str*) – multicast destination; can be a name, "all" (will pass all subscribers' destination filters), or "any" (will only be passed to subscribers specifically subscribed to multicasts with "any" destination).
- **tag** (*str*) – multicast tag.
- **value** – multicast value.

pylablib.core.thread.notifier module

class pylablib.core.thread.notifier.**ISkippableNotifier**(*skippable=False*)
Bases: `object`

Generic skippable notifier.

The main methods are `wait()` (wait until the event happened) and `notify()` (notify that the event happened). Only calls underlying waiting and notifying methods once, duplicate calls are ignored.

Parameters **skippable** (*bool*) – if True, allows for skippable wait events (if *notify()* is called before *wait()*, neither methods are actually called).

wait(**args*, ***kwargs*)
Wait for the notification.

Can only be called once per notifier lifetime. If the notifier allows skipping, and this method is called after *notify()*, return immediately.

notify(**args*, ***kwargs*)
Notify the waiting process.

Can only be called once per notifier lifetime. If the notifier allows skipping, and this method is called before *wait()*, return immediately.

waiting()
Check if waiting is in progress

done_wait()
Check if waiting is done

success_wait()
Check if waiting is done successfully

done_notify()
Check if notifying is done

waiting_state()

notifying_state()

pylablib.core.thread.synchronizing module

class pylablib.core.thread.synchronizing.**QThreadNotifier**(*skippable=True*)
Bases: *pylablib.core.thread.notifier.ISkippableNotifier*

Wait-notify thread synchronizer for controlled Qt threads based on *notifier.ISkippableNotifier*.

Like *notifier.ISkippableNotifier*, the main functions are *ISkippableNotifier.wait()* (wait in a message loop until notified or until timeout expires) and *ISkippableNotifier.notify()* (notify the waiting thread). Both of these can only be called once and will raise and error on repeating calls. Along with notifying a variable can be passed, which can be accessed using *get_value()* and *get_value_sync()*.

Parameters **skippable** (*bool*) – if True, allows for skippable wait events (if *ISkippableNotifier.notify()* is called before *ISkippableNotifier.wait()*, neither methods are actually called).

get_value()
Get the value passed by the notifier (doesn't check if it has been passed already)

get_value_sync(*timeout=None*)
Wait (with the given *timeout*) for the value passed by the notifier

done_notify()
Check if notifying is done

done_wait()
Check if waiting is done

notify(**args*, ***kwargs*)
Notify the waiting process.

Can only be called once per notifier lifetime. If the notifier allows skipping, and this method is called before `wait()`, return immediately.

notifying_state()

success_wait()
Check if waiting is done successfully

wait(**args*, ***kwargs*)
Wait for the notification.

Can only be called once per notifier lifetime. If the notifier allows skipping, and this method is called after `notify()`, return immediately.

waiting()
Check if waiting is in progress

waiting_state()

class pylablib.core.thread.synchronizing.**QMultiThreadNotifier**
Bases: `object`

Wait-notify thread synchronizer that can be used for multiple threads and called multiple times.

Performs similar function to conditional variables. The synchronizer has an internal counter which is increased by 1 every time it is notified. The wait functions have an option to wait until the counter reaches the specific counter value (usually, 1 above the last wait call).

wait(*state=1*, *timeout=None*)
Wait until notifier counter is equal to at least *state*

Return current counter state plus 1, which is the next smallest value resulting in waiting.

wait_until(*condition*, *timeout=None*)
Wait until *condition* is met.

condition is a function which is called (in the waiting thread) every time the synchronizer is notified. If it return non-`False`, the waiting is complete and its result is returned.

notify()
Notify all waiting threads

fail()
Mark notifier as fails

Fails all waiting notifiers. All subsequent wait calls raise an error

class pylablib.core.thread.synchronizing.**QLockNotifier**
Bases: `object`

Resource lock.

Behaves similarly to the regular lock, but waiting is done in the message loop, which still allows interrupts.

acquire(*timeout=None*)

release()

pylablib.core.thread.threadprop module

exception pylablib.core.thread.threadprop.**ThreadError**(*msg=None*)

Bases: RuntimeError

Generic thread error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.core.thread.threadprop.**NoControllerThreadError**(*msg=None*)

Bases: *pylablib.core.thread.threadprop.ThreadError*

Thread error for a case of thread having no controllers

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.core.thread.threadprop.**DuplicateControllerThreadError**(*msg=None*)

Bases: *pylablib.core.thread.threadprop.ThreadError*

Thread error for a case of a duplicate thread controller

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.core.thread.threadprop.**TimeoutThreadError**(*msg=None*)

Bases: *pylablib.core.thread.threadprop.ThreadError*, TimeoutError

Thread error for a case of a wait timeout

args

characters_written

errno

POSIX exception code

filename

exception filename

filename2

second exception filename

strerror

exception strerror

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.core.thread.threadprop.**NoMessageThreadError**(*msg=None*)

Bases: *pylablib.core.thread.threadprop.ThreadError*

Thread error for a case of trying to get a non-existing message

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.core.thread.threadprop.**SkippedCallError**(*msg=None*)
Bases: *pylablib.core.thread.threadprop.ThreadError*

Thread error for a case of external call getting skipped (unscheduled)

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.core.thread.threadprop.**InterruptException**(*msg=None*)
Bases: Exception

Generic interrupt exception (raised by some function to signal interrupts from other threads)

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.core.thread.threadprop.**InterruptExceptionStop**(*msg=None*)
Bases: *pylablib.core.thread.threadprop.InterruptException*

Interrupt exception denoting thread stop request

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

pylablib.core.thread.threadprop.**get_app**()
Get current application instance

pylablib.core.thread.threadprop.**get_gui_thread**()
Get main (GUI) thread, or None if application is not running

pylablib.core.thread.threadprop.**is_gui_running**()
Check if GUI is running

pylablib.core.thread.threadprop.**is_gui_thread**()
Check if the current thread is the one running the GUI loop

pylablib.core.thread.threadprop.**current_controller**(*require_controller=True*)
Get controller of the current thread.

If the current thread has not controller *and 'require_controller==True'*, raise an error; otherwise, return None.

Module contents

pylablib.core.utils package

Submodules

pylablib.core.utils.array_utils module

pylablib.core.utils.array_utils.**as_array**(*data*, *force_copy=False*, *try_object=True*)
Turn *data* into a numpy array.

If force_copy==True, copy the data if it's already a numpy array. If try_object==False, only try to convert to numerical numpy arrays; otherwise, generic numpy arrays (with dtype=="object") are acceptable.

pylablib.core.utils.array_utils.**get_shape**(*data*, *strict=False*)

Get the data shape.

If the data is a nested list and `strict==True`, raise an error unless all sublists have the same length (i.e., the data is rectangular).

pylablib.core.utils.ctypes_wrap module

pylablib.core.utils.ctypes_wrap.**get_value**(*rval*)

Get value of a ctypes variable

pylablib.core.utils.ctypes_wrap.**setup_func**(*func*, *argtypes*, *restype=None*, *errcheck=None*)

Setup a ctypes function.

Assign argtypes (list of argument types), restype (return value type) and errcheck (error checking function called for the return value).

class pylablib.core.utils.ctypes_wrap.**CFunctionWrapper**(*restype=None*, *errcheck=None*, *tuple_single_retval=False*, *return_res='auto'*, *default_rvals='rest'*, *pointer_byref=False*)

Bases: `object`

Wrapper object for ctypes function.

The main methods are *wrap_annotated()* and *wrap_bare()*, which wrap a ctypes function and returns a Python function with a proper signature. These methods can also handle some standard use cases such as passing parameters by reference, or setting up the function arguments, or parsing the results. These methods can also be invoked when the wrapper is used as a callable; in this case, the exact method is determined by the presence of `.argtypes` attribute in the supplied function.

Parameters

- **restype** – default type of the function return value when calling *wrap_bare()* and `restype` is not supplied there explicitly (defaults to `ctypes.int`)
- **errcheck** – default error-checking function which is automatically called for the return value; can also be overridden explicitly when calling wrapping methods if `None`, no error checking method
- **tuple_single_retval** (*bool*) – determines if a single return values gets turned into a single-element tuple
- **return_res** (*bool*) – determined if the function result gets returned; only used when list of return arguments (`rvals`) to wrapping functions is not explicitly supplied; can also be set to `"auto"` (default), which means that function returns its return value when no other rvals are found, and omits it otherwise.
- **default_rvals** – default value for `rvals` in *wrap_annotated()* and *wrap_bare()*, if it is specified as `None` (default for those methods).
- **pointer_byref** (*bool*) – if `True`, use explicit pointer creation instead of byref (in rare cases use of byref crashes the call).

byref(*value*)

wrap_bare (*func*, *argtypes*, *argnames=None*, *restype=None*, *args='nonrval'*, *rvals='default'*, *argprep=None*, *rconv=None*, *byref='all'*, *errcheck=None*)

Annotate and wrap bare C function in a Python call.

Same as `wrap_annotated()`, but annotates the function first.

Parameters

- **func** – C function
- **argtypes** – list of ctypes types corresponding to function arguments; gets assigned as `func.argtypes`
- **argnames** – list of argument names; if not supplied, generated automatically as `"arg1"`, `"arg2"`, etc. Same for names which are defined as `None`.
- **restype** – type of the function return value; if `None`, use the value supplied to the wrapper constructor (defaults to `ctypes.int`)
- **args** – names of Python function arguments; can also be `"all"` (all C function arguments in that order), or `"nonrval"` (same, but with return value arguments excluded) by default, use `"nonrval"`
- **rvals** – names of return value arguments; can include either a C function argument name, or `None` (which means the function return value); can also be `"rest"` (listsall the arguments not included into `args`; if `args=="nonrval"`, assume that there are no rvals), `"pointer"` (assume that all pointer arguments are rvals; this does not include `c_void_p`, `c_char_p`, or `c_wchar_p`); by default, use the value supplied on the wrapper creation (`"rest"` by default)
- **argprep** – dictionary `{name: prep}` of ways to prepare of C function arguments; each `prep` can be a value (which is assumed to be default argument value), or a callable, which is given values of Python function arguments
- **rconv** – dictionary `{name: conv}` of converters of the return values; each `conv` is a function which takes 3 arguments: unconverted ctypes value, dictionary of all C function arguments, and dictionary of all Python function arguments if `conv` takes less than 3 argument, then the arguments list is trimmed (e.g., if it takes only one argument, it will be an unconverted value) `conv` can also be `"ctypes"` (return raw ctypes value), or `"raw"` (return raw value for buffers).
- **byref** – list of all argument names which should by passed by reference; by default, it includes all arguments listed in `rvals`
- **errcheck** – error-checking function which is automatically called for the return value; if `None`, use the value supplied to the wrapper constructor (none by default)

wrap_annotated (*func*, *args='nonrval'*, *rvals='default'*, *alias=None*, *argprep=None*, *rconv=None*, *byref='all'*, *errcheck=None*)

Wrap annotated C function in a Python call.

Assumes that the functions has defined `.argtypes` (list of argument types) and `.argnames` (list of argument names) attributes.

Parameters

- **func** – C function
- **args** – names of Python function arguments; can also be `"all"` (all C function arguments in that order), or `"nonrval"` (same, but with return value arguments excluded); by default, use `"nonrval"`

- **rvals** – names of return value arguments; can include either a C function argument name, or `None` (which means the function return value); can also be `"rest"` (listsall the arguments not included into `args`; if `args=="nonrval"`, assume that there are no rvals), `"pointer"` (assume that all pointer arguments are rvals; this does not include `c_void_p`, `c_char_p`, or `c_wchar_p`); by default, use the value supplied on the wrapper creation (`"rest"` by default)
- **alias** – either a list of argument names which replace `.argnames`, or a dictionary `{argname: alias}` which transforms names; all names in all other parameters (`rvals`, `argprep`, `rconv`, and `byref`) take aliased names
- **argprep** – dictionary `{name: prep}` of ways to prepare of C function arguments; each `prep` can be a value (which is assumed to be default argument value), or a callable, which is given values of Python function arguments
- **rconv** – dictionary `{name: conv}` of converters of the return values; each `conv` is a function which takes 3 arguments: unconverted ctypes value, dictionary of all C function arguments, and dictionary of all Python function arguments if `conv` takes less than 3 argument, then the arguments list is trimmed (e.g., if it takes only one argument, it will be an unconverted value)
- **byref** – list of all argument names which should by passed by reference; by default, it includes all arguments listed in `rvals`
- **errcheck** – error-checking function which is automatically called for the return value; if `None`, use the value supplied to the wrapper constructor (none by default)

`pylablib.core.utils.ctypes_wrap.`**`strprep`**`(`*l*, *ctype=None*, *unicode=False*`)`

Make a string preparation function.

Return a function which creates a string with a fixed length of *l* bytes and returns a pointer to it. *ctype* can specify the type of the result (by default, `ctypes.c_char_p`).

`pylablib.core.utils.ctypes_wrap.`**`buffprep`**`(`*size_arg_pos*, *dtype*`)`

Make a buffer preparation function.

Return a function which creates a string with a variable size (specified by an argument at a position *size_arg_pos*). The buffer size is given in elements. *dtype* specifies the datatype of the buffer, whose size is used to determine buffer size in bytes.

`pylablib.core.utils.ctypes_wrap.`**`buffconv`**`(`*size_arg_pos*, *dtype*`)`

Make a buffer conversion function.

Return a function which converts a pointer of a variable size (specified by an argument at a position *size_arg_pos*) into a numpy array. The buffer size is given in elements. *dtype* specifies the datatype of the resulting array.

`class` `pylablib.core.utils.ctypes_wrap.`**`CStructWrapper`**`(`*struct=None*`)`

Bases: `object`

Wrapper around a ctypes structure, which allows for easier creation of parsing of these structures.

When created, all structure fields can be accessed/modified as attributes of the wrapper object. It can also be converted into tuple using *`tup()`* method, or back into C structure using *`to_struct()`* method.

Class variable `_struct` should be set to the ctypes structure which is being wrapped. Several other class variables determine the behavior when generating and parsing:

- `_prep`: dictionary `{name: prep}` of methods to prepare individual structure parameters; can be either a value or a function (which takes as ordered arguments all structure fields as ctypes values)

- `_conv`: dictionary `{name: conv}` of methods to convert individual structure parameters when parsing a C structure; can be either a function (which takes ctypes value of the field as a single argument) or a value; also can be used as a source of default values on wrapper creation
- `_tup`: dictionary `{name: conv}` of functions to convert structure values when generating a tuple
- `_tup_exc`: list of values to exclude from the resulting tuple
- `_tup_inc`: list of values to include in the resulting tuple (if `None`, include all)
- `_tup_add`: list of values to add to the resulting tuple (these values must then exist either as attributes, or as entries in `_tup` dictionary)
- `_tup_order`: order of fields in the returned tuple (by default, same as structure order)

Also specifies two overloaded methods for a more flexible preparation/conversion of structures. `conv()` takes no arguments and is called in the end of wrapper creation to finish setting up attributes. `prep()` takes a single argument (C structure) and is called when converting into a C structure to finish setting up the fields (e.g., size field).

Parameters **struct** – C structure to wrap (if `None`, create a new 'blank' structure).

to_struct()
Convert wrapper into a C structure

prep(*struct*)
Prepare C structure after creation (by default, do nothing)

conv()
Prepare wrapper after setting up the fields from the wrapped structure

tup()
Convert wrapper into a named tuple

classmethod prep_struct(**args*, ***kwargs*)
Prepare a blank C structure

classmethod prep_struct_args(***kwargs*)
Prepare a C structure with the given supplied fields

classmethod tup_struct(*struct*, **args*, ***kwargs*)
Convert C structure into a named tuple

`pylablib.core.utils.ctypes_wrap.`**class_tuple_to_dict**(*val*, *norm_strings=True*, *expand_lists=False*)
Convert a named tuple (usually, a tuple returned by `CStructWrapper.tup()`) into a dictionary.

Iterate recursively over all named tuple elements as well. If `norm_strings==True`, automatically translate byte strings into regular ones. If `expand_lists==True`, iterate recursively over lists members.

pylablib.core.utils.dictionary module

Tree-like multi-level dictionary with advanced indexing options.

`pylablib.core.utils.dictionary.`**split_path**(*path*, *omit_empty=True*, *sep=None*)
Split generic path into individual path entries.

Parameters

- **path** – Generic path. Lists and tuples (possible nested) are flattened; strings are split according to separators; non-strings are converted into strings first.
- **omit_empty** (`bool`) – Determines if empty entries are skipped.

- **sep** (*str*) – If not None, defines regex for path separators; default separator is '/'.

Returns A list of individual entries.

Return type list

pylablib.core.utils.dictionary.**normalize_path_entry**(*entry*, *case_normalization=None*)

Normalize the case of the entry if it's not case-sensitive. Normalization is either None (no normalization, names are case-sensitive), 'lower' or 'upper'

pylablib.core.utils.dictionary.**normalize_path**(*path*, *omit_empty=True*, *case_normalization=None*, *sep=None*, *force=False*)

Split and normalize generic path into individual path entries.

Parameters

- **path** – Generic path. Lists and tuples (possible nested) are flattened; strings are split according to separators; non-strings are converted into strings first.
- **omit_empty** (*bool*) – Determines if empty entries are skipped.
- **case_normalization** (*str*) – Case normalization rules; can be None (no normalization, names are case-sensitive), 'lower' or 'upper'.
- **sep** (*str*) – If not None, defines regex for path separators; default separator is '/'.
- **force** (*bool*) – If False, treat lists as if they're already normalized.

Returns A list of individual normalized entries.

Return type list

pylablib.core.utils.dictionary.**is_dictionary**(*obj*, *generic=False*)

Determine if the object is a dictionary.

Parameters

- **obj** – object
- **generic** (*bool*) – if False, passes only *Dictionary* (or subclasses) objects; otherwise, passes any dictionary-like object.

Returns bool

pylablib.core.utils.dictionary.**as_dictionary**(*obj*, *case_normalization=None*)

Convert object into *Dictionary* with the given parameters.

If object is already a *Dictionary* (or its subclass), return unchanged, even if its parameters are different.

pylablib.core.utils.dictionary.**as_dict**(*obj*, *style='nested'*, *copy=True*)

Convert object into standard *dict* with the given parameters.

If object is already a *dict*, return unchanged, even if the parameters are different.

class pylablib.core.utils.dictionary.**Dictionary**(*root=None*, *case_normalization=None*, *copy=True*)

Bases: object

Multi-level dictionary.

Access is done by path (all path elements are converted into strings and concatenated to form a single string path). If dictionary is not case-sensitive, all inserted and accessed paths are normalized to lower or upper case.

Parameters

- **root** (*dict or Dictionary*) – Initial value.
- **case_normalization** (*str*) – Case normalization rules; can be `None` (no normalization, names are case-sensitive), `'lower'` or `'upper'`.
- **copy** (*bool*) – If `True`, make copy of the supplied data; otherwise, just make it the root.

> **Warning:** If `copy==False`, the root data is already assumed to be normalized. If it isn't, the behavior might be incorrect.

static is_dictionary(*obj*, *generic=True*)
Determine if the object is a dictionary.

Parameters

- **obj** –
- **generic** (*bool*) – if False, passes only *Dictionary* (or subclasses) objects; otherwise, passes any dictionary-like object.

Returns bool

static as_dictionary(*obj*, *case_normalization=None*)
Convert object into *Dictionary* with the given parameters.

If object is already a *Dictionary* (or its subclass), return unchanged, even if its parameters are different.

add_entry(*path*, *value*, *force=False*, *branch_option='normalize'*)
Add value to a given path (overwrite leaf value if necessary).

Doesn't replace leaves with branches and vice-verse if `force==False`.

Parameters

- **path** –
- **value** –
- **force** (*bool*) – If `True`, change leaf into a branch and vice-versa; otherwise, raises `ValueError` if the conversion is necessary.
- **branch_option** (*str*) –

  Decides what to do if the value is dictionary-like:

  – `'attach'` – just attach the root,

  – `'copy'` – copy and attach,

  – `'normalize'` – copy while normalizing all the keys according to the current rules.

get_entry(*path*, *as_pointer=False*)
Get entry at a given path

Parameters

- **path** –
- **as_pointer** (*bool*) – If `True` and entry is not a leaf, return *DictionaryPointer*; otherwise, return *Dictionary*

has_entry(*path*, *kind='all'*)
Determine if the path is in the dictionary.

kind determines which kind of path to consider and can be `'leaf'`, `'branch'` or `'all'`.

is_leaf_path(*path*)
Determine if the path is in the dictionary and points to a leaf

is_branch_path(*path*)
Determine if the path is in the dictionary and points to a branch

get_max_prefix(*path*, *kind='all'*)
Find the longest prefix of *path* contained in the dictionary.

Return tuple `(prefix, rest)`, where both path entries are normalized according to the dictionary rules (i.e., these are lists representing normalized paths). *kind* determines which kind of path to consider and can be `'leaf'`, `'branch'` or `'all'`. If the longest prefix is of a different kind, return `(None, None)`.

del_entry(*path*)
Delete entry from the dictionary.

Return `True` if the path was present. Note that it never raises *KeyError*.

size()
Return the total size of the dictionary (number of nodes)

get(*path*, *default=None*)
Analog of `dict.get()`: `D.get(k,d) -> D[k] if k in D else d`

pop(*path*, *default=None*)
Analog of `dict.pop()`: remove value at *path* and return it if `path in D`, otherwise return *default*

Note that it never raises `KeyError`.

setdefault(*path*, *default=None*)
Analog of `dict.setdefault()`: `D.setdefault(k,d) -> D.get(k,d)`, also sets `D[k]=d` if `k not in D`.

items(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)
- **wrap_branches** (*bool*) – if `True`, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

iteritems(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)

- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

viewitems(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of dict.items(), by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

values(*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of dict.values(), iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

viewvalues(*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of dict.values(), iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

itervalues(*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of dict.values(), iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

keys (*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

viewkeys (*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

iterkeys (*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

paths (*ordered=False*, *topdown=False*, *path_kind='split'*)
Return list of all paths (leafs and nodes).

Parameters

- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **topdown** (*bool*) – If `True`, return node's leafs before its subtrees leafs.
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

iternodes (*to_visit='leafs'*, *ordered=False*, *include_path=False*, *topdown=False*)
Iterate over nodes.

Parameters

- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary are visited.
- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **include_path** (*bool*) – Include in the return value.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.

Yields Values for leafs and *DictionaryPointer* for branches. If `include_path==True`, yields tuple `(path, value)`, where *path* is in the form of a normalized list.

nodes (*to_visit='leafs'*, *ordered=False*, *include_path=False*, *topdown=False*)
Iterate over nodes.

Parameters

- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary are visited.
- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **include_path** (*bool*) – Include in the return value.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.

Yields Values for leafs and *DictionaryPointer* for branches. If `include_path==True`, yields tuple `(path, value)`, where *path* is in the form of a normalized list.

merge (*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)
Attach source (`dict` or other *Dictionary*) to a given branch; source is automatically deep-copied.

If *source* is not a dictionary, simply assign it (i.e., `D.merge(v,p)` is equivalent to `D.add_entry(p, v,force=True)` in this case). Compared to *add_entry()*, merges two branches instead of removing the old branch completely.

Parameters

- **source** (*dict or Dictionary*) –
- **branch** (*tuple or str*) – Destination path.
- **overwrite** (*bool*) – If `True`, replaces the old entries with the new ones (it only matters for leaf assignments).
- **normalize_paths** (*bool*) – If `True` and the dictionary isn't case sensitive, perform normalization if the *source*.

update (*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)
Attach source (`dict` or other *Dictionary*) to a given branch; source is automatically deep-copied.

If *source* is not a dictionary, simply assign it (i.e., `D.merge(v,p)` is equivalent to `D.add_entry(p, v,force=True)` in this case). Compared to *add_entry()*, merges two branches instead of removing the old branch completely.

Parameters

- **source** (*dict or Dictionary*) –
- **branch** (*tuple or str*) – Destination path.
- **overwrite** (*bool*) – If `True`, replaces the old entries with the new ones (it only matters for leaf assignments).
- **normalize_paths** (*bool*) – If `True` and the dictionary isn't case sensitive, perform normalization if the *source*.

detach (*path*)
Remove a branch or a leaf from the current dictionary.

Branch is returned as a separate *Dictionary*. If *path* is missing, raise a `KeyError`.

branch_copy(*branch=''*)
Get a copy of the branch as a `Dictionary`

copy()
Get a full copy the dictionary

updated(*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)
Get a copy of the dictionary and attach a new branch to it.

Parameters are the same as in the `Dictionary.merge()`.

as_dict(*style='nested'*, *copy=True*)
Convert into a `dict` object.

Parameters

- **style** (*str*) –

  Determines style of the result:

  - `'nested'` – subtrees are turned into nested dictionaries,
  - `'flat'` – single dictionary is formed with full paths as keys.

- **copy** (*bool*) – If `False` and `style=='nested'`, return the root dictionary.

asdict(*style='nested'*, *copy=True*)
Convert into a `dict` object.

Parameters

- **style** (*str*) –

  Determines style of the result:

  - `'nested'` – subtrees are turned into nested dictionaries,
  - `'flat'` – single dictionary is formed with full paths as keys.

- **copy** (*bool*) – If `False` and `style=='nested'`, return the root dictionary.

as_json(*style='nested'*)
Convert into a JSON string.

Parameters **style** (*str*) – Determines style of the result: - `'nested'` – subtrees are turned into nested dictionaries, - `'flat'` – single dictionary is formed with full paths as keys.

classmethod from_json(*data*, *case_normalization=None*)
Convert JSON representations of a dictionary into a `Dictionary` object

as_pandas(*index_key=True*, *as_series=True*)
Convert into a pandas DataFrame or Series object.

Parameters

- **index_key** (*bool*) – If `False`, create a 2-column table with the first column (`"key"`) containing string path and the second column (`"value"`) containing value; otherwise, move key to the table index.
- **as_series** (*bool*) – If `index_key==True` and `as_series==True`, convert the resulting DataFrame into 1D Series (the key is the index); otherwise, keep it as a single-column table

get_path()

branch_pointer(*branch=''*)
Get a `DictionaryPointer` of a given branch

map_self(*func*, *to_visit='leafs'*, *pass_path=False*, *topdown=False*, *branch_option='normalize'*)
Apply *func* to the nodes in the dictionary.

Note that any pointers to the replaced branches or their sub-branches will become invalid.

Parameters

- **func** (*callable*) – Mapping function. Leafs are passed by value, branches (if visited) are passed as `DictionaryPointer`.
- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary passed to the map function.
- **pass_path** (*bool*) – If `True`, pass the node path (in the form of a normalized list) as a first argument to *func*.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.
- **branch_option** (*str*) – If the function returns a dict-like object, determines how to incorporate into the dictionary; can be `"normalize"` (make a copy with normalized paths and insert that), `"copy"` (make a copy without normalization), or `"attach"` (simply replace the value without copying and normalization)

filter_self(*pred*, *to_visit='leafs'*, *pass_path=False*, *topdown=False*)
Remove all the nodes from the dictionary for which *pred* returns `False`.

Parameters

- **pred** (*callable*) – Filter function. Leafs are passed to *pred* by value, branches (if visited) are passed as `DictionaryPointer`.
- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary passed to the predicate.
- **pass_path** (*bool*) – If `True`, pass the node path (in the form of a normalized list) as a first argument to *pred*.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.

diff(*other*)
Perform an element-wise comparison to another Dictionary.

If the other Dictionary has a different case sensitivity, raise `ValueError`.

Returns `DictionaryDiff`

static diff_flatdict(*first*, *second*)
Find the difference between flat `dict` objects.

Returns `DictionaryDiff`

static find_intersection(*dicts*, *use_flatten=False*)
Find intersection of multiple dictionaries.

Parameters

- **dicts** ([`Dictionary`]) –
- **use_flatten** (*bool*) – If `True` flatten all dictionaries before comparison (works faster for a large number of dictionaries).

Returns `DictionaryIntersection`

get_matching_paths (*pattern*, *wildkey='*'*, *wildpath='**'*, *only_leaves=True*)
Get all paths in the tree that match the provided pattern.

Parameters

- **wildkey** (*str*) – Pattern symbol that matches any key.
- **wildpath** (*str*) – Pattern symbol that matches any subpath (possibly empty).
- **only_leaves** (*bool*) – If `True`, only check leaf paths; otherwise, check subtree paths (i.e., incomplete leaf paths) as well. Basically, `only_leaves=False` is analogous to adding wildpath at the end of the pattern.

get_matching_subtree (*pattern*, *wildkey='*'*, *wildpath='**'*, *only_leaves=True*)
Get a subtree containing nodes with paths matching the provided pattern.

Parameters

- **wildkey** (*str*) – Pattern symbol that matches any key.
- **wildpath** (*str*) – Pattern symbol that matches any subpath (possibly empty).
- **only_leaves** (*bool*) – If `True`, only check leaf paths; otherwise, check subtree paths (i.e., incomplete leaf paths) as well. Basically, `only_leaves=False` is analogous to adding wildpath at the end of the pattern.

class pylablib.core.utils.dictionary.**DictionaryDiff**
Bases: *pylablib.core.utils.dictionary.DictionaryDiff*

Describes a difference between the two dictionaries.

same
Contains the leafs which is the same.

Type *Dictionary*

changed_from
Contains the leafs from the first dictionary which have different values in the second dictionary.

Type *Dictionary*

changed_to
Contains the leafs from the second dictionary which have different values in the first dictionary.

Type *Dictionary*

removed
Contains the leafs from the first dictionary which are absent in the second dictionary.

Type *Dictionary*

added
Contains the leafs from the second dictionary which are absent in the first dictionary.

Type *Dictionary*

added

changed_from

changed_to

count ()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

removed

same

class pylablib.core.utils.dictionary.**DictionaryIntersection**
Bases: *pylablib.core.utils.dictionary.DictionaryIntersection*

Describes the result of finding intersection of multiple dictionaries.

common
Contains the intersection of all dictionaries.

Type *Dictionary*

individual
Contains list of difference from intersection for all dictionaries.

Type [*Dictionary*]

common

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

individual

class pylablib.core.utils.dictionary.**DictionaryPointer**(*root=None*, *pointer=None*, *case_normalization=None*, *copy=True*)
Bases: *pylablib.core.utils.dictionary.Dictionary*

Similar to *Dictionary*, but can point at one of the branches instead of the full dictionary.

Effect is mostly equivalent to prepending some path to all queries.

Parameters

- **root** (*dict or* Dictionary) – Complete tree.
- **pointer** – Path to the pointer location.
- **case_normalization** (*str*) – Case normalization rules; can be None (no normalization, names are case-sensitive), 'lower' or 'upper'.
- **copy** (*bool*) – If True, make copy of the supplied data; otherwise, just make it the root.

> **Warning:** If copy==False, the root data is already assumed to be normalized. If it isn't, the behavior might be incorrect.

get_path()
Return pointer path in the whole dictionary.

move_to (*path=''*, *absolute=True*)
Move the pointer to a new path.

Parameters

- **path** –
- **absolute** (*bool*) – If `True`, path is specified with respect to the root; otherwise, it's specified with respect to the current position (and can only go deeper).

move_up (*levels*, *strict=True*)
Move the pointer by the given number of levels up.

If `strict==True` and there are not enough levels above, raise an error. Otherwise, stop at the top dictionary level.

branch_pointer (*branch=''*)
Get a `DictionaryPointer` of a given branch.

add_entry (*path*, *value*, *force=False*, *branch_option='normalize'*)
Add value to a given path (overwrite leaf value if necessary).

Doesn't replace leaves with branches and vice-verse if `force==False`.

Parameters

- **path** –
- **value** –
- **force** (*bool*) – If `True`, change leaf into a branch and vice-versa; otherwise, raises `ValueError` if the conversion is necessary.
- **branch_option** (*str*) –

  Decides what to do if the value is dictionary-like:

  - `'attach'` – just attach the root,
  - `'copy'` – copy and attach,
  - `'normalize'` – copy while normalizing all the keys according to the current rules.

as_dict (*style='nested'*, *copy=True*)
Convert into a `dict` object.

Parameters

- **style** (*str*) –

  Determines style of the result:

  - `'nested'` – subtrees are turned into nested dictionaries,
  - `'flat'` – single dictionary is formed with full paths as keys.
- **copy** (*bool*) – If `False` and `style=='nested'`, return the root dictionary.

static as_dictionary (*obj*, *case_normalization=None*)
Convert object into `Dictionary` with the given parameters.

If object is already a `Dictionary` (or its subclass), return unchanged, even if its parameters are different.

as_json (*style='nested'*)
Convert into a JSON string.

Parameters **style** (*str*) – Determines style of the result: - 'nested' – subtrees are turned into nested dictionaries, - 'flat' – single dictionary is formed with full paths as keys.

as_pandas (*index_key=True*, *as_series=True*)
Convert into a pandas DataFrame or Series object.

Parameters

- **index_key** (*bool*) – If False, create a 2-column table with the first column ("key") containing string path and the second column ("value") containing value; otherwise, move key to the table index.
- **as_series** (*bool*) – If index_key==True and as_series==True, convert the resulting DataFrame into 1D Series (the key is the index); otherwise, keep it as a single-column table

asdict (*style='nested'*, *copy=True*)
Convert into a dict object.

Parameters

- **style** (*str*) –

  Determines style of the result:

  – 'nested' – subtrees are turned into nested dictionaries,

  – 'flat' – single dictionary is formed with full paths as keys.
- **copy** (*bool*) – If False and style=='nested', return the root dictionary.

branch_copy (*branch=''*)
Get a copy of the branch as a *Dictionary*

copy ()
Get a full copy the dictionary

del_entry (*path*)
Delete entry from the dictionary.

Return True if the path was present. Note that it never raises *KeyError*.

detach (*path*)
Remove a branch or a leaf from the current dictionary.

Branch is returned as a separate *Dictionary*. If *path* is missing, raise a KeyError.

diff (*other*)
Perform an element-wise comparison to another Dictionary.

If the other Dictionary has a different case sensitivity, raise ValueError.

Returns *DictionaryDiff*

static diff_flatdict (*first*, *second*)
Find the difference between flat dict objects.

Returns *DictionaryDiff*

filter_self (*pred*, *to_visit='leafs'*, *pass_path=False*, *topdown=False*)
Remove all the nodes from the dictionary for which *pred* returns False.

Parameters

- **pred** (*callable*) – Filter function. Leafs are passed to *pred* by value, branches (if visited) are passed as *DictionaryPointer*.
- **to_visit** (*str*) – Can be 'leafs', 'branches' or 'all' and determines which parts of the dictionary passed to the predicate.
- **pass_path** (*bool*) – If True, pass the node path (in the form of a normalized list) as a first argument to *pred*.
- **topdown** (*bool*) – If True, visit node and its leafs before its subtrees leafs.

static find_intersection(*dicts*, *use_flatten=False*)
Find intersection of multiple dictionaries.

Parameters

- **dicts** ([*Dictionary*]) –
- **use_flatten** (*bool*) – If True flatten all dictionaries before comparison (works faster for a large number of dictionaries).

Returns *DictionaryIntersection*

classmethod from_json(*data*, *case_normalization=None*)
Convert JSON representations of a dictionary into a *Dictionary* object

get(*path*, *default=None*)
Analog of `dict.get()`: `D.get(k,d) -> D[k] if k in D else d`

get_entry(*path*, *as_pointer=False*)
Get entry at a given path

Parameters

- **path** –
- **as_pointer** (*bool*) – If True and entry is not a leaf, return *DictionaryPointer*; otherwise, return *Dictionary*

get_matching_paths(*pattern*, *wildkey='*'*, *wildpath='**'*, *only_leaves=True*)
Get all paths in the tree that match the provided pattern.

Parameters

- **wildkey** (*str*) – Pattern symbol that matches any key.
- **wildpath** (*str*) – Pattern symbol that matches any subpath (possibly empty).
- **only_leaves** (*bool*) – If True, only check leaf paths; otherwise, check subtree paths (i.e., incomplete leaf paths) as well. Basically, `only_leaves=False` is analogous to adding wildpath at the end of the pattern.

get_matching_subtree(*pattern*, *wildkey='*'*, *wildpath='**'*, *only_leaves=True*)
Get a subtree containing nodes with paths matching the provided pattern.

Parameters

- **wildkey** (*str*) – Pattern symbol that matches any key.
- **wildpath** (*str*) – Pattern symbol that matches any subpath (possibly empty).
- **only_leaves** (*bool*) – If True, only check leaf paths; otherwise, check subtree paths (i.e., incomplete leaf paths) as well. Basically, `only_leaves=False` is analogous to adding wildpath at the end of the pattern.

get_max_prefix(*path*, *kind='all'*)
Find the longest prefix of *path* contained in the dictionary.

Return tuple `(prefix, rest)`, where both path entries are normalized according to the dictionary rules (i.e., these are lists representing normalized paths). *kind* determines which kind of path to consider and can be `'leaf'`, `'branch'` or `'all'`. If the longest prefix is of a different kind, return `(None, None)`.

has_entry(*path*, *kind='all'*)
Determine if the path is in the dictionary.

kind determines which kind of path to consider and can be `'leaf'`, `'branch'` or `'all'`.

is_branch_path(*path*)
Determine if the path is in the dictionary and points to a branch

static is_dictionary(*obj*, *generic=True*)
Determine if the object is a dictionary.

Parameters

- **obj** –
- **generic** (*bool*) – if False, passes only *Dictionary* (or subclasses) objects; otherwise, passes any dictionary-like object.

Returns bool

is_leaf_path(*path*)
Determine if the path is in the dictionary and points to a leaf

items(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)
- **wrap_branches** (*bool*) – if `True`, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

iteritems(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)
- **wrap_branches** (*bool*) – if `True`, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

iterkeys(*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

iternodes(*to_visit='leafs'*, *ordered=False*, *include_path=False*, *topdown=False*)
Iterate over nodes.

Parameters

- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary are visited.
- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **include_path** (*bool*) – Include in the return value.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.

Yields Values for leafs and `DictionaryPointer` for branches. If `include_path==True`, yields tuple `(path, value)`, where *path* is in the form of a normalized list.

itervalues(*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of `dict.values()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if `True`, wrap sub-branches into `DictionaryPointer` objects; otherwise, return them as nested built-in dictionaries

keys(*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

map_self(*func*, *to_visit='leafs'*, *pass_path=False*, *topdown=False*, *branch_option='normalize'*)
Apply *func* to the nodes in the dictionary.

Note that any pointers to the replaced branches or their sub-branches will become invalid.

Parameters

- **func** (*callable*) – Mapping function. Leafs are passed by value, branches (if visited) are passed as *DictionaryPointer*.
- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary passed to the map function.
- **pass_path** (*bool*) – If `True`, pass the node path (in the form of a normalized list) as a first argument to *func*.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.
- **branch_option** (*str*) – If the function returns a dict-like object, determines how to incorporate into the dictionary; can be `"normalize"` (make a copy with normalized paths and insert that), `"copy"` (make a copy without normalization), or `"attach"` (simply replace the value without copying and normalization)

merge(*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)

Attach source (`dict` or other *Dictionary*) to a given branch; source is automatically deep-copied.

If *source* is not a dictionary, simply assign it (i.e., `D.merge(v,p)` is equivalent to `D.add_entry(p, v,force=True)` in this case). Compared to *add_entry()*, merges two branches instead of removing the old branch completely.

Parameters

- **source** (*dict or Dictionary*) –
- **branch** (*tuple or str*) – Destination path.
- **overwrite** (*bool*) – If `True`, replaces the old entries with the new ones (it only matters for leaf assignments).
- **normalize_paths** (*bool*) – If `True` and the dictionary isn't case sensitive, perform normalization if the *source*.

nodes(*to_visit='leafs'*, *ordered=False*, *include_path=False*, *topdown=False*)

Iterate over nodes.

Parameters

- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary are visited.
- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **include_path** (*bool*) – Include in the return value.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.

Yields Values for leafs and *DictionaryPointer* for branches. If `include_path==True`, yields tuple `(path, value)`, where *path* is in the form of a normalized list.

paths(*ordered=False*, *topdown=False*, *path_kind='split'*)

Return list of all paths (leafs and nodes).

Parameters

- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **topdown** (*bool*) – If `True`, return node's leafs before its subtrees leafs.
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

pop(*path*, *default=None*)
Analog of `dict.pop()`: remove value at *path* and return it if `path in D`, otherwise return *default*

Note that it never raises `KeyError`.

setdefault(*path*, *default=None*)
Analog of `dict.setdefault()`: `D.setdefault(k,d) -> D.get(k,d)`, also sets `D[k]=d` if `k not in D`.

size()
Return the total size of the dictionary (number of nodes)

update(*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)
Attach source (`dict` or other *`Dictionary`*) to a given branch; source is automatically deep-copied.

If *source* is not a dictionary, simply assign it (i.e., `D.merge(v,p)` is equivalent to `D.add_entry(p, v,force=True)` in this case). Compared to *`add_entry()`*, merges two branches instead of removing the old branch completely.

Parameters

- **source** (*dict or* `Dictionary`) –
- **branch** (*tuple or str*) – Destination path.
- **overwrite** (*bool*) – If `True`, replaces the old entries with the new ones (it only matters for leaf assignments).
- **normalize_paths** (*bool*) – If `True` and the dictionary isn't case sensitive, perform normalization if the *source*.

updated(*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)
Get a copy of the dictionary and attach a new branch to it.

Parameters are the same as in the *`Dictionary.merge()`*.

values(*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of `dict.values()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if `True`, wrap sub-branches into *`DictionaryPointer`* objects; otherwise, return them as nested built-in dictionaries

viewitems(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

viewkeys (*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)

viewvalues (*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of `dict.values()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

pylablib.core.utils.dictionary.**combine_dictionaries** (*dicts*, *func*, *select='all'*, *pass_missing=False*)
Combine several dictionaries element-wise (only for leafs) using a given function.

Parameters

- **dicts** (*list or tuple*) – list of dictionaries (*Dictionary* or dict) to be combined
- **func** (*callable*) – combination function. Takes a single argument, which is a list of elements to be combined.
- **select** (*str*) – determines which keys are selected for the resulting dictionary. Can be either "all" (only keep keys which are present in all the dictionaries), or "any" (keep keys which are present in at least one dictionary). Only keys that point to leafs count; if a key points to a non-leaf branch in some dictionary, it is considered absent from this dictionary.
- **pass_missing** (*bool*) – if select=="any", this parameter determines whether missing elements will be passed to *func* as None, or omitted entirely.

class pylablib.core.utils.dictionary.**PrefixTree** (*root=None*, *case_normalization=None*, *wildcard='*'*, *matchcard='.'*, *copy=True*)
Bases: *pylablib.core.utils.dictionary.Dictionary*

Expansion of a *Dictionary* designed to store data related to prefixes.

Each branch node can have a leaf with a name given by wildcard ('*' by default) or matchcard ('.' by default). Wildcard assumes that the branch node path is a prefix; matchcard assumes exact match. These leafs are inspected when specific prefix tree functions (*find_largest_prefix()* and *find_all_prefixes()*) are used.

Parameters

- **root** (*dict or Dictionary*) – Complete tree.
- **case_normalization** (*str*) – Case normalization rules; can be `None` (no normalization, names are case-sensitive), `'lower'` or `'upper'`.
- **wildcard** (*str*) – Symbol for a wildcard entry.
- **matchcard** (*str*) – Symbol for a matchcard entry.
- **copy** (*bool*) – If `True`, make copy of the supplied data; otherwise, just make it the root.

> **Warning:** If `copy==False`, the root data is already assumed to be normalized. If it isn't, the behavior might be incorrect.

copy()
Get a full copy the prefix tree

find_largest_prefix(*path*, *default=None*, *allow_nomatch_exact=True*, *return_path=False*, *return_subpath=False*)
Find the entry which is the largest prefix of a given path.

Parameters

- **path** –
- **default** – Default value if the path isn't found.
- **allow_nomatch_exact** (*bool*) – If `True`, just element with the given path can be returned; otherwise, only elements stored under wildcards and matchcards are considered.
- **return_path** (*bool*) – If `True`, return path to the element (i.e., the largest prefix) instead of the element itself.
- **return_subpath** (*bool*) – If `True`, return tuple with a second element being part of the *path* left after subtraction of the prefix.

find_all_prefixes(*path*, *allow_nomatch_exact=True*, *return_path=True*, *return_subpath=False*)
Find list of all the entries which are prefixes of a given path.

Parameters

- **path** –
- **default** – Default value if the path isn't found.
- **allow_nomatch_exact** (*bool*) – If `True`, just element with the given path can be returned; otherwise, only elements stored under wildcards and matchcards are considered.
- **return_path** (*bool*) – If `True`, return path to the element (i.e., the largest prefix) instead of the element itself.
- **return_subpath** (*bool*) – If `True`, return tuple with a second element being part of the *path* left after subtraction of the prefix.

add_entry(*path*, *value*, *force=False*, *branch_option='normalize'*)
Add value to a given path (overwrite leaf value if necessary).

Doesn't replace leaves with branches and vice-verse if `force==False`.

Parameters

- **path** –
- **value** –
- **force** (*bool*) – If `True`, change leaf into a branch and vice-versa; otherwise, raises `ValueError` if the conversion is necessary.
- **branch_option** (*str*) –

  Decides what to do if the value is dictionary-like:

  – `'attach'` – just attach the root,

  – `'copy'` – copy and attach,

  – `'normalize'` – copy while normalizing all the keys according to the current rules.

as_dict (*style='nested'*, *copy=True*)

Convert into a `dict` object.

Parameters

- **style** (*str*) –

  Determines style of the result:

  – `'nested'` – subtrees are turned into nested dictionaries,

  – `'flat'` – single dictionary is formed with full paths as keys.

- **copy** (*bool*) – If `False` and `style=='nested'`, return the root dictionary.

static as_dictionary (*obj*, *case_normalization=None*)

Convert object into *Dictionary* with the given parameters.

If object is already a *Dictionary* (or its subclass), return unchanged, even if its parameters are different.

as_json (*style='nested'*)

Convert into a JSON string.

Parameters **style** (*str*) – Determines style of the result: - `'nested'` – subtrees are turned into nested dictionaries, - `'flat'` – single dictionary is formed with full paths as keys.

as_pandas (*index_key=True*, *as_series=True*)

Convert into a pandas DataFrame or Series object.

Parameters

- **index_key** (*bool*) – If `False`, create a 2-column table with the first column (`"key"`) containing string path and the second column (`"value"`) containing value; otherwise, move key to the table index.
- **as_series** (*bool*) – If `index_key==True` and `as_series==True`, convert the resulting DataFrame into 1D Series (the key is the index); otherwise, keep it as a single-column table

asdict (*style='nested'*, *copy=True*)

Convert into a `dict` object.

Parameters

- **style** (*str*) –

  Determines style of the result:

  – 'nested' – subtrees are turned into nested dictionaries,

  – 'flat' – single dictionary is formed with full paths as keys.

- **copy** (*bool*) – If False and style=='nested', return the root dictionary.

branch_copy (*branch=''*)
Get a copy of the branch as a *Dictionary*

branch_pointer (*branch=''*)
Get a *DictionaryPointer* of a given branch

del_entry (*path*)
Delete entry from the dictionary.

Return True if the path was present. Note that it never raises *KeyError*.

detach (*path*)
Remove a branch or a leaf from the current dictionary.

Branch is returned as a separate *Dictionary*. If *path* is missing, raise a KeyError.

diff (*other*)
Perform an element-wise comparison to another Dictionary.

If the other Dictionary has a different case sensitivity, raise ValueError.

Returns *DictionaryDiff*

static diff_flatdict (*first*, *second*)
Find the difference between flat dict objects.

Returns *DictionaryDiff*

filter_self (*pred*, *to_visit='leafs'*, *pass_path=False*, *topdown=False*)
Remove all the nodes from the dictionary for which *pred* returns False.

Parameters

- **pred** (*callable*) – Filter function. Leafs are passed to *pred* by value, branches (if visited) are passed as *DictionaryPointer*.
- **to_visit** (*str*) – Can be 'leafs', 'branches' or 'all' and determines which parts of the dictionary passed to the predicate.
- **pass_path** (*bool*) – If True, pass the node path (in the form of a normalized list) as a first argument to *pred*.
- **topdown** (*bool*) – If True, visit node and its leafs before its subtrees leafs.

static find_intersection (*dicts*, *use_flatten=False*)
Find intersection of multiple dictionaries.

Parameters

- **dicts** ([*Dictionary*]) –
- **use_flatten** (*bool*) – If True flatten all dictionaries before comparison (works faster for a large number of dictionaries).

Returns *DictionaryIntersection*

classmethod from_json(*data*, *case_normalization=None*)
Convert JSON representations of a dictionary into a `Dictionary` object

get(*path*, *default=None*)
Analog of `dict.get()`: `D.get(k,d) -> D[k] if k in D else d`

get_entry(*path*, *as_pointer=False*)
Get entry at a given path

Parameters

- **path** –
- **as_pointer** (*bool*) – If `True` and entry is not a leaf, return `DictionaryPointer`; otherwise, return `Dictionary`

get_matching_paths(*pattern*, *wildkey='*'*, *wildpath='**'*, *only_leaves=True*)
Get all paths in the tree that match the provided pattern.

Parameters

- **wildkey** (*str*) – Pattern symbol that matches any key.
- **wildpath** (*str*) – Pattern symbol that matches any subpath (possibly empty).
- **only_leaves** (*bool*) – If `True`, only check leaf paths; otherwise, check subtree paths (i.e., incomplete leaf paths) as well. Basically, `only_leaves=False` is analogous to adding wildpath at the end of the pattern.

get_matching_subtree(*pattern*, *wildkey='*'*, *wildpath='**'*, *only_leaves=True*)
Get a subtree containing nodes with paths matching the provided pattern.

Parameters

- **wildkey** (*str*) – Pattern symbol that matches any key.
- **wildpath** (*str*) – Pattern symbol that matches any subpath (possibly empty).
- **only_leaves** (*bool*) – If `True`, only check leaf paths; otherwise, check subtree paths (i.e., incomplete leaf paths) as well. Basically, `only_leaves=False` is analogous to adding wildpath at the end of the pattern.

get_max_prefix(*path*, *kind='all'*)
Find the longest prefix of *path* contained in the dictionary.

Return tuple `(prefix, rest)`, where both path entries are normalized according to the dictionary rules (i.e., these are lists representing normalized paths). *kind* determines which kind of path to consider and can be `'leaf'`, `'branch'` or `'all'`. If the longest prefix is of a different kind, return `(None, None)`.

get_path()

has_entry(*path*, *kind='all'*)
Determine if the path is in the dictionary.

kind determines which kind of path to consider and can be `'leaf'`, `'branch'` or `'all'`.

is_branch_path(*path*)
Determine if the path is in the dictionary and points to a branch

static is_dictionary(*obj*, *generic=True*)
Determine if the object is a dictionary.

Parameters

- **obj** –

- **generic** (*bool*) – if False, passes only *Dictionary* (or subclasses) objects; otherwise, passes any dictionary-like object.

Returns bool

is_leaf_path(*path*)
Determine if the path is in the dictionary and points to a leaf

items(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)
- **wrap_branches** (*bool*) – if `True`, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

iteritems(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)
- **wrap_branches** (*bool*) – if `True`, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

iterkeys(*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If `True`, loop over keys in alphabetic order.
- **leafs** (*bool*) – If `True`, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

iternodes(*to_visit='leafs'*, *ordered=False*, *include_path=False*, *topdown=False*)
Iterate over nodes.

Parameters

- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary are visited.
- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **include_path** (*bool*) – Include in the return value.

- **topdown** (*bool*) – If True, visit node and its leafs before its subtrees leafs.

Yields Values for leafs and *DictionaryPointer* for branches. If include_path==True, yields tuple (path, value), where *path* is in the form of a normalized list.

itervalues(*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of dict.values(), iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

keys(*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of dict.keys(), iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)

map_self(*func*, *to_visit='leafs'*, *pass_path=False*, *topdown=False*, *branch_option='normalize'*)
Apply *func* to the nodes in the dictionary.

Note that any pointers to the replaced branches or their sub-branches will become invalid.

Parameters

- **func** (*callable*) – Mapping function. Leafs are passed by value, branches (if visited) are passed as *DictionaryPointer*.
- **to_visit** (*str*) – Can be 'leafs', 'branches' or 'all' and determines which parts of the dictionary passed to the map function.
- **pass_path** (*bool*) – If True, pass the node path (in the form of a normalized list) as a first argument to *func*.
- **topdown** (*bool*) – If True, visit node and its leafs before its subtrees leafs.
- **branch_option** (*str*) – If the function returns a dict-like object, determines how to incorporate into the dictionary; can be "normalize" (make a copy with normalized paths and insert that), "copy" (make a copy without normalization), or "attach" (simply replace the value without copying and normalization)

merge(*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)
Attach source (dict or other *Dictionary*) to a given branch; source is automatically deep-copied.

If *source* is not a dictionary, simply assign it (i.e., D.merge(v,p) is equivalent to D.add_entry(p, v,force=True) in this case). Compared to *add_entry()*, merges two branches instead of removing the old branch completely.

Parameters

- **source** (*dict or Dictionary*) –
- **branch** (*tuple or str*) – Destination path.
- **overwrite** (*bool*) – If `True`, replaces the old entries with the new ones (it only matters for leaf assignments).
- **normalize_paths** (*bool*) – If `True` and the dictionary isn't case sensitive, perform normalization if the *source*.

nodes (*to_visit='leafs'*, *ordered=False*, *include_path=False*, *topdown=False*)
Iterate over nodes.

Parameters

- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary are visited.
- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **include_path** (*bool*) – Include in the return value.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.

Yields Values for leafs and *DictionaryPointer* for branches. If `include_path==True`, yields tuple `(path, value)`, where *path* is in the form of a normalized list.

paths (*ordered=False*, *topdown=False*, *path_kind='split'*)
Return list of all paths (leafs and nodes).

Parameters

- **ordered** (*bool*) – If `True`, loop over paths in alphabetic order.
- **topdown** (*bool*) – If `True`, return node's leafs before its subtrees leafs.
- **path_kind** (*str*) – either `"split"` (each path is a tuple of individual keys), or `"joined"` (each path is a single string)

pop (*path*, *default=None*)
Analog of `dict.pop()`: remove value at *path* and return it if `path in D`, otherwise return *default*

Note that it never raises `KeyError`.

setdefault (*path*, *default=None*)
Analog of `dict.setdefault()`: `D.setdefault(k,d) -> D.get(k,d)`, also sets `D[k]=d` if `k not in D`.

size ()
Return the total size of the dictionary (number of nodes)

update (*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)
Attach source (`dict` or other *Dictionary*) to a given branch; source is automatically deep-copied.

If *source* is not a dictionary, simply assign it (i.e., `D.merge(v,p)` is equivalent to `D.add_entry(p, v,force=True)` in this case). Compared to `add_entry()`, merges two branches instead of removing the old branch completely.

Parameters

- **source** (*dict or Dictionary*) –
- **branch** (*tuple or str*) – Destination path.

- **overwrite** (*bool*) – If True, replaces the old entries with the new ones (it only matters for leaf assignments).
- **normalize_paths** (*bool*) – If True and the dictionary isn't case sensitive, perform normalization if the *source*.

updated (*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)
Get a copy of the dictionary and attach a new branch to it.

Parameters are the same as in the *Dictionary.merge()*.

values (*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of `dict.values()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

viewitems (*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

viewkeys (*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)

viewvalues (*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of `dict.values()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)

- **wrap_branches** (*bool*) – if `True`, wrap sub-branches into `DictionaryPointer` objects; otherwise, return them as nested built-in dictionaries

class pylablib.core.utils.dictionary.**FilterTree**(*root=None*, *case_normalization=None*, *default=False*, *match_prefix=False*, *copy=True*)

Bases: `pylablib.core.utils.dictionary.Dictionary`

Expansion of a `Dictionary` designed to store hierarchical path filtering rules.

Store path templates and the corresponding values (usually `True` or `False` for a filter tree, but other values are possible). The `match()` method is then tested against this templates, and the value of the closest matching template (or default value, if none match) is returned. The templates can contain direct matches (e.g., `"a/b/c"`, which matches only `"a/b/c/"` path), `"*"` path entries for a single level wildcard (e.g., `"a/*/c"` matches `"a/b/c"` or `'a/d/c"`, but not `"a/c"` or `"a/b/d/c"`), or `"**"` path entries for a multi-level wildcard (e.g., `"a/**/c"` matches `"a/b/c"`, `"a/c"`, or `"a/b/d/c"`). The paths are always tested first for direct match, then for `"*"` match, then for `"**"` match starting from the smallest subpath matching `"**"`.

Parameters

- **root** (*dict or Dictionary*) – A filter tree or a list of filter tree paths (which are all assumed to be have the `True` value).s
- **case_normalization** (*str*) – Case normalization rules; can be `None` (no normalization, names are case-sensitive), `'lower'` or `'upper'`.
- **default** – Default value to return if no match is found.
- **match_prefix** – if `True`, match the result even if only its prefix matches the tree content (same effect as adding `"/**"` to every tree path)
- **copy** (*bool*) – If `True`, make copy of the supplied data; otherwise, just make it the root.

> **Warning:** If `copy==False`, the root data is already assumed to be normalized. If it isn't, the behavior might be incorrect.

copy()

Get a full copy the prefix tree

match(*path*)

Return the match result for the path

add_entry(*path*, *value*, *force=False*, *branch_option='normalize'*)

Add value to a given path (overwrite leaf value if necessary).

Doesn't replace leaves with branches and vice-verse if `force==False`.

Parameters

- **path** –
- **value** –
- **force** (*bool*) – If `True`, change leaf into a branch and vice-versa; otherwise, raises `ValueError` if the conversion is necessary.
- **branch_option** (*str*) –

  Decides what to do if the value is dictionary-like:

  - 'attach' – just attach the root,
  - 'copy' – copy and attach,
  - 'normalize' – copy while normalizing all the keys according to the current rules.

as_dict (*style='nested'*, *copy=True*)
Convert into a dict object.

Parameters

- **style** (*str*) –

  Determines style of the result:

  - 'nested' – subtrees are turned into nested dictionaries,
  - 'flat' – single dictionary is formed with full paths as keys.

- **copy** (*bool*) – If False and style=='nested', return the root dictionary.

static as_dictionary (*obj*, *case_normalization=None*)
Convert object into *Dictionary* with the given parameters.

If object is already a *Dictionary* (or its subclass), return unchanged, even if its parameters are different.

as_json (*style='nested'*)
Convert into a JSON string.

Parameters style (*str*) – Determines style of the result: - 'nested' – subtrees are turned into nested dictionaries, - 'flat' – single dictionary is formed with full paths as keys.

as_pandas (*index_key=True*, *as_series=True*)
Convert into a pandas DataFrame or Series object.

Parameters

- **index_key** (*bool*) – If False, create a 2-column table with the first column ("key") containing string path and the second column ("value") containing value; otherwise, move key to the table index.
- **as_series** (*bool*) – If index_key==True and as_series==True, convert the resulting DataFrame into 1D Series (the key is the index); otherwise, keep it as a single-column table

asdict (*style='nested'*, *copy=True*)
Convert into a dict object.

Parameters

- **style** (*str*) –

  Determines style of the result:

  - 'nested' – subtrees are turned into nested dictionaries,
  - 'flat' – single dictionary is formed with full paths as keys.

- **copy** (*bool*) – If False and style=='nested', return the root dictionary.

branch_copy (*branch=''*)
Get a copy of the branch as a *Dictionary*

branch_pointer (*branch=''*)
Get a `DictionaryPointer` of a given branch

del_entry (*path*)
Delete entry from the dictionary.

Return `True` if the path was present. Note that it never raises *KeyError*.

detach (*path*)
Remove a branch or a leaf from the current dictionary.

Branch is returned as a separate `Dictionary`. If *path* is missing, raise a `KeyError`.

diff (*other*)
Perform an element-wise comparison to another Dictionary.

If the other Dictionary has a different case sensitivity, raise `ValueError`.

Returns `DictionaryDiff`

static diff_flatdict (*first*, *second*)
Find the difference between flat `dict` objects.

Returns `DictionaryDiff`

filter_self (*pred*, *to_visit='leafs'*, *pass_path=False*, *topdown=False*)
Remove all the nodes from the dictionary for which *pred* returns `False`.

Parameters

- **pred** (*callable*) – Filter function. Leafs are passed to *pred* by value, branches (if visited) are passed as `DictionaryPointer`.
- **to_visit** (*str*) – Can be `'leafs'`, `'branches'` or `'all'` and determines which parts of the dictionary passed to the predicate.
- **pass_path** (*bool*) – If `True`, pass the node path (in the form of a normalized list) as a first argument to *pred*.
- **topdown** (*bool*) – If `True`, visit node and its leafs before its subtrees leafs.

static find_intersection (*dicts*, *use_flatten=False*)
Find intersection of multiple dictionaries.

Parameters

- **dicts** ([*Dictionary*]) –
- **use_flatten** (*bool*) – If `True` flatten all dictionaries before comparison (works faster for a large number of dictionaries).

Returns `DictionaryIntersection`

classmethod from_json (*data*, *case_normalization=None*)
Convert JSON representations of a dictionary into a `Dictionary` object

get (*path*, *default=None*)
Analog of `dict.get()`: `D.get(k,d) -> D[k] if k in D else d`

get_entry (*path*, *as_pointer=False*)
Get entry at a given path

Parameters

- **path** –

- **as_pointer** (*bool*) – If `True` and entry is not a leaf, return *DictionaryPointer*; otherwise, return *Dictionary*

get_matching_paths (*pattern*, *wildkey='*'*, *wildpath='**'*, *only_leaves=True*)
Get all paths in the tree that match the provided pattern.

Parameters

- **wildkey** (*str*) – Pattern symbol that matches any key.
- **wildpath** (*str*) – Pattern symbol that matches any subpath (possibly empty).
- **only_leaves** (*bool*) – If `True`, only check leaf paths; otherwise, check subtree paths (i.e., incomplete leaf paths) as well. Basically, `only_leaves=False` is analogous to adding wildpath at the end of the pattern.

get_matching_subtree (*pattern*, *wildkey='*'*, *wildpath='**'*, *only_leaves=True*)
Get a subtree containing nodes with paths matching the provided pattern.

Parameters

- **wildkey** (*str*) – Pattern symbol that matches any key.
- **wildpath** (*str*) – Pattern symbol that matches any subpath (possibly empty).
- **only_leaves** (*bool*) – If `True`, only check leaf paths; otherwise, check subtree paths (i.e., incomplete leaf paths) as well. Basically, `only_leaves=False` is analogous to adding wildpath at the end of the pattern.

get_max_prefix (*path*, *kind='all'*)
Find the longest prefix of *path* contained in the dictionary.

Return tuple `(prefix, rest)`, where both path entries are normalized according to the dictionary rules (i.e., these are lists representing normalized paths). *kind* determines which kind of path to consider and can be `'leaf'`, `'branch'` or `'all'`. If the longest prefix is of a different kind, return `(None, None)`.

get_path ()

has_entry (*path*, *kind='all'*)
Determine if the path is in the dictionary.

kind determines which kind of path to consider and can be `'leaf'`, `'branch'` or `'all'`.

is_branch_path (*path*)
Determine if the path is in the dictionary and points to a branch

static is_dictionary (*obj*, *generic=True*)
Determine if the object is a dictionary.

Parameters

- **obj** –
- **generic** (*bool*) – if False, passes only *Dictionary* (or subclasses) objects; otherwise, passes any dictionary-like object.

Returns bool

is_leaf_path (*path*)
Determine if the path is in the dictionary and points to a leaf

items (*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

iteritems(*ordered=False*, *leafs=False*, *path_kind='split'*, *wrap_branches=True*)
Analog of dict.items(), by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

iterkeys(*ordered=False*, *leafs=False*, *path_kind='split'*)
Analog of dict.keys(), iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)

iternodes(*to_visit='leafs'*, *ordered=False*, *include_path=False*, *topdown=False*)
Iterate over nodes.

Parameters

- **to_visit** (*str*) – Can be 'leafs', 'branches' or 'all' and determines which parts of the dictionary are visited.
- **ordered** (*bool*) – If True, loop over paths in alphabetic order.
- **include_path** (*bool*) – Include in the return value.
- **topdown** (*bool*) – If True, visit node and its leafs before its subtrees leafs.

Yields Values for leafs and *DictionaryPointer* for branches. If include_path==True, yields tuple (path, value), where *path* is in the form of a normalized list.

itervalues(*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of dict.values(), iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.

- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

keys (*ordered=False*, *leafs=False*, *path_kind='split'*)

Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)

map_self (*func*, *to_visit='leafs'*, *pass_path=False*, *topdown=False*, *branch_option='normalize'*)

Apply *func* to the nodes in the dictionary.

Note that any pointers to the replaced branches or their sub-branches will become invalid.

Parameters

- **func** (*callable*) – Mapping function. Leafs are passed by value, branches (if visited) are passed as *DictionaryPointer*.
- **to_visit** (*str*) – Can be 'leafs', 'branches' or 'all' and determines which parts of the dictionary passed to the map function.
- **pass_path** (*bool*) – If True, pass the node path (in the form of a normalized list) as a first argument to *func*.
- **topdown** (*bool*) – If True, visit node and its leafs before its subtrees leafs.
- **branch_option** (*str*) – If the function returns a dict-like object, determines how to incorporate into the dictionary; can be "normalize" (make a copy with normalized paths and insert that), "copy" (make a copy without normalization), or "attach" (simply replace the value without copying and normalization)

merge (*source*, *path=''*, *overwrite=True*, *normalize_paths=True*)

Attach source (dict or other *Dictionary*) to a given branch; source is automatically deep-copied.

If *source* is not a dictionary, simply assign it (i.e., `D.merge(v,p)` is equivalent to `D.add_entry(p, v,force=True)` in this case). Compared to *add_entry()*, merges two branches instead of removing the old branch completely.

Parameters

- **source** (*dict or Dictionary*) –
- **branch** (*tuple or str*) – Destination path.
- **overwrite** (*bool*) – If True, replaces the old entries with the new ones (it only matters for leaf assignments).
- **normalize_paths** (*bool*) – If True and the dictionary isn't case sensitive, perform normalization if the *source*.

nodes (*to_visit='leafs'*, *ordered=False*, *include_path=False*, *topdown=False*)

Iterate over nodes.

Parameters

- **to_visit** (*str*) – Can be 'leafs', 'branches' or 'all' and determines which parts of the dictionary are visited.
- **ordered** (*bool*) – If True, loop over paths in alphabetic order.
- **include_path** (*bool*) – Include in the return value.
- **topdown** (*bool*) – If True, visit node and its leafs before its subtrees leafs.

Yields Values for leafs and *DictionaryPointer* for branches. If include_path==True, yields tuple (path, value), where *path* is in the form of a normalized list.

paths(*ordered=False*, *topdown=False*, *path_kind='split'*)
Return list of all paths (leafs and nodes).

Parameters

- **ordered** (*bool*) – If True, loop over paths in alphabetic order.
- **topdown** (*bool*) – If True, return node's leafs before its subtrees leafs.
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)

pop(*path*, *default=None*)
Analog of dict.pop(): remove value at *path* and return it if path in D, otherwise return *default*

Note that it never raises KeyError.

setdefault(*path*, *default=None*)
Analog of dict.setdefault(): D.setdefault(k,d) -> D.get(k,d), also sets D[k]=d if k not in D.

size()
Return the total size of the dictionary (number of nodes)

update(*source*, *path="*, *overwrite=True*, *normalize_paths=True*)
Attach source (dict or other *Dictionary*) to a given branch; source is automatically deep-copied.

If *source* is not a dictionary, simply assign it (i.e., D.merge(v,p) is equivalent to D.add_entry(p, v,force=True) in this case). Compared to *add_entry()*, merges two branches instead of removing the old branch completely.

Parameters

- **source** (*dict or Dictionary*) –
- **branch** (*tuple or str*) – Destination path.
- **overwrite** (*bool*) – If True, replaces the old entries with the new ones (it only matters for leaf assignments).
- **normalize_paths** (*bool*) – If True and the dictionary isn't case sensitive, perform normalization if the *source*.

updated(*source*, *path="*, *overwrite=True*, *normalize_paths=True*)
Get a copy of the dictionary and attach a new branch to it.

Parameters are the same as in the *Dictionary.merge()*.

values(*ordered=False*, *leafs=False*, *wrap_branches=True*)
Analog of dict.values(), iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

viewitems (*ordered=False, leafs=False, path_kind='split', wrap_branches=True*)

Analog of `dict.items()`, by default iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

viewkeys (*ordered=False, leafs=False, path_kind='split'*)

Analog of `dict.keys()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **path_kind** (*str*) – either "split" (each path is a tuple of individual keys), or "joined" (each path is a single string)

viewvalues (*ordered=False, leafs=False, wrap_branches=True*)

Analog of `dict.values()`, iterating only over the immediate children of the root.

Parameters

- **ordered** (*bool*) – If True, loop over keys in alphabetic order.
- **leafs** (*bool*) – If True, loop over leaf nodes (i.e., behave as 'flat' dictionary); otherwise, loop over immediate children (i.e., behave as 'nested' dictionary)
- **wrap_branches** (*bool*) – if True, wrap sub-branches into *DictionaryPointer* objects; otherwise, return them as nested built-in dictionaries

class pylablib.core.utils.dictionary.**PrefixShortcutTree** (*shortcuts=None*)

Bases: `object`

Convenient storage for dictionary path shortcuts.

Parameters **shortcuts** (*dict*) – Dictionary of shortcuts `{shortcut: full_path}`.

copy ()

Return full copy

add_shortcut (*source*, *dest*, *exact=False*)
Add a single shortcut.

Parameters

- **source** – Shortcut path.
- **dest** – expanded path corresponding to the shortcut.
- **exact** (*bool*) – If `True`, the shortcut works only for the exact path; otherwise, it works for any path with 'source' as a prefix.

add_shortcuts (*shortcuts*, *exact=False*)
Add a dictionary of shortcuts `{shortcut: full_path}`.

Arguments are the same as in `PrefixShortcutTree.add_shortcut()`.

remove_shortcut (*source*)
Remove a shortcut from the tree

updated (*shortcuts*, *exact=False*)
Make a copy and add additional shortcuts.

Arguments are the same as in `PrefixShortcutTree.add_shortcuts()`.

class pylablib.core.utils.dictionary.**DictionaryNode** (***vargs*)
Bases: `object`

pylablib.core.utils.dictionary.**dict_to_object_local** (*data*, *name=None*, *object_generator=<function _default_object_generator>*)

class pylablib.core.utils.dictionary.**ItemAccessor** (*getter=None*, *setter=None*, *deleter=None*, *contains_checker='auto'*, *normalize_names=True*, *path_separator=None*, *missing_error=None*)
Bases: `object`

Simple wrapper which implements array interface using supplied methods.

Also has an option to normalize requested paths (enabled by default)

Parameters

- **getter** – method for getting values (`None` means none is supplied, so getting raises an error)
- **setter** – method for setting values (`None` means none is supplied, so setting raises an error)
- **deleter** – method for deleting values (`None` means none is supplied, so deleting raises an error)
- **contains_checker** – method for checking if variable is present (`None` means none is supplied, so checking containment raises an error; `"auto"` means that getter raising `KeyError` is used for checking)
- **normalize_names** – if `True`, normalize a supplied path using the standard `Dictionary` rules and join it into a single string using the supplied separator
- **path_separator** – path separator regex used for splitting and joining the supplied paths (by default, the standard `"/"` separator)

- **missing_error** – if not None, specifies the error raised on the missing value; used in `__contains__`, `get()` and `setdefault()` to determine if the value is missing

get (*name*, *default=None*)

setdefault (*name*, *default=None*)

pylablib.core.utils.files module

Utilities for working with the file system: creating/removing/listing folders, comparing folders and files, working with zip archives.

`pylablib.core.utils.files.`**eof** (*f*, *strict=False*)
Standard EOF function.

Return `True` if the the marker is at the end of the file. If `strict==True`, only return `True` if the marker is exactly at the end of file; otherwise, return `True` if it's at the end of further.

`pylablib.core.utils.files.`**get_file_creation_time** (*path*, *timestamp=True*)
Try to find a file creation time. Return current time if an error occurs.

If `timestamp==True`, return UNIX timestamp; otherwise, return `datetime.datetime`.

`pylablib.core.utils.files.`**get_file_modification_time** (*path*, *timestamp=True*)
Try to find a file modification time. Return current time if an error occurs.

If `timestamp==True`, return UNIX timestamp; otherwise, return `datetime.datetime`

`pylablib.core.utils.files.`**touch** (*fname*, *times=None*)
Update file access and modification times.

Parameters **times** (`tuple`) – Access and modification times; if *times* is `None`, use current time.

`pylablib.core.utils.files.`**generate_indexed_filename** (*name_format*, *idx_start=0*, *folder=''*)
Generate an unused indexed filename in *folder*.

The name has *name_format* (using standard Python `format()` rules, e.g., `"data_{:03d}.dat"`), and the index starts with *idx_start*.

`pylablib.core.utils.files.`**generate_prefixed_filename** (*prefix=''*, *suffix=''*, *idx_start=None*, *idx_fmt='d'*, *folder=None*)
Generate an unused filename with the given *prefix* and *suffix* in the given *folder*.

By default, the format is `prefix_{:d}_suffix`, where the parameter is the index starting with *idx_start*. If *idx_start* is `None`, first check simply `prefix+suffix` name before using numbered indices.

`pylablib.core.utils.files.`**generate_temp_filename** (*prefix='__tmp__'*, *idx_start=0*, *idx_template='d'*, *folder=''*)
Generate a temporary filename with a given prefix.

idx_template is the number index format (only the parameter itself, not the whole string).

`pylablib.core.utils.files.`**fullsplit** (*path*, *ignore_empty=True*)
Split path into a list.

If `ignore_empty==True`, exclude empty folder names.

`pylablib.core.utils.files.`**normalize_path** (*p*)
Normalize filesystem path (case and origin). If two paths are identical, they should be equal when normalized

`pylablib.core.utils.files.`**`case_sensitive_path`**`()`

Check if OS path names are case-sensitive (e.g., Linux)

`pylablib.core.utils.files.`**`paths_equal`**`(`*a*, *b*`)`

Determine if the two paths are equal (can be local or have different case)

`pylablib.core.utils.files.`**`relative_path`**`(`*a*, *b*, *check_paths=True*`)`

Determine return path *a* as seen from *b*.

If `check_paths==True`, check if *a* is contained in *b* and raise the `OSError` if it isn't.

`pylablib.core.utils.files.`**`is_path_valid`**`(`*p*`)`

Check if the string is a valid path.

Not guaranteed to have complete success rate, but catches most likely errors (invalid characters, reserved file names, too long, etc.) Does not check if the path actually exists or if it can be written into.

`class` `pylablib.core.utils.files.`**`TempFile`**`(`*folder=''*, *name=None*, *mode='w'*, *wait_time=None*, *rep_time=None*`)`

Bases: `object`

Temporary file context manager.

Upon creation, generate an unused temporary filename. Upon entry, create the file using supplied mode and return self. Upon exit, close and remove the file.

Can be mostly substituted by `tempfile.TemporaryFile()`, but generates file locally, and with specified/determined name. Preserved largely for legacy reasons.

Parameters

- **`folder`** (*`str`*) – Containing folder.
- **`name`** (*`str`*) – File name. If `None`, generate new temporary name.
- **`mode`** (*`str`*) – File opening mode.
- **`wait_time`** (*`float`*) – Waiting time between attempts to create the file if the first try fails.
- **`rep_time`** (*`int`*) – Number of attempts to create the file if the first try fails.

`f`

File object.

`name`

File name.

Type str

`full_name`

File name including containing folder.

Type str

`pylablib.core.utils.files.`**`copy_file`**`(`*source*, *dest*, *overwrite=True*, *cmp_on_overwrite=True*, *preserve_metadata=True*`)`

Copy file, creating a containing folder if necessary. Return `True` if the operation was performed.

Parameters

- **`overwrite`** (*`bool`*) – If `True`, overwrite existing file.
- **`cmp_on_overwrite`** (*`bool`*) – If `True` and the two files are compared to be the same, don't perform overwrite.

- **preserve_metadata** (*bool*) – If True, preserve file metadata (such as creation time) by using shutil.copy2(); otherwise, use shutil.copy()

pylablib.core.utils.files.**move_file**(*source*, *dest*, *overwrite=True*, *cmp_on_overwrite=True*, *preserve_if_not_move=False*)

Move file, creating a containing folder if necessary. Returns True if the operation was performed.

Parameters

- **overwrite** (*bool*) – If True, overwrite existing file (if the existing file isn't overwritten, preserve the original).
- **cmp_on_overwrite** (*bool*) – If True and the two files are compared to be the same, don't perform overwrite.
- **preserve_if_not_move** (*bool*) – If True and the files are identical, preserve the original.

pylablib.core.utils.files.**ensure_dir_singlelevel**(*path*, *error_on_file=True*)

pylablib.core.utils.files.**ensure_dir**(*path*, *error_on_file=True*)

Ensure that the folder exists (create a new one if necessary).

If error_on_file==True, raise OSError if there's a file with the same name.

pylablib.core.utils.files.**remove_dir**(*path*, *error_on_file=True*)

Remove the folder recursively if it exists.

If error_on_file==True, raise OSError if there's a file with the same name.

pylablib.core.utils.files.**remove_dir_if_empty**(*path*, *error_on_file=True*)

Remove the folder only if it's empty.

If error_on_file==True, raise OSError if there's a file with the same name.

pylablib.core.utils.files.**clean_dir**(*path*, *error_on_file=True*)

Remove the folder and then recreate it.

If error_on_file==True, raise OSError if there's a file with the same name.

class pylablib.core.utils.files.**FolderList**

Bases: *pylablib.core.utils.files.FolderList*

Describes folder content

count()

Return number of occurrences of value.

files

folders

index()

Return first index of value.

Raises ValueError if the value is not present.

pylablib.core.utils.files.**list_dir**(*folder=''*, *folder_filter=None*, *file_filter=None*, *separate_kinds=True*, *error_on_file=True*)

Return folder content filtered by *folder_filter* and *file_filter*.

Parameters

- **folder** (*str*) – Path to the folder.
- **folder_filter** – Folder filter function (more description at *string.get_string_filter()*).

- **file_filter** – File filter function (more description at *string.get_string_filter()*).
- **separate_kinds** (*bool*) – if `True`, return *FolderList* with files and folder separate; otherwise, return a single list (works much faster).
- **error_on_file** (*bool*) – if `True`, raise `OSError` if there's a file with the same name as the target folder.

`pylablib.core.utils.files.`**`dir_empty`**(*folder*, *folder_filter=None*, *file_filter=None*, *level='single'*, *error_on_file=True*)

Check if the folder is empty (only checks content filtered by *folder_filter* and *file_filter*).

Parameters

- **folder** (*str*) – Path to the folder.
- **folder_filter** – Folder filter function (more description at *string.get_string_filter()*).
- **file_filter** – File filter function (more description at *string.get_string_filter()*).
- **level** (*str*) – if `'single'`, check only immediate folder content; if `'recursive'`, follow recursively in all folders passing *folder_filter*.
- **error_on_file** (*bool*) – if `True`, raise `OSError` if there's a file with the same name as the target folder.

`pylablib.core.utils.files.`**`walk_dir`**(*folder*, *folder_filter=None*, *file_filter=None*, *rel_path=True*, *topdown=True*, *visit_folder_filter=None*, *max_depth=None*)

Modification of `os.walk()` function.

Acts in a similar way, but *followlinks* is always `False` and errors of `os.listdir()` are always passed.

Parameters

- **folder** (*str*) – Path to the folder.
- **folder_filter** – Folder filter function (more description at *string.get_string_filter()*).
- **file_filter** – File filter function (more description at *string.get_string_filter()*).
- **rel_path** (*bool*) – If `True`, the returned folder path is specified relative to the initial path.
- **topdown** (*bool*) – If `True`, return folder before its subfolders.
- **visit_folder_filter** – Filter for visiting folders (more description at *string.get_string_filter()*). If not `None`, specifies filter for visiting folders which is different from *folder_filter* (filter for returned folders).
- **max_depth** (*int*) – If not `None`, limits the recursion depth.

Yields

For each folder (including the original) yields a tuple `(folder_path, folders, files)`, where *folder_path* is the containing folder name and *folders* and *files* are its content (similar to *list_dir()*).

pylablib.core.utils.files.**list_dir_recursive**(*folder, folder_filter=None, file_filter=None, topdown=True, visit_folder_filter=None, max_depth=None*)

Recursive walk analog of `list_dir()`.

Parameters are the same as `walk_dir()`.

Returns `FolderList`

pylablib.core.utils.files.**copy_dir**(*source, dest, folder_filter=None, file_filter=None, overwrite=True, cmp_on_overwrite=True, preserve_metadata=True*)

Copy files satisfying the filtering conditions.

Parameters

- **source** (`str`) – Source path.
- **dest** (`str`) – Destination path.
- **folder_filter** – Folder filter function (more description at `string.get_string_filter()`).
- **file_filter** – File filter function (more description at `string.get_string_filter()`).
- **overwrite** (`bool`) – If `True`, overwrite existing files.
- **cmp_on_overwrite** (`bool`) – If `True` and the two files are compared to be the same, don't perform overwrite.
- **preserve_metadata** (`bool`) – If `True`, preserve file metadata (such as creation time) by using `shutil.copy2()`; otherwise, use `shutil.copy()`

pylablib.core.utils.files.**move_dir**(*source, dest, folder_filter=None, file_filter=None, overwrite=True, cmp_on_overwrite=True, preserve_if_not_move=False*)

Move files satisfying the filtering conditions.

Parameters

- **source** (`str`) – Source path.
- **dest** (`str`) – Destination path.
- **folder_filter** – Folder filter function (more description at `string.get_string_filter()`).
- **file_filter** – File filter function (more description at `string.get_string_filter()`).
- **overwrite** (`bool`) – If `True`, overwrite existing files (if the existing file isn't overwritten, preserve the original).
- **cmp_on_overwrite** (`bool`) – If `True` and the two files are compared to be the same, don't perform overwrite.
- **preserve_if_not_move** (`bool`) – If `True` and the files are identical, preserve the original.

pylablib.core.utils.files.**combine_diff**(*d1, d2*)

pylablib.core.utils.files.**cmp_dirs**(*a, b, folder_filter=None, file_filter=None, shallow=True, return_difference=False*)

Compare the folders based on the content filtered by *folder_filter* and *file_filter*.

Parameters

- **a** (*str*) – First folder path
- **b** (*str*) – Second folder path
- **folder_filter** – Folder filter function (more description at `string.get_string_filter()`).
- **file_filter** – File filter function (more description at `string.get_string_filter()`).
- **shallow** – If `True`, do shallow comparison of the files (see `filecmp.cmp()`).
- **return_difference** – If `False`, simply return *bool*; otherwise, return difference type (`'='`, `'+'`, `'-'` or `'*'`).

`pylablib.core.utils.files.`**retry_copy**(*source*, *dest*, *overwrite=True*, *cmp_on_overwrite=True*, *preserve_metadata=True*, *try_times=5*, *delay=0.3*)

Retrying version of `copy_file()`.

If the operation raises error, wait for *delay* (in seconds) and call it again. Try total of *try_times* times.

`pylablib.core.utils.files.`**retry_move**(*source*, *dest*, *overwrite=True*, *cmp_on_overwrite=True*, *preserve_if_not_move=False*, *try_times=5*, *delay=0.3*)

Retrying version of `move_file()` (see `retry_copy()` for details on retrying).

`pylablib.core.utils.files.`**retry_remove**(*path*, *try_times=5*, *delay=0.3*)

Retrying version of `os.remove()` (see `retry_copy()` for details on retrying).

`pylablib.core.utils.files.`**retry_ensure_dir**(*path*, *error_on_file=True*, *try_times=5*, *delay=0.3*)

Retrying version of `ensure_dir()` (see `retry_copy()` for details on retrying).

`pylablib.core.utils.files.`**retry_copy_dir**(*source*, *dest*, *folder_filter=None*, *file_filter=None*, *overwrite=True*, *cmp_on_overwrite=True*, *preserve_metadata=True*, *try_times=5*, *delay=0.3*)

Retrying version of `copy_dir()` (see `retry_copy()` for details on retrying).

`pylablib.core.utils.files.`**retry_move_dir**(*source*, *dest*, *folder_filter=None*, *file_filter=None*, *overwrite=True*, *cmp_on_overwrite=True*, *preserve_if_not_move=False*, *try_times=5*, *delay=0.3*)

Retrying version of `move_dir()` (see `retry_copy()` for details on retrying).

`pylablib.core.utils.files.`**retry_remove_dir**(*path*, *error_on_file=True*, *try_times=5*, *delay=0.3*)

Retrying version of `remove_dir()` (see `retry_copy()` for details on retrying).

`pylablib.core.utils.files.`**retry_remove_dir_if_empty**(*path*, *error_on_file=True*, *try_times=5*, *delay=0.3*)

Retrying version of `remove_dir_if_empty()` (see `retry_copy()` for details on retrying).

`pylablib.core.utils.files.`**retry_clean_dir**(*path*, *error_on_file=True*, *try_times=5*, *delay=0.3*)

Retrying version of `clean_dir()` (see `retry_copy()` for details on retrying).

`pylablib.core.utils.files.`**zip_folder**(*zip_path*, *source_path*, *inside_path=''*, *folder_filter=None*, *file_filter=None*, *mode='a'*, *compression=8*, *compresslevel=None*)

Add a folder into a zip archive.

Parameters

- **zip_path** (*str*) – Path to the .zip file.

- **source_path** (*str*) – Path to the source folder.
- **inside_path** (*str*) – Destination path inside the zip archive.
- **folder_filter** – Folder filter function (more description at *string.get_string_filter()*).
- **file_filter** – File filter function (more description at *string.get_string_filter()*).
- **mode** (*str*) – Zip archive adding mode (see zipfile.ZipFile).
- **compression** – Zip archive compression (see zipfile.ZipFile).
- **compresslevel** – Zip archive compression level (see zipfile.ZipFile); ignored for Python version below 3.7.

pylablib.core.utils.files.**zip_file**(*zip_path, source_path, inside_name=None, mode='a', compression=8, compresslevel=None*)

Add a file into a zip archive.

Parameters

- **zip_path** (*str*) – Path to the .zip file.
- **source_path** (*str*) – Path to the source file.
- **inside_name** (*str*) – Destination file name inside the zip archive (source name on the top level by default).
- **mode** (*str*) – Zip archive adding mode (see zipfile.ZipFile).
- **compression** – Zip archive compression (see zipfile.ZipFile).
- **compresslevel** – Zip archive compression level (see zipfile.ZipFile); ignored for Python version below 3.7.

pylablib.core.utils.files.**zip_multiple_files**(*zip_path, source_paths, inside_names=None, mode='a', compression=8, compresslevel=None*)

Add a multiple files into a zip archive.

Parameters

- **zip_path** (*str*) – Path to the .zip file.
- **source_paths** (*[str]*) – List of path to the source files.
- **inside_names** (*[str] or None*) – List of destination file names inside the zip archive (source name on the top level by default).
- **mode** (*str*) – Zip archive adding mode (see zipfile.ZipFile).
- **compression** – Zip archive compression (see zipfile.ZipFile).
- **compresslevel** – Zip archive compression level (see zipfile.ZipFile); ignored for Python version below 3.7.

pylablib.core.utils.files.**unzip_folder**(*zip_path, dest_path, inside_path='', folder_filter=None, file_filter=None*)

Extract a folder from a zip archive (create containing folder if necessary).

Parameters

- **zip_path** (*str*) – Path to the .zip file.
- **dest_path** (*str*) – Path to the destination folder.

- **inside_path** (*str*) – Source path inside the zip archive; extracted data paths are relative (i.e., they don't include *inside_path*).
- **folder_filter** – Folder filter function (more description at *string.get_string_filter()*).
- **file_filter** – File filter function (more description at *string.get_string_filter()*).

pylablib.core.utils.files.**unzip_file**(*zip_path*, *dest_path*, *inside_path*)
Extract a file from a zip archive (create containing folder if necessary).

Parameters

- **zip_path** (*str*) – Path to the .zip file.
- **dest_path** (*str*) – Destination file path.
- **inside_path** (*str*) – Source path inside the zip archive.

pylablib.core.utils.funcargparse module

Contains routines for checking arguments passed into a function for better flexibility.

pylablib.core.utils.funcargparse.**parameter_value_error**(*par_val*, *par_name*, *message=None*, *error_type=None*)
Raise parameter value error (`ValueError` by default).

pylablib.core.utils.funcargparse.**parameter_range_error**(*par_val*, *par_name*, *par_set=None*, *message=None*, *error_type=None*)
Raise parameter range error (`ValueError` by default).

pylablib.core.utils.funcargparse.**check_parameter_range**(*par_val*, *par_name*, *par_set*, *message=None*, *error_type=None*)
Raise error if *par_val* is not in in the *par_set* (*par_name* is used in the error message).

pylablib.core.utils.funcargparse.**getdefault**(*value*, *default_value*, *unassigned_value=None*, *conflict_action='ignore'*, *message=None*, *error_type=None*)
Analog of `dict`'s `getdefault`.

If *value* is *unassigned_value*, return *default_value* instead. If `conflict_action=='error'` and `value!=default_value`, raise value error using *message* and *error_type*.

pylablib.core.utils.funcargparse.**is_sequence**(*value*, *sequence_type='builtin;nostring'*)
Check if *value* is a sequence.

***sequence_type* is semicolon separated list of possible sequence types:**

- `'builtin'` - `list`, `tuple` or `str`
- `'nostring'` - `str` is not allows
- `'array'` - `list`, `tuple` or `numpy.ndarray`
- `'indexable'` - anything which can be indexed
- `'haslength'` - anything with length property

pylablib.core.utils.funcargparse.**make_sequence**(*element*, *length=1*, *sequence_type='list'*)

Turn element into a sequence of *sequence_type* ('list' or 'tuple') repeated *length* times.

pylablib.core.utils.funcargparse.**as_sequence**(*value*, *multiply_length=1*, *allowed_type='builtin;nostring'*, *wrapping_type='list'*, *length_conflict_action='ignore'*, *message=None*, *error_type=None*)

Ensure that *value* is a sequence.

If *value* is not a sequence of *allowed_type* (as checked by `is_sequence()`), turn it into a sequence specified by *wrapping_type* and *multiply_length*.

If value is a sequence and `length_conflict_action=='error'`, raise error with *error_type* and *error_message* if the length doesn't match *multiply_length*. Otherwise, return value unchanged.

pylablib.core.utils.functions module

Utilities for dealing with function, methods and function signatures.

class pylablib.core.utils.functions.**FunctionSignature**(*arg_names=None*, *defaults=None*, *varg_name=None*, *kwarg_name=None*, *kwonly_arg_names=None*, *cls=None*, *obj=None*, *name=None*, *doc=None*)

Bases: `object`

Description of a function signature, including name, argument names, default values, names of varg and kwarg arguments, class and object (for methods) and docstring.

Parameters

- **arg_names** (*list*) – Names of the arguments.
- **default** (*dict*) – Dictionary `{name: value}` of default values.
- **varg_name** (*str*) – Name of `*varg` parameter (`None` means no such parameter).
- **kwarg_name** (*str*) – Name of `**kwarg` parameter (`None` means no such parameter).
- **cls** – Caller class, for methods.
- **obj** – Caller object, for methods.
- **name** (*str*) – Function name.
- **doc** (*str*) – Function docstring.

get_defaults_list()

Get list of default values for arguments in the order specified in the signature.

signature(*pass_order=None*)

Get string containing a signature (arguments list) of the function (call or definition), including `*vargs` and `**kwargs`.

If *pass_order* is not `None`, it specifies the order in which the arguments are passed.

wrap_function(*func*, *pass_order=None*)

Wrap a function *func* into a containing function with this signature.

Sets function name, argument names, default values, object and class (for methods) and docstring. If *pass_order* is not `None`, it determines the order in which the positional arguments are passed to the wrapped function.

as_kwargs(*args*, *kwargs*, *add_defaults=False*, *exclude=None*)
Turn `args` and `kwargs` into a single `kwargs` dictionary using the names of positional arguments.

If `add_defaults==True`, add all the non-specified default arguments as well. If the function takes `*args` argument and some of the supplied arguments go there, place them into a list under `"*"` key in the result. If *exclude* is not `None` is specifies arguments which should be excluded.

arg_value(*argname*, *args=None*, *kwargs=None*)
Get the value of the argument with the given name for given args and kwargs

mandatory_args_num()
Get minimal number of arguments which have to be passed to the function.

The mandatory arguments are the ones which are not bound to caller object (i.e., not `self`) and don't have default values.

max_args_num(*include_positional=True*, *include_keywords=True*)
Get maximal number of arguments which can be passed to the function.

Parameters

- **include_positional** (*bool*) – If `True` and function accepts `*vargs`, return `None` (unlimited number of arguments).
- **include_keywords** (*bool*) – If `True` and function accepts `**kwargs`, return `None` (unlimited number of arguments).

static from_function(*func*, *follow_wrapped=True*)
Get signature of the given function or method.

If `follow_wrapped==True`, follow __wrapped__ attributes until the innermost function (useful for getting signatures of functions wrapped using `functools` methods).

copy()
Return a copy

as_simple_func()
Turn the signature into a simple function (as opposed to a bound method).

If the signature corresponds to a bound method, get rid of the first argument in the signature (`self`) and the bound object. Otherwise, return unchanged.

static merge(*inner*, *outer*, *add_place='front'*, *merge_duplicates=True*, *overwrite=None*, *hide_outer_obj=False*)
Merge two signatures (used for wrapping functions).

The signature describes the function would take arguments according to the *outer* signature and pass them according to the *inner* signature.

The arguments are combined:

- if `add_place=='front'`, the outer arguments are placed in the beginning, followed by inner arguments not already listed;
- if `add_place=='back'`, the inner arguments are placed in the beginning, followed by outer arguments not already listed.

The default values are joined, with the outer values superseding the inner values.

overwrite is a set or a list specifying which inner parameters are overwritten by the outer. It includes `'name'`, `'doc'`, `'cls'`, `'obj'`, `'varg_name'` and `'kwarg_name'`; the default value is all parameters.

If the inner signature is a bound method and `hide_inner_obj==True`, treat it as a function (with `self` argument missing). In this case, the wrapped signature `.obj` field will be `None`.

Returns

`(signature, pass_order)`

pass_order is the order in which the arguments of the combined signature may be passed to the inner signature; it may be different from the signature order if `add_place=='front'`. If `merge_duplicates==True`, duplicate entries in *pass_order* are omitted; otherwise, they're repeated.

Return type tuple

`pylablib.core.utils.functions.`**`funcsig`**(*func*, *follow_wrapped=True*)
Return a function signature object

`pylablib.core.utils.functions.`**`getargsfrom`**(*source*, ***merge_params*)
Decorator factory.

Returns decorator that conforms function signature to the source function. `**merge_params` are passed to the `FunctionSignature.merge()` method merging wrapped and source signature.

The default behavior (conforming parameter names, default values args and kwargs names) is useful for wrapping universal functions like `g(*args, **kwargs)`.

Example:

```
def f(x, y=2):
    return x+y

@getargsfrom(f)
def g(*args): # Now g has the same signature as f, including parameter names and␣
↪default values.
    return prod(args)
```

`pylablib.core.utils.functions.`**`call_cut_args`**(*func*, **args*, ***kwargs*)
Call *func* with the given arguments, omitting the ones that don't fit its signature.

`pylablib.core.utils.functions.`**`getattr_call`**(*obj*, *attr_name*, **args*, ***vargs*)
Call the getter for the attribute *attr_name* of *obj*.

If the attribute is a property, pass `*args` and `**kwargs` to the getter (*fget*); otherwise, ignore them.

`pylablib.core.utils.functions.`**`setattr_call`**(*obj*, *attr_name*, **args*, ***vargs*)
Call the setter for the attribute *attr_name* of *obj*.

If the attribute is a property, pass `*args` and `**kwargs` to the setter (*fset*); otherwise, the set value is assumed to be either the first argument, or the keyword argument with the name `'value'`.

`pylablib.core.utils.functions.`**`delattr_call`**(*obj*, *attr_name*, **args*, ***vargs*)
Call the deleter for the attribute *attr_name* of *obj*.

If the attribute is a property, pass `*args` and `**kwargs` to the deleter (*fdel*); otherwise, ignore them.

class `pylablib.core.utils.functions.`**`IObjectCall`**
Bases: `object`

Universal interface for object method call (makes methods, attributes and properties look like methods).

Should be called with an object as a first argument.

class pylablib.core.utils.functions.**MethodObjectCall**(*method*)

Bases: *pylablib.core.utils.functions.IObjectCall*

Object call created from an object method.

Parameters **method** – Either a method object or a method name which is used for the call.

class pylablib.core.utils.functions.**AttrObjectCall**(*name*, *as_getter*)

Bases: *pylablib.core.utils.functions.IObjectCall*

Object call created from an object attribute (makes attributes and properties look like methods).

Parameters

- **name** (*str*) – Attribute name.
- **as_getter** (*bool*) – If `True`, call the getter when invoked; otherwise, call the setter.

If an attribute is a simple attribute, than getter gets no arguments and setter gets one argument (either the first argument, or the keyword argument named `'value'`). If it's a property, pass all the parameters to the property call.

class pylablib.core.utils.functions.**IObjectProperty**

Bases: `object`

Universal interface for an object property (makes methods, attributes and properties look like properties).

Can be used to get, set or remove a property.

get(*obj*, *params=None*)

set(*obj*, *value*)

rem(*obj*, *params=None*)

class pylablib.core.utils.functions.**MethodObjectProperty**(*getter=None*, *setter=None*, *remover=None*, *expand_tuple=True*)

Bases: *pylablib.core.utils.functions.IObjectProperty*

Object property created from object methods (makes methods look like properties).

Parameters

- **getter** (*callable*) – Method invoked on `get()`. If `None`, raise `RuntimeError` when called.
- **setter** (*callable*) – Method invoked on `set()`. If `None`, raise `RuntimeError` when called.
- **remover** (*callable*) – Method invoked on `rem()`. If `None`, raise `RuntimeError` when called.
- **expand_tuple** (*bool*) – If `True` and if the first argument in the method call is a tuple, expand it as an argument list for the underlying function call.

get(*obj*, *params=None*)

set(*obj*, *value*)

rem(*obj*, *params=None*)

class pylablib.core.utils.functions.**AttrObjectProperty**(*name*, *use_getter=True*, *use_setter=True*, *use_remover=True*, *expand_tuple=True*)

Bases: *pylablib.core.utils.functions.IObjectProperty*

Object property created from object attribute. Works with attributes or properties.

Parameters

- **name** (*str*) – Attribute name.
- **use_getter** (*bool*) – If False, raise RuntimeError when calling get method.
- **use_setter** (*bool*) – If False, raise RuntimeError when calling set method.
- **use_remover** (*bool*) – If False, raise RuntimeError when calling rem method.
- **expand_tuple** (*bool*) – If True and if the first argument in the method call is a tuple, expand it as an argument list for the underlying function call.

get(*obj*, *params=None*)

set(*obj*, *value*)

rem(*obj*, *params=None*)

pylablib.core.utils.functions.**empty_object_property**(*value=None*)

Dummy property which does nothing and returns *value* on *get* (None by default).

pylablib.core.utils.functions.**obj_prop**(**args*, ***kwargs*)

Build an object property wrapper.

If no arguments (or a single None argument) are supplied, return a dummy property. If one argument is supplied, return *AttrObjectProperty* for a property with a given name. Otherwise, return *MethodObjectProperty* property.

pylablib.core.utils.functions.**as_obj_prop**(*value*)

Turn value into an object property using *obj_prop()* function.

If it's already *IObjectProperty*, return unchanged. If *value* is a tuple, expand as an argument list.

pylablib.core.utils.functions.**delaydef**(*gen*)

Wrapper for a delayed definition of a function inside of a module.

Useful if defining a function is computationally costly. The wrapped function should be a generator of the target function rather than the function itself.

On the first call the generator is executed to define the target function, which is then substituted for all subsequent calls.

pylablib.core.utils.general module

Collection of small utilities.

pylablib.core.utils.general.**set_props**(*obj*, *prop_names*, *props*)

Set multiple attributes of *obj*.

Names are given by *prop_names* list and values are given by *props* list.

pylablib.core.utils.general.**get_props**(*obj*, *prop_names*)

Get multiple attributes of *obj*.

Names are given by *prop_names* list.

`pylablib.core.utils.general.`**`using_method`**(*func*, *method_name=None*, *inherit_signature=True*)

Decorator that makes the function attempt to call the first argument's method instead of *func*.

Before calling the function, try and call a method of the first argument named *method_name* (*func* name by default). If the method exists, call it instead of the wrapped function. If `inherit_signature==True`, completely copy the signature of the wrapped method (name, args list, docstring, etc.).

`pylablib.core.utils.general.`**`to_predicate`**(*x*)

Turn *x* into a predicate.

If *x* is callable, it will be called with a single argument and returned value determines if the argument passes. If *x* is a container, an argument passes if it's contained in *x*.

`pylablib.core.utils.general.`**`map_container`**(*value*, *func*)

Map values in the container.

value can be a `tuple`, a `list` or a `dict` (mapping is applied to the values) raises `ValueError` if it's something else.

`pylablib.core.utils.general.`**`as_container`**(*val*, *t*)

Turn iterable into a container of type *t*.

Can handle named tuples, which have different constructor signature.

`pylablib.core.utils.general.`**`recursive_map`**(*value*, *func*)

Map container recursively.

value can be a `tuple`, a `list` or a `dict` (mapping is applied to the values).

`pylablib.core.utils.general.`**`make_flat_namedtuple`**(*nt*, *fields=None*, *name=None*, *subfield_fmt='{field:}_{subfield:}'*)

Turn a nested structure of named tuples into a single flat namedtuple.

Parameters

- **`nt`** – toplevel namedtuple class to be flattened
- **`fields`** – a dictionary `{name: desc}` of the fields, where `name` is the named tuple name, and `desc` is either a nested namedtuple class, or a list of arguments which are passed to the recursive call to this function (e.g., `[TTuple, {"field": TNestedTuple}]`). Any tuple field which is present in this dictionary gets recursively flattened, and the field names of the corresponding returned tuple are added to the full list of fields
- **`name`** – name of the resulting tuple
- **`subfield_fmt`** – format string, which describes how the combined field name is built out of the original field name and the subtuple field name; by default, connect with `"_"`, i.e., `t.field.subfiled` turns into `t.field_subfield`.

Returns a new namedtuple class, which describes the flattened structure

`pylablib.core.utils.general.`**`any_item`**(*d*)

Return arbitrary tuple `(key, value)` contained in the dictionary (works both in Python 2 and 3)

`pylablib.core.utils.general.`**`merge_dicts`**(**dicts*)

Combine multiple `dict` objects together.

If multiple dictionaries have the same keys, later arguments have higher priority.

`pylablib.core.utils.general.`**`filter_dict`**(*pred*, *d*, *exclude=False*)

Filter dictionary based on a predicate.

pred can be a callable or a container (in which case the predicate is true if a value is in the container). If `exclude==True`, the predicate is inverted.

`pylablib.core.utils.general.`**`map_dict_keys`**(*func*, *d*)
Map dictionary keys with *func*

`pylablib.core.utils.general.`**`map_dict_values`**(*func*, *d*)
Map dictionary values with *func*

`pylablib.core.utils.general.`**`to_dict`**(*d*, *default=None*)
Convert a `dict` or a `list` of pairs or single keys (or mixed) into a `dict`.

If a list element is single, *default* value is used.

`pylablib.core.utils.general.`**`to_pairs_list`**(*d*, *default=None*)
Convert a `dict` or a `list` of pairs or single keys (or mixed) into a `list` of pairs.

If a list element is single, *default* value is used. When converting `list` into `list`, the order is preserved.

`pylablib.core.utils.general.`**`invert_dict`**(*d*, *kmap=None*)
Invert dictionary (switch keys and values).

If *kmap* is supplied, it's a function mapping dictionary values into inverted dictionary keys (identity by default).

`pylablib.core.utils.general.`**`flatten_list`**(*l*)
Flatten nested `list`/`tuple` structure into a single list.

`pylablib.core.utils.general.`**`partition_list`**(*pred*, *l*)
Split the lis' *l* into two parts based on the predicate.

`pylablib.core.utils.general.`**`split_in_groups`**(*key_func*, *l*, *continuous=True*, *max_group_size=None*)
Split the list *l* into groups according to the *key_func*.

Go over the list and group the elements with the same key value together. If `continuous==False`, groups all elements with the same key together regardless of where they are in the list. otherwise, group only continuous sequences of the elements with the same key together (element with different key in the middle will result in two groups). If `continuous==True` and *max_group_size* is not `None`, it determines the maximal size of a group; larger groups are split into separate groups.

`pylablib.core.utils.general.`**`sort_set_by_list`**(*s*, *l*, *keep_duplicates=True*)
Convert the set *s* into a list ordered by a list *l*.

Elements in *s* which are not in *l* are omitted. If `keep_duplicates==True`, keep duplicate occurrences in *l* in the result; otherwise, only keep the first occurrence.

`pylablib.core.utils.general.`**`compare_lists`**(*l1*, *l2*, *sort_lists=False*, *keep_duplicates=True*)
Return three lists `(l1 and l2, l1-l2, l2-l1)`.

If `sort_lists==True`, sort the first two lists by *l1*, and the last one by *l2*; otherwise, the order is undefined. If `sort_lists==True`, *keep_duplicated* determines if duplicate elements show up in the result.

`pylablib.core.utils.general.`**`topological_order`**(*graph*, *visit_order=None*)
Get a topological order of a graph.

Return a list of nodes where each node is listed after its children. If *visit_order* is not `None`, it is a list specifying nodes visiting order (nodes earlier in the list are visited first). Otherwise, the visit order is undefined. *graph* is a dictionary `{node: [children]}`. If graph contains loops, raise `ValueError`.

class `pylablib.core.utils.general.`**`DummyResource`**
Bases: `object`

Object that acts as a resource (has `__enter__` and `__exit__` methods), but doesn't do anything.

Analog of:

```
@contextlib.contextmanager
def dummy_resource():
    yield
```

class pylablib.core.utils.general.**RetryOnException**(*tries=None*, *exceptions=None*)

Bases: object

Wrapper for repeating the same block of code several time if an exception occurs

Useful for filesystem or communication operations, where retrying a failed operation is a valid option.

Parameters

- **tries** (*int*) – Determines how many time will the chunk of code execute before re-raising the exception; None (default) means no limit
- **exceptions** (*Exception or list*) – A single exception class or a list of exception classes which are going to be silenced.

Example:

```
for t in RetryOnException(tries,exceptions):
    with t:
        ... do stuff ...
```

is analogue of:

```
for i in range(tries):
    try:
        ... do stuff ...
    except exceptions:
        if i==tries-1:
            raise
```

class ExceptionCatcher(*retrier*, *try_number*)

Bases: object

reraise()

pylablib.core.utils.general.**retry_wait**(*func*, *try_times=1*, *delay=0.0*, *exceptions=None*)

Try calling function (with no arguments) at most *try_times* as long as it keeps raising exception.

If *exceptions* is not None, it specifies which exception types should be silenced. If an exception has been raised, wait *delay* seconds before retrying.

class pylablib.core.utils.general.**SilenceException**(*exceptions=None*, *on_exception=None*, *reraise=False*)

Bases: object

Context which silences exceptions raised in a block of code.

Parameters

- **exceptions** (*Exception or list*) – A single exception class or a list of exception classes which are going to be silenced.
- **on_exception** (*callable*) – A callback to be invoked if an exception occurs.
- **reraise** (*bool*) – Defines if the exception is re-raised after the callback has been invoked.

A simple bit of syntax sugar. The code:

```
with SilenceException(exceptions,on_exception,reraise):
    ... do stuff ...
```

is exactly analogous to:

```
try:
    ... do stuff ...
except exceptions:
    on_exception()
    if reraise:
        raise
```

`pylablib.core.utils.general.`**`full_exit`**`(`*code=<Signals.SIGTERM: 15>*`)`

Terminate the current process and all of its threads.

Doesn't perform any cleanup or resource release; should only be used if the process is irrevocably damaged.

`class` `pylablib.core.utils.general.`**`UIDGenerator`**`(`*thread_safe=False*`)`

Bases: `object`

Generator of unique numeric IDs.

Parameters **`thread_safe`** (*bool*) – If `True`, using lock to ensure that simultaneous calls from different threads are handled properly.

`reset``(`*value=0*`)`

Reset the generator to the given value

`class` `pylablib.core.utils.general.`**`NamedUIDGenerator`**`(`*name_template='{0}{1:03d}'*, *thread_safe=False*`)`

Bases: `object`

Generator of unique string IDs based on a name.

Parameters

- **`name_template`** (*str*) – Format string with two parameters (name and numeric ID) used to generate string IDs.
- **`thread_safe`** (*bool*) – If `True`, using lock to ensure that simultaneous calls from different threads are handled properly.

`pylablib.core.utils.general.`**`call_limit`**`(`*func*, *period=1*, *cooldown=0.0*, *limit=None*, *default=None*`)`

Wrap *func* such that calls to it are forwarded only under certain conditions.

If `period>1`, then *func* is called after at least *period* calls to the wrapped function. If `cooldown>0`, then *func* is called after at least *cooldown* seconds passed since the last call. if `limit is not None`, then *func* is called only first *limit* times. If several conditions are specified, they should be satisfied simultaneously. *default* specifies return value if *func* wasn't called. Returned function also has an added method `reset`, which resets the internal call and time counters.

`pylablib.core.utils.general.`**`doc_inherit`**`(`*parent*`)`

Wrapper for inheriting docstrings from parent classes.

Takes parent class as an argument and replaces the docstring of the wrapped function by the docstring of the same-named function from the parent class (if available).

`class` `pylablib.core.utils.general.`**`Countdown`**`(`*timeout*, *start=True*`)`

Bases: `object`

Object for convenient handling of timeouts and countdowns with interrupts.

Parameters

- **timeout** (*float*) – Countdown timeout; if `None`, assumed to be infinite.
- **start** (*bool*) – if `True`, automatically start the countdown; otherwise, wait until `trigger()` is called explicitly

reset (*start=True*)
Restart the countdown from the current moment

trigger (*restart=True*)
Trigger the countdown.

If `restart==True`, restart the countdown if it's running; otherwise, do nothing in that situation.

running ()
Check if the countdown is running

stop ()
Stop the timer if currently running

time_left (*t=None*, *bound_below=True*)
Return the amount of time left. For infinite timeout, return `None`.

If `bound_below==True`, instead of negative time return zero. If *t* is supplied, it indicates the current time; otherwise, use `time.time()`.

add_time (*dt*, *t=None*, *bound_below=True*)
Add a given amount of time (positive or negative) to the start time (timeout stays the same).

If `bound_below==True`, do not let the end time (start time plus timeout) to get below the current time. If *t* is supplied, it indicates the current time; otherwise, use `time.time()`.

set_timeout (*timeout*)
Change the timer timeout

time_passed ()
Return the amount of time passed since the countdown start/reset, or `None` if it is not started

passed ()
Check if the timeout has passed

class `pylablib.core.utils.general.`**Timer** (*period*, *skip_first=False*)
Bases: `object`

Object for keeping time of repeating tasks.

Parameters **period** (*float*) – Timer period.

change_period (*period*, *method='current'*)
Change the timer period.

method specifies the changing method. Could be `"current"` (change the period of the ongoing tick), `"next"` (change the period starting from the next tick), `"reset_skip"` (reset the timer and skip the first tick) or `"reset_noskip"` (reset the timer and don't skip the first tick).

reset (*skip_first=False*)
Reset the timer.

If `skip_first==False`, timer ticks immediately; otherwise, it starts ticking only after one period.

time_left (*t=None*, *bound_below=True*)
Return the amount of time left before the next tick.

If `bound_below==True`, instead of negative time return zero.

passed(*t=None*)
Return the number of ticks passed.

If timer period is zero, always return 1.

acknowledge(*n=None*, *nmin=0*)
Acknowledge the timer tick.

n specifies the number of tick to acknowledge (by default, all passed). Return number of actually acknowledged ticks (0 if the timer hasn't ticked since the last acknowledgement).

class pylablib.core.utils.general.**StreamFileLogger**(*path*, *stream=None*, *lock=None*, *autoflush=False*)

Bases: `object`

Stream logger that replaces standard output stream (usually stdout or stderr) and logs them into a file.

Parameters

- **path** – path to the destination logfile. The file is always appended.
- **stream** – an optional output stream into which the output will be duplicated; usually, the original stream which is being replaced
- **lock** – a thread lock object, which is used for any file writing operation; necessary if replacing standard streams (such as `sys.stdout` or `sys.stderr`) in a multithreading environment.
- **autoflush** – if `True`, flush after any write operation into *stream*

It is also possible to subclass the file and overload `write_header()` method to write a header before the first file write operation during the execution.

The intended use is to log stdout or stderr streams:

```
import sys, threading
sys.stderr = StreamFileLogger("error_log.txt", stream=sys.stderr, lock=threading.
↪Lock())
```

write_header(*f*)
Write header to file stream *f*

add_path(*path*)
Add another logging path to the list

remove_path(*path*)
Remove logging path to the list

write(*s*)

flush()

pylablib.core.utils.general.**setbp**()

pylablib.core.utils.general.**timing**(*n=1*, *name=None*, *profile=False*)
Context manager for timing a piece of code.

Measures the time it takes to execute the wrapped code and prints the result.

Parameters

- **n** – can specify the number of repetitions, which is used to show time per single repetition.
- **name** – name which is printed alongside the time
- **profile** – if `True`, use `cProfile` and print its output instead of a simple timing

class pylablib.core.utils.general.**AccessIterator**(*obj*, *access_function=None*)

Bases: `object`

Simple sequential access iterator with customizable access function (by default it's 1D indexing).

Determines end of iterations by `IndexError`.

Parameters

- **obj** – Container to be iterated over.
- **access_function** (*callable*) – A function which takes two parameters *obj* and *idx* and either returns the element or raises `IndexError`. By default, a simple `__getitem__` operation.

next()

pylablib.core.utils.general.**muxcall**(*argname*, *all_arg_value='all'*, *all_arg_func=None*, *mux_argnames=None*, *return_kind='list'*, *allow_partial=False*)

Wrap a function such that it can become multiplexable over a given argument.

Parameters

- **argname** – name of the argument to loop over
- **all_arg_value** – value of *argname* argument which indicates that the function should be multiplexed over all argument values
- **all_arg_func** – function which takes the same arguments as the wrapped function and returns a list of values for *argname* to loop over
- **mux_argnames** – names of additional arguments which, when supplied list or dict values, and when the *argname* value is a list, specify different values for different calls
- **return_kind** – method to combined multiple returned values; can be `"list"`, `"dict"` (return dict `{arg: result}`), or `"none"` (simply return `None`)
- **allow_partial** – if `True` and some of *mux_argnames* argument do not specify value for the full range of *argname* value, do not call the function for those unspecified values; otherwise (*allow_partial* is `True`), the error will be raised

pylablib.core.utils.general.**wait_for_keypress**(*message='Waiting...'*)

pylablib.core.utils.general.**restart**()

Restart the script.

Execution will not resume after this call. Note: due to Windows limitations, this function does not replace the current process with a new one, but rather calls a new process and makes the current one wait for its execution. Hence, each nested call adds an additional loaded application into the memory. Therefore, nesting restart calls (i.e., calling several restarts in a row) should be avoided.

pylablib.core.utils.indexing module

Processing and normalization of different indexing styles.

pylablib.core.utils.indexing.**string_list_idx**(*names_to_find*, *names_list*, *only_exact=False*)

Index through a list of strings in *names_list*.

Return corresponding numerical indices. Case sensitive; first look for exact matching, then for prefix matching (unless `only_exact=True`).

`pylablib.core.utils.indexing.`**`is_slice`**(*idx*)
Check if *idx* is slice.

`pylablib.core.utils.indexing.`**`is_range`**(*idx*)
Check if *idx* is iterable (list, numpy array, or *builtins.range*).

`pylablib.core.utils.indexing.`**`is_bool_array`**(*idx*)
Check if *idx* is a boolean array.

`pylablib.core.utils.indexing.`**`to_range`**(*idx, length*)
Turn list, array, builtins.range, slice into an iterable.

`pylablib.core.utils.indexing.`**`covers_all`**(*idx, length, strict=False, ordered=True*)
Check if *idx* covers all of the elements (indices from 0 to *length*).

If `strict==True`, strictly checks the condition; otherwise may return `False` even if *idx* actually covers everything, but takes less time (i.e., can be used for optimization). If `ordered==True`, only returns `True` when indices follow in order.

class `pylablib.core.utils.indexing.`**`IIndex`**
Bases: `object`

A generic index object.

Used to transform a variety of indexes into a subset applicable for specific objects (numpy arrays or lists).

Allowed input index types:

- scalar: integer, string
- vector: integer lists or numpy arrays, bool lists or numpy arrays, string lists or numpy arrays, builtin.ranges, slices and string slices

`tup`()
Represent index as a tuple for easy unpacking.

class `pylablib.core.utils.indexing.`**`NumpyIndex`**(*idx, ndim=None*)
Bases: *`pylablib.core.utils.indexing.IIndex`*

NumPy compatible index: allows for integers, slices, numpy integer or boolean arrays, integer lists or builtin.ranges.

Parameters

- **`idx`** – raw index
- **`ndim`** – index dimensionality (either 0 or 1); if supplied, assume that *idx* is already normalized

`tup`()
Represent index as a tuple for easy unpacking.

class `pylablib.core.utils.indexing.`**`ListIndex`**(*idx, names=None, ndim=None*)
Bases: *`pylablib.core.utils.indexing.IIndex`*

List compatible index: allows for integers, slices, numpy integer arrays, integer lists or builtin.ranges.

Parameters

- **`idx`** – raw index
- **`names`** – list of allowed index string values, which is used to convert them into integers
- **`ndim`** – index dimensionality (either 0 or 1); if supplied, assume that *idx* is already normalized

tup()

Represent index as a tuple for easy unpacking.

class pylablib.core.utils.indexing.**ListIndexNoSlice**(*idx*, *names=None*, *length=None*, *ndim=None*)

Bases: *pylablib.core.utils.indexing.ListIndex*

List compatible index with slice unwrapped into builtin.range: allows for integers, numpy integer arrays, integer lists or builtin.ranges.

Parameters

- **idx** – raw index
- **names** – list of allowed index string values, which is used to convert them into integers
- **length** – length of the list (used to expand slice indices)
- **ndim** – index dimensionality (either 0 or 1); if supplied, assume that *idx* is already normalized

tup()

Represent index as a tuple for easy unpacking.

pylablib.core.utils.indexing.**to_double_index**(*idx*, *names*)

Convert double index into a pair of indexes.

Assume that one index is purely numerical, while the other can take names (out of the supplied list).

Parameters

- **idx** – raw double index
- **names** – list of allowed index string values, which is used to convert them into integers

pylablib.core.utils.ipc module

Universal interface for inter-process communication.

Focus on higher throughput for large numpy arrays via shared memory.

class pylablib.core.utils.ipc.**IIPCChannel**

Bases: object

Generic IPC channel interface

send(*data*)

Send data

recv(*timeout=None*)

Receive data

send_numpy(*data*)

Send numpy array

recv_numpy(*timeout=None*)

Receive numpy array

get_peer_args()

Get arguments required to create a peer connection

classmethod from_args(**args*)

Create a peer connection from the supplied arguments

class pylablib.core.utils.ipc.**TPipeMsg**(*id*, *data*)
Bases: tuple

count()
Return number of occurrences of value.

data

id

index()
Return first index of value.

Raises ValueError if the value is not present.

class pylablib.core.utils.ipc.**PipeIPCChannel**(*pipe_conn=None*)
Bases: *pylablib.core.utils.ipc.IIPCChannel*

Generic IPC channel interface using pipe.

get_peer_args()
Get arguments required to create a peer connection

send(*data*)
Send data

recv(*timeout=None*)
Receive data

classmethod from_args(**args*)
Create a peer connection from the supplied arguments

recv_numpy(*timeout=None*)
Receive numpy array

send_numpy(*data*)
Send numpy array

class pylablib.core.utils.ipc.**SharedMemIPCChannel**(*pipe_conn=None*, *arr=None*, *arr_size=None*)
Bases: *pylablib.core.utils.ipc.PipeIPCChannel*

Generic IPC channel interface using pipe and shared memory for large arrays.

get_peer_args()
Get arguments required to create a peer connection

send_numpy(*data*, *method='auto'*, *timeout=None*)
Send numpy array

recv_numpy(*timeout=None*)
Receive numpy array

classmethod from_args(**args*)
Create a peer connection from the supplied arguments

recv(*timeout=None*)
Receive data

send(*data*)
Send data

class pylablib.core.utils.ipc.**TShmemVarDesc**(*offset*, *size*, *kind*, *fixed_size*)
Bases: tuple

count()
Return number of occurrences of value.

fixed_size

index()
Return first index of value.

Raises ValueError if the value is not present.

kind

offset

size

class pylablib.core.utils.ipc.**SharedMemIPCTable**(*pipe_conn=None*, *arr=None*, *arr_size=None*, *lock=True*)
Bases: `object`

Shared memory table for exchanging shared variables between processes.

Can be used instead of channels for variables which are rarely changed but frequently checked (e.g., status), or when synchronization of sending and receiving might be difficult

add_variable(*name*, *size*, *kind='pickle'*)
Add a variable with a given name.

The variable info is also communicated to the other endpoint. *size* determines maximal variable size in bytes. If the actual size ever exceeds it, an exception will be raised. *kind* determines the way to convert variable into bytes; can be `"pickle"` (universal, but large size overhead), `"nps_###"`` (where `###` can be any numpy scalar dtype description, e.g., `"float"` or `"<u2"`) for numpy scalars, or `"npa_###"`` (where `###` means the same as for `nps`) for numpy arrays (in this case the array size and shape need to be communicated separately).

set_variable(*name*, *value*)
Set a variable with a given name.

If the variable is missing, raise an exception.

get_variable(*name*, *default=None*)
Get a variable with a given name.

If the variable is missing, return *default*.

is_peer_connected()
Check if the peer is connected (i.e., the other side of the pipe is initialized)

close_connection()
Mark the connection as closed

is_peer_closed()
Check if the peer is closed

get_peer_args()
Get arguments required to create a peer connection

classmethod from_args(**args*)
Create a peer connection from the supplied arguments

pylablib.core.utils.library_parameters module

Storage for global library parameters

pylablib.core.utils.library_parameters.**temp_library_parameters**(*restore=None*)
Context manager, which restores library parameters upon exit.

If `rester is not None`, it can specify a list of parameters to be restored (by default, all parameters).

pylablib.core.utils.module module

Library for dealing with python module properties.

pylablib.core.utils.module.**get_package_version**(*pkg*)
Get the version of the package.

If the package version is unavailable, return `None`.

pylablib.core.utils.module.**cmp_versions**(*ver1*, *ver2*)
Compare two package versions.

Return `'<'` if the first version is older (smaller), `'>'` if it's younger (larger) or `'='` if it's the same.

pylablib.core.utils.module.**cmp_package_version**(*pkg*, *ver*)
Compare current package version to *ver*.

ver should be a name of the package (rather than the module). Return `'<'` if current version is older (smaller), `'>'` if it's younger (larger) or `'='` if it's the same. If the package version is unavailable, return `None`.

pylablib.core.utils.module.**expand_relative_path**(*module_name*, *rel_path*)
Turn a relative module path into an absolute one.

module_name is the absolute name of the reference module, *rel_path* is the path relative to this module.

pylablib.core.utils.module.**get_loaded_package_modules**(*pkg_name*)
Get all modules in the package *pkg_name*.

Returns a dict `{name: module}`.

pylablib.core.utils.module.**get_imported_modules**(*module*, *explicit=False*)
Get modules imported within a given module.

If `explicit==True`, take into account only toplevel objects which are modules (corresponds to `import module` or `from package import module` statements) If `explicit==False`, also include all modules containing toplevel objects (corresponds to `from module import Class` or `from package import function` statements). Return a dictionary `{name: module}` (modules with the same name are considered to be the same).

pylablib.core.utils.module.**get_reload_order**(*modules*)
Find reload order for modules which respects dependencies (a module is loaded before its dependents).

modules is a dict `{name: module}`.

The module dependencies (i.e., the modules which the current module depends on) are determined based on imported modules and modules containing toplevel module objects.

pylablib.core.utils.module.**reload_package_modules**(*pkg_name*, *ignore_errors=False*)
Reload package *pkg_name*, while respecting dependencies of its submodules.

If `ignore_errors=True`, ignore `ImportError` exceptions during the reloading process.

pylablib.core.utils.module.**unload_package_modules**(*pkg_name*, *ignore_errors=False*)
Reload package *pkg_name*, while respecting dependencies of its submodules.

If `ignore_errors=True`, ignore `ImportError` exceptions during the reloading process.

`pylablib.core.utils.module.`**`get_library_path`**`()`
: Get a filesystem path for the pyLabLib library (the one containing current the module).

`pylablib.core.utils.module.`**`get_library_name`**`()`
: Get the name for the pyLabLib library (the one containing current the module).

`pylablib.core.utils.module.`**`pip_install`**`(`*pkg*, *upgrade=False*`)`
: Call `pip install` for a given package.

  If `upgrade==True`, call with `--upgrade` key (upgrade current version if it is already installed).

`pylablib.core.utils.module.`**`install_if_older`**`(`*pkg*, *min_ver=''*`)`
: Install *pkg* from the default PyPI repository if its version is lower that *min_ver*

  If *min_ver* is `None`, upgrade to the newest version regardless; if `min_ver==""`, install only if no version is installed. Return `True` if the package was installed.

pylablib.core.utils.net module

A wrapper for built-in TCP/IP routines.

`exception` `pylablib.core.utils.net.`**`SocketError`**
: Bases: `OSError`

  Base socket error class.

  `args`

  `characters_written`

  `errno`
  : POSIX exception code

  `filename`
  : exception filename

  `filename2`
  : second exception filename

  `strerror`
  : exception strerror

  `with_traceback``()`
  : Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

`exception` `pylablib.core.utils.net.`**`SocketTimeout`**
: Bases: *`pylablib.core.utils.net.SocketError`*

  Socket timeout error.

  `args`

  `characters_written`

  `errno`
  : POSIX exception code

  `filename`
  : exception filename

  `filename2`
  : second exception filename

strerror
exception strerror

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

pylablib.core.utils.net.**get_local_addr**()
Get local IP address

pylablib.core.utils.net.**get_all_local_addr**()
Get a list of all local IP addresses

pylablib.core.utils.net.**get_local_hostname**(*full=True*)
Get a local host name

pylablib.core.utils.net.**get_all_remote_addr**(*hostname*)
Get a list of all remote addresses of a remote host by name

pylablib.core.utils.net.**get_remote_hostname**(*addr*, *error_on_missing=False*)
Get a remote host name by its address

pylablib.core.utils.net.**as_addr_port**(*addr*, *port*)
Parse the given address and port combination.

addr can be a host address, a tuple `(addr, port)`, or a string `"addr:port"`; in the first case the given *port* is used, while in the other two it is ignore. Return tuple `(addr, port)`.

class pylablib.core.utils.net.**ClientSocket**(*sock=None*, *timeout=None*, *wait_callback=None*, *send_method='decllen'*, *recv_method='decllen'*, *datatype='auto'*, *nodelay=False*)

Bases: `object`

A client socket (used to connect to a server socket).

Parameters

- **sock** (*socket.socket*) – Socket to wrap; if `None` create a new one.
- **timeout** (*float*) – The timeout used for connecting and sending/receiving (`None` means no timeout).
- **wait_callback** (*callable*) – Called periodically (every 100ms by default) while waiting for connecting or sending/receiving.
- **send_method** (*str*) – Default sending method.
- **recv_method** (*str*) – Default receiving method.
- **datatype** (*str*) – Type of the returned data; can be `"bytes"` (return *bytes* object), `"str"` (return *str* object), or `"auto"` (default Python result: *str* in Python 2 and *bytes* in Python 3)
- **nodelay** (*bool*) – Whether to enable `TCP_NODELAY`.

Possible sending/receiving methods are:

- `'fixedlen'`: data is sent as is, and receiving requires to know the length of the message;
- `'decllen'`: data is prepended by a length, and receiving reads this length and doesn't need predetermined length info.

sock
Corresponding Python socket.

Type socket.socket

decllen_bo
Byteorder of the prepended length for `'decllen'` sending method. Can be either `'>'` (big-endian, default) or `'<'`.

Type str

decllen_ll
Length of the prepended length for `'decllen'` sending method; default is 4 bytes (corresponding to maximum of 4Gb per single length-prepended message)

Type int

set_wait_callback(*wait_callback=None*)
Set callback function for waiting during connecting or sending/receiving

set_timeout(*timeout=None*)
Set timeout for connecting or sending/receiving

get_timeout()
Get timeout for connecting or sending/receiving

using_timeout(*timeout=None*)
Context manager for usage of a different timeout inside a block

connect(*host*, *port*)
Connect to a remote host

close()
Close the connection

is_connected()
Check if the connection is opened

get_local_name()
Return IP address and port of this socket

get_peer_name()
Return IP address and port of the peer socket

recv_fixedlen(*l*)
Receive fixed-length message of length *l*

recv_delimiter(*delim*, *lmax=None*, *chunk_l=1024*, *strict=False*)
Receive a single message ending with a delimiter *delim* (can be several characters, or list several possible delimiter strings).

lmax specifies the maximal received length (*None* means no limit). *chunk_l* specifies the size of data chunk to be read in one try. If `strict==False`, keep receiving as much data as possible until a delimiter is found in the end (only works properly if a single line is expected); otherwise, receive the data byte-by-byte and stop as soon as a delimiter is found (equivalent to setting `chunk_l=1`).

recv_decllen()
Receive variable-length message (prepended by its length).

Length format is described by *decllen_bo* and *decllen_ll* attributes.

recv(*l=None*)
Receive a message using the default method.

recv_all(*chunk_l=1024*)
Receive all of the data currently in the socket.

chunk_l specifies the size of data chunk to be read in one try. For technical reasons, use 1ms timeout (i.e., this operation takes 1ms).

recv_ack (*l=None*)
Receive a message using the default method and send an acknowledgement (message length)

send_fixedlen (*msg*)
Send a message as is

send_decllen (*msg*)
Send a message as a variable-length (prepending its length in the sent message).

Length format is described by *decllen_bo* and *decllen_ll* attributes.

send_delimiter (*msg*, *delimiter*)
Send a message with a delimiter *delim* (can be several characters)

send (*msg*)
Send a message using the default method.

send_ack (*msg*)
Send a message using default method and wait for acknowledgement (message length).

If the acknowledgement message length doesn't agree, raise `SocketError`.

pylablib.core.utils.net.**recv_JSON** (*sock*, *chunk_l=1024*, *strict=True*)
Receive a complete JSON token from the socket.

chunk_l specifies the size of data chunk to be read in one try. If `strict==False`, keep receiving as much data as possible until the received data forms a complete JSON token. otherwise, receive the data byte-by-byte and stop as soon as a token is formed (equivalent to setting `chunk_l=1`).

pylablib.core.utils.net.**listen** (*host*, *port*, *conn_func*, *port_func=None*, *wait_callback=None*, *timeout=None*, *backlog=10*, *wrap_socket=True*, *connections_number=None*, *socket_kwargs=None*)
Run a server socket at the given host and port.

Parameters

- **host** (*str*) – Server host address. If `None`, use the local host defined by `socket.gethostname()`.
- **port** (*int*) – Server port. If `0`, generate an arbitrary free port.
- **conn_func** (*callable*) – Called with the client socket as a single argument every time a connection is established.
- **port_func** (*callable*) – Called with the port as a single argument when the listening starts (useful with `port=0`).
- **wait_callback** (*callable*) – A callback function which is called periodically (every 100ms by default) while awaiting for connections.
- **timeout** (*float*) – Timeout for waiting for the connections (`None` is no timeout).
- **backlog** (*int*) – Backlog length for the socket (see `socket.socket.listen()`).
- **wrap_socket** (*bool*) – If `True`, wrap the client socket of the connection into `ClientSocket` class; otherwise, return `socket.socket` object.
- **connections_number** (*int*) – Specifies maximal number of connections before the listening function returns (by default, the number is unlimited).
- **socket_kwargs** (*dict*) – additional keyword arguments passed to `ClientSocket` constructor.

Checking for connections is paused until *conn_func* returns. If multiple simultaneous connections are expected, *conn_func* should spawn a separate processing thread and return. If `connections_number is None` (i.e., there's no limit on the number of connections before closing), this function never returns.

pylablib.core.utils.numerical module

Numerical functions that don't deal with sequences.

`pylablib.core.utils.numerical.`**`gcd`**(**numbers*)
Euclid's algorithm for GCD. Arguments are cast to integer

`pylablib.core.utils.numerical.`**`integer_distance`**(*x*)
Get distance to the closes integer

`pylablib.core.utils.numerical.`**`gcd_approx`**(*a*, *b*, *min_fraction=1e-08*, *tolerance=1e-05*)
Approximate Euclid's algorithm for possible non-integer values.

Try to find a number *d* such that `a/d` and `b/d` are less than *tolerance* away from a closest integer. If GCD becomes less than `min_fraction * min(a, b)`, raise `ArithmeticError`.

`pylablib.core.utils.numerical.`**`round_significant`**(*x*, *n*)
Rounds *x* to *n* significant digits (not the same as *n* decimal places!).

`pylablib.core.utils.numerical.`**`limit_to_range`**(*x*, *min_val=None*, *max_val=None*, *default=0*)
Confine *x* to the given limit.

Default limit values are `None`, which means no limit. *default* specifies returned value if both *x*, *min_val* and *max_val* are `None`.

class `pylablib.core.utils.numerical.`**`infinite_list`**(*start=0*, *step=1*)
Bases: `object`

Mimics the behavior of the usual list, but is infinite and immutable.

Supports accessing elements, slicing (including slices giving infinite lists) and iterating. Iterating over it naturally leads to an infinite loop, so it should only be used either for finite slices or for loops with break condition.

Parameters

- **start** – The first element of the list.
- **step** – List step.

class counter(*lst*)
Bases: `object`

next()

`pylablib.core.utils.numerical.`**`unity`**()
Return a unity function

`pylablib.core.utils.numerical.`**`constant`**(*c*)
Return a function which returns a constant *c*.

c can only be either a scalar, or an array-like object with the shape matching the expected argument.

`pylablib.core.utils.numerical.`**`polynomial`**(*coeffs*)
Return a polynomial function which with coefficients *coeffs*.

Coefficients are list lowest-order first, so that `coeffs[i]` is the coefficient in front of `x**i`.

pylablib.core.utils.observer_pool module

A simple observer pool (notification pool) implementation.

class pylablib.core.utils.observer_pool.**ObserverPool**(*expand_tuple=True*)

Bases: `object`

An observer pool.

Stores notification functions (callbacks), and calls them whenever `notify()` is called. The callbacks can have priority (higher priority ones are called first) and filter (observer is only called if the filter function passes the notification tag).

Parameters **expand_tuple** (*bool*) – if `True` and the notification value is a tuple, treat it as an argument list for the callback functions.

class Observer(*filt*, *callback*, *priority*, *attr*, *cacheable*)

Bases: `tuple`

attr

cacheable

callback

count()

Return number of occurrences of value.

filt

index()

Return first index of value.

Raises ValueError if the value is not present.

priority

add_observer(*callback*, *name=None*, *filt=None*, *priority=0*, *attr=None*, *cacheable=False*)

Add the observer callback.

Parameters

- **callback** (*callable*) – callback function; takes at least one argument (notification tag), and possible more depending on the notification value.
- **name** (*str*) – stored callback name; by default, a unique name is auto-generated
- **filt** (*callable or None*) – a filter function for this observer (the observer is called only if the `notify()` function tag and value pass the filter); by default, all tags are accepted
- **priority** (*int*) – callback priority; higher priority callback are invoked first.
- **attr** – additional observer attributes (can be used by `ObserverPool` subclasses to change their behavior).
- **cacheable** (*bool*) – if `True`, assumes that the filter function only depends on the tag, so its calls can be cached.

Returns callback name (equal to *name* if supplied; an automatically generated name otherwise).

remove_observer(*name*)

Remove the observer callback with the given name

find_observers(*tag*, *value*)

notify(*tag*, *value=()*)
Notify the observers by calling their callbacks.

Return a dictionary of the callback results. By default the value is an empty tuple: for `expand_tuple==True` this means that only one argument (*tag*) is passed to the callbacks.

pylablib.core.utils.py3 module

Dealing with Python2 / Python3 compatibility.

`pylablib.core.utils.py3.`**as_str**(*data*)
Convert a string into a text string

`pylablib.core.utils.py3.`**as_bytes**(*data*)
Convert a string into bytes

`pylablib.core.utils.py3.`**as_builtin_bytes**(*data*)
Convert a string into bytes

`pylablib.core.utils.py3.`**as_datatype**(*data*, *datatype*)
Convert a string into a given datatypes.

datatype can be `"str"` (text string), `"bytes"` (byte string), or `"auto"` (no conversion).

pylablib.core.utils.rpyc_utils module

Routines and classes related to RPyC package

`pylablib.core.utils.rpyc_utils.`**obtain**(*proxy*, *serv=None*, *deep=False*, *direct=False*)
Obtain a remote netref object by value (i.e., copy it to the local Python instance).

Wrapper around `rpyc.utils.classic.obtain()` with some special cases handling. *serv* specifies the current remote service. If it is of type *SocketTunnelService*, use its socket tunnel for faster transfer. If `deep==True` and `proxy` is a container (tuple, list, or dict), run the function recursively for all its sub-elements. If `direct==True`, directly use RPyC `obtain` method; otherwise use the custom method, which works better with large numpy arrays, but worse with composite types (e.g., lists).

`pylablib.core.utils.rpyc_utils.`**transfer**(*obj*, *serv*)
Send a local object to the remote PC by value (i.e., copy it to the remote Python instance).

A 'reversed' version of *obtain()*.

class `pylablib.core.utils.rpyc_utils.`**SocketTunnelService**(*server=False*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Extension of the standard `rpyc.core.service.SlaveService` with built-in network socket tunnel for faster data transfer.

In order for the tunnel to work, services on both ends need to be subclasses of *SocketTunnelService*. Because of the initial setup protocol, the two services are asymmetric: one should be 'server' (corresponding to the listening server), and one should be 'client' (external connection). The roles are decided by the *server* constructor parameter.

tunnel_send(*obj*, *packer=None*)
Send data through the socket tunnel.

If *packer* is not `None`, it defines a function to convert *obj* to a bytes string.

tunnel_recv(*unpacker=None*)
Receive data sent through the socket tunnel.

If *unpacker* is not `None`, it defines a function to convert the received bytes string into an object.

obtain(*proxy*)
Execute `obtain()` on the local instance

transfer(*obj*)
Execute `transfer()` on the local instance

on_connect(*conn*)

on_disconnect(*conn*)

class pylablib.core.utils.rpyc_utils.**DeviceService**(*verbose=False*)
Bases: `pylablib.core.utils.rpyc_utils.SocketTunnelService`

Device RPyC service.

Expands on `SocketTunnelService` by adding `get_device()` method, which opens local devices, tracks them, and closes them automatically on disconnect.

on_connect(*conn*)

on_disconnect(*conn*)

get_device_class(*cls*)
Get remote device class.

cls is the full class name, including the module within `pylablib.devices` (e.g., `Attocube.ANC300`).

get_device(*cls*, **args*, ***kwargs*)
Connect to a device.

cls is the full class name, including the module within `pylablib.devices` (e.g., `Attocube.ANC300`). Stores reference to the connected device and closes it automatically on disconnect.

obtain(*proxy*)
Execute `obtain()` on the local instance

transfer(*obj*)
Execute `transfer()` on the local instance

tunnel_recv(*unpacker=None*)
Receive data sent through the socket tunnel.

If *unpacker* is not `None`, it defines a function to convert the received bytes string into an object.

tunnel_send(*obj*, *packer=None*)
Send data through the socket tunnel.

If *packer* is not `None`, it defines a function to convert *obj* to a bytes string.

pylablib.core.utils.rpyc_utils.**run_device_service**(*port=18812*, *verbose=False*)
Start `DeviceService` at the given port

pylablib.core.utils.rpyc_utils.**connect_device_service**(*addr*, *port=18812*, *timeout=3*, *attempts=2*, *error_on_fail=True*)
Connect to the `DeviceService` running at the given address and port

timeout and *attempts* define respectively timeout of a single connection attempt, and the number of attempts (RPyC default is 3 seconds timeout and 6 attempts). If `error_on_fail==True`, raise error if the connection failed; otherwise, return `None`

pylablib.core.utils.strdump module

Utils for converting variables into standard python objects (lists, dictionaries, strings, etc.) and back (e.g., for a more predictable LAN transfer). Provides an extension for pickle for more customized classes (numpy arrays, Dictionary).

class pylablib.core.utils.strdump.**StrDumper**
Bases: `object`

Class for dumping and loading an object.

Stores procedures for dumping and loading, i.e., conversion from complex classes (such as *Dictionary*) to simple built-in classes (such as `dict` or `str`).

add_class(*cls*, *dumpf=None*, *loadf=None*, *name=None*, *allow_subclass=True*, *recursive=False*)
Add a rule for dumping/loading an object of class *cls*.

Parameters

- **cls** –
- **dumpf** (*callable*) – Function for dumping an object of the class; `None` means identity function.
- **loadf** (*callable*) – Function for loading an object of the class; `None` means identity function.
- **name** (*str*) – Name of class, which is stored in the packed data (`cls.__name__` by default).
- **allow_subclass** (*bool*) – If `True`, this rule is also used for subclasses of this class.
- **recursive** (*bool*) – If `True`, the functions are given a second argument, which is a dumping/loading function for their sub-elements.

dump(*obj*)
Convert an object into a dumped value

load(*obj*)
Convert a dumped value into an object

loads(*s*)
Convert a pickled string of a damped object into an object

dumps(*obj*)
Dump an object into a pickled string

pylablib.core.utils.strdump.**dumper = <pylablib.core.utils.strdump.StrDumper object>**
Default dumper for converting into standard Python classes and pickling.

Converts `numpy.ndarray` and *Dictionary* objects (these conversion routines are defined when corresponding modules are imported). The converted values include non-printable characters (conversion uses `numpy.load()` and `numpy.ndarray.dump()`), so they can't be saved into text files. However, they're suited for pickling.

pylablib.core.utils.strdump.**dump**(*obj*)
Convert obj into standard Python classes using the default dumper

pylablib.core.utils.strdump.**load**(*s*)
Convert standard Python class representation *s* into an object using the default dumper

pylablib.core.utils.strdump.**dumps**(*obj*)
Convert obj into a pickled string using the default dumper

`pylablib.core.utils.strdump.loads(s)`
Convert a pickled string into an object using the default dumper

pylablib.core.utils.string module

String search, manipulation and conversion routines.

`pylablib.core.utils.string.string_equal(name1, name2, case_sensitive=True, as_prefix=False)`
Determine if *name1* and *name2* are equal with taking special rules (*case_sensitive* and *as_prefix*) into account.

If `as_prefix==True`, strings match even if *name1* is just a prefix of *name2* (not the other wait around).

`pylablib.core.utils.string.find_list_string(name, str_list, case_sensitive=True, as_prefix=False, first_matched=False)`
Find *name* in the string list.

Comparison parameters are defined in `string_equal()`. If `first_matched==True`, stop at the first match; otherwise if multiple occurrences happen, raise `ValueError`.

Returns tuple `(index, value)`.

`pylablib.core.utils.string.find_dict_string(name, str_dict, case_sensitive=True, as_prefix=False)`
Find name in the string dictionary.

Comparison parameters are defined in `string_equal()`. If multiple occurrences happen, raise `ValueError`.

Returns tuple `(key, value)`.

`pylablib.core.utils.string.find_first_entry(line, elements, start=0, not_found_value=-1)`
Find the index of the earliest position inside the *line* of any of the strings in *elements*, starting from *start*.

If none are found, return *not_found_value*.

`pylablib.core.utils.string.find_all_first_locations(line, elements, start=0, not_found_value=-1, known_locations=None)`
Find the indices of the earliest position inside the *line* of all of the strings in *elements*, starting from *start*.

Return dict `{element: pos}`, where `pos` is either position in the string, or *not_found_value* if no entries are present. *known_locations* can specify a dictionary of already known locations of some of the elements. In this case, only missing elements or elements located before *start* will be re-evaluated.

`pylablib.core.utils.string.translate_string_filter(filt, syntax, match_case=True, default=False)`
Turns *filt* into a matching function.

The matching function takes single `str` argument, returns `bool` value.

***filt* can be**

- `None`: function always returns default,
- `bool`: function always returns this value,
- `str`: pattern, determined by *syntax*,
- anything else: returned as is (assumed to already be a callable).

syntax can be `'re'` (`re`), `'glob'` (`glob`) or `'pred'` (simply matching predicate). *match_case* determines whether the filter cares about the string case when matching.

class pylablib.core.utils.string.**StringFilter**(*include=None*, *exclude=None*, *syntax='re'*, *match_case=False*)

Bases: object

String filter function.

Matches string if it matches include (matches all strings by default) and doesn't match exclude (matches nothing by default).

Parameters

- **include** – Inclusion filter (translated by *translate_string_filter()* with syntax specified by *syntax*); include all by default.
- **exclude** – Exclusion filter (translated by *translate_string_filter()* with syntax specified by *syntax*); exclude none by default.
- **syntax** – Default syntax for pattern filters. Can be `'re'` (re), `'glob'` (glob) or `'pred'` (simply matching predicate).
- **match_case** (*bool*) – Determines whether filter ignores case when matching.

pylablib.core.utils.string.**get_string_filter**(*include=None*, *exclude=None*, *syntax='re'*, *match_case=False*)

Generate *StringFilter* with the given parameters.

If the first argument is already *StringFilter*, return as is. If it's a tuple, expand as argument list.

pylablib.core.utils.string.**sfglob**(*include=None*, *exclude=None*)

Return string filter based on glob syntax

pylablib.core.utils.string.**sfregex**(*include=None*, *exclude=None*, *match_case=False*)

Return string filter based on re syntax

pylablib.core.utils.string.**filter_string_list**(*l*, *filt*)

Filter string list based on the filter

pylablib.core.utils.string.**escape_string**(*value*, *location='element'*, *escape_convertible=True*, *quote_type='"'*)

Escape string.

Escaping can be partially skipped depending on *location*:

- **`"parameter"`: escape only if it contains hard delimiters (`"\n\t\v\r"`) anywhere** or `_border_escaped` (`"`, `'` or space) on the sides (suited for parameters taking the full string);
- `"entry"`: same as above, plus containing soft delimiters (`,` or space) anywhere (suited for entries of a table);
- `"element"`: always escaped

If `escape_convertible==True`, escape strings which can be misinterpreted as other values, such as `"1"` or `"[]"`; otherwise, escape only strings which contain special characters.

If *quote_type* is not `None`, automatically put the string into the specified quotation marks; if *quote_type* is `None`, all quotation marks are escaped; if it's not `None`, only *quote_type* marks are escaped.

class pylablib.core.utils.string.**TConversionClass**(*label*, *cls*, *rep*, *conv*)

Bases: tuple

cls

conv

count()

Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

label

rep

pylablib.core.utils.string.**add_conversion_class**(*label*, *cls*, *rep*, *conv*)
Add a string conversion class.

Some values (e.g., numpy arrays or named tuples) lose some of their associated information when converted into strings. With this function is possible to define custom conversion rules for such classes.

Parameters

- **label** (*str*) – class label (e.g., "array")
- **cls** – class which is used to determine if the value should use this conversion functions (e.g., np.ndarray)
- **rep** – function which takes a single argument (object of class *cls*) and returns its representations; can return a string or an object which is easier to convert to a string (e.g., a list or a tuple)
- **conv** – function which takes one or several arguments (converted values of the class representation) and returns the corresponding object; if *rep* returns a tuple, treat it as a list of several arguments, which are passed to *conv* separately; otherwise, *conv* gets a single argument which is the result of *rep*

When converting to string, if an object of class *cls* is encountered, it is converted in a string label(str_rep) (e.g., "array([0, 1, 2])"), where str_rep is the result of calling *rep* (if this result is a tuple, avoid double parentheses, e.g., if the result is a tuple (1, 2), the string becomes "label(1, 2)" instead of "label((1, 2))"). When converting from string, the values inside the parentheses are passed as arguments to *conv* function to get the resulting value.

pylablib.core.utils.string.**add_namedtuple_class**(*cls*)
Add conversion class for a given named tuple class.

For details, see *add_conversion_class()*.

pylablib.core.utils.string.**to_string**(*value*, *location='element'*, *value_formats=None*, *parenthesis_rules='text'*, *use_classes=False*)
Convert value to string with an option of modifying format string.

Parameters

- **value** –
- **location** (*str*) – Used for converting strings (see *escape_string()*).
- **value_formats** (*dict*) – dictionary {value_type: fmt}, where value type can be int, float or complex and *fmt* is a format string used to represent value of this type (e.g., "5.3f"); default formats are {float:".12E", complex:".12E", int:"d"}.
- **parenthesis_rules** (*str*) – determine how to deal with single-element tuples and complex numbers can be "text" (single-element tuples are represented with simple parentheses, e.g., "(1)"; complex number are represented without parentheses, e.g., "1+2j") or "python" (single-element tuples are represented with a comma in the end, e.g., "(1,)"; complex number are represented with parentheses, e.g., "(1+2j)")

- **use_classes** (*bool*) – if True, use additional representation classes for special objects (e.g., numpy arrays will be represented as "array([1, 2, 3])" instead of just "[1, 2, 3]"). This improves conversion fidelity, but makes result harder to parse (e.g., by external string parsers). See *add_conversion_class()* for more explanation.

pylablib.core.utils.string.**is_convertible**(*value*)
Check if the value can be converted to a string using standard *to_string()* function.

pylablib.core.utils.string.**extract_escaped_string**(*line*, *start=0*)
Extract escaped string in quotation marks from the *line*, starting from *start*.

line[start] should be a quotation mark (' or ") or r or b followed by a quotation mark (for raw or binary strings).

Returns tuple (end position, un-escaped string).

pylablib.core.utils.string.**unescape_string**(*value*)
Un-escape string.

Only attempt if the string starts a quotation mark " or '. Otherwise (including strings like 'r""' or 'b""'), return the string as is. Raise an error if the string starts with a quotation mark, but does not correspond to a proper escaped string (e.g., '"abc or '"abc"def).

pylablib.core.utils.string.**to_range**(*range_tuple*)

pylablib.core.utils.string.**from_string**(*value*, *case_sensitive=True*, *parenthesis_rules='text'*, *use_classes=True*)
Parse a string.

Recognizes integers, floats, complex numbers (with i or j for complex part), strings (in quotation marks), dicts, sets, list and tuples, booleans and None. If item is unrecognizable, assumed to be a string.

Parameters

- **case_sensitive** (*bool*) – applied when compared to None, True or False.
- **parenthesis_rules** (*str*) – determines how to deal with empty entries (e.g., [1, ,3]) and complex number representation ("1+2j" vs. "(1+2j)"):
  - 'text': any empty entries are translated into empty_string (i.e., [,] -> [empty_string, empty_string]), except for completely empty structures ([] or ()); complex numbers are represented without parentheses, so that "(1+2j)" will be interpreted as a single-element tuple (1+2j,).
  - 'python': empty entries in the middle are not allowed; empty entries at the end are ignored (i.e., [2,] -> [2]) (single-element tuple can still be expressed in two ways: (e,) or (e)); complex numbers are by default represented with parentheses, so that "(1+2j)" will be interpreted as a complex number, and only (1+2j,), ((1+2j)) or ((1+2j),) as a single-element tuple.
- **use_classes** (*bool*) – if True, use additional representation classes for special objects (e.g., "array([1, 2, 3])" will be converted into a numpy array instead of raising an error). See *add_conversion_class()* for more explanation.

pylablib.core.utils.string.**from_string_partial**(*value*, *delimiters=re.compile('\\s*, \\s*|\\s+')*, *case_sensitive=True*, *parenthesis_rules='text'*, *use_classes=True*, *return_string=False*)
Convert the first part of the supplied string (bounded by *delimiters*) into a value.

delimiters is a string or a regexp (default is `"\s*,\s*|\s+"`, i.e., comma or spaces). If `return_string==False`, convert the value string and return tuple `(end_position, converted_value)`; otherwise, return tuple `(end_position, value_string)`.

The rest of the parameters is the same as in `from_string()`.

`pylablib.core.utils.string.`**`from_row_string`**`(`*value*, *delimiters=re.compile('\\s*, \\s*|\\s+')*, *case_sensitive=True*, *parenthesis_rules='text'*, *use_classes=True*, *return_string=False*`)`

Convert the row string into a list of values, separated by delimiters.

If `return_string==False`, return list of converted objects; otherwise, return list of unconverted strings.

The rest of the parameters is the same as in `from_string_partial()`.

pylablib.core.utils.strpack module

Utilities for packing values into bitstrings. Small extension of the struct module.

`pylablib.core.utils.strpack.`**`int2bytes`**`(`*val*, *l*, *bo='>'*`)`

Convert integer into a list of bytes of length *l*.

bo determines byte order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

`pylablib.core.utils.strpack.`**`bytes2int`**`(`*val*, *bo='>'*`)`

Convert a list of bytes into an integer.

bo determines byte order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

`pylablib.core.utils.strpack.`**`int2bits`**`(`*val*, *l*, *bo='>'*`)`

Convert integer into a list of bits of length *l*.

bo determines byte (and bit) order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

`pylablib.core.utils.strpack.`**`bits2int`**`(`*val*, *bo='>'*`)`

Convert a list of bits into an integer.

bo determines byte (and bit) order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

`pylablib.core.utils.strpack.`**`pack_uint`**`(`*val*, *l*, *bo='>'*`)`

Convert an unsigned integer into a bytestring of length *l*.

Return `bytes` object. *bo* determines byte order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

`pylablib.core.utils.strpack.`**`pack_int`**`(`*val*, *l*, *bo='>'*`)`

Convert a signed integer into a bytestring of length *l*.

Return `bytes` object. *bo* determines byte order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

`pylablib.core.utils.strpack.`**`unpack_uint`**`(`*msg*, *bo='>'*`)`

Convert a bytestring into an unsigned integer.

bo determines byte order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

`pylablib.core.utils.strpack.`**`unpack_int`**`(`*msg*, *bo='>'*`)`

Convert a bytestring into an signed integer.

bo determines byte order: `'>'` is big-endian (MSB first), `'<'` is little-endian (LSB first).

`pylablib.core.utils.strpack.`**`unpack_numpy_u12bit`**`(`*buffer*, *byteorder='<'*, *count=-1*`)`

pylablib.core.utils.units module

Routines for conversion of physical units.

pylablib.core.utils.units.**split_units**(*value*)
Split string value with a dimension.

Return tuple `(val, unit)`, where `val` is the float part of the value, and `unit` is the string representing units.

pylablib.core.utils.units.**convert_length_units**(*value*, *value_unit='m'*, *result_unit='m'*, *case_sensitive=True*)
Convert *value* from *value_unit* to *result_unit*.

The possible length units are `'m'`, `'mm'`, `'um'`, `'nm'`, `'pm'`, `'fm'`. If `case_sensitive==True`, matching units is case sensitive.

pylablib.core.utils.units.**convert_time_units**(*value*, *value_unit='s'*, *result_unit='s'*, *case_sensitive=True*)
Convert *value* from *value_unit* to *result_unit*.

The possible time units are `'s'`, `'ms'`, `'us'`, `'ns'`, `'ps'`, `'fs'`, `'as'`. If `case_sensitive==True`, matching units is case sensitive.

pylablib.core.utils.units.**convert_frequency_units**(*value*, *value_unit='Hz'*, *result_unit='Hz'*, *case_sensitive=True*)
Convert *value* from *value_unit* to *result_unit*.

The possible frequency units are `'Hz'`, `'kHz'`, `'MHz'`, `'GHz'`. If `case_sensitive==True`, matching units is case sensitive.

pylablib.core.utils.units.**convert_power_units**(*value*, *value_unit='dBm'*, *result_unit='dBm'*, *case_sensitive=True*, *impedance=50.0*)
Convert *value* from *value_unit* to *result_unit*.

For conversion between voltage and power, assume RMS voltage and the given *impedance*. The possible power units are `'dBm'`, `'dBmV'`, `'dBuV'`, `'W'`, `'mW'`, `'uW'`, `'nW'`, `'mV'`, `'nV'`. If `case_sensitive==True`, matching units is case sensitive.

Module contents

Module contents

pylablib.devices package

Subpackages

pylablib.devices.AWG package

Submodules

pylablib.devices.AWG.generic module

exception pylablib.devices.AWG.generic.**GenericAWGError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic AWG error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.AWG.generic.**GenericAWGBackendError**(*exc*)
Bases: *pylablib.devices.AWG.generic.GenericAWGError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

AWG backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.AWG.generic.**GenericAWG**(*addr*)
Bases: *pylablib.core.devio.SCPI.SCPIDevice*

Generic arbitrary wave generator, based on Agilent 33500.

With slight modifications works for many other AWGs using largely the same syntax.

Error
alias of *GenericAWGError*

ReraiseError
alias of *GenericAWGBackendError*

get_channels_number()
Get the number of channels

get_current_channel()
Get current channel

select_current_channel(*channel*)
Select current default channel

is_output_enabled(*channel=None*)
Check if the output is enabled

enable_output(*enabled=True*, *channel=None*)
Turn the output on or off

get_output_polarity(*channel=None*)
Get output polarity.

Can be either "norm" or "inv".

set_output_polarity(*polarity='norm'*, *channel=None*)
Set output polarity.

Can be either "norm" or "inv".

is_sync_output_enabled(*channel=None*)
Check if SYNC output is enabled

enable_sync_output(*enabled=True*, *channel=None*)
Enable or disable SYNC output

get_load(*channel=None*)
Get the output load

set_load(*load=None*, *channel=None*)
Set the output load (`None` means High-Z)

get_function(*channel=None*)
Get output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

set_function(*func*, *channel=None*)
Set output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

get_amplitude(*channel=None*)
Get output amplitude (i.e., half of the span)

set_amplitude(*amplitude*, *channel=None*)
Set output amplitude (i.e., half of the span)

get_offset(*channel=None*)
Get output offset

set_offset(*offset*, *channel=None*)
Set output offset

get_output_range(*channel=None*)
Get output voltage range.

Return tuple `(vmin, vmax)` with the low and high voltage values (i.e., `offset-amplitude` and `offset+amplitude`).

set_output_range(*rng*, *channel=None*)
Set output voltage range.

If span is less than `1E-4`, automatically switch to DC mode.

get_frequency(*channel=None*)
Get output frequency

set_frequency(*frequency*, *channel=None*)
Set output frequency

get_phase(*channel=None*)
Get output phase (in degrees)

set_phase(*phase*, *channel=None*)
Set output phase (in degrees)

sync_phase()
Synchronize phase between two channels

get_duty_cycle(*channel=None*)
Get output duty cycle (in percent).

Only applies to `"square"` output function.

set_duty_cycle(*dcycle*, *channel=None*)
Set output duty cycle (in percent).

Only applies to `"square"` output function.

get_ramp_symmetry(*channel=None*)
Get output ramp symmetry (in percent).

Only applies to "ramp" output function.

set_ramp_symmetry(*rsymm*, *channel=None*)
Set output ramp symmetry (in percent).

Only applies to "ramp" output function.

get_pulse_width(*channel=None*)
Get output pulse width (in seconds).

Only applies to "pulse" output function.

set_pulse_width(*width*, *channel=None*)
Set output pulse width (in seconds).

Only applies to "pulse" output function.

is_burst_enabled(*channel=None*)
Check if the burst mode is enabled

enable_burst(*enabled=True*, *channel=None*)
Enable burst mode

get_burst_mode(*channel=None*)
Get burst mode.

Can be either "trig" or "gate".

set_burst_mode(*mode*, *channel=None*)
Set burst mode.

Can be either "trig" or "gate".

get_burst_ncycles(*channel=None*)
Get burst mode ncycles.

Infinite corresponds to a large value (>1E37).

set_burst_ncycles(*ncycles=1*, *channel=None*)
Set burst mode ncycles.

Infinite corresponds to None

get_gate_polarity(*channel=None*)
Get burst gate polarity.

Can be either "norm" or "inv".

set_gate_polarity(*polarity='norm'*, *channel=None*)
Set burst gate polarity.

Can be either "norm" or "inv".

get_trigger_source(*channel=None*)
Get trigger source.

Can be either "imm", "ext", or "bus".

set_trigger_source(*src*, *channel=None*)
Set trigger source.

Can be either "imm", "ext", or "bus".

get_trigger_slope(*channel=None*)
Get trigger slope.

Can be either "pos", or "neg".

set_trigger_slope(*slope*, *channel=None*)
Set trigger slope.

Can be either "pos", or "neg".

is_trigger_output_enabled(*channel=None*)
Check if the trigger output is enabled

enable_trigger_output(*enabled=True*, *channel=None*)
Enable trigger output

get_output_trigger_slope(*channel=None*)
Get output trigger slope.

Can be either "pos", or "neg".

BackendError
alias of pylablib.core.devio.comm_backend.DeviceBackendError

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in read(). If read_echo==True, assume that the device first echoes the input and skip it.

close()
Close the backend

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If one_line==True, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is DataFormat description in numpy format (e.g., "<u2").

If include_header==True, the data is assumed to be in a (somewhat) standard SCPI format: b'#', then a single digit s denoting length of the size block, then s digits denoting length of the data (in bytes) followed by the actual data. Otherwise (include_header==False), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be 'raw' (just raw data), 'string' (with trailing and leading spaces stripped), 'int', 'float', 'bool' (interprets 0 or 'off' as False, anything else as True), 'value' (returns tuple (value, unit), where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of "#" symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If include_header==True, return the data with the header; otherwise, return only the content. If flush_term==True, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If new_instrument==True, create a new backend instance. If ignore_error==True, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_output_trigger_slope(*slope*, *channel=None*)
Set output trigger slope.

Can be either `"pos"`, or `"neg"`.

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.

- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

pylablib.devices.AWG.specific module

class `pylablib.devices.AWG.specific.`**`Agilent33500`**(*addr*, *channels_number='auto'*)

Bases: *pylablib.devices.AWG.generic.GenericAWG*

Agilent 33500 AWG.

Parameters **channels_number** – number of channels; if `"auto"`, try to determine automatically (by certain commands causing errors)

BackendError

alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error

alias of *pylablib.devices.AWG.generic.GenericAWGError*

class NoParameterCaller(*device*, *kind*)

Bases: `object`

Class to simplify calling functions without a parameter

ReraiseError

alias of *pylablib.devices.AWG.generic.GenericAWGBackendError*

apply_settings(*settings*)

Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)

Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()

Close the backend

enable_burst(*enabled=True*, *channel=None*)

Enable burst mode

enable_output(*enabled=True*, *channel=None*)

Turn the output on or off

enable_sync_output(*enabled=True*, *channel=None*)

Enable or disable SYNC output

enable_trigger_output(*enabled=True*, *channel=None*)

Enable trigger output

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

get_amplitude(*channel=None*)
Get output amplitude (i.e., half of the span)

static get_arg_type(*arg*)
Autodetect argument type

get_burst_mode(*channel=None*)
Get burst mode.

Can be either `"trig"` or `"gate"`.

get_burst_ncycles(*channel=None*)
Get burst mode ncycles.

Infinite corresponds to a large value (>1E37).

get_channels_number()
Get the number of channels

get_current_channel()
Get current channel

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_duty_cycle(*channel=None*)
Get output duty cycle (in percent).

Only applies to `"square"` output function.

get_esr(*timeout=None*)
Get the device status register (by default, `"*ESR?"` command)

get_frequency(*channel=None*)
Get output frequency

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_function(*channel=None*)
Get output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

get_gate_polarity(*channel=None*)
Get burst gate polarity.

Can be either `"norm"` or `"inv"`.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_load(*channel=None*)
Get the output load

get_offset(*channel=None*)
Get output offset

get_output_polarity(*channel=None*)
Get output polarity.

Can be either `"norm"` or `"inv"`.

get_output_range(*channel=None*)
Get output voltage range.

Return tuple `(vmin, vmax)` with the low and high voltage values (i.e., `offset-amplitude` and `offset+amplitude`).

get_output_trigger_slope(*channel=None*)
Get output trigger slope.

Can be either `"pos"`, or `"neg"`.

get_phase(*channel=None*)
Get output phase (in degrees)

get_pulse_width(*channel=None*)
Get output pulse width (in seconds).

Only applies to `"pulse"` output function.

get_ramp_symmetry(*channel=None*)
Get output ramp symmetry (in percent).

Only applies to `"ramp"` output function.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_trigger_slope(*channel=None*)
Get trigger slope.

Can be either `"pos"`, or `"neg"`.

get_trigger_source(*channel=None*)
Get trigger source.

Can be either `"imm"`, `"ext"`, or `"bus"`.

is_burst_enabled(*channel=None*)
Check if the burst mode is enabled

is_opened()
Check if the device is connected

is_output_enabled(*channel=None*)
Check if the output is enabled

is_sync_output_enabled(*channel=None*)
Check if SYNC output is enabled

is_trigger_output_enabled(*channel=None*)
Check if the trigger output is enabled

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

select_current_channel(*channel*)
Select current default channel

set_amplitude(*amplitude*, *channel=None*)
Set output amplitude (i.e., half of the span)

set_burst_mode(*mode*, *channel=None*)
Set burst mode.

Can be either `"trig"` or `"gate"`.

set_burst_ncycles(*ncycles=1*, *channel=None*)
Set burst mode ncycles.

Infinite corresponds to `None`

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_duty_cycle(*dcycle*, *channel=None*)
Set output duty cycle (in percent).

Only applies to `"square"` output function.

set_frequency(*frequency*, *channel=None*)
Set output frequency

set_function(*func*, *channel=None*)
Set output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

set_gate_polarity(*polarity='norm'*, *channel=None*)
Set burst gate polarity.

Can be either `"norm"` or `"inv"`.

set_load(*load=None*, *channel=None*)
Set the output load (`None` means High-Z)

set_offset(*offset*, *channel=None*)
Set output offset

set_output_polarity(*polarity='norm'*, *channel=None*)
Set output polarity.

Can be either `"norm"` or `"inv"`.

set_output_range(*rng*, *channel=None*)
Set output voltage range.

If span is less than `1E-4`, automatically switch to DC mode.

set_output_trigger_slope(*slope*, *channel=None*)
Set output trigger slope.

Can be either `"pos"`, or `"neg"`.

set_phase(*phase*, *channel=None*)
Set output phase (in degrees)

set_pulse_width(*width*, *channel=None*)
Set output pulse width (in seconds).

Only applies to `"pulse"` output function.

set_ramp_symmetry(*rsymm*, *channel=None*)
Set output ramp symmetry (in percent).

Only applies to `"ramp"` output function.

set_trigger_slope(*slope*, *channel=None*)
Set trigger slope.

Can be either `"pos"`, or `"neg"`.

set_trigger_source(*src*, *channel=None*)
Set trigger source.

Can be either "imm", "ext", or "bus".

sleep(*delay*)
Wait for *delay* seconds

sync_phase()
Synchronize phase between two channels

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either 'sync' (perform `wait_sync()`), 'dev' (perform `wait_dev()`) or 'none' (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with ",".
- **arg_type** (*str*) – Argument type. Can be 'raw' (in which case data is sent raw), 'string', 'int', 'float', 'bool', a format string (such as '{:.3f}') or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., arg_type='{0};{1}' with arg=[1,2] will produce a string '1;2'); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with ",".
- **unit** (*str*) – If not None, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple (false_value, true_value) of two strings to represent bool argument; by default, use ._bool_selector attribute.
- **wait_sync** – if True, append the sync command (specified as ._wait_sync_comm attribute, "*OPC?" by default) after the message and pause the execution command is complete; useful in long set operations, where the

device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).

- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class `pylablib.devices.AWG.specific.`**Agilent33220A**(*addr*)

Bases: *pylablib.devices.AWG.generic.GenericAWG*

Agilent 33220A AWG.

BackendError

alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error

alias of *pylablib.devices.AWG.generic.GenericAWGError*

class NoParameterCaller(*device*, *kind*)

Bases: `object`

Class to simplify calling functions without a parameter

ReraiseError

alias of *pylablib.devices.AWG.generic.GenericAWGBackendError*

apply_settings(*settings*)

Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)

Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()

Close the backend

enable_burst(*enabled=True*, *channel=None*)

Enable burst mode

enable_output(*enabled=True*, *channel=None*)

Turn the output on or off

enable_sync_output(*enabled=True*, *channel=None*)

Enable or disable SYNC output

enable_trigger_output(*enabled=True*, *channel=None*)

Enable trigger output

flush(*one_line=False*)

Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

get_amplitude(*channel=None*)

Get output amplitude (i.e., half of the span)

static get_arg_type(*arg*)

Autodetect argument type

get_burst_mode(*channel=None*)
Get burst mode.

Can be either "trig" or "gate".

get_burst_ncycles(*channel=None*)
Get burst mode ncycles.

Infinite corresponds to a large value (>1E37).

get_channels_number()
Get the number of channels

get_current_channel()
Get current channel

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_duty_cycle(*channel=None*)
Get output duty cycle (in percent).

Only applies to "square" output function.

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_frequency(*channel=None*)
Get output frequency

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_function(*channel=None*)
Get output function.

Can be one of the following: "sine", "square", "ramp", "pulse", "noise", "prbs", "DC", "user", "arb". Not all functions can be available, depending on the particular model of the generator.

get_gate_polarity(*channel=None*)
Get burst gate polarity.

Can be either "norm" or "inv".

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_load(*channel=None*)
Get the output load

get_offset(*channel=None*)
Get output offset

get_output_polarity(*channel=None*)
Get output polarity.

Can be either "norm" or "inv".

get_output_range(*channel=None*)
Get output voltage range.

Return tuple (vmin, vmax) with the low and high voltage values (i.e., offset-amplitude and offset+amplitude).

get_output_trigger_slope(*channel=None*)
Get output trigger slope.

Can be either "pos", or "neg".

get_phase(*channel=None*)
Get output phase (in degrees)

get_pulse_width(*channel=None*)
Get output pulse width (in seconds).

Only applies to "pulse" output function.

get_ramp_symmetry(*channel=None*)
Get output ramp symmetry (in percent).

Only applies to "ramp" output function.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_trigger_slope(*channel=None*)
Get trigger slope.

Can be either "pos", or "neg".

get_trigger_source(*channel=None*)
Get trigger source.

Can be either "imm", "ext", or "bus".

is_burst_enabled(*channel=None*)
Check if the burst mode is enabled

is_opened()
Check if the device is connected

is_output_enabled(*channel=None*)
Check if the output is enabled

is_sync_output_enabled(*channel=None*)
Check if SYNC output is enabled

is_trigger_output_enabled(*channel=None*)
Check if the trigger output is enabled

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

select_current_channel(*channel*)
Select current default channel

set_amplitude(*amplitude*, *channel=None*)
Set output amplitude (i.e., half of the span)

set_burst_mode(*mode*, *channel=None*)
Set burst mode.

Can be either `"trig"` or `"gate"`.

set_burst_ncycles(*ncycles=1*, *channel=None*)
Set burst mode ncycles.

Infinite corresponds to `None`

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_duty_cycle(*dcycle*, *channel=None*)
Set output duty cycle (in percent).

Only applies to "square" output function.

set_frequency(*frequency*, *channel=None*)
Set output frequency

set_function(*func*, *channel=None*)
Set output function.

Can be one of the following: "sine", "square", "ramp", "pulse", "noise", "prbs", "DC", "user", "arb". Not all functions can be available, depending on the particular model of the generator.

set_gate_polarity(*polarity='norm'*, *channel=None*)
Set burst gate polarity.

Can be either "norm" or "inv".

set_load(*load=None*, *channel=None*)
Set the output load (None means High-Z)

set_offset(*offset*, *channel=None*)
Set output offset

set_output_polarity(*polarity='norm'*, *channel=None*)
Set output polarity.

Can be either "norm" or "inv".

set_output_range(*rng*, *channel=None*)
Set output voltage range.

If span is less than 1E-4, automatically switch to DC mode.

set_output_trigger_slope(*slope*, *channel=None*)
Set output trigger slope.

Can be either "pos", or "neg".

set_phase(*phase*, *channel=None*)
Set output phase (in degrees)

set_pulse_width(*width*, *channel=None*)
Set output pulse width (in seconds).

Only applies to "pulse" output function.

set_ramp_symmetry(*rsymm*, *channel=None*)
Set output ramp symmetry (in percent).

Only applies to "ramp" output function.

set_trigger_slope(*slope*, *channel=None*)
Set trigger slope.

Can be either "pos", or "neg".

set_trigger_source(*src*, *channel=None*)
Set trigger source.

Can be either "imm", "ext", or "bus".

sleep(*delay*)
Wait for *delay* seconds

sync_phase()
Synchronize phase between two channels

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive *write()* operations are bundled together with `;` delimiter. The actual write is performed at the *read()*/*ask()* operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform *wait_sync()*), `'dev'` (perform *wait_dev()*) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class pylablib.devices.AWG.specific.**InstekAFG2225**(*addr*)

Bases: *pylablib.devices.AWG.generic.GenericAWG*

Instek AFG2225 AWG.

Compared to 2000/2100 series, has one extra channel and a bit more capabilities (burst trigger, pulse function)

get_offset(*channel=None*)
Get output offset

set_offset(*offset*, *channel=None*)
Set output offset

get_amplitude(*channel=None*)
Get output amplitude (i.e., half of the span)

set_amplitude(*amplitude*, *channel=None*)
Set output amplitude (i.e., half of the span)

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *pylablib.devices.AWG.generic.GenericAWGError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

ReraiseError
alias of *pylablib.devices.AWG.generic.GenericAWGBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()
Close the backend

enable_burst(*enabled=True*, *channel=None*)
Enable burst mode

enable_output(*enabled=True*, *channel=None*)
Turn the output on or off

enable_sync_output(*enabled=True*, *channel=None*)
Enable or disable SYNC output

enable_trigger_output(*enabled=True*, *channel=None*)
Enable trigger output

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_burst_mode(*channel=None*)
Get burst mode.

Can be either `"trig"` or `"gate"`.

get_burst_ncycles(*channel=None*)
Get burst mode ncycles.

Infinite corresponds to a large value (>1E37).

get_channels_number()
Get the number of channels

get_current_channel()
Get current channel

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_duty_cycle(*channel=None*)
Get output duty cycle (in percent).

Only applies to `"square"` output function.

get_esr(*timeout=None*)
Get the device status register (by default, `"*ESR?"` command)

get_frequency(*channel=None*)
Get output frequency

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_function(*channel=None*)
Get output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

get_gate_polarity(*channel=None*)
Get burst gate polarity.

Can be either `"norm"` or `"inv"`.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_load(*channel=None*)
Get the output load

get_output_polarity(*channel=None*)
Get output polarity.

Can be either "norm" or "inv".

get_output_range(*channel=None*)
Get output voltage range.

Return tuple (vmin, vmax) with the low and high voltage values (i.e., offset-amplitude and offset+amplitude).

get_output_trigger_slope(*channel=None*)
Get output trigger slope.

Can be either "pos", or "neg".

get_phase(*channel=None*)
Get output phase (in degrees)

get_pulse_width(*channel=None*)
Get output pulse width (in seconds).

Only applies to "pulse" output function.

get_ramp_symmetry(*channel=None*)
Get output ramp symmetry (in percent).

Only applies to "ramp" output function.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_trigger_slope(*channel=None*)
Get trigger slope.

Can be either "pos", or "neg".

get_trigger_source(*channel=None*)
Get trigger source.

Can be either "imm", "ext", or "bus".

is_burst_enabled(*channel=None*)
Check if the burst mode is enabled

is_opened()
Check if the device is connected

is_output_enabled(*channel=None*)
Check if the output is enabled

is_sync_output_enabled(*channel=None*)
Check if SYNC output is enabled

is_trigger_output_enabled(*channel=None*)
Check if the trigger output is enabled

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data, fmt, include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string', timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False, timeout=None, flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True, ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

select_current_channel(*channel*)
Select current default channel

set_burst_mode(*mode, channel=None*)
Set burst mode.

Can be either `"trig"` or `"gate"`.

set_burst_ncycles(*ncycles=1, channel=None*)
Set burst mode ncycles.

Infinite corresponds to `None`

set_device_variable(*key, value*)
Set the value of a settings parameter

set_duty_cycle(*dcycle, channel=None*)
Set output duty cycle (in percent).

Only applies to `"square"` output function.

set_frequency(*frequency*, *channel=None*)
Set output frequency

set_function(*func*, *channel=None*)
Set output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

set_gate_polarity(*polarity='norm'*, *channel=None*)
Set burst gate polarity.

Can be either `"norm"` or `"inv"`.

set_load(*load=None*, *channel=None*)
Set the output load (`None` means High-Z)

set_output_polarity(*polarity='norm'*, *channel=None*)
Set output polarity.

Can be either `"norm"` or `"inv"`.

set_output_range(*rng*, *channel=None*)
Set output voltage range.

If span is less than `1E-4`, automatically switch to DC mode.

set_output_trigger_slope(*slope*, *channel=None*)
Set output trigger slope.

Can be either `"pos"`, or `"neg"`.

set_phase(*phase*, *channel=None*)
Set output phase (in degrees)

set_pulse_width(*width*, *channel=None*)
Set output pulse width (in seconds).

Only applies to `"pulse"` output function.

set_ramp_symmetry(*rsymm*, *channel=None*)
Set output ramp symmetry (in percent).

Only applies to `"ramp"` output function.

set_trigger_slope(*slope*, *channel=None*)
Set trigger slope.

Can be either `"pos"`, or `"neg"`.

set_trigger_source(*src*, *channel=None*)
Set trigger source.

Can be either `"imm"`, `"ext"`, or `"bus"`.

sleep(*delay*)
Wait for *delay* seconds

sync_phase()
Synchronize phase between two channels

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive *write()* operations are bundled together with ; delimiter. The actual write is performed at the *read()*/*ask()* operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either 'sync' (perform *wait_sync()*), 'dev' (perform *wait_dev()*) or 'none' (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with ",".
- **arg_type** (*str*) – Argument type. Can be 'raw' (in which case data is sent raw), 'string', 'int', 'float', 'bool', a format string (such as '{:.3f}') or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., arg_type='{0};{1}' with arg=[1,2] will produce a string '1;2'); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with ",".
- **unit** (*str*) – If not None, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple (false_value, true_value) of two strings to represent bool argument; by default, use ._bool_selector attribute.
- **wait_sync** – if True, append the sync command (specified as ._wait_sync_comm attribute, "*OPC?" by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if None, use the class default ._default_write_sync attribute (False by default).
- **read_echo** (*bool*) – If True, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if read_echo==True.

class pylablib.devices.AWG.specific.**InstekAFG2000**(*addr*)
Bases: *pylablib.devices.AWG.specific.InstekAFG2225*

Instek AFG2000/2100 series AWG.

Compared to AFG2225, has only one channel and fewer capabilities.

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *pylablib.devices.AWG.generic.GenericAWGError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

ReraiseError
alias of *pylablib.devices.AWG.generic.GenericAWGBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()
Close the backend

enable_burst(*enabled=True*, *channel=None*)
Enable burst mode

enable_output(*enabled=True*, *channel=None*)
Turn the output on or off

enable_sync_output(*enabled=True*, *channel=None*)
Enable or disable SYNC output

enable_trigger_output(*enabled=True*, *channel=None*)
Enable trigger output

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

get_amplitude(*channel=None*)
Get output amplitude (i.e., half of the span)

static get_arg_type(*arg*)
Autodetect argument type

get_burst_mode(*channel=None*)
Get burst mode.

Can be either `"trig"` or `"gate"`.

get_burst_ncycles(*channel=None*)
Get burst mode ncycles.

Infinite corresponds to a large value (>1E37).

get_channels_number()
Get the number of channels

get_current_channel()
Get current channel

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_duty_cycle(*channel=None*)
Get output duty cycle (in percent).

Only applies to `"square"` output function.

get_esr(*timeout=None*)
Get the device status register (by default, `"*ESR?"` command)

get_frequency(*channel=None*)
Get output frequency

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_function(*channel=None*)
Get output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

get_gate_polarity(*channel=None*)
Get burst gate polarity.

Can be either `"norm"` or `"inv"`.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_load(*channel=None*)
Get the output load

get_offset(*channel=None*)
Get output offset

get_output_polarity(*channel=None*)
Get output polarity.

Can be either `"norm"` or `"inv"`.

get_output_range(*channel=None*)
Get output voltage range.

Return tuple `(vmin, vmax)` with the low and high voltage values (i.e., `offset-amplitude` and `offset+amplitude`).

get_output_trigger_slope(*channel=None*)
Get output trigger slope.

Can be either "pos", or "neg".

get_phase(*channel=None*)
Get output phase (in degrees)

get_pulse_width(*channel=None*)
Get output pulse width (in seconds).

Only applies to "pulse" output function.

get_ramp_symmetry(*channel=None*)
Get output ramp symmetry (in percent).

Only applies to "ramp" output function.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_trigger_slope(*channel=None*)
Get trigger slope.

Can be either "pos", or "neg".

get_trigger_source(*channel=None*)
Get trigger source.

Can be either "imm", "ext", or "bus".

is_burst_enabled(*channel=None*)
Check if the burst mode is enabled

is_opened()
Check if the device is connected

is_output_enabled(*channel=None*)
Check if the output is enabled

is_sync_output_enabled(*channel=None*)
Check if SYNC output is enabled

is_trigger_output_enabled(*channel=None*)
Check if the trigger output is enabled

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is DataFormat description in numpy format (e.g., "<u2").

If include_header==True, the data is assumed to be in a (somewhat) standard SCPI format: b'#', then a single digit s denoting length of the size block, then s digits denoting length of the data (in bytes) followed by the actual data. Otherwise (include_header==False), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

select_current_channel(*channel*)
Select current default channel

set_amplitude(*amplitude*, *channel=None*)
Set output amplitude (i.e., half of the span)

set_burst_mode(*mode*, *channel=None*)
Set burst mode.

Can be either `"trig"` or `"gate"`.

set_burst_ncycles(*ncycles=1*, *channel=None*)
Set burst mode ncycles.

Infinite corresponds to `None`

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_duty_cycle(*dcycle*, *channel=None*)
Set output duty cycle (in percent).

Only applies to `"square"` output function.

set_frequency(*frequency*, *channel=None*)
Set output frequency

set_function(*func*, *channel=None*)
Set output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

set_gate_polarity(*polarity='norm'*, *channel=None*)
Set burst gate polarity.

Can be either "norm" or "inv".

set_load(*load=None*, *channel=None*)
Set the output load (None means High-Z)

set_offset(*offset*, *channel=None*)
Set output offset

set_output_polarity(*polarity='norm'*, *channel=None*)
Set output polarity.

Can be either "norm" or "inv".

set_output_range(*rng*, *channel=None*)
Set output voltage range.

If span is less than 1E-4, automatically switch to DC mode.

set_output_trigger_slope(*slope*, *channel=None*)
Set output trigger slope.

Can be either "pos", or "neg".

set_phase(*phase*, *channel=None*)
Set output phase (in degrees)

set_pulse_width(*width*, *channel=None*)
Set output pulse width (in seconds).

Only applies to "pulse" output function.

set_ramp_symmetry(*rsymm*, *channel=None*)
Set output ramp symmetry (in percent).

Only applies to "ramp" output function.

set_trigger_slope(*slope*, *channel=None*)
Set trigger slope.

Can be either "pos", or "neg".

set_trigger_source(*src*, *channel=None*)
Set trigger source.

Can be either "imm", "ext", or "bus".

sleep(*delay*)
Wait for *delay* seconds

sync_phase()
Synchronize phase between two channels

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive *write()* operations are bundled together with **;** delimiter. The actual write is performed at the *read()*/*ask()* operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform *wait_sync()*), `'dev'` (perform *wait_dev()*) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class pylablib.devices.AWG.specific.**RSInstekAFG21000**(*addr*)
Bases: *pylablib.devices.AWG.specific.InstekAFG2000*

RS Instek AFG21000 series AWG.

Compared to Instek AFG2000, it takes care of the amplitude output bug.

get_offset(*channel=None*)
Get output offset

get_amplitude(*channel=None*)
Get output amplitude (i.e., half of the span)

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *pylablib.devices.AWG.generic.GenericAWGError*

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

ReraiseError
alias of *pylablib.devices.AWG.generic.GenericAWGBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If read_echo==True, assume that the device first echoes the input and skip it.

close()
Close the backend

enable_burst(*enabled=True*, *channel=None*)
Enable burst mode

enable_output(*enabled=True*, *channel=None*)
Turn the output on or off

enable_sync_output(*enabled=True*, *channel=None*)
Enable or disable SYNC output

enable_trigger_output(*enabled=True*, *channel=None*)
Enable trigger output

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If one_line==True, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_burst_mode(*channel=None*)
Get burst mode.

Can be either "trig" or "gate".

get_burst_ncycles(*channel=None*)
Get burst mode ncycles.

Infinite corresponds to a large value (>1E37).

get_channels_number()
Get the number of channels

get_current_channel()
Get current channel

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_duty_cycle(*channel=None*)
Get output duty cycle (in percent).

Only applies to "square" output function.

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_frequency(*channel=None*)
Get output frequency

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_function(*channel=None*)
Get output function.

Can be one of the following: "sine", "square", "ramp", "pulse", "noise", "prbs", "DC", "user", "arb". Not all functions can be available, depending on the particular model of the generator.

get_gate_polarity(*channel=None*)
Get burst gate polarity.

Can be either "norm" or "inv".

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_load(*channel=None*)
Get the output load

get_output_polarity(*channel=None*)
Get output polarity.

Can be either "norm" or "inv".

get_output_range(*channel=None*)
Get output voltage range.

Return tuple (vmin, vmax) with the low and high voltage values (i.e., offset-amplitude and offset+amplitude).

get_output_trigger_slope(*channel=None*)
Get output trigger slope.

Can be either "pos", or "neg".

get_phase(*channel=None*)
Get output phase (in degrees)

get_pulse_width(*channel=None*)
Get output pulse width (in seconds).

Only applies to "pulse" output function.

get_ramp_symmetry(*channel=None*)
Get output ramp symmetry (in percent).

Only applies to "ramp" output function.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_trigger_slope(*channel=None*)
Get trigger slope.

Can be either "pos", or "neg".

get_trigger_source(*channel=None*)
Get trigger source.

Can be either "imm", "ext", or "bus".

is_burst_enabled(*channel=None*)
Check if the burst mode is enabled

is_opened()
Check if the device is connected

is_output_enabled(*channel=None*)
Check if the output is enabled

is_sync_output_enabled(*channel=None*)
Check if SYNC output is enabled

is_trigger_output_enabled(*channel=None*)
Check if the trigger output is enabled

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is DataFormat description in numpy format (e.g., "<u2").

If include_header==True, the data is assumed to be in a (somewhat) standard SCPI format: b'#', then a single digit s denoting length of the size block, then s digits denoting length of the data (in bytes) followed by the actual data. Otherwise (include_header==False), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be 'raw' (just raw data), 'string' (with trailing and leading spaces stripped), 'int', 'float', 'bool' (interprets 0 or 'off' as False, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

select_current_channel(*channel*)
Select current default channel

set_amplitude(*amplitude*, *channel=None*)
Set output amplitude (i.e., half of the span)

set_burst_mode(*mode*, *channel=None*)
Set burst mode.

Can be either `"trig"` or `"gate"`.

set_burst_ncycles(*ncycles=1*, *channel=None*)
Set burst mode ncycles.

Infinite corresponds to `None`

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_duty_cycle(*dcycle*, *channel=None*)
Set output duty cycle (in percent).

Only applies to `"square"` output function.

set_frequency(*frequency*, *channel=None*)
Set output frequency

set_function(*func*, *channel=None*)
Set output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

set_gate_polarity(*polarity='norm'*, *channel=None*)
Set burst gate polarity.

Can be either `"norm"` or `"inv"`.

set_load(*load=None*, *channel=None*)
Set the output load (`None` means High-Z)

set_offset (*offset*, *channel=None*)
Set output offset

set_output_polarity (*polarity='norm'*, *channel=None*)
Set output polarity.

Can be either `"norm"` or `"inv"`.

set_output_range (*rng*, *channel=None*)
Set output voltage range.

If span is less than `1E-4`, automatically switch to DC mode.

set_output_trigger_slope (*slope*, *channel=None*)
Set output trigger slope.

Can be either `"pos"`, or `"neg"`.

set_phase (*phase*, *channel=None*)
Set output phase (in degrees)

set_pulse_width (*width*, *channel=None*)
Set output pulse width (in seconds).

Only applies to `"pulse"` output function.

set_ramp_symmetry (*rsymm*, *channel=None*)
Set output ramp symmetry (in percent).

Only applies to `"ramp"` output function.

set_trigger_slope (*slope*, *channel=None*)
Set trigger slope.

Can be either `"pos"`, or `"neg"`.

set_trigger_source (*src*, *channel=None*)
Set trigger source.

Can be either `"imm"`, `"ext"`, or `"bus"`.

sleep (*delay*)
Wait for *delay* seconds

sync_phase ()
Synchronize phase between two channels

unlock ()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer ()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait (*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev ()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class pylablib.devices.AWG.specific.**TektronixAFG1000**(*addr*, *channels_number='auto'*)
Bases: *pylablib.devices.AWG.generic.GenericAWG*

get_pulse_width(*channel=None*)
Get output pulse width (in seconds).

Only applies to `"pulse"` output function.

set_pulse_width(*width*, *channel=None*)
Set output pulse width (in seconds).

Only applies to `"pulse"` output function.

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *pylablib.devices.AWG.generic.GenericAWGError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

RaiseError
alias of *pylablib.devices.AWG.generic.GenericAWGBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If read_echo==True, assume that the device first echoes the input and skip it.

close()
Close the backend

enable_burst(*enabled=True*, *channel=None*)
Enable burst mode

enable_output(*enabled=True*, *channel=None*)
Turn the output on or off

enable_sync_output(*enabled=True*, *channel=None*)
Enable or disable SYNC output

enable_trigger_output(*enabled=True*, *channel=None*)
Enable trigger output

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If one_line==True, read only a single line.

get_amplitude(*channel=None*)
Get output amplitude (i.e., half of the span)

static get_arg_type(*arg*)
Autodetect argument type

get_burst_mode(*channel=None*)
Get burst mode.

Can be either "trig" or "gate".

get_burst_ncycles(*channel=None*)
Get burst mode ncycles.

Infinite corresponds to a large value (>1E37).

get_channels_number()
Get the number of channels

get_current_channel()
Get current channel

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_duty_cycle(*channel=None*)
Get output duty cycle (in percent).

Only applies to "square" output function.

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_frequency(*channel=None*)
Get output frequency

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_function(*channel=None*)
Get output function.

Can be one of the following: "sine", "square", "ramp", "pulse", "noise", "prbs", "DC", "user", "arb". Not all functions can be available, depending on the particular model of the generator.

get_gate_polarity(*channel=None*)
Get burst gate polarity.

Can be either "norm" or "inv".

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_load(*channel=None*)
Get the output load

get_offset(*channel=None*)
Get output offset

get_output_polarity(*channel=None*)
Get output polarity.

Can be either "norm" or "inv".

get_output_range(*channel=None*)
Get output voltage range.

Return tuple (vmin, vmax) with the low and high voltage values (i.e., offset-amplitude and offset+amplitude).

get_output_trigger_slope(*channel=None*)
Get output trigger slope.

Can be either "pos", or "neg".

get_phase(*channel=None*)
Get output phase (in degrees)

get_ramp_symmetry(*channel=None*)
Get output ramp symmetry (in percent).

Only applies to "ramp" output function.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_trigger_slope(*channel=None*)
Get trigger slope.

Can be either `"pos"`, or `"neg"`.

get_trigger_source(*channel=None*)
Get trigger source.

Can be either `"imm"`, `"ext"`, or `"bus"`.

is_burst_enabled(*channel=None*)
Check if the burst mode is enabled

is_opened()
Check if the device is connected

is_output_enabled(*channel=None*)
Check if the output is enabled

is_sync_output_enabled(*channel=None*)
Check if SYNC output is enabled

is_trigger_output_enabled(*channel=None*)
Check if the trigger output is enabled

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

select_current_channel(*channel*)
Select current default channel

set_amplitude(*amplitude*, *channel=None*)
Set output amplitude (i.e., half of the span)

set_burst_mode(*mode*, *channel=None*)
Set burst mode.

Can be either `"trig"` or `"gate"`.

set_burst_ncycles(*ncycles=1*, *channel=None*)
Set burst mode ncycles.

Infinite corresponds to `None`

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_duty_cycle(*dcycle*, *channel=None*)
Set output duty cycle (in percent).

Only applies to `"square"` output function.

set_frequency(*frequency*, *channel=None*)
Set output frequency

set_function(*func*, *channel=None*)
Set output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

set_gate_polarity(*polarity='norm'*, *channel=None*)
Set burst gate polarity.

Can be either `"norm"` or `"inv"`.

set_load(*load=None*, *channel=None*)
Set the output load (`None` means High-Z)

set_offset(*offset*, *channel=None*)
Set output offset

set_output_polarity(*polarity='norm'*, *channel=None*)
Set output polarity.

Can be either `"norm"` or `"inv"`.

set_output_range (*rng*, *channel=None*)
Set output voltage range.

If span is less than `1E-4`, automatically switch to DC mode.

set_output_trigger_slope (*slope*, *channel=None*)
Set output trigger slope.

Can be either `"pos"`, or `"neg"`.

set_phase (*phase*, *channel=None*)
Set output phase (in degrees)

set_ramp_symmetry (*rsymm*, *channel=None*)
Set output ramp symmetry (in percent).

Only applies to `"ramp"` output function.

set_trigger_slope (*slope*, *channel=None*)
Set trigger slope.

Can be either `"pos"`, or `"neg"`.

set_trigger_source (*src*, *channel=None*)
Set trigger source.

Can be either `"imm"`, `"ext"`, or `"bus"`.

sleep (*delay*)
Wait for *delay* seconds

sync_phase ()
Synchronize phase between two channels

unlock ()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer ()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait (*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev ()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync (*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write (*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with ",".
- **arg_type** (*str*) – Argument type. Can be 'raw' (in which case data is sent raw), 'string', 'int', 'float', 'bool', a format string (such as '{:.3f}') or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., arg_type='{0};{1}' with arg=[1,2] will produce a string '1;2'); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with ",".
- **unit** (*str*) – If not None, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple (false_value, true_value) of two strings to represent bool argument; by default, use ._bool_selector attribute.
- **wait_sync** – if True, append the sync command (specified as ._wait_sync_comm attribute, "*OPC?" by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if None, use the class default ._default_write_sync attribute (False by default).
- **read_echo** (*bool*) – If True, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if read_echo==True.

class pylablib.devices.AWG.specific.**RigolDG1000**(*addr*)
Bases: *pylablib.devices.AWG.generic.GenericAWG*

Rigol DG1000 AWG.

sync_phase()
Synchronize phase between two channels

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *pylablib.devices.AWG.generic.GenericAWGError*

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

ReraiseError
alias of *pylablib.devices.AWG.generic.GenericAWGBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If read_echo==True, assume that the device first echoes the input and skip it.

close()
Close the backend

enable_burst(*enabled=True*, *channel=None*)
Enable burst mode

enable_output(*enabled=True*, *channel=None*)
Turn the output on or off

enable_sync_output(*enabled=True*, *channel=None*)
Enable or disable SYNC output

enable_trigger_output(*enabled=True*, *channel=None*)
Enable trigger output

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

get_amplitude(*channel=None*)
Get output amplitude (i.e., half of the span)

static get_arg_type(*arg*)
Autodetect argument type

get_burst_mode(*channel=None*)
Get burst mode.

Can be either `"trig"` or `"gate"`.

get_burst_ncycles(*channel=None*)
Get burst mode ncycles.

Infinite corresponds to a large value (>1E37).

get_channels_number()
Get the number of channels

get_current_channel()
Get current channel

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_duty_cycle(*channel=None*)
Get output duty cycle (in percent).

Only applies to `"square"` output function.

get_esr(*timeout=None*)
Get the device status register (by default, `"*ESR?"` command)

get_frequency(*channel=None*)
Get output frequency

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_function(*channel=None*)
Get output function.

Can be one of the following: `"sine"`, `"square"`, `"ramp"`, `"pulse"`, `"noise"`, `"prbs"`, `"DC"`, `"user"`, `"arb"`. Not all functions can be available, depending on the particular model of the generator.

get_gate_polarity(*channel=None*)
Get burst gate polarity.

Can be either `"norm"` or `"inv"`.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_load(*channel=None*)
Get the output load

get_offset(*channel=None*)
Get output offset

get_output_polarity(*channel=None*)
Get output polarity.

Can be either `"norm"` or `"inv"`.

get_output_range(*channel=None*)
Get output voltage range.

Return tuple `(vmin, vmax)` with the low and high voltage values (i.e., `offset-amplitude` and `offset+amplitude`).

get_output_trigger_slope(*channel=None*)
Get output trigger slope.

Can be either `"pos"`, or `"neg"`.

get_phase(*channel=None*)
Get output phase (in degrees)

get_pulse_width(*channel=None*)
Get output pulse width (in seconds).

Only applies to `"pulse"` output function.

get_ramp_symmetry(*channel=None*)
Get output ramp symmetry (in percent).

Only applies to `"ramp"` output function.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_trigger_slope(*channel=None*)
Get trigger slope.

Can be either `"pos"`, or `"neg"`.

get_trigger_source(*channel=None*)
Get trigger source.

Can be either "imm", "ext", or "bus".

is_burst_enabled(*channel=None*)
Check if the burst mode is enabled

is_opened()
Check if the device is connected

is_output_enabled(*channel=None*)
Check if the output is enabled

is_sync_output_enabled(*channel=None*)
Check if SYNC output is enabled

is_trigger_output_enabled(*channel=None*)
Check if the trigger output is enabled

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., "<u2").

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of "#" symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, "*RST" command)

select_current_channel(*channel*)
Select current default channel

set_amplitude(*amplitude*, *channel=None*)
Set output amplitude (i.e., half of the span)

set_burst_mode(*mode*, *channel=None*)
Set burst mode.

Can be either "trig" or "gate".

set_burst_ncycles(*ncycles=1*, *channel=None*)
Set burst mode ncycles.

Infinite corresponds to None

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_duty_cycle(*dcycle*, *channel=None*)
Set output duty cycle (in percent).

Only applies to "square" output function.

set_frequency(*frequency*, *channel=None*)
Set output frequency

set_function(*func*, *channel=None*)
Set output function.

Can be one of the following: "sine", "square", "ramp", "pulse", "noise", "prbs", "DC", "user", "arb". Not all functions can be available, depending on the particular model of the generator.

set_gate_polarity(*polarity='norm'*, *channel=None*)
Set burst gate polarity.

Can be either "norm" or "inv".

set_load(*load=None*, *channel=None*)
Set the output load (None means High-Z)

set_offset(*offset*, *channel=None*)
Set output offset

set_output_polarity(*polarity='norm'*, *channel=None*)
Set output polarity.

Can be either "norm" or "inv".

set_output_range(*rng*, *channel=None*)
Set output voltage range.

If span is less than 1E-4, automatically switch to DC mode.

set_output_trigger_slope(*slope*, *channel=None*)
Set output trigger slope.

Can be either "pos", or "neg".

set_phase(*phase*, *channel=None*)
Set output phase (in degrees)

set_pulse_width(*width*, *channel=None*)
Set output pulse width (in seconds).

Only applies to `"pulse"` output function.

set_ramp_symmetry(*rsymm*, *channel=None*)
Set output ramp symmetry (in percent).

Only applies to `"ramp"` output function.

set_trigger_slope(*slope*, *channel=None*)
Set trigger slope.

Can be either `"pos"`, or `"neg"`.

set_trigger_source(*src*, *channel=None*)
Set trigger source.

Can be either `"imm"`, `"ext"`, or `"bus"`.

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (`str`) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (`str`) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list

of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.

- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

Module contents

pylablib.devices.Andor package

Submodules

pylablib.devices.Andor.AndorSDK2 module

class `pylablib.devices.Andor.AndorSDK2.`**`LibraryController`**(*lib*)

Bases: *pylablib.devices.utils.load_lib.LibraryController*

`close`(*opid*)

Mark device closing.

Return tuple `(close_result, uninit_result)` with the results of the closing and the shutdown. If library does not need to be shut down yet, set `uninit_result=None`

`open`()

Mark device opening.

Return tuple `(init_result, open_result, opid)` with the results of the initialization and the opening, and the opening ID which should afterwards be used for closing. If library is already initialized, set `init_result=None`

`preinit`()

Pre-initialize the library, if it hasn't been done already

`shutdown`()

Close all opened connections and shutdown the library

`temp_open`()

Context for temporarily opening a new device connection

`pylablib.devices.Andor.AndorSDK2.`**`restart_lib`**()

`pylablib.devices.Andor.AndorSDK2.`**`get_cameras_number`**()

Get number of connected Andor cameras

class `pylablib.devices.Andor.AndorSDK2.`**`TDeviceInfo`**(*controller_model*, *head_model*, *serial_number*)

Bases: `tuple`

controller_model

count()
Return number of occurrences of value.

head_model

index()
Return first index of value.

Raises ValueError if the value is not present.

serial_number

class pylablib.devices.Andor.AndorSDK2.**TCycleTimings**(*exposure*, *accum_cycle_time*, *kinetic_cycle_time*)
Bases: tuple

accum_cycle_time

count()
Return number of occurrences of value.

exposure

index()
Return first index of value.

Raises ValueError if the value is not present.

kinetic_cycle_time

class pylablib.devices.Andor.AndorSDK2.**TAcqProgress**(*frames_done*, *cycles_done*)
Bases: tuple

count()
Return number of occurrences of value.

cycles_done

frames_done

index()
Return first index of value.

Raises ValueError if the value is not present.

class pylablib.devices.Andor.AndorSDK2.**AndorSDK2Camera**(*idx=0*, *ini_path=''*, *temperature=None*, *fan_mode='off'*, *amp_mode=None*)
Bases: *pylablib.devices.interface.camera.IBinROICamera*, *pylablib.devices.interface.camera.IExposureCamera*

Andor SDK2 camera.

Due to the library features, the camera needs to set up all of the parameters to some default values upon connection. Most of these parameters are chosen as reasonable defaults: full ROI, minimal exposure time, closed shutter, internal trigger, fastest recommended verticals shift speed, no EMCCD gain. However, some should be supplied during the connection: temperature setpoint (where appropriate), fan mode, and amplifier mode; while there is still a possibility to have default values of these parameters, they might not be appropriate in some settings, and frequently need to be changed.

Caution: the manufacturer DLL is designed such that if the camera is not closed on the program termination, the allocated resources are never released. If this happens, these resources are blocked until the complete OS restart.

Parameters

- **idx** (*int*) – camera index (use *get_cameras_number()* to get the total number of connected cameras)
- **ini_path** (*str*) – path to .ini file, if required by the camera
- **temperature** – initial temperature setpoint (in C); can also be `None` (select the bottom 20% of the whole range), or `"off"` (turn the cooler off and set the maximal of the whole range)
- **fan_mode** – initial fan mode
- **amp_mode** – initial amplifier mode (a tuple like the one returned by *get_amp_mode()*); can also be `None`, which selects the slowest, smallest gain mode

Error
alias of *pylablib.devices.Andor.base.AndorError*

TimeoutError
alias of *pylablib.devices.Andor.base.AndorTimeoutError*

open()
Open connection to the camera

close()
Close connection to the camera

is_opened()
Check if the device is connected

get_device_info()
Get camera device info.

Return tuple `(controller_mode, head_model, serial_number)`.

get_status()
Get camera status.

Return either `"idle"` (no acquisition), `"acquiring"` (acquisition in progress) or `"temp_cycle"` (temperature cycle in progress).

acquisition_in_progress()
Check if acquisition is in progress

get_capabilities()
Get camera capabilities.

For description of the structure, see Andor SDK manual.

get_pixel_size()
Get camera pixel size (in m)

is_cooler_on()
Check if the cooler is on

set_cooler(*on=True*)
Set the cooler on or off

get_temperature_status()
Get temperature status.

Can return `"off"` (cooler off), `"not_reached"` (cooling in progress), `"not_stabilized"` (reached but not stabilized yet), `"stabilized"` (completely stabilized) or `"drifted"`.

get_temperature()
Get the current camera temperature (in C)

set_temperature(*temperature*, *enable_cooler=True*)
Change the temperature setpoint (in C).

If `enable_cooler==True`, turn the cooler on automatically.

get_temperature_setpoint()
Get the temperature setpoint (in C)

get_temperature_range()
Return the available range of temperatures (in C)

get_all_amp_modes()
Get all available preamp modes.

Each preamp mode is characterized by an AD channel index, amplifier index, channel speed (horizontal scan speed) index and preamp gain index. Return list of tuples `(channel, channel_bitdepth, oamp, oamp_kind, hsspeed, hsspeed_MHz, preamp, preamp_gain)`, where `channel`, `oamp`, `hsspeed` and `preamp` are indices, while `channel_bitdepth`, `oamp_kind`, `hsspeed_MHz` and `preamp_gain` are descriptions.

get_max_vsspeed()
Get maximal recommended vertical scan speed

set_amp_mode(*channel=None*, *oamp=None*, *hsspeed=None*, *preamp=None*)
Setup preamp mode.

Can specify AD channel index, amplifier index, channel speed (horizontal scan speed) index and preamp gain index. `None` (default) means leaving the current value.

get_amp_mode(*full=True*)
Return the current amplifier mode.

If `full==True`, return a full description (e.g., actual preamp gain or channel name); otherwise, return just the essential indices information (enough to set the mode for this camera, but no explanations).

set_vsspeed(*vsspeed*)
Set vertical scan speed index

get_channel()
Get current channel index

get_channel_bitdepth(*channel=None*)
Get channel bit depth corresponding to the given channel index (current by default)

get_oamp()
Get current output amplifier index

get_oamp_desc(*oamp=None*)
Get output amplifier kind corresponding to the given oamp index (current by default)

get_hsspeed()
Get current horizontal speed index

get_hsspeed_frequency(*hsspeed=None*)
Get horizontal scan frequency (in Hz) corresponding to the given hsspeed index (current by default)

get_preamp()
: Get current preamp index

get_preamp_gain(*preamp=None*)
: Get preamp gain corresponding to the given preamp index (current by default)

get_vsspeed()
: Get current vertical speed index

get_vsspeed_period(*vsspeed=None*)
: Get vertical scan period corresponding to the given vsspeed index (current by default)

get_EMCCD_gain()
: Get current EMCCD gain.

  Return tuple `(gain, advanced)`.

set_EMCCD_gain(*gain*, *advanced=None*)
: Set EMCCD gain.

  Gain goes up to 300 if `advanced==False` or higher if `advanced==True` (in this mode the sensor can be permanently damaged by strong light).

init_amp_mode(*mode=None*)
: Initialize the camera channel, frequencies and amp settings to some default mode.

  If `mode` is supplied, use this mode; otherwise, use the slowest, lowest gain mode (the first one returned by *get_all_amp_modes()*). Also set the maximal recommended vertical shift speed and no EMCCD gain.

get_min_shutter_times()
: Get minimal shutter opening and closing times

setup_shutter(*mode*, *ttl_mode=0*, *open_time=None*, *close_time=None*)
: Setup shutter.

  mode can be `"auto"`, `"open"` or `"closed"`, ttl_mode can be 0 (low is open) or 1 (high is open), *open_time* and *close_time* specify opening and closing times (required to calculate the minimal exposure times). By default, these time are minimal allowed times.

get_shutter_parameters()
: Return shutter parameters as a tuple `(mode, ttl_mode, open_time, close_time)`

get_shutter()
: Get shutter state (`"auto"`, `"open"`, or `"closed"`)

set_fan_mode(*mode*)
: Set fan mode.

  Can be `"full"`, `"low"` or `"off"`.

get_fan_mode()
: Return fan mode (`"full"`, `"low"`, or `"off"`)

read_in_aux_port(*port*)
: Get state at a given auxiliary port

set_out_aux_port(*port*, *state*)
: Set state at a given auxiliary port

set_trigger_mode(*mode*)
: Set trigger mode.

Can be "int" (internal), "ext" (external), "ext_start" (external start), "ext_exp" (external exposure), "ext_fvb_em" (external FVB EM), "software" (software trigger) or "ext_charge_shift" (external charge shifting).

For description, see Andor SDK manual.

get_trigger_mode()
Return trigger mode

get_trigger_level_limits()
Get limits on the trigger level

setup_ext_trigger(*level=None*, *invert=None*, *term_highZ=None*)
Setup external trigger (level, inversion, and high-Z termination).

Any None values are not changed. If any returned values are None, it means that this option is not supported.

get_ext_trigger_parameters()
Return external trigger parameters (level, inversion, high-Z termination).

If any returned values are None, it means that this option is not supported.

send_software_trigger()
Send software trigger signal

set_acquisition_mode(*mode*, *setup_params=True*)
Set the acquisition mode.

Can be "single", "accum", "kinetic", "fast_kinetic" or "cont" (continuous). If setup_params==True, make sure that the last specified parameters for this mode are set up. For description of each mode, see Andor SDK manual and corresponding setup_*_mode functions.

get_acquisition_mode()
Get the current acquisition mode

setup_accum_mode(*num_acc*, *cycle_time_acc=0*)
Switch into the accum acquisition mode and set up its parameters.

num_acc is the number of accumulated frames, *cycle_time_acc* is the acquisition period (by default the minimal possible based on exposure and transfer time).

get_accum_mode_parameters()
Return accum acquisition mode parameters (num_acc, cycle_time_acc)

setup_kinetic_mode(*num_cycle*, *cycle_time=0.0*, *num_acc=1*, *cycle_time_acc=0*, *num_prescan=0*)
Switch into the kinetic acquisition mode and set up its parameters.

num_cycle is the number of kinetic cycles frames, *cycle_time* is the acquisition period between accum frames, *num_accum* is the number of accumulated frames, *cycle_time_acc* is the accum acquisition period, *num_prescan* is the number of prescans.

get_kinetic_mode_parameters()
Return kinetic acquisition mode parameters (num_cycle, cycle_time, num_acc, cycle_time_acc, num_prescan)

setup_fast_kinetic_mode(*num_acc*, *cycle_time_acc=0.0*)
Switch into the fast kinetic acquisition mode and set up its parameters.

num_acc is the number of accumulated frames, *cycle_time_acc* is the acquisition period (by default the minimal possible based on exposure and transfer time).

get_fast_kinetic_mode_parameters()
Return fast kinetic acquisition mode parameters `(num_acc, cycle_time_acc)`

setup_cont_mode(*cycle_time=0*)
Switch into the continuous acquisition mode and set up its parameters.

cycle_time is the acquisition period (by default the minimal possible based on exposure and transfer time).

get_cont_mode_parameters()
Return continuous acquisition mode parameters `cycle_time`

set_exposure(*exposure*)
Set camera exposure

get_exposure()
Get current exposure

set_frame_period(*frame_period*)
Set frame acquisition period for the continuous mode

enable_frame_transfer_mode(*enable=True*)
Enable frame transfer mode.

For description, see Andor SDK manual.

is_frame_transfer_enabled()
Return whether the frame transfer mode is enabled

get_cycle_timings()
Get acquisition timing.

Return tuple `(exposure, accum_cycle_time, kinetic_cycle_time)`. In continuous mode, the relevant cycle time is `kinetic_cycle_time`.

get_frame_timings()
Get acquisition timing.

Return tuple `(exposure, frame_period)`. Frame period is the rate of frame generation, not of internal frame acquisition (e.g., in accumulator or kinetic mode this is the rate of generating a single accumulated frame, which is `num_acc` times larger than the internal frame period).

get_readout_time()
Get frame readout time

get_keepclean_time()
Get sensor keep-clean time

set_read_mode(*mode*)
Set camera read mode.

Can be `"fvb"` (average all image vertically and return it as one row), `"single_track"` (read a single row or several rows averaged together), `"multi_track"` (read multiple rows or averaged sets of rows), `"random_track"` (read several arbitrary lines), or `"image"` (read a whole image or its rectangular part).

get_read_mode()
Get the current read mode

setup_single_track_mode(*center=0*, *width=1*)
Switch into the singe-track read mode and set up its parameters.

center and *width* specify selection of the rows to be averaged together.

get_single_track_mode_parameters()
Return singe-track read mode parameters (center, width)

setup_multi_track_mode(*number=1*, *height=1*, *offset=1*)
Switch into the multi-track read mode and set up its parameters.

number is the number of rows (or row sets) to read, *height* is number of one row set (1 for a single row), *offset* is the distance between the row sets. Return a tuple (number, height, offset, top, gap), where top is the offset of the first row from the top, and gap is the gap between the tracks.

get_multi_track_mode_parameters()
Return multi-track read mode parameters (number, height, offset)

setup_random_track_mode(*tracks=None*)
Switch into the random-track read mode and set up its parameters.

tracks is a list of tuples (start, stop) specifying track span (start are inclusive, stop are exclusive, starting from 0). Note that it does not affect the current read mode, which should be set using *set_read_mode()*.

get_random_track_mode_parameters()
Return random-track read mode parameters, i.e., the list of track positions

setup_image_mode(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*, *hbin=1*, *vbin=1*)
Switch into the image read mode and set up its parameters.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start are inclusive, stop are exclusive, starting from 0), *hbin* and *vbin* specify binning. By default, all non-supplied parameters take extreme values.

get_image_mode_parameters()
Return image read mode parameters, (hstart, hend, vstart, vend, hbin, vbin)

get_detector_size()
Get camera detector size (in pixels) as a tuple (width, height)

get_roi()
Get current ROI.

Return tuple (hstart, hend, vstart, vend, hbin, vbin). *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0), *hbin* and *vbin* specify binning.

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*, *hbin=1*, *vbin=1*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0), *hbin* and *vbin* specify binning. By default, all non-supplied parameters take extreme values (0 for start, maximal for end, 1 for binning).

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple (hlim, vlim), where each element is in turn a limit 5-tuple (min, max, pstep, sstep, maxbin) with, correspondingly, minimal and maximal size, position and size step, and the maximal binning. In some cameras, the step and the minimal size depend on the binning, which can be supplied.

setup_acquisition(*mode=None*, *nframes=None*)
Setup acquisition.

Any non-specified acquisition parameters are assumed to be the same as previously set (or default, if not explicitly set before). Return the new acquisition parameters.

clear_acquisition()
Clear acquisition settings

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

get_acquisition_progress()
Get acquisition progress.

Return tuple `(frames_done, acc_done)` with the number of full transferred frames and the number of acquired sub-frames in the current accumulation cycle.

get_buffer_size()
Get the size of the image ring buffer

FrameTransferError
alias of *pylablib.devices.interface.camera.DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

get_frames_status()
Get acquisition and buffer status.

Return tuple (acquired, unread, skipped, size), where acquired is the total number of acquired frames, unread is the number of acquired but not read frames, skipped is the number of skipped (not read and then written over) frames, and buffer_size is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple (first, last) with images range (first inclusive). If no images are available, return None. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if None, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them, while still keeping total returned frames number to *n*). If return_info==True, return tuple (frames, infos), where infos is a list of frame info tuples (camera-dependent); if some frames are missing and missing_frame!="skip", the corresponding frame info is None.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return True.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of with block.

Useful for applying certain settings which can't be changed during the acquisition. If clear==True, clear acquisition in addition to pausing (by default, use the class default specified as _clear_pausing_acquisition attribute). Yields tuple (acq_in_progress, acq_params), which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple (first, last) with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return None. If peek==True, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them). If return_info==True, return tuple (frames, infos), where infos is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and missing_frame!="skip", the corresponding frame info is None.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return None. If peek==True, return the image but not mark it as read. If return_info==True, return tuple (frame, info), where info is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return None. If peek==True, return the image but not mark it as read. If return_info==True, return tuple (frame, info), where info is an info tuples (camera-dependent, see *read_multiple_images()*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array). Note that if the format is set to "array", the frame info format is also automatically set to "array".

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff

readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes) If *include_fields* is not None, it specifies the fields included for non-"tuple" formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"`` (from the last read frame), "lastwait" (wait for the last successful *wait_for_frame()* call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

pylablib.devices.Andor.AndorSDK3 module

class pylablib.devices.Andor.AndorSDK3.**LibraryController**(*lib*)
Bases: *pylablib.devices.utils.load_lib.LibraryController*

close(*opid*)
Mark device closing.

Return tuple (close_result, uninit_result) with the results of the closing and the shutdown. If library does not need to be shut down yet, set uninit_result=None

open()
Mark device opening.

Return tuple (init_result, open_result, opid) with the results of the initialization and the opening, and the opening ID which should afterwards be used for closing. If library is already initialized, set init_result=None

preinit()
Pre-initialize the library, if it hasn't been done already

shutdown()
Close all opened connections and shutdown the library

temp_open()
Context for temporarily opening a new device connection

`pylablib.devices.Andor.AndorSDK3.`**`restart_lib`**`()`

`pylablib.devices.Andor.AndorSDK3.`**`get_cameras_number`**`()`

Get number of connected Andor cameras

`class` `pylablib.devices.Andor.AndorSDK3.`**`AndorSDK3Attribute`**`(`*handle*, *name*, *kind='auto'*`)`

Bases: `object`

Andor SDK3 camera attribute.

Allows to query and set values and get additional information. Usually created automatically by a Andor SDK3 camera instance, but could also be created manually.

Parameters

- **`handle`** – Andor SDK3 camera handle
- **`pid`** – attribute id
- **`kind`** – attribute kind; can be `"float"`, `"int"`, `"str"`, `"bool"`, `"enum"`, or `"comm"` (command); can also be `"auto"` (default), in which case it is obtained from the stored feature table; newer features might be missing, in which case kind needs to be supplied explicitly, or it raises an error

`name`

attribute name

`implemented`

whether attribute is implemented

Type bool

`readable`

whether attribute is readable

Type bool

`writable`

whether attribute is writable

Type bool

`min`

minimal attribute value (if applicable)

Type float or int

`max`

maximal attribute value (if applicable)

Type float or int

`values`

list of possible attribute values (if applicable)

`kind`

attribute kind (autodetected or supplied upon creation)

Type str

`is_command`

whether attribute is a command (same as `kind=="comm"`)

Type bool

update_properties()
Update all attribute properties: implemented, readable, writable, limits

get_value(*enum_str=True*, *not_implemented_error=True*, *default=None*)
Get current value.

If `enum_str==True`, return enum values as strings; otherwise, return as indices. If `not_implemented_error==True` and the feature is not implemented, raise *AndorError*; otherwise, return *default* if it is not implemented.

set_value(*value*, *not_implemented_error=True*)
Set current value.

If `not_implemented_error==True` and the feature is not implemented, raise *AndorError*; otherwise, do nothing.

call_command()
Execute the given command

get_range(*enum_str=True*)
Get allowed range of the given value.

For `"int"` or `"float"` values return tuple `(min, max)` (inclusive); for `"enum"` return list of possible values (if `enum_str==True`, return list of string values, otherwise return list of indices). For all other value kinds return `None`.

update_limits()
Update minimal and maximal attribute limits and return tuple `(min, max)`

truncate_value(*value*)
Limit value to lie within the allowed range

class pylablib.devices.Andor.AndorSDK3.**TDeviceInfo**(*camera_model*, *serial_number*, *firmware_version*, *software_version*)
Bases: `tuple`

camera_model

count()
Return number of occurrences of value.

firmware_version

index()
Return first index of value.

Raises ValueError if the value is not present.

serial_number

software_version

class pylablib.devices.Andor.AndorSDK3.**TMissedFramesStatus**(*skipped*, *overflows*)
Bases: `tuple`

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

overflows

skipped

class pylablib.devices.Andor.AndorSDK3.**TFrameInfo**(*frame_index*, *timestamp_dev*, *size*, *pixeltype*, *stride*)

Bases: tuple

count()
Return number of occurrences of value.

frame_index

index()
Return first index of value.

Raises ValueError if the value is not present.

pixeltype

size

stride

timestamp_dev

class pylablib.devices.Andor.AndorSDK3.**AndorSDK3Camera**(*idx=0*)

Bases: *pylablib.devices.interface.camera.IBinROICamera*, *pylablib.devices.interface.camera.IExposureCamera*, *pylablib.devices.interface.camera.IAttributeCamera*

Andor SDK3 camera.

Parameters **idx** (*int*) – camera index (use *get_cameras_number()* to get the total number of connected cameras)

Error
alias of *pylablib.devices.Andor.base.AndorError*

TimeoutError
alias of *pylablib.devices.Andor.base.AndorTimeoutError*

FrameTransferError
alias of *pylablib.devices.Andor.base.AndorFrameTransferError*

open()
Open connection to the camera

close()
Close connection to the camera

is_opened()
Check if the device is connected

add_attribute(*name*, *kind*)
Add a new attribute which is not currently present in the dictionary.

kind can be "float", "int", "str", "bool", "enum", or "comm" (command).

get_attribute(*name*, *update_properties=False*, *error_on_missing=True*)
Get the camera attribute with the given name.

If update_properties==True, automatically update all attribute properties.

get_attribute_value(*name*, *update_properties=False*, *error_on_missing=True*, *default=None*)
Get value of an attribute with the given name.

If `update_properties==True`, automatically update all attribute properties before settings. If the value doesn't exist or can not be read and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, assume that `error_on_missing==False`.

set_attribute_value(*name*, *value*, *update_properties=True*, *error_on_missing=True*)
Set value of an attribute with the given name.

If the value doesn't exist or can not be written and `error_on_missing==True`, raise error; otherwise, do nothing. If `update_properties==True`, automatically update all attribute properties before settings.

get_all_attribute_values(*update_properties=False*)
Get values of all attributes.

If `update_properties==True`, automatically update all attribute properties before settings.

set_all_attribute_values(*settings*, *update_properties=True*)
Set values of all attribute in the given dictionary.

If `update_properties==True`, automatically update all attribute properties before settings.

call_command(*name*)
Execute the given command

get_device_info()
Get camera info.

Return tuple `(camera_model, serial_number, firmware_version, software_version)`.

get_trigger_mode()
Get trigger mode.

Can be "int" (internal), "ext" (external), "software" (software trigger), `"ext_start"` (external start), or `"ext_exp"` (external exposure).

set_trigger_mode(*mode*)
Set trigger mode.

Can be `"int"` (internal), `"ext"` (external), or `"software"` (software trigger).

get_shutter()
Get current shutter mode

set_shutter(*mode*)
Set trigger mode.

Can be `"open"`, `"closed"`, or `"auto"`.

is_cooler_on()
Check if the cooler is on

set_cooler(*on=True*)
Set the cooler on or off

get_temperature()
Get the current camera temperature

get_temperature_setpoint()
Get current temperature setpoint

set_temperature(*temperature*, *enable_cooler=True*)
Change the temperature setpoint.

If `enable_cooler==True`, turn the cooler on automatically.

get_exposure()
Get current exposure

set_exposure(*exposure*)
Set camera exposure

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

set_frame_period(*frame_period*)

get_frame_timings()
Get acquisition timing.

Return tuple `(exposure, frame_period)`.

is_metadata_enabled()
Check if the metadata enabled

enable_metadata(*enable=True*)
Enable or disable metadata streaming

class BufferManager(*cam*)
Bases: `object`

Buffer manager: stores, constantly reads and re-schedules buffers, keeps track of acquired frames and buffer overflow events

allocate_buffers(*nbuff*, *size*, *queued_buffers=None*)
Allocate and queue buffers.

queued_buffers' specifies number of allocated buffers to keep queued at a given time (by default, all of them)

deallocate_buffers()
Deallocated buffers (flushing should be done manually)

reset()
Reset counter (on frame acquisition)

read(*idx*)
Return the oldest available acquired but not read buffer, and mark it as read

start_loop()
Start buffer scheduling loop

stop_loop()
Stop buffer scheduling loop

wait_for_frame(*idx=None*, *timeout=None*)
Wait for a new frame acquisition

on_overflow()
Process buffer overflow event

new_overflow()

get_status()
Get counter status: tuple `(acquired, total_length)`

setup_acquisition(*mode='sequence'*, *nframes=100*)
Setup acquisition.

mode can be either `"snap"` (single frame or a fixed number of frames) or `"sequence"` (continuous acquisition). *nframes* determines number of frames to acquire in the single mode, or size of the ring buffer in the `"sequence"` mode (by default, 100).

clear_acquisition()
Clear acquisition settings

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:``setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

acquisition_in_progress()
Check if acquisition is in progress

get_missed_frames_status()
Get missed frames status.

Return tuple `(skipped, overflows)` with the number skipped frames (sent from camera to the PC, but not read and overwritten) and number of buffer overflows (events when the frame rate is too for the data transfer, so some unknown number of frames is skipped).

reset_overflows_counter()
Reset buffer overflows counter

set_overflow_behavior(*behavior*)
Choose the camera behavior if buffer overflow is encountered when waiting for a new frame.

Can be `"error"` (raise `AndorFrameTransferError`), `"restart"` (restart the acquisition), or `"ignore"` (ignore the overflow, which will cause the wait to time out).

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend, hbin, vbin)`.

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*, *hbin=1*, *vbin=1*)
Set current ROI.

By default, all non-supplied parameters take extreme values. Binning is the same for both axes.

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning.

Note that the minimal ROI size depends on the current (not just supplied) binning settings. For more accurate results, is it only after setting up the binning.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_all_attributes(*copy=False*)
Return a dictionary of all available attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be “lastread”‘‘ (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of `TFrameInfo` instances describing frame index and frame metadata, which contains timestamp, image size, pixel format, and row stride; if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

pylablib.devices.Andor.atcore_features module

pylablib.devices.Andor.base module

exception pylablib.devices.Andor.base.**AndorError**
Bases: `pylablib.core.devio.base.DeviceError`

Generic Andor error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Andor.base.**AndorTimeoutError**
Bases: `pylablib.devices.Andor.base.AndorError`

Andor timeout error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Andor.base.**AndorFrameTransferError**
Bases: `pylablib.devices.Andor.base.AndorError`

Andor frame transfer error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Andor.base.**AndorNotSupportedError**
Bases: `pylablib.devices.Andor.base.AndorError`

Option not supported error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

Module contents

pylablib.devices.Arcus package

Submodules

pylablib.devices.Arcus.performax module

pylablib.devices.Arcus.performax.**get_usb_device_info**(*devid*)
Get info for the given device index (starting from 0).

Return tuple `(index, serial, model, desc, vid, pid)`.

pylablib.devices.Arcus.performax.**list_usb_performax_devices**()
List all performax devices.

Return list of tuples `(index, serial, model, desc, vid, pid)`, one per device.

class pylablib.devices.Arcus.performax.**GenericPerformaxStage**(*idx=0*)
Bases: *pylablib.devices.interface.stage.IMultiaxisStage*

Generic Arcus Performax translation stage.

Parameters **idx** (*int*) – stage index

Error = <Mock name='mock.ArcusError' id='140318679849040'>

open()
Open the connection to the stage

close()
Close the connection to the stage

is_opened()
Check if the device is connected

get_device_info()
Get the device info

query(*comm*)
Send a query to the stage and return the reply

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_all_axes()
Get the list of all available axes (taking mapping into account)

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

remap_axes(*mapping*, *accept_original=True*)
Rename axes to the new labels.

mapping is the new axes mapping, which can be a list of new axes name (corresponding to the old axes in order returned by `get_all_axes()`), or a dictionary `{alias: original}` of the new axes aliases.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

class pylablib.devices.Arcus.performax.**Performax4EXStage**(*idx=0*, *enable=True*)
Bases: `pylablib.devices.Arcus.performax.GenericPerformaxStage`

Arcus Performax 4EX/4ET translation stage.

Parameters

- **idx** (*int*) – stage index
- **enable** – if `True`, enable all axes on startup

enable_absolute_mode(*enable=True*)
Set absolute motion mode

enable_limit_errors(*enable=True*, *autoclear=True*)
Enable limit errors.

If on, reaching limit switch on an axis puts it into an error state, which immediately stops this an all other axes; any further motion command on this axis will raise an error (it is still possible to restart motion on other axes); the axis motion can only be resumed by calling `clear_limit_error()`. If off, the limited axis still stops, but the other axes are unaffected. If `autoclear==True` and `enable==False`, also clear the current limit errors on all exs.

limit_errors_enabled()
Check if global limit errors are enabled.

If on, reaching limit switch on an axis puts it into an error state, which immediately stops this an all other axes; any further motion command on this axis will raise an error (it is still possible to restart motion on other axes); the axis motion can only be resumed by calling `clear_limit_error()`. If off, the limited axis still stops, but the other axes are unaffected.

is_enabled(*axis='all'*)
Check if the axis output is enabled

enable_axis(*axis='all'*, *enable=True*)
Enable axis output.

If the output is disabled, the steps are generated by the controller, but not sent to the motors.

get_position(*axis='all'*)
Get the current axis pulse position

set_position_reference (*axis*, *position=0*)
Set the current axis pulse position as a reference.

Re-calibrate the pulse position counter so that the current position is set as *position* (0 by default).

get_encoder (*axis='all'*)
Get the current axis encoder value

set_encoder_reference (*axis*, *position=0*)
Set the current axis encoder value as a reference.

Re-calibrate the encoder counter so that the current position is set as *position* (0 by default).

move_to (*axis*, *position*)
Move a given axis to a given position

move_by (*axis*, *steps=1*)
Move a given axis for a given number of steps

jog (*axis*, *direction*)
Jog a given axis in a given direction.

direction can be either `"-"` (negative) or `"+"` (positive). The motion continues until it is explicitly stopped, or until a limit is hit.

stop (*axis='all'*, *immediate=False*)
Stop motion of a given axis.

If `immediate==True` make an abrupt stop; otherwise, slow down gradually.

home (*axis*, *direction*, *home_mode*)
Home the given axis using a given home mode.

direction can be `"+"` or `"-"` The mode can be `"only_home_input"`, `"only_home_input_lowspeed"`, `"only_limit_input"`, `"only_zidx_input"`, or `"home_and_zidx_input"`. For meaning, see Arcus PMX manual.

get_global_speed ()
Get the global speed setting (in Hz); overridden by a non-zero axis speed

get_axis_speed (*axis='all'*)
Get the individual axis speed setting (in Hz); 0 means that the global speed is used

set_global_speed (*speed*)
Set the global speed setting (in Hz); overridden by a non-zero axis speed

set_axis_speed (*axis*, *speed*)
Set the individual axis speed setting (in Hz); 0 means that the global speed is used

get_current_axis_speed (*axis='all'*)
Get the instantaneous speed (in Hz)

get_status_n (*axis='all'*)
Get the axis status as an integer

get_status (*axis='all'*)
Get the axis status as a set of string descriptors

is_moving (*axis='all'*)
Check if a given axis is moving

wait_move (*axis*, *timeout=None*, *period=0.05*)
Wait until motion is done

check_limit_error(*axis='all'*)
Check if the axis hit limit errors.

Return `""` (not errors), `"+"` (positive limit error) or `"-"` (negative limit error).

clear_limit_error(*axis='all'*)
Clear axis limit errors

get_analog_input(*channel*)
Get voltage (in V) at a given input (starting with 1)

get_digital_input(*channel*)
Get value (0 or 1) at a given digital input (1 through 8)

get_digital_input_register()
Get all 8 digital inputs as a single 8-bit integer

get_digital_output(*channel*)
Get value (0 or 1) at a given digital output (1 through 8)

get_digital_output_register()
Get all 8 digital inputs as a single 8-bit integer

set_digital_output(*channel*, *value*)
Set value (0 or 1) at a given digital output (1 through 8)

set_digital_output_register(*value*)
Set all 8 digital inputs as a single 8-bit integer

Error = <Mock name='mock.ArcusError' id='140318679849040'>

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the connection to the stage

get_all_axes()
Get the list of all available axes (taking mapping into account)

get_device_info()
Get the device info

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

open()
Open the connection to the stage

query(*comm*)
Send a query to the stage and return the reply

remap_axes(*mapping*, *accept_original=True*)
Rename axes to the new labels.

mapping is the new axes mapping, which can be a list of new axes name (corresponding to the old axes in order returned by `get_all_axes()`), or a dictionary `{alias: original}` of the new axes aliases.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

class pylablib.devices.Arcus.performax.**Performax2EXStage**(*idx=0*, *enable=True*)
Bases: `pylablib.devices.Arcus.performax.Performax4EXStage`

Arcus Performax 2EX/2ED translation stage.

Parameters

- **idx** (`int`) – stage index
- **enable** – if `True`, enable all axes on startup

Error = <Mock name='mock.ArcusError' id='140318679849040'>

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

check_limit_error(*axis='all'*)
Check if the axis hit limit errors.

Return `""` (not errors), `"+"` (positive limit error) or `"-"` (negative limit error).

clear_limit_error(*axis='all'*)
Clear axis limit errors

close()
Close the connection to the stage

enable_absolute_mode(*enable=True*)
Set absolute motion mode

enable_axis(*axis='all'*, *enable=True*)
Enable axis output.

If the output is disabled, the steps are generated by the controller, but not sent to the motors.

enable_limit_errors(*enable=True*, *autoclear=True*)
Enable limit errors.

If on, reaching limit switch on an axis puts it into an error state, which immediately stops this an all other axes; any further motion command on this axis will raise an error (it is still possible to restart motion on other axes); the axis motion can only be resumed by calling `clear_limit_error()`. If off, the limited axis still stops, but the other axes are unaffected. If `autoclear==True` and `enable==False`, also clear the current limit errors on all exs.

get_all_axes()
Get the list of all available axes (taking mapping into account)

get_analog_input(*channel*)
Get voltage (in V) at a given input (starting with 1)

get_axis_speed(*axis='all'*)
Get the individual axis speed setting (in Hz); 0 means that the global speed is used

get_current_axis_speed(*axis='all'*)
Get the instantaneous speed (in Hz)

get_device_info()
Get the device info

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_digital_input(*channel*)
Get value (0 or 1) at a given digital input (1 through 8)

get_digital_input_register()
Get all 8 digital inputs as a single 8-bit integer

get_digital_output(*channel*)
Get value (0 or 1) at a given digital output (1 through 8)

get_digital_output_register()
Get all 8 digital inputs as a single 8-bit integer

get_encoder(*axis='all'*)
Get the current axis encoder value

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_global_speed()
Get the global speed setting (in Hz); overridden by a non-zero axis speed

get_position(*axis='all'*)
Get the current axis pulse position

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_status(*axis='all'*)
Get the axis status as a set of string descriptors

get_status_n(*axis='all'*)
Get the axis status as an integer

home(*axis*, *direction*, *home_mode*)
Home the given axis using a given home mode.

direction can be `"+"` or `"-"` The mode can be `"only_home_input"`, `"only_home_input_lowspeed"`, `"only_limit_input"`, `"only_zidx_input"`, or `"home_and_zidx_input"`. For meaning, see Arcus PMX manual.

is_enabled(*axis='all'*)
Check if the axis output is enabled

is_moving(*axis='all'*)
Check if a given axis is moving

is_opened()
Check if the device is connected

jog(*axis*, *direction*)
Jog a given axis in a given direction.

direction can be either `"-"` (negative) or `"+"` (positive). The motion continues until it is explicitly stopped, or until a limit is hit.

limit_errors_enabled()
Check if global limit errors are enabled.

If on, reaching limit switch on an axis puts it into an error state, which immediately stops this an all other axes; any further motion command on this axis will raise an error (it is still possible to restart motion on other axes); the axis motion can only be resumed by calling *clear_limit_error()*. If off, the limited axis still stops, but the other axes are unaffected.

move_by(*axis*, *steps=1*)
Move a given axis for a given number of steps

move_to(*axis*, *position*)
Move a given axis to a given position

open()
Open the connection to the stage

query(*comm*)
Send a query to the stage and return the reply

remap_axes(*mapping*, *accept_original=True*)
Rename axes to the new labels.

mapping is the new axes mapping, which can be a list of new axes name (corresponding to the old axes in order returned by `get_all_axes()`), or a dictionary `{alias: original}` of the new axes aliases.

set_axis_speed(*axis*, *speed*)
Set the individual axis speed setting (in Hz); 0 means that the global speed is used

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_digital_output(*channel*, *value*)
Set value (0 or 1) at a given digital output (1 through 8)

set_digital_output_register(*value*)
Set all 8 digital inputs as a single 8-bit integer

set_encoder_reference(*axis*, *position=0*)
Set the current axis encoder value as a reference.

Re-calibrate the encoder counter so that the current position is set as *position* (0 by default).

set_global_speed(*speed*)
Set the global speed setting (in Hz); overridden by a non-zero axis speed

set_position_reference(*axis*, *position=0*)
Set the current axis pulse position as a reference.

Re-calibrate the pulse position counter so that the current position is set as *position* (0 by default).

stop(*axis='all'*, *immediate=False*)
Stop motion of a given axis.

If `immediate==True` make an abrupt stop; otherwise, slow down gradually.

wait_move(*axis*, *timeout=None*, *period=0.05*)
Wait until motion is done

Module contents

pylablib.devices.Arduino package

Submodules

pylablib.devices.Arduino.base module

exception pylablib.devices.Arduino.base.**ArduinoError**
Bases: `pylablib.core.devio.base.DeviceError`

Generic Arduino devices error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Arduino.base.**ArduinoBackendError**(*exc*)
Bases: `pylablib.devices.Arduino.base.ArduinoError`, `pylablib.core.devio.comm_backend.DeviceBackendError`

Generic Arduino backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.Arduino.base.**IArduinoDevice**(*port*, *rate=9600*, *timeout=10.0*, *term_write='n'*, *term_read='n'*, *flush_before_op=True*)

Bases: `pylablib.core.devio.comm_backend.ICommBackendWrapper`

Generic Arduino device.

Parameters

- **port** – serial port name
- **rate** – baud rate
- **timeout** – default communication timeout
- **term_write** – default write terminating character (automatically appended on every sent message)
- **term_read** – default read terminating character (used to determine when the incoming message is received completely)
- **flush_before_op** – if `True` (default), automatically flush input buffer on comm/query

Error
alias of `ArduinoError`

reset_board()
Reset the board by pulsing the DTR and RTS lines

comm(*comm*, *timeout=None*, *flush=False*, *flush_delay=0.02*)
Send a device command.

If *timeout* is not `None`, it specifies a custom timeout for the operation. If `flush==True`, then wait for *flush_delay* seconds after the write and read everything returned by the device.

query(*query*, *timeout=None*, *query_delay=0*, *flush=False*, *flush_delay=0.02*)
Send a device query and return the reply.

If *timeout* is not `None`, it specifies a custom timeout for the reply read operation. If `query_delay>0`, it specifies the delay between write and subsequent read attempt. If `flush==True`, then wait for *flush_delay* seconds after the write and read everything returned by the device.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key, value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.Attocube package

Submodules

pylablib.devices.Attocube.anc300 module

pylablib.devices.Attocube.anc300.**muxaxis**(**args, **kwargs*)

class pylablib.devices.Attocube.anc300.**TDeviceInfo**(*serial, version*)
Bases: tuple

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

serial

version

class pylablib.devices.Attocube.anc300.**ANC300**(*conn*, *backend='auto'*, *pwd='123456'*)

Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*, *pylablib.devices.interface.stage.IMultiaxisStage*

Attocube ANC300 controller.

Parameters

- **conn** – connection parameters; for Ethernet connection is a tuple `(addr, port)`, a string `"addr:port"`, or a string `"addr"` (default port 7240 us assumed)
- **backend** (*str*) – communication backend; by default, try to determine from the communication parameters
- **pwd** (*str*) – connection password for Ethernet connection (default is `"123456"`)

Error

alias of *pylablib.devices.Attocube.base.AttocubeError*

open()

Open the connection to the stage

query(*msg*)

Send a query to the stage and return the reply

update_available_axes()

Update the list of available axes.

Need to call only if the hardware configuration of the ANC module has changed.

get_device_info()

Get the device info of the controller board: `(serial, version)`

get_axis_serial(*axis='all'*)

Get serial number of the controller board

set_mode(*axis='all'*, *mode='stp'*)

Set axis mode.

axis is either an axis index (starting from 1), or `"all"` (all axes). *mode* can be `"gnd"` (ground), `"stp"` (step), `"cap"` (measure capacitance, then ground), `"offs"` (offset only, no stepping), `"stp+"` (offset with added stepping waveform), `"stp-"` (offset with subtracted stepping). Note that not all modes are supported by all modules: ANM150 doesn't support offset voltage (`"offs"`, `"stp+"`, `"stp-"` modes), ANM200 doesn't support stepping (`"stp"`, `"stp+"`, `"stp-"` modes).

get_mode(*axis='all'*)

Get axis mode.

axis is either an axis index (starting from 1), or `"all"` (all axes). See *set_mode()* for the description of the modes.

is_enabled(*axis='all'*)

Check if the axis is enabled

enable_axis(*axis='all'*, *mode='stp'*)

Enable specific axis (set to step mode)

disable_axis(*axis='all'*)

Disable specific axis (set to ground mode)

measure_capacitance (*axis='all'*, *wait=True*)
Measure axis capacitance; finish in the GND mode.

If `wait==True`, wait until the capacitance measurement is finished (takes about a second per axis).

get_voltage (*axis='all'*)
Get axis step amplitude in Volts

set_voltage (*axis*, *voltage*)
Set axis step amplitude in Volts

get_offset (*axis='all'*)
Get axis offset voltage in Volts

set_offset (*axis*, *voltage*)
Set axis offset voltage in Volts

get_output (*axis='all'*)
Get axis current output voltage in Volts

get_frequency (*axis='all'*)
Get axis step frequency in Hz

set_frequency (*axis*, *freq*)
Set axis step frequency in Hz

get_capacitance (*axis='all'*, *measure=False*)
Get capacitance measurement on the axis.

If `measure==True`, re-measure axis capacitance (takes about a second); otherwise, get the last measurement value.

get_voltage_pattern (*axis*, *kind*)
Get axis voltage pattern.

kind be either `"up"` for up pattern or `"down"` for down pattern. The pattern is a numpy array of 256 numbers from 0 to 255 corresponding to the output voltage from 0 to the axis voltage. This pattern is output (repeatedly) for each step. The default is a simple linear ramp.

set_voltage_pattern (*axis*, *kind*, *pattern=None*)
Set axis voltage pattern.

kind be either `"up"` for up pattern or `"down"` for down pattern. The pattern is an array of 256 numbers from 0 to 255 corresponding to the output voltage from 0 to the axis voltage. This pattern is output (repeatedly) for each step. The default is a simple linear ramp, which is set if `pattern is None`.

get_trigger_input (*axis='all'*)
Get trigger input lines for the given axis.

Return tuple `(up, down)` with values for up and down step triggers, which can be either integer with the trigger line number, or `"off"` if the trigger is off.

set_trigger_input (*axis*, *up=None*, *down=None*)
Set trigger input lines for the given axis.

up and *down* are can be integer with the trigger line number, `"off"` if the trigger is off, or `None` (keep the value unchanged).

get_external_input_modes (*axis='all'*)
Get external BNC input modes.

Return tuple `(acin, dcin)` indicating whether AC-IN and DC-IN channels are enabled.

set_external_input_modes (*axis*, *acin=None*, *dcin=None*)
Enable or disable external BNC inputs.

acin and *dcin* are can be boolean indicating if the corresponding input is enabled, or `None` (keep the value unchanged).

get_axis_correction (*axis*)
Get axis correction factor.

The factor is automatically applied when the motion is in the negative direction.

set_axis_correction (*axis*, *factor=1.0*)
Set axis correction factor.

The factor is automatically applied when the motion is in the negative direction.

jog (*axis*, *direction*)
Jog continuously in the given direction (`"+"` or `"-"`).

The motion will continue until another move or stop command is called.

move_by (*axis*, *steps=1*)
Move a given axis for a given number of steps

wait_move (*axis*, *timeout=30.0*)
Wait for a given axis to stop moving.

If the motion is not finished after *timeout* seconds, raise a backend error.

is_moving (*axis*)
Check if a given axis is moving

stop (*axis='all'*)
Stop motion of a given axis

class NoParameterCaller (*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings (*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close ()
Close the backend

get_all_axes ()
Get the list of all available axes (taking mapping into account)

get_device_variable (*key*)
Get the value of a settings, status, or full info parameter

get_full_info (*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status (*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

remap_axes(*mapping*, *accept_original=True*)
Rename axes to the new labels.

mapping is the new axes mapping, which can be a list of new axes name (corresponding to the old axes in order returned by `get_all_axes()`), or a dictionary `{alias: original}` of the new axes aliases.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

pylablib.devices.Attocube.anc350 module

pylablib.devices.Attocube.anc350.**get_usb_devices_number**()
Get the number of controllers connected via USB

class pylablib.devices.Attocube.anc350.**ANC350**(*conn=0*, *timeout=5.0*)
Bases: `pylablib.core.devio.comm_backend.ICommBackendWrapper`, `pylablib.devices.interface.stage.IMultiaxisStage`

Attocube ANC350 controller.

Parameters

- **conn** – connection parameters - index of the Attocube ANC350 in the system (for a single controller leave 0)
- **timeout** (`float`) – default operation timeout

Error
alias of `pylablib.devices.Attocube.base.AttocubeError`

class Telegram(*opcode*, *address*, *index*, *data*, *corr_number*)
Bases: `tuple`

address

corr_number

count()
Return number of occurrences of value.

data

index

opcode

class Reply(*address*, *index*, *reason*, *data*)
Bases: `tuple`

address

count()
Return number of occurrences of value.

data

index

reason

check_tell(*timeout=0.01*)
Check for queued TELL (periodic value update) commands

set_value(*address*, *index*, *value*, *ack=False*)
Set device value at the given address and index.

If `ack==True`, request ACK responds and return its value; otherwise, return immediately after set.

get_value(*address*, *index*, *as_int=True*)
Get device value at the given address and index.

If `as_int==True`, convert the result into a signed integer; otherwise return raw byte string.

enable_updates(*enabled=True*)
Enable or disable periodic TELL updates

get_hardware_id()
Return device HWID (by default -1)

set_hardware_id(*hwid*, *persist=False*)
Set device HWID (can be used to identify different devices).

If `persist==True`, the value persists after power cycling.

is_connected(*axis='all'*)
Check if axis is connected

is_enabled(*axis='all'*)
Check if axis is enabled

enable_axis(*axis='all'*, *enabled=True*)
Enable a specific axis or all axes

disable_axis(*axis='all'*)
Disable a specific axis or all axes

is_moving(*axis='all'*)
Move a given axis for a given number of steps

check_limit(*axis='all'*)
Check if the ent of travel has been reached.

Return `None` if no limits are reached, `"fwd"` if forward limit is reached, `"bwd"` if backward limit is reached, or `"both"` if both are reached together (normally shouldn't happen).

get_status_n(*axis='all'*)
Get numerical status of the axis.

For details, see ANC350 protocol.

status_bits = [(1, 'running'), (2, 'limit'), (256, 'sens_err'), (1024, 'sens_disconn')

get_status(*axis='all'*)
Get device status.

Return list of status strings, which can include `"running"` (axis is moving), `"limit"` (one of the limits is reached), `"sens_err"` (sensor error), `"sens_disconn"` (sensor disconnected), or `"ref_valid"` (reference is valid).

get_target_position(*axis='all'*)
Get the target position for the given axis (the position towards which it is moving)

get_precision(*axis='all'*)
Get the axis precision in m (used for checking if the target is reached)

set_precision(*axis='all'*, *precision=1e-06*)
Set the axis precision in m (used for checking if the target is reached)

is_target_reached(*axis='all'*, *precision=None*)
Check if the target position is reached.

If *precision* is not `None`, it sets final position tolerance (in m).

get_sensor_voltage()
Get position sensor voltage in Volts

set_sensor_voltage(*voltage*)
Set position sensor voltage in Volts

get_voltage(*axis='all'*)
Get axis step voltage in Volts

set_voltage(*axis*, *voltage*)
Set axis step voltage in Volts

get_offset(*axis='all'*)
Get axis offset voltage in Volts

set_offset(*axis*, *voltage*)
Set axis offset voltage in Volts

get_frequency(*axis='all'*)
Get axis step frequency in Hz

set_frequency(*axis*, *freq*)
Set axis step frequency in Hz

get_capacitance(*axis='all'*, *measure=False*, *delay=0.5*)
Get axis capacitance in F.

If `measure==True`, initialize the measurement and get the result after the measurement *delay*. Otherwise, return the last measured value.

get_position(*axis='all'*)
Get axis position (in m)

move_to(*axis*, *position*, *precision=None*)
Move to target position (in m).

If *precision* is not `None`, it sets final position tolerance.

move_by(*axis*, *dist*)
Move along a given axis by a given distance (in m)

move_by_steps(*axis*, *steps=1*)
Move along a given axis by a given number of steps

wait_move(*axis*, *precision=1e-06*, *timeout=10.0*, *period=0.01*)
Wait for a given axis to stop moving or to reach target position.

If the motion is not finished after *timeout* seconds, raise a backend error. Precision sets the final positioning precision (in m).

stop(*axis='all'*)
Stop motion of a given axis

jog(*axis*, *direction*)
Jog a given axis in a given direction.

direction can be either `"-"` (negative) or `"+"` (positive). The motion continues until it is explicitly stopped, or until a limit is hit.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_all_axes()
Get the list of all available axes (taking mapping into account)

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

remap_axes(*mapping*, *accept_original=True*)
Rename axes to the new labels.

mapping is the new axes mapping, which can be a list of new axes name (corresponding to the old axes in order returned by *get_all_axes()*), or a dictionary `{alias: original}` of the new axes aliases.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

pylablib.devices.Attocube.base module

exception pylablib.devices.Attocube.base.**AttocubeError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Attocube error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Attocube.base.**AttocubeBackendError**(*exc*)
Bases: *pylablib.devices.Attocube.base.AttocubeError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Attocube backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

Module contents

pylablib.devices.Conrad package

Submodules

pylablib.devices.Conrad.base module

exception pylablib.devices.Conrad.base.**ConradError**

Bases: *pylablib.core.devio.base.DeviceError*

Generic Conrad devices error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Conrad.base.**ConradBackendError**(*exc*)

Bases: *pylablib.devices.Conrad.base.ConradError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Generic Conrad backend communication error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.Conrad.base.**RelayBoard**(*conn*, *start_addr=1*)

Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*

Conrad relay board controller

Parameters

- **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)
- **start_addr** – address which is assigned to the first board in the chain upon initialization; all following boards increase the address by 1

Error

alias of *ConradError*

open()

Open the connection to the board

class TMessage(*comm*, *addr*, *data*)

Bases: tuple

addr

comm

count()

Return number of occurrences of value.

data

index()

Return first index of value.

Raises ValueError if the value is not present.

query(*comm*, *addr=1*, *data=0*, *multi_result=False*)

Send a query with the given command, address and data.

If `multi_result==False`, read a single reply frame; otherwise, keep reading until reply with the same command as sent is received (used in initialization and broadcast queries).

get_all_relays(*addr=1*)
Get all relay states.

If *addr* is not 0, return dictionary `{relay:value}`, where `relay` is the relay index on the board (between 1 and 8 inclusive). If `addr==0` (broadcast), return dictionary `{addr:board_state}`, where `board_state` is in turn a state dictionary is described above.

set_all_relays(*values*, *addr=1*)
Set all relay states.

values can be a list (listing relay states from lowest to highest), or a dictionary `{relay:value}`, where relays are numbered from 1 to 8. Relays without values are kept unchanged. If `addr==0`, broadcast to all boards

get_relay(*relay*, *addr=1*)
Get the state at a given relay (indexed from 1 to 8 inclusive)

set_relay(*relay*, *enable=True*, *addr=1*)
Get the state at a given relay (indexed from 1 to 8 inclusive)

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.Cryomagnetics package

Submodules

pylablib.devices.Cryomagnetics.base module

exception pylablib.devices.Cryomagnetics.base.**CryomagneticsError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Cryomagnetics devices error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Cryomagnetics.base.**CryomagneticsBackendError**(*exc*)
Bases: *pylablib.devices.Cryomagnetics.base.CryomagneticsError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Generic Cryomagnetics backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.Cryomagnetics.base.**LM500**(*conn*)
Bases: *pylablib.core.devio.SCPI.SCPIDevice*

Cryomagnetics LM500/510 level monitor.

Channels are enumerated from 1. To abort filling or reset a timeout, call *SCPIDevice.reset()* method.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

Error
alias of *CryomagneticsError*

ReraiseError
alias of *CryomagneticsBackendError*

close()
Close connection to the device

get_channel()
Get current measurement channel

select_channel(*channel=1*)
Select the current measurement channel

get_type(*channel=None*)
Get type of a given channel (`"lhe"` or `"ln"`)

get_mode(*channel=None*)
Get measurement mode at the given channel (`None` for the currently selected channel).

Can be either `'sample_hold'`, or `'continuous'`.

set_mode(*mode*, *channel=None*)
Set measurement mode at the given channel (`None` for the current channel).

Can be either `'sample_hold'`, or `'continuous'`.

get_interval(*channel=None*)
Get measurement interval (in seconds) in sample/hold mode at the given channel (`None` for the current channel)

set_interval(*intvl*, *channel=None*)
Set measurement interval (in seconds) in sample/hold mode at the given channel (`None` for the current channel)

start_measurement(*channel=None*)
Initialize measurement on a given channel

wait_for_measurement(*channel=None*, *timeout=None*)
Wait for the measurement on a given channel to finish

get_level(*channel=None*)
Get level reading on a given channel

measure_level(*channel=None*)
Measure the level (perform the measurement and return the result) on a given channel

start_fill(*channel=None*)
Initialize filling at a given channel (`None` for the current channel)

get_fill_status(*channel=None*)
Get filling status at a given channels (`None` for the current channel).

Return either `"off"` (filling is off), `"timeout"` (filling timed out) or a float (time since filling started, in seconds).

get_low_level(*channel=None*)
Get low level (automated refill start) setting on a given channel (`None` for the current channel)

set_low_level(*level*, *channel=None*)
Set low level (automated refill start) setting on a given channel (`None` for the current channel)

get_high_level(*channel=None*)
Get high level (automated refill stop) setting on a given channel (`None` for the current channel)

set_high_level(*level*, *channel=None*)
Set high level (automated refill stop) setting on a given channel (`None` for the current channel)

BackendError
alias of `pylablib.core.devio.comm_backend.DeviceBackendError`

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in read(). If read_echo==True, assume that the device first echoes the input and skip it.

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If one_line==True, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is DataFormat description in numpy format (e.g., "<u2").

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

set_device_variable(*key*, *value*)
Set the value of a settings parameter

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)

Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)

Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

Module contents

pylablib.devices.DCAM package

Submodules

pylablib.devices.DCAM.DCAM module

class `pylablib.devices.DCAM.DCAM.`**LibraryController**(*lib*)

Bases: *pylablib.devices.utils.load_lib.LibraryController*

close(*opid*)

Mark device closing.

Return tuple `(close_result, uninit_result)` with the results of the closing and the shutdown. If library does not need to be shut down yet, set `uninit_result=None`

open()

Mark device opening.

Return tuple `(init_result, open_result, opid)` with the results of the initialization and the opening, and the opening ID which should afterwards be used for closing. If library is already initialized, set `init_result=None`

preinit()
Pre-initialize the library, if it hasn't been done already

shutdown()
Close all opened connections and shutdown the library

temp_open()
Context for temporarily opening a new device connection

pylablib.devices.DCAM.DCAM.**restart_lib**()

pylablib.devices.DCAM.DCAM.**get_cameras_number**()
Get number of connected DCAM cameras

class pylablib.devices.DCAM.DCAM.**DCAMAttribute**(*handle*, *pid*)
Bases: `object`

DCAM camera attribute.

Allows to query and set values and get additional information. Usually created automatically by a DCAM camera instance, but could also be created manually.

Parameters

- **handle** – DCAM camera handle
- **pid** – attribute id

name
attribute name

min
minimal attribute value (if applicable)

Type float

max
maximal attribute value (if applicable)

Type float

step
attribute value step (if applicable)

Type float

unit
attribute units (index value)

Type int

as_text(*value=None*)
Get the given attribute value as text (by default, current value)

update_limits()
Update minimal and maximal attribute limits and return tuple `(min, max)`

get_value(*enum_str=False*)
Get current attribute value.

If `enum_str==True`, try to represent enums as their string values; otherwise, return their integer values (only integers can be used for setting).

set_value (*value*)
Set attribute value

class pylablib.devices.DCAM.DCAM.**TDeviceInfo** (*vendor, model, serial_number, camera_version*)
Bases: tuple

camera_version

count ()
Return number of occurrences of value.

index ()
Return first index of value.

Raises ValueError if the value is not present.

model

serial_number

vendor

class pylablib.devices.DCAM.DCAM.**TFrameInfo** (*frame_index, framestamp, timestamp_us, camerastamp, position, pixeltype*)
Bases: tuple

camerastamp

count ()
Return number of occurrences of value.

frame_index

framestamp

index ()
Return first index of value.

Raises ValueError if the value is not present.

pixeltype

position

timestamp_us

class pylablib.devices.DCAM.DCAM.**DCAMCamera** (*idx=0*)
Bases: *pylablib.devices.interface.camera.IBinROICamera*, *pylablib.devices.interface.camera.IExposureCamera*, *pylablib.devices.interface.camera.IAttributeCamera*

Error = <Mock name='mock.DCAMError' id='140318678549968'>

TimeoutError = <Mock spec='str' id='140318687804240'>

open ()
Open connection to the camera

close ()
Close connection to the camera

is_opened ()
Check if the device is connected

get_device_info()
Get camera model data.

Return tuple (vendor, model, serial_number, camera_version).

get_attribute_value(*name*, *enum_str=False*, *error_on_missing=True*, *default=None*)
Get value of an attribute with the given name.

If the value doesn't exist or can not be read and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, assume that `error_on_missing==False`. If `enum_str==True`, try to represent enums as their string values; otherwise, return their integer values (only integers can be used for setting).

set_attribute_value(*name*, *value*, *error_on_missing=True*)
Set value of an attribute with the given name.

If the value doesn't exist or can not be written and `error_on_missing==True`, raise error; otherwise, do nothing.

get_all_attribute_values(*enum_str=False*)
Get values of all attributes.

If `enum_str==True`, try to represent enums as their string values; otherwise, return their integer values (only integers can be used for setting).

set_all_attribute_values(*settings*)
Set values of all attribute in the given dictionary

set_trigger_mode(*mode*)
Set trigger mode.

Can be `"int"` (internal), `"ext"` (external), or `"software"` (software trigger).

get_trigger_mode()
Get trigger mode.

Can be `"int"` (internal), `"ext"` (external), or `"software"` (software trigger).

get_all_trigger_modes()
Return the list of all available trigger modes

setup_ext_trigger(*invert=False*, *delay=0.0*)
Setup external trigger (inversion and delay)

get_ext_trigger_parameters()
Return external trigger parameters (inversion and delay)

send_software_trigger()
Send software trigger signal

set_exposure(*exposure*)
Set camera exposure

get_exposure()
Set current exposure

set_readout_speed(*speed='fast'*)
Set readout speed (can be `"fast"` or `"slow"`)

get_readout_speed()
Set current readout speed

get_all_readout_speeds()
Return the list of all available readout speeds

get_frame_readout_time()
Set current frame readout time

get_frame_timings()
Get acquisition timing.

Return tuple `(exposure, frame_period)`.

get_defect_correct_mode()
Check if the defect pixel correction mode is on

set_defect_correct_mode(*enabled=True*)
Enable or disable the defect pixel correction mode

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend, hbin, vbin)`.

set_roi(*hstart=0, hend=None, vstart=0, vend=None, hbin=1, vbin=1*)
Set current ROI.

By default, all non-supplied parameters take extreme values. Binning is the same for both axes, so value of *vbin* is ignored (it is left for compatibility).

get_roi_limits(*hbin=1, vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning. In some cameras, the step and the minimal size depend on the binning, which can be supplied.

setup_acquisition(*mode='sequence', nframes=100*)
Setup acquisition.

mode can be either `"snap"` (single frame or a fixed number of frames) or `"sequence"` (continuous acquisition). *nframes* determines number of frames to acquire in the single mode, or size of the ring buffer in the `"sequence"` mode (by default, 100).

clear_acquisition()
Clear acquisition settings

start_acquisition(**args, **kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:``setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

get_status()
Get acquisition status.

Can be `"busy"` (capturing in progress), `"ready"` (ready for capturing), `"stable"` (not prepared for capturing), `"unstable"` (can't be prepared for capturing), or `"error"` (some other error).

acquisition_in_progress()
Check if acquisition is in progress

get_transfer_info()
Get frame transfer info.

Return tuple `(last_buff, frame_count)`, where `last_buff` is the index of the last filled buffer, and `frame_count` is the total number of acquired frames.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of `TFrameInfo` instances describing frame index, framestamp and timestamp, camera stamp, frame location on the sensor, and pixel type; if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

FrameTransferError
alias of `pylablib.devices.interface.camera.DefaultFrameTransferError`

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_all_attributes(*copy=False*)
Return a dictionary of all available attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is "list" or "array".

get_frame_info_format()
Get format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

get_frames_status()
Get acquisition and buffer status.

Return tuple (acquired, unread, skipped, size), where acquired is the total number of acquired frames, unread is the number of acquired but not read frames, skipped is the number of skipped (not read and then written over) frames, and buffer_size is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple (first, last) with images range (first inclusive). If no images are available, return None. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"‘‘ (from the last read frame), "lastwait" (wait for the last successful wait_for_frame() call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

Module contents

pylablib.devices.HighFinesse package

Submodules

pylablib.devices.HighFinesse.wlm module

pylablib.devices.HighFinesse.wlm.**muxchannel**(**args*, ***kwargs*)
Multiplex the function over its channel argument

class pylablib.devices.HighFinesse.wlm.**TDeviceInfo**(*model*, *serial_number*, *revision_number*, *compilation_number*)

Bases: tuple

compilation_number

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

model

revision_number

serial_number

class pylablib.devices.HighFinesse.wlm.**WLM**(*version=None*, *dll_path=None*, *app_path=None*, *autostart=True*)

Bases: *pylablib.core.devio.interface.IDevice*

Generic HighFinesse wavemeter.

Parameters

- **version** (*int*) – wavemeter version; if None, use any available version
- **dll_path** – path to wlmData.dll; if None, use standard locations or search based on the version
- **app_path** – path to the wavemeter server application (looks like wlm_ws.exe or wlm_ws7.exe); if None, try to autodetect, or rely on the server already running
- **autostart** – if True, start measurements automatically (if the wavemeter server app is not running, it will launch with the measurements stopped).

Error = <Mock name='mock.HighFinesseError' id='140318671239952'>

open()
Open the connection to the wavemeter

close()
Close the connection to the wavemeter

is_opened()
Check if the device is connected

get_device_info()
Get the wavemeter info.

Return tuple (model, serial_number, revision_number, compilation_number).

start_measurement()
Start wavemeter measurement

stop_measurement()
Stop wavemeter measurement

is_measurement_running()
Check if the measurement is running

get_channels_number()
Get number of channels in the wavemeter

get_default_channel()
Get the default channel (starting from 1) which is used for querying

set_default_channel(*channel*)
Set the default channel (starting from 1) which is used for querying

get_frequency(*channel=None*, *error_on_invalid=True*)
Get the wavemeter readings (in Hz) on a given channel.

channel is the measurement channel (starting from 1); if None, use the default channel. If error_on_invalid==True, raise an error if the measurement is invalid (e.g., over- or underexposure); otherwise, the method can return "under" if the meter is underexposed or "over" is it is overexposed, "badsig" if there is no calculable signal, "noval" if there are no values acquire yet, or "nosig" if there is no signal.

get_wavelength(*channel=None*, *error_on_invalid=True*)
Get the wavemeter readings (in m, and in vacuum).

channel is the measurement channel (starting from 1); if `None`, use the default channel. If `error_on_invalid==True`, raise an error if the measurement is invalid (e.g., over- or underexposure); otherwise, the method can return `"under"` if the meter is underexposed or `"over"` is it is overexposed, `"badsig"` if there is no calculable signal, or `"nosig"` if there is no signal.

get_exposure_mode(*channel=None*)
Get the exposure mode (`"manual"` or `"auto"`) at the given channel

set_exposure_mode(*mode='auto'*, *channel=None*)
Set the exposure mode (`"manual"` or `"auto"`) at the given channel

get_exposure(*sensor=1*, *channel=None*)
Get the exposure for a given channel and sensor (starting from 1)

set_exposure(*exposure*, *sensor=1*, *channel=None*)
Manually set the exposure for a given channel and sensor (starting from 1)

get_switcher_mode()
Get the switcher mode (`"off"` for manual switching or `"on"` for cycling mode)

set_switcher_mode(*mode='on'*)
Set the switcher mode (`"off"` for manual switching or `"on"` for cycling mode)

get_active_channel()
Get the current active channel

set_active_channel(*channel*, *automode=True*)
Set the current switcher channel.

Only makes sense in the manual (`"off"`) switcher mode. If `automode==True`, switch to this mode automatically.

is_switcher_channel_enabled(*channel*, *automode=True*)
Check whether the switcher channel enabled.

Only works in the cycling (`"on"`) switcher mode. If `automode==True`, switch to this mode automatically.

is_switcher_channel_shown(*channel*, *automode=True*)
Check whether the switcher channel is shown in the wavemeter control application.

Only works in the cycling (`"on"`) switcher mode. If `automode==True`, switch to this mode automatically.

enable_switcher_channel(*channel*, *enable=True*, *show=None*, *automode=True*)
Enable or disable the current switcher channel in the switch mode.

Only works in the cycling (`"on"`) switcher mode. If `automode==True`, switch to this mode automatically.

get_pulse_mode()
Get the current pulse mode.

Can be `"cw"` (CW laser mode), `"int"` (standard single-laser internally triggered mode), `"ext"` (single- or double-laser mode with external TTL trigger), or `"opt"` (double-laser mode with optical triggering).

set_pulse_mode(*mode*)
Set the current pulse mode.

Can be `"cw"` (CW laser mode), `"int"` (standard single-laser internally triggered mode), `"ext"` (single- or double-laser mode with external TTL trigger), or `"opt"` (double-laser mode with optical triggering).

get_precision_mode()
Set the current precision mode (`"fine"`, `"wide"`, or `"grating"`)

set_precision_mode(*mode*)
Set the current precision mode ("fine", "wide", or "grating")

get_measurement_interval()
Set measurement interval (per channel), or `None` if the interval mode is off

set_measurement_interval(*interval=None*)
Set measurement interval (per channel).

`None` means that the interval mode is off.

calibrate(*source_type*, *source_frequency*, *channel=None*)
Initialize the calibration.

source_type is the calibration source type, which can be "hene_633" (HeNe 633nm laser), "hene_1152" (HeNe 1152nm laser), "hene_free" (free-running HeNe laser), "ne1" (Ne lamp), or "other" (other source). *source_frequency* is the exact source frequency (in Hz) sent through the given *channel*.

get_autocalibration_parameters()
Get up the automatic calibration parameters.

Return tuple `(enable, unit, period)`, where `enable` determines if it is enabled, and `unit` and `period` together specify the calibration period. `unit` can be "start" (once on the measurement start; `period` is irrelevant here), "meas" (once every `period` frequency measurements), "min" (once every `period` minutes), "hours", or "days".

setup_autocalibration(*enable=True*, *unit=None*, *period=None*)
Set up the automatic calibration parameters.

enable determines if it is enabled. *unit* and *period* together specify the calibration period. *unit* can be "start" (once on the measurement start; *period* is irrelevant here), "meas" (once every *period* frequency measurements), "min" (once every *period* minutes), "hours", or "days". Any `None` parameters are kept at the present value.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

Module contents

pylablib.devices.IMAQ package

Submodules

pylablib.devices.IMAQ.IMAQ module

pylablib.devices.IMAQ.IMAQ.**list_cameras**()
List all cameras available through IMAQ interface

pylablib.devices.IMAQ.IMAQ.**get_cameras_number**()
Get number of connected IMAQ cameras

class pylablib.devices.IMAQ.IMAQ.**TDeviceInfo**(*serial_number*, *interface*)
Bases: `tuple`

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

interface

serial_number

class pylablib.devices.IMAQ.IMAQ.**IMAQFrameGrabber**(*imaq_name='img0'*, *do_open=True*, ***kwargs*)
Bases: *pylablib.devices.interface.camera.IROICamera*

Generic IMAQ frame grabber interface.

Compared to *IMAQCamera*, has more permissive initialization arguments, which simplifies its use as a base class for expanded cameras.

Parameters

- **imaq_name** – interface name (can be learned by *list_cameras()*; usually, but not always, starts with `"cam"` or `"img"`)
- **do_open** – if `False`, skip the last step of opening the device (should be opened in a subclass)

Error = <Mock name='mock.IMAQError' id='140318668797968'>

TimeoutError = <Mock spec='str' id='140318667689104'>

open()
Open connection to the camera

close()
Close connection to the camera

reset()
Reset connection to the camera

is_opened()
Check if the device is connected

get_grabber_attribute_value(*attr*, *default=None*, *kind='auto'*)
Get value of an attribute with a given name or index.

If *default* is not `None`, return *default* if the attribute is not supported; otherwise, raise an error. *kind* is the attribute kind, and it can be `"uint32"`, `"uint64"`, `"double"`, or `"auto"` (autodetect based on the stored list of attribute kinds).

set_grabber_attribute_value(*attr*, *value*, *kind='int32'*)
Set value of an attribute with a given name or index.

kind is the attribute kind, and it can be `"uint32"`, `"uint64"`, `"double"`, or `"auto"` (autodetect based on the stored list of attribute kinds).

get_all_grabber_attribute_values()
Get a dictionary of all readable attributes.

The attributes types are autodetected, and some of the types of uncommon attributes may be misrepresented.

get_device_info()
Get camera model data.

Return tuple `(serial, interface)` with the board serial number and an the interface type (e.g., `"1430"` for NI PCIe-1430)

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_grabber_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_grabber_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

set_grabber_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_grabber_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

configure_trigger_in(*trig_type*, *trig_line=0*, *trig_pol='high'*, *trig_action='none'*, *timeout=None*, *reset_acquisition=True*)
Configure input trigger.

Parameters

- **trig_type** (*str*) – trigger source type; can be `"ext"`, `"rtsi"`, `"iso_in"`, or `"software"`
- **trig_line** (*int*) – trigger line number
- **trig_pol** (*str*) – trigger polarity; can be `"high"` or `"low"`
- **trig_action** (*str*) – trigger action; can be `"none"` (disable trigger), `"capture"` (start capturing), `"stop"` (stop capturing), `"buffer"` (capture a single frame), or `"bufflist"` (capture the whole buffer list once)
- **timeout** (*float*) – timeout in seconds; `None` means not timeout.
- **reset_acquisition** (*bool*) – if the input triggers configuration has been changed, acquisition needs to be restart; if `True`, perform it automatically

send_software_trigger()
Send software trigger signal

configure_trigger_out(*trig_type*, *trig_line=0*, *trig_pol='high'*, *trig_drive='disable'*)
Configure trigger output.

Parameters

- **trig_type** (*str*) – trigger drive destination type; can be `"ext"`, `"rtsi"`, or `"iso_out"`
- **trig_line** (*int*) – trigger line number
- **trig_pol** (*str*) – trigger polarity; can be `"high"` or `"low"`
- **trig_drive** (*str*) – trigger output signal; can be `"disable"` (disable drive), `"acq_in_progress"` (asserted when acquisition is started), `"acq_done"` (asserted when acquisition is done), `"unasserted"` (force unasserted level), `"asserted"` (force asserted level), `"hsync"` (asserted on start of a single line start), `"vsync"` (asserted on start of a frame scan), `"frame_start"` (asserted when a single frame is captured), or `"frame_done"` (asserted when a single frame is done)

read_trigger(*trig_type*, *trig_line=0*, *trig_pol='high'*)
Read current value of a trigger (input or output).

Parameters

- **trig_type** (*str*) – trigger drive destination type; can be "ext", "rtsi", "iso_in", or "iso_out"
- **trig_line** (*int*) – trigger line number
- **trig_pol** (*str*) – trigger polarity; can be "high" or "low"

clear_all_triggers(*reset_acquisition=True*)
Disable all triggers of the session

If the input triggers configuration has been changed, acquisition needs to be restart; if `reset_acquisition==True`, perform it automatically.

setup_serial_params(*write_term=''*, *datatype='bytes'*)
Setup default serial communication parameters.

Parameters

- **write_term** – default terminator character to be added to the sent messages
- **datatype** – type of the result of read commands; can be "bytes" (return raw bytes), or "str" (convert into UTF-8 string)

get_serial_params()
Return serial parameters as a tuple `(write_term, datatype)`

serial_write(*msg*, *timeout=3.0*, *term=None*)
Write message into CameraLink serial port.

Parameters

- **timeout** – operation timeout (in seconds)
- **term** – additional write terminator character to add to the message; if `None`, use the value set up using `setup_serial_params()` (by default, no additional terminator)

serial_read(*n*, *timeout=3.0*, *datatype=None*)
Read specified number of bytes from CameraLink serial port.

Parameters

- **n** – number of bytes to read
- **timeout** – operation timeout (in seconds)
- **datatype** – return datatype; can be "bytes" (return raw bytes), or "str" (convert into UTF-8 string) if `None`, use the value set up using `setup_serial_params()` (by default, "bytes")

serial_readline(*timeout=3.0*, *datatype=None*, *maxn=1024*)
Read bytes from CameraLink serial port until the termination character (defined in camera file) is encountered.

Parameters

- **timeout** – operation timeout (in seconds)
- **datatype** – return datatype; can be "bytes" (return raw bytes), or "str" (convert into UTF-8 string) if `None`, use the value set up using `setup_serial_params()` (by default, "bytes")

- **maxn** – maximal number of bytes to read

serial_flush()
Flush CameraLink serial port

setup_acquisition(*mode='sequence'*, *nframes=100*)
Setup acquisition mode.

mode can be either `"snap"` (single frame or a fixed number of frames) or `"sequence"` (continuous acquisition). (note that *IMAQCamera.acquisition_in_progress()* would still return `True` in this case, even though new frames are no longer acquired). *nframes* sets up number of frame buffers.

clear_acquisition()
Clear all acquisition details and free all buffers

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

acquisition_in_progress()
Check if acquisition is in progress

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*, *fastbuff=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of `TFrameInfo` single-element tuples containing frame index; if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`. If `fastbuff==False`, return a list of individual frames (2D numpy arrays). Otherwise, return a list of ‘chunks’, which are 3D numpy arrays containing several frames; in this case, if *return_info* is `True`, then `frame_info` will automatically be in an `"array"` format, with the rows corresponding to the frames within the chunks, and the columns corresponding to the frames. Using `fastbuff` results in faster operation at high frame rates (>~1kFPS), at the expense of a more complicated frame processing in the following code.

FrameTransferError
alias of *pylablib.devices.interface.camera.DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by `set_frame_info_format()`) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple (first, last) with images range (first inclusive). If no images are available, return None. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if None, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them, while still keeping total returned frames number to *n*). If return_info==True, return tuple (frames, infos), where infos is a list of frame info tuples (camera-dependent); if some frames are missing and missing_frame!="skip", the corresponding frame info is None.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return True.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of with block.

Useful for applying certain settings which can't be changed during the acquisition. If clear==True, clear acquisition in addition to pausing (by default, use the class default specified as _clear_pausing_acquisition attribute). Yields tuple (acq_in_progress, acq_params), which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return None. If peek==True, return the image but not mark it as read. If return_info==True, return tuple (frame, info), where info is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return None. If peek==True, return the image but not mark it as read. If return_info==True, return tuple (frame, info), where info is an info tuples (camera-dependent, see *read_multiple_images()*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array). Note that if the format is set to "array", the frame info format is also automatically set to "array".

set_frame_info_format (*fmt*, *include_fields=None*)
Set format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes) If *include_fields* is not None, it specifies the fields included for non-"tuple" formats.

set_frame_info_period (*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing (*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

snap (*timeout=5.0*, *return_info=False*)
Snap a single frame

wait_for_frame (*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"`` (from the last read frame), "lastwait" (wait for the last successful *wait_for_frame()* call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

class pylablib.devices.IMAQ.IMAQ.**IMAQCamera** (*name='img0'*)
Bases: *pylablib.devices.IMAQ.IMAQ.IMAQFrameGrabber*

Generic IMAQ camera interface.

Parameters name – interface name (can be learned by *list_cameras()*; usually, but not always, starts with "cam" or "img")

Error = <Mock name='mock.IMAQError' id='140318668797968'>

FrameTransferError
alias of *pylablib.devices.interface.camera.DefaultFrameTransferError*

class NoParameterCaller (*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

TimeoutError = <Mock spec='str' id='140318667689104'>

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

clear_acquisition()
Clear all acquisition details and free all buffers

clear_all_triggers(*reset_acquisition=True*)
Disable all triggers of the session

If the input triggers configuration has been changed, acquisition needs to be restart; if reset_acquisition==True, perform it automatically.

close()
Close connection to the camera

configure_trigger_in(*trig_type*, *trig_line=0*, *trig_pol='high'*, *trig_action='none'*, *timeout=None*, *reset_acquisition=True*)
Configure input trigger.

Parameters

- **trig_type** (*str*) – trigger source type; can be "ext", "rtsi", "iso_in", or "software"
- **trig_line** (*int*) – trigger line number
- **trig_pol** (*str*) – trigger polarity; can be "high" or "low"
- **trig_action** (*str*) – trigger action; can be "none" (disable trigger), "capture" (start capturing), "stop" (stop capturing), "buffer" (capture a single frame), or "bufflist" (capture the whole buffer list once)
- **timeout** (*float*) – timeout in seconds; None means not timeout.
- **reset_acquisition** (*bool*) – if the input triggers configuration has been changed, acquisition needs to be restart; if True, perform it automatically

configure_trigger_out(*trig_type*, *trig_line=0*, *trig_pol='high'*, *trig_drive='disable'*)
Configure trigger output.

Parameters

- **trig_type** (*str*) – trigger drive destination type; can be "ext", "rtsi", or "iso_out"
- **trig_line** (*int*) – trigger line number
- **trig_pol** (*str*) – trigger polarity; can be "high" or "low"
- **trig_drive** (*str*) – trigger output signal; can be "disable" (disable drive), "acq_in_progress" (asserted when acquisition is started), "acq_done" (asserted when acquisition is done), "unasserted" (force unasserted level), "asserted" (force asserted level), "hsync" (asserted on start of a single line start), "vsync" (asserted on start of a frame scan), "frame_start" (asserted when a single frame is captured), or "frame_done" (asserted when a single frame is done)

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_all_grabber_attribute_values()
Get a dictionary of all readable attributes.

The attributes types are autodetected, and some of the types of uncommon attributes may be misrepresented.

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_info()
Get camera model data.

Return tuple `(serial, interface)` with the board serial number and an the interface type (e.g., `"1430"` for NI PCIe-1430)

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by `set_frame_info_format()`) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_grabber_attribute_value(*attr*, *default=None*, *kind='auto'*)
Get value of an attribute with a given name or index.

If *default* is not `None`, return *default* if the attribute is not supported; otherwise, raise an error. *kind* is the attribute kind, and it can be `"uint32"`, `"uint64"`, `"double"`, or `"auto"` (autodetect based on the stored list of attribute kinds).

get_grabber_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_grabber_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_grabber_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_serial_params()
Return serial parameters as a tuple (write_term, datatype)

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if None, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them, while still keeping total returned frames number to *n*). If return_info==True, return tuple (frames, infos), where infos is a list of frame info tuples (camera-dependent); if some frames are missing and missing_frame!="skip", the corresponding frame info is None.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return True.

is_opened()
Check if the device is connected

open()
Open connection to the camera

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of with block.

Useful for applying certain settings which can't be changed during the acquisition. If clear==True, clear acquisition in addition to pausing (by default, use the class default specified as _clear_pausing_acquisition attribute). Yields tuple (acq_in_progress, acq_params), which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*, *fastbuff=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple (first, last) with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return None. If peek==True, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them). If return_info==True, return tuple (frames, infos), where infos is a list of TFrameInfo single-element tuples containing frame index; if some frames are missing and missing_frame!="skip", the corresponding frame info is None. If fastbuff==False, return a list of individual frames (2D numpy arrays). Otherwise, return a list of 'chunks', which are 3D numpy arrays containing several frames; in this case, if *return_info* is True, then frame_info will automatically be in an "array" format, with the rows corresponding to the frames within the chunks, and the columns corresponding to the frames. Using fastbuff results in faster operation at high frame rates (>~1kFPS), at the expense of a more complicated frame processing in the following code.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

`read_oldest_image`(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

`read_trigger`(*trig_type*, *trig_line=0*, *trig_pol='high'*)
Read current value of a trigger (input or output).

Parameters

- **`trig_type`** (*`str`*) – trigger drive destination type; can be `"ext"`, `"rtsi"`, `"iso_in"`, or `"iso_out"`
- **`trig_line`** (*`int`*) – trigger line number
- **`trig_pol`** (*`str`*) – trigger polarity; can be `"high"` or `"low"`

`reset`()
Reset connection to the camera

`send_software_trigger`()
Send software trigger signal

`serial_flush`()
Flush CameraLink serial port

`serial_read`(*n*, *timeout=3.0*, *datatype=None*)
Read specified number of bytes from CameraLink serial port.

Parameters

- **`n`** – number of bytes to read
- **`timeout`** – operation timeout (in seconds)
- **`datatype`** – return datatype; can be `"bytes"` (return raw bytes), or `"str"` (convert into UTF-8 string) if `None`, use the value set up using `setup_serial_params()` (by default, `"bytes"`)

`serial_readline`(*timeout=3.0*, *datatype=None*, *maxn=1024*)
Read bytes from CameraLink serial port until the termination character (defined in camera file) is encountered.

Parameters

- **`timeout`** – operation timeout (in seconds)
- **`datatype`** – return datatype; can be `"bytes"` (return raw bytes), or `"str"` (convert into UTF-8 string) if `None`, use the value set up using `setup_serial_params()` (by default, `"bytes"`)
- **`maxn`** – maximal number of bytes to read

`serial_write`(*msg*, *timeout=3.0*, *term=None*)
Write message into CameraLink serial port.

Parameters

- **`timeout`** – operation timeout (in seconds)

- **term** – additional write terminator character to add to the message; if `None`, use the value set up using `setup_serial_params()` (by default, no additional terminator)

set_device_variable (*key*, *value*)
Set the value of a settings parameter

set_frame_format (*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format (*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period (*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_grabber_attribute_value (*attr*, *value*, *kind='int32'*)
Set value of an attribute with a given name or index.

kind is the attribute kind, and it can be `"uint32"`, `"uint64"`, `"double"`, or `"auto"` (autodetect based on the stored list of attribute kinds).

set_grabber_roi (*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

set_image_indexing (*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

set_roi (*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

setup_acquisition (*mode='sequence'*, *nframes=100*)
Setup acquisition mode.

mode can be either "snap" (single frame or a fixed number of frames) or "sequence" (continuous acquisition). (note that *IMAQCamera.acquisition_in_progress()* would still return True in this case, even though new frames are no longer acquired). *nframes* sets up number of frame buffers.

setup_serial_params (*write_term=”*, *datatype='bytes'*)
Setup default serial communication parameters.

Parameters

- **write_term** – default terminator character to be added to the sent messages
- **datatype** – type of the result of read commands; can be "bytes" (return raw bytes), or "str" (convert into UTF-8 string)

snap (*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition (**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is None). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition ()
Stop acquisition

wait_for_frame (*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be “lastread”‘‘ (from the last read frame), "lastwait" (wait for the last successful *wait_for_frame()* call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

pylablib.devices.IMAQ.niimaq_attrtypes module

Module contents

pylablib.devices.IMAQdx package

Submodules

pylablib.devices.IMAQdx.IMAQdx module

class pylablib.devices.IMAQdx.IMAQdx.**TCameraInfo** (*name*, *type*, *version*, *flags*, *serial_number*, *bus*, *vendor*, *model*, *camera_file*, *attr_url*)

Bases: tuple

attr_url

bus

camera_file

count()
Return number of occurrences of value.

flags

index()
Return first index of value.

Raises ValueError if the value is not present.

model

name

serial_number

type

vendor

version

pylablib.devices.IMAQdx.IMAQdx.**list_cameras**(*connected=True*, *desc=True*)
List all cameras available through IMAQdx interface

If `desc==True`, return complete camera descriptions; otherwise, simply return the names.

pylablib.devices.IMAQdx.IMAQdx.**get_cameras_number**()
Get number of connected dx cameras

class pylablib.devices.IMAQdx.IMAQdx.**IMAQdxAttribute**(*sid*, *name*)
Bases: `object`

Object representing an IMAQdx camera parameter.

Allows to query and set values and get additional information. Usually created automatically by an *IMAQdxCamera* instance, but could be created manually.

Parameters

- **sid** – camera session ID
- **name** – attribute text name

name
attribute name

display_name
attribute display name (short description name)

tooltip
longer attribute description

description
full attribute description (usually, same as *tooltip*)

units
attribute units (if applicable)

readable
whether attribute is readable

Type bool

writable
whether attribute is writable

Type bool

min
minimal attribute value (if applicable)

Type float or int

max
maximal attribute value (if applicable)

Type float or int

inc
minimal attribute increment value (if applicable)

Type float or int

values
list of possible attribute values (if applicable)

update_limits()
Update minimal and maximal attribute limits and return tuple `(min, max, inc)`

truncate_value(*value*)
Truncate value to lie within attribute limits

get_value(*enum_as_str=True*)
Get attribute value.

If `enum_as_str==True`, return enum-style values as strings; otherwise, return corresponding integer values.

set_value(*value*, *truncate=True*)
Get attribute value.

If `truncate==True`, automatically truncate value to lie within allowed range.

class pylablib.devices.IMAQdx.IMAQdx.**TDeviceInfo**(*vendor*, *model*, *serial_number*, *bus_type*)
Bases: `tuple`

bus_type

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

model

serial_number

vendor

class pylablib.devices.IMAQdx.IMAQdx.**IMAQdxCamera**(*name='cam0'*, *mode='controller'*, *visibility='advanced'*)
Bases: *pylablib.devices.interface.camera.IROICamera*, *pylablib.devices.interface.camera.IAttributeCamera*

Generic IMAQdx camera interface.

Parameters

- **name** – interface name (can be learned by `list_cameras()`; usually, but not always, starts with `"cam"`)
- **mode** – connection mode; can be `"controller"` (full control) or `"listener"` (only reading)
- **visibility** – attribute visibility when listing attributes; can be `"simple"`, `"intermediate"` or `"advanced"` (higher mode exposes more attributes).

Error = <Mock name='mock.IMAQdxError' id='140318679809872'>

TimeoutError = <Mock spec='str' id='140318700982224'>

open()
Open connection to the camera

close()
Close connection to the camera

reset()
Reset connection to the camera

is_opened()
Check if the device is connected

post_open()
Additional setup after camera opening

get_attribute_value(*name*, *error_on_missing=True*, *default=None*)
Get value of an attribute with the given name.

If the value doesn't exist or can not be read and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, assume that `error_on_missing==False`. If *name* points at a dictionary branch, return a dictionary with all values in this branch.

set_attribute_value(*name*, *value*, *truncate=True*, *error_on_missing=True*)
Set value of an attribute with the given name.

If the value doesn't exist or can not be written and `error_on_missing==True`, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). If `truncate==True`, truncate value to lie within attribute range.

get_all_attribute_values(*root=”*)
Get values of all attributes with the given *root*

set_all_attribute_values(*settings*, *root=”*, *truncate=True*)
Set values of all attributes with the given *root*.

If `truncate==True`, truncate value to lie within attribute range.

get_device_info()
Get camera information.

Return tuple `(vendor, model, serial_number, bus_type)`.

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

set_roi (*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

get_roi_limits (*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

setup_acquisition (*mode='sequence'*, *nframes=100*)
Setup acquisition mode.

mode can be either `"snap"` (single frame or a fixed number of frames) or `"sequence"` (continuous acquisition). (note that `IMAQdxCamera.acquisition_in_progress()` would still return `True` in this case, even though new frames are no longer acquired). *nframes* sets up number of frame buffers.

clear_acquisition ()
Clear acquisition settings

start_acquisition (**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:``setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition ()
Stop acquisition

acquisition_in_progress ()
Check if acquisition is in progress

refresh_acquisition (*delay=0.005*)
Stop and restart the acquisition, waiting *delay* seconds in between

enable_raw_readout (*enable='rows'*)
Enable raw frame transfer.

Should be used if the camera uses unsupported pixel format. Can be `"frame"` (return the whole frame as a 1D `"u1"` numpy array), `"rows"` (return a 2D array, where each row corresponds to a single image row), or `False` (convert to image data, or raise an error if the format is not supported; default)

FrameTransferError
alias of `pylablib.devices.interface.camera.DefaultFrameTransferError`

class NoParameterCaller (*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings (*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_all_attributes(*copy=False*)
Return a dictionary of all available attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by `set_frame_info_format()`) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple (first, last) with images range (first inclusive). If no images are available, return None. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if None, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them, while still keeping total returned frames number to *n*). If return_info==True, return tuple (frames, infos), where infos is a list of frame info tuples (camera-dependent); if some frames are missing and missing_frame!="skip", the corresponding frame info is None.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return True.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of with block.

Useful for applying certain settings which can't be changed during the acquisition. If clear==True, clear acquisition in addition to pausing (by default, use the class default specified as _clear_pausing_acquisition attribute). Yields tuple (acq_in_progress, acq_params), which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple (first, last) with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return None. If peek==True, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them). If return_info==True, return tuple (frames, infos), where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"‘‘ (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

class pylablib.devices.IMAQdx.IMAQdx.**EthernetIMAQdxCamera**(*name='cam0'*, *mode='controller'*, *visibility='advanced'*, *small_packet=False*)

Bases: *pylablib.devices.IMAQdx.IMAQdx.IMAQdxCamera*

LAN-controlled IMAQdx camera.

Compared to the standard camera, has an option of automatically switching to a smaller TCP/IP packet size (can be useful if the PC network adapter can't handle jumbo packets).

Parameters

- **name** – interface name (can be learned by *list_cameras()*; usually, but not always, starts with "cam")
- **mode** – connection mode; can be "controller" (full control) or "listener" (only reading)
- **visibility** – default attribute visibility when listing attributes; can be "simple", "intermediate" or "advanced" (higher mode exposes more attributes).
- **small_packet** – if True, automatically set small packet size (1500 bytes).

Error = <Mock name='mock.IMAQdxError' id='140318679809872'>

FrameTransferError

alias of *pylablib.devices.interface.camera.DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)

Bases: object

Class to simplify calling functions without a parameter

TimeoutError = <Mock spec='str' id='140318700982224'>

acquisition_in_progress()

Check if acquisition is in progress

apply_settings(*settings*)

Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

clear_acquisition()

Clear acquisition settings

close()

Close connection to the camera

enable_raw_readout(*enable='rows'*)

Enable raw frame transfer.

Should be used if the camera uses unsupported pixel format. Can be "frame" (return the whole frame as a 1D "u1" numpy array), "rows" (return a 2D array, where each row corresponds to a single image row), or False (convert to image data, or raise an error if the format is not supported; default)

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_all_attribute_values(*root=''*)
Get values of all attributes with the given *root*

get_all_attributes(*copy=False*)
Return a dictionary of all available attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_attribute_value(*name*, *error_on_missing=True*, *default=None*)
Get value of an attribute with the given name.

If the value doesn't exist or can not be read and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, assume that `error_on_missing==False`. If *name* points at a dictionary branch, return a dictionary with all values in this branch.

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_info()
Get camera information.

Return tuple `(vendor, model, serial_number, bus_type)`.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by `set_frame_info_format()`) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

open()
Open connection to the camera

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

refresh_acquisition(*delay=0.005*)
Stop and restart the acquisition, waiting *delay* seconds in between

reset()
Reset connection to the camera

set_all_attribute_values(*settings*, *root=''*, *truncate=True*)
Set values of all attributes with the given *root*.

If `truncate==True`, truncate value to lie within attribute range.

set_attribute_value (*name*, *value*, *truncate=True*, *error_on_missing=True*)
Set value of an attribute with the given name.

If the value doesn't exist or can not be written and `error_on_missing==True`, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). If `truncate==True`, truncate value to lie within attribute range.

set_device_variable (*key*, *value*)
Set the value of a settings parameter

set_frame_format (*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format (*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period (*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing (*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

set_roi (*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

setup_acquisition (*mode='sequence'*, *nframes=100*)
Setup acquisition mode.

mode can be either `"snap"` (single frame or a fixed number of frames) or `"sequence"` (continuous acquisition). (note that `IMAQdxCamera.acquisition_in_progress()` would still return `True` in this case, even though new frames are no longer acquired). *nframes* sets up number of frame buffers.

snap (*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition (**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be “lastread”‘‘ (from the last read frame), `"lastwait"` (wait for the last successful *wait_for_frame()* call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

post_open()
Additional setup after camera opening

Module contents

pylablib.devices.Lakeshore package

Submodules

pylablib.devices.Lakeshore.base module

exception pylablib.devices.Lakeshore.base.**LakeshoreError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Lakeshore devices error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Lakeshore.base.**LakeshoreBackendError**(*exc*)
Bases: *pylablib.devices.Lakeshore.base.LakeshoreError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Generic Lakeshore backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.Lakeshore.base.**TLakeshore218AnalogSettings**(*bipolar*, *mode*, *channel*, *source*, *high_value*, *low_value*, *man_value*)

Bases: tuple

bipolar

channel

count()
Return number of occurrences of value.

high_value

index()
Return first index of value.

Raises ValueError if the value is not present.

low_value

man_value

mode

source

class pylablib.devices.Lakeshore.base.**TLakeshore218FilterSettings**(*enabled*, *points*, *window*)

Bases: tuple

count()
Return number of occurrences of value.

enabled

index()
Return first index of value.

Raises ValueError if the value is not present.

points

window

class pylablib.devices.Lakeshore.base.**TLakeshore218CurveHeader**(*name*, *serial*, *fmt*, *limit*, *coeff*)

Bases: tuple

coeff

count()
Return number of occurrences of value.

fmt

index()
Return first index of value.

Raises ValueError if the value is not present.

limit

name

serial

class pylablib.devices.Lakeshore.base.**Lakeshore218**(*conn*)

Bases: *pylablib.core.devio.SCPI.SCPIDevice*

Lakeshore 218 temperature controller.

The channels are enumerated from 1 to 8 and are split into 2 groups: "A" for 1-4 and "B" for 5-8.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

Error

alias of *LakeshoreError*

ReraiseError

alias of *LakeshoreBackendError*

is_enabled(*channel*)

Check if a given channel is enabled

set_enabled(*channel*, *enabled=True*)

Enable or disable a given channel

get_sensor_type(*group*)

Get sensor type for a given group ("A" for sensors 1-4 or "B" for sensors 5-8).

For types, see INTYPE command description in the Lakeshore 218 programming manual.

set_sensor_type(*group*, *sensor_type*)

Set sensor type for a given group ("A" for sensors 1-4 or "B" for sensors 5-8).

For types, see INTYPE command description in the Lakeshore 218 programming manual.

get_sensor_curve_index(*channel*)

Get sensor curve index for a given channel (1 to 8).

For curve descriptions, see INCRV command description in the Lakeshore 218 programming manual.

set_sensor_curve_index(*channel*, *index*)

Get sensor curve index for a given channel (1 to 8).

For curve descriptions, see INCRV command description in the Lakeshore 218 programming manual.

get_curve_header(*index*)

Get header of a given curve (1-9 or 21-28).

Return tuple (name, serial, fmt, limit, coeff). For values descriptions, see CRVHDR command description in the Lakeshore 218 programming manual.

set_curve_header(*index*, *name=None*, *serial=None*, *fmt=None*, *limit=None*, *coeff=None*)

Set header of a given user curve (21-28).

For values descriptions, see CRVHDR command description in the Lakeshore 218 programming manual.

get_curve(*index*, *trim_zeros=True*)

Get values of a given curve (1-9 or 21-28).

Return 2-column numpy array with up to 200 points, where the first column is sensor reading, and the second is temperature; for associated sensor units, see *get_curve_header()*. If trim_zeros==True, trim the trailing zero-valued points. Note, that it takes about 10 seconds to complete.

set_curve(*index*, *curve*)
Set values of a given user curve (21-28).

curve is a 2-column numpy array with up to 200 points, where the first column is sensor reading, and the second is temperature; for associated sensor units, see `get_curve_header()`. Note, that it takes about 20 seconds to complete.

get_temperature(*channel*)
Get readings (in Kelvin) on a given channel (1 to 8)

get_all_temperatures()
Get readings (in Kelvin) on all channels

get_sensor_reading(*channel*)
Get readings (in sensor units) on a given channel (1 to 8)

get_all_sensor_readings()
Get readings (in sensor units) on all channels

get_analog_output_settings(*output*)
Get analog output settings for a given output (1 or 2).

For parameters, see `setup_analog_output()` and `ANALOG` command description in the Lakeshore 218 programming manual.

setup_analog_output(*output*, *bipolar=None*, *mode=None*, *channel=None*, *source=None*, *high_value=None*, *low_value=None*, *man_value=None*)
Setup analog output settings for a given output (1 or 2).

For parameters, see `ANALOG` command description in the Lakeshore 218 programming manual. Value of `None` means keeping the current parameter value.

set_analog_output_value(*output*, *value*, *bipolar=False*, *enabled=True*)
Set manual analog output value.

A simplified version of `setup_analog_output()`.

get_analog_output(*output*)
Get value (in percents of the total range) at a given output (1 or 2)

get_filter_settings(*channel*)
Get input filter settings for a given channel (1 to 8).

For parameters, see `setup_filter()` and `FILTER` command description in the Lakeshore 218 programming manual.

setup_filter(*channel*, *enabled=None*, *points=None*, *window=None*)
Setup input filter settings for a given channel (1 to 8).

For parameters, see `FILTER` command description in the Lakeshore 218 programming manual. Value of `None` means keeping the current parameter value.

BackendError
alias of `pylablib.core.devio.comm_backend.DeviceBackendError`

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in `read()`. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()
Close the backend

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)
Get the device status register (by default, `"*ESR?"` command)

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

set_device_variable(*key*, *value*)
Set the value of a settings parameter

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class `pylablib.devices.Lakeshore.base.`**TLakeshore370RangeSettings**(*exc_mode*, *exc_range*, *res_range*, *autorange*, *enable*)

Bases: `tuple`

autorange

count()
Return number of occurrences of value.

enable

exc_mode

exc_range

index()
Return first index of value.

Raises ValueError if the value is not present.

res_range

class pylablib.devices.Lakeshore.base.**TLakeshore370AnalogSettings**(*bipolar, mode, channel, source, high_value, low_value, man_value*)

Bases: tuple

bipolar

channel

count()

Return number of occurrences of value.

high_value

index()

Return first index of value.

Raises ValueError if the value is not present.

low_value

man_value

mode

source

class pylablib.devices.Lakeshore.base.**TLakeshore370FilterSettings**(*enabled, settle_time, window*)

Bases: tuple

count()

Return number of occurrences of value.

enabled

index()

Return first index of value.

Raises ValueError if the value is not present.

settle_time

window

class pylablib.devices.Lakeshore.base.**Lakeshore370**(*conn*)

Bases: *pylablib.core.devio.SCPI.SCPIDevice*

Lakeshore 370 resistance bridge / temperature controller.

All channels are enumerated from 0.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

Error

alias of *LakeshoreError*

ReraiseError

alias of *LakeshoreBackendError*

get_temperature(*channel*)
Get temperature readings (in K) on a given channel

get_resistance(*channel*)
Get resistance readings (in Ohm) on a given channel

get_sensor_power(*channel*)
Get dissipated power (in W) on a given channel

select_channel(*channel*)
Select measurement channel

get_channel()
Get current measurement channel

get_channel_range_settings(*channel*)
Setup the current measurement channel range parameters.

For parameters, see `setup_channel_range()` and `RDGRNG` command description in the Lakeshore 370 programming manual.

setup_channel_range(*channel=None*, *exc_mode='v'*, *exc_range=1*, *res_range=22*, *autorange=True*, *enable=True*)
Setup the measurement channel range (all channels by default).

exc_mode is the excitation mode (`"i"` or `"v"`), *exc_range* is the excitation range (1 is smallest), *res_range* is the resistance range (1 is smallest). For range descriptions, see Lakeshore 370 programming manual.

get_analog_output_settings(*output*)
Get analog output settings for a given output (1 or 2).

For parameters, see `setup_analog_output()` and `ANALOG` command description in the Lakeshore 370 programming manual.

setup_analog_output(*output*, *bipolar=None*, *mode=None*, *channel=None*, *source=None*, *high_value=None*, *low_value=None*, *man_value=None*)
Setup analog output settings for a given output (1 or 2).

For parameters, see `ANALOG` command description in the Lakeshore 370 programming manual. Value of `None` means keeping the current parameter value.

set_analog_output_value(*output*, *value*, *bipolar=False*, *enabled=True*)
Set manual analog output value.

A simplified version of `setup_analog_output()`.

get_analog_output(*output*)
Get value (in percents of the total range) at a given output (1 or 2)

get_filter_settings(*channel*)
Get input filter settings for a given channel (1 to 16).

For parameters, see `setup_filter()` and `FILTER` command description in the Lakeshore 370 programming manual.

setup_filter(*channel*, *enabled=None*, *settle_time=None*, *window=None*)
Setup input filter settings for a given channel (1 to 16).

For parameters, see `FILTER` command description in the Lakeshore 370 programming manual. Value of `None` means keeping the current parameter value.

BackendError
alias of `pylablib.core.devio.comm_backend.DeviceBackendError`

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in `read()`. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()
Close the backend

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)
Get the device status register (by default, `"*ESR?"` command)

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

set_device_variable(*key*, *value*)
Set the value of a settings parameter

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

Module contents

pylablib.devices.LaserQuantum package

Submodules

pylablib.devices.LaserQuantum.base module

exception pylablib.devices.LaserQuantum.base.**LaserQuantumError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Laser Quantum devices error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.LaserQuantum.base.**LaserQuantumBackendError**(*exc*)
Bases: *pylablib.devices.LaserQuantum.base.LaserQuantumError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Generic Laser Quantum backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.LaserQuantum.base.**TDeviceInfo**(*serial*, *software_version*, *cal_date*)
Bases: tuple

cal_date

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

serial

software_version

class pylablib.devices.LaserQuantum.base.**TWorkHours**(*psu*, *laser_enabled*, *laser_threshold*)
Bases: tuple

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

laser_enabled

laser_threshold

psu

class pylablib.devices.LaserQuantum.base.**TTemperatures**(*head*, *psu*)
Bases: tuple

count()
Return number of occurrences of value.

head

index()
Return first index of value.

Raises ValueError if the value is not present.

psu

class pylablib.devices.LaserQuantum.base.**Finesse**(*conn*)
Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*

Laser Quantum Finesse pump laser.

Parameters **conn** – serial connection parameters (usually port)

Error
alias of *LaserQuantumError*

query(*comm*, *reply_lines=1*)
Send a query to the device and read the reply.

reply_lines specify the number of lines to read as a reply (almost all queries have only one line).

get_device_info()
Get device information `(serial, software_version, cal_date)`

get_work_hours()
Get the work hours (PSU run time, laser run time, laser above threshold time)

get_temperatures()
Get device status, head temperature, and PSU temperature

get_output_status()
Get output status.

Can be `"enabled"` or `"disabled"`.

get_interlock_status()
Get manual interlock status

get_shutter_status()
Get the shutter status

is_shutter_opened()
Check if shutter is opened

set_shutter(*opened=True*)
Open or close the shutter

is_enabled()
Check if the output is enabled

enable(*enabled=True*)
Turn the output on or off

get_output_power()
Get the output power (in Watts)

get_output_setpoint()
Get the output setpoint power (in Watts)

set_output_power(*level*)
Set the output power setpoint (in Watts)

get_current()
Get the laser drive current (in %)

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key, value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.LighthousePhotonics package

Submodules

pylablib.devices.LighthousePhotonics.base module

exception `pylablib.devices.LighthousePhotonics.base.`**`LighthousePhotonicsError`**

Bases: *`pylablib.core.devio.base.DeviceError`*

Generic Lighthouse Photonics devices error

`args`

`with_traceback``()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `pylablib.devices.LighthousePhotonics.base.`**`LighthousePhotonicsBackendError`**`(`*exc*`)`

Bases: *`pylablib.devices.LighthousePhotonics.base.LighthousePhotonicsError`*, *`pylablib.core.devio.comm_backend.DeviceBackendError`*

Generic Lighthouse Photonics backend communication error

`args`

`with_traceback``()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class `pylablib.devices.LighthousePhotonics.base.`**`TDeviceInfo`**`(`*product*, *version*, *serial*, *configuration*`)`

Bases: `tuple`

`configuration`

`count``()`

Return number of occurrences of value.

`index``()`

Return first index of value.

Raises ValueError if the value is not present.

`product`

`serial`

`version`

class `pylablib.devices.LighthousePhotonics.base.`**`TWorkHours`**`(`*controller*, *laser*`)`

Bases: `tuple`

`controller`

`count``()`

Return number of occurrences of value.

`index``()`

Return first index of value.

Raises ValueError if the value is not present.

`laser`

class `pylablib.devices.LighthousePhotonics.base.`**`SproutG`**`(`*conn*`)`

Bases: *`pylablib.core.devio.comm_backend.ICommBackendWrapper`*

Lighthouse Photonics Sprout G laser.

Parameters **`conn`** – serial connection parameters (usually port)

Error
alias of *LighthousePhotonicsError*

query(*comm*, *allowed_replies=('0',)*)
Send a query to the device and parse the reply

get_device_info()
Get device information (product name, product version, serial number, configuration)

get_work_hours()
Return device operation hours (controller on) and run hours (laser on)

get_warning_status()
Get device warnings

get_interlock_status()
Get manual interlock status

get_shutter_status()
Get manual shutter status (`"open"` or `"close"`)

get_output_mode()
Get output mode.

Can be `"on"`, `"off"`, `"idle"` (power standby mode), `"calibrate"`, `"interlock"` (manual interlock is off), `"warmup"` (warmup mode), or `"calibration"` (calibration mode).

set_output_mode(*mode='on'*)
Set output mode.

mode can be `"on"`, `"off"`, `"idle"` (power standby mode), or `"calibrate"` (calibration mode).

is_enabled()
Check if the output is on (idle or warmup don't count as on)

enable(*enabled=True*)
Turn the output on or off

get_output_power()
Set the actual output power (in Watts)

get_output_setpoint()
Get the output setpoint power (in Watts)

set_output_power(*level*)
Get the output power setpoint (in Watts)

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.M2 package

Submodules

pylablib.devices.M2.solstis module

exception pylablib.devices.M2.solstis.**M2Error**
Bases: *pylablib.core.devio.base.DeviceError*

Generic M2 error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.M2.solstis.**M2CommunicationError**(*exc*)
Bases: *pylablib.devices.M2.solstis.M2Error*, *pylablib.core.devio.comm_backend.DeviceBackendError*

M2 network communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.M2.solstis.**Solstis**(*addr*, *port*, *timeout=5.0*, *start_link=True*, *use_websocket='auto'*, *use_cavity=True*)
Bases: *pylablib.core.devio.interface.IDevice*

M2 Solstis Ice Bloc device.

Parameters

- **addr** (*str*) – IP address of the Ice Bloc device.
- **port** (*int*) – port of the Ice Bloc device.
- **timeout** (*float*) – default timeout of synchronous operations.
- **start_link** (*bool*) – if `True`, initialize device link on creation.
- **use_websocket** (*bool*) – if `True`, use websocket interface (same as used by the web interface) for additional functionality (wavemeter connection, etalon value, improved operation stopping); `"auto"` enables it if websocket package is installed, and disables otherwise
- **use_cavity** – if `False` and any reference cavity methods are used, either ignore them, or use closest available methods instead

Error
alias of *M2Error*

ReraiseError
alias of *M2CommunicationError*

BackendError
alias of `builtins.OSError`

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

set_timeout(*timeout*)
Set timeout for connecting or sending/receiving

flush()
Flush read buffer

query(*op*, *params*, *reply_op='auto'*, *report=False*)
Send a query using the standard device interface.

reply_op is the name of the reply operation (by default, its the operation name plus `"_reply"`). If `report==True`, request completion report (does not apply to all operation).

update_reports(*timeout=0.0*)
Check for fresh operation reports

get_last_report(*op*)
Get the latest report for the given operation

check_report(*op*)
Check and return the latest report for the given operation

wait_for_report(*op*, *error_msg=None*, *timeout=None*)
Wait for a report for the given operation

error_msg specifies the exception message if the report results in an error.

start_link()
Initialize device link (called automatically on creation)

connect_wavemeter(*sync=True*)
Connect to the wavemeter (if `sync==True`, wait until the connection is established)

disconnect_wavemeter(*sync=True*)
Disconnect from the wavemeter (if `sync==True`, wait until the connection is broken)

is_wavemeter_connected()
Check if the wavemeter is connected

get_system_status()
Get the device system status

get_full_web_status()
Get full websocket status.

Return a large dictionary containing all the information available in the web interface.

get_full_fine_tuning_status()
Get full fine-tuning status (see M2 Solstis JSON protocol manual for `"poll_wave_m"` command)

lock_wavemeter(*lock=True*, *sync=True*, *error_on_fail=True*)
Lock or unlock the laser to the wavemeter (if `sync==True`, wait until the operation is complete)

is_wavemeter_lock_on()
Check if the laser is locked to the wavemeter

fine_tune_wavelength(*wavelength*, *sync=True*, *timeout=None*)
Fine-tune the wavelength.

Only works if the wavemeter is connected. If `sync==True`, wait until the operation is complete (might take from several seconds up to several minutes).

check_fine_tuning_report()
Check wavelength fine-tuning report

Return `"success"` or `"fail"` if the operation is complete, or `None` if it is still in progress.

wait_for_fine_tuning(*timeout=None*)
Wait until wavelength fine-tuning is complete

get_fine_tuning_status()
Get fine-tuning status.

Return either `"idle"` (no tuning or locking), `"nolink"` (no wavemeter link), `"tuning"` (tuning in progress), or `"locked"` (tuned and locked to the wavemeter).

get_fine_wavelength()
Get fine-tuned wavelength.

Only works if the wavemeter is connected.

stop_fine_tuning()
Stop fine wavelength tuning

coarse_tune_wavelength(*wavelength*, *sync=True*)
Coarse-tune the wavelength.

Only works if the wavemeter is disconnected. If `sync==True`, wait until the operation is complete.

get_full_coarse_tuning_status()
Get full coarse-tuning status (see M2 M2 Solstis JSON protocol manual for `"poll_move_wave_t"` command)

get_coarse_tuning_status()
Get coarse-tuning status.

Return either `"done"` (tuning is done), `"tuning"` (tuning in progress), or `"fail"` (tuning failed).

get_coarse_wavelength()
Get course-tuned wavelength.

Only works if the wavemeter is disconnected.

stop_coarse_tuning()
Stop coarse wavelength tuning

tune_etalon(*value*, *sync=True*)
Tune the etalon to *value* percent.

Only works if the wavemeter is disconnected. If `sync==True`, wait until the operation is complete.

lock_etalon(*sync=True*)
Lock the etalon.

If `sync==True`, wait until the operation is complete.

unlock_etalon(*sync=True*)
Unlock the etalon .

If `sync==True`, wait until the operation is complete. Automatically unlock the reference cavity first (otherwise the operation fails).

get_etalon_lock_status()
Get etalon lock status.

Return either `"off"` (lock is off), `"on"` (lock is on), `"debug"` (lock in debug condition), `"error"` (lock had an error), `"search"` (lock is searching), or `"low"` (lock is off due to low output).

tune_laser_resonator(*value*, *fine=False*, *sync=True*)
Tune the laser cavity to *value* percent.

If `fine==True`, adjust fine tuning; otherwise, adjust coarse tuning. Only works if the wavemeter is disconnected. If `sync==True`, wait until the operation is complete.

tune_reference_cavity(*value*, *fine=False*, *sync=True*)
Tune the reference cavity to *value* percent.

If `fine==True`, adjust fine tuning; otherwise, adjust coarse tuning. Only works if the wavemeter is disconnected. If `sync==True`, wait until the operation is complete. If reference cavity is disabled by setting `use_cavity=False` on creation, do nothing.

lock_reference_cavity(*sync=True*)
Lock the laser to the reference cavity.

Automatically lock etalon first (otherwise the operation fails). If `sync==True`, wait until the operation is complete. If reference cavity is disabled by setting `use_cavity=False` on creation, do nothing.

unlock_reference_cavity(*sync=True*)
Unlock the laser from the reference cavity.

If `sync==True`, wait until the operation is complete. If reference cavity is disabled by setting `use_cavity=False` on creation, do nothing.

get_reference_cavity_lock_status()
Get the reference cavity lock status.

Return either `"off"` (lock is off), `"on"` (lock is on), `"debug"` (lock in debug condition), `"error"` (lock had an error), `"search"` (lock is searching), `"low"` (lock is off due to low output), or `"disabled"` (reference cavity is disabled by setting `use_cavity=False` on creation).

setup_terascan(*scan_type*, *scan_range*, *rate*, *trunc_rate=True*)
Setup terascan.

Parameters

- **scan_type** (*str*) – scan type. Can be `"medium"` (BRF+etalon, rate from 100 GHz/s to 1 GHz/s), `"fine"` (all elements, rate from 20 GHz/s to 1 MHz/s), or `"line"` (all elements, rate from 20 GHz/s to 50 kHz/s).
- **scan_range** (*tuple*) – tuple `(start, stop)` with the scan range (in Hz).
- **rate** (*float*) – scan rate (in Hz/s).
- **trunc_rate** (*bool*) – if `True`, truncate the scan rate to the nearest available rate (otherwise, incorrect rate would raise an error).

If reference cavity is disabled by setting `use_cavity=False` on creation and *scan_type* is `"line"`, use `"fine"` instead.

start_terascan(*scan_type*, *sync=False*, *sync_done=False*)
Start terascan.

Scan parameters are set up separately using `setup_terascan()`. Scan type can be `"medium"` (BRF+etalon, rate from 100 GHz/s to 1 GHz/s), `"fine"` (all elements, rate from 20 GHz/s to 1 MHz/s), or `"line"` (all elements, rate from 20 GHz/s to 50 kHz/s). If reference cavity is disabled by setting `use_cavity=False` on creation and *scan_type* is `"line"`, use `"fine"` instead. If `sync==True`, wait until the scan is set up (not until the whole scan is complete). If `sync_done==True`, wait until the whole scan is complete (not recommended, as it can take hours).

enable_terascan_updates(*enable=True*, *update_period=0*)
Enable sending periodic terascan updates.

If enabled, laser will send updates in the beginning and in the end of every terascan segment. If `update_period!=0`, it will also send updates every `update_period` percents of the segment (this option is not currently supported by M2 firmware).

check_terascan_update()
Check the latest terascan update.

Return `None` if none are available, or a dictionary `{"wavelength":current_wavelength, "operation":op}`, where `op` is `"scanning"` (scanning in progress), `"stitching"` (stitching in progress), `"finished"` (scan is finished), or `"repeat"` (segment is repeated).

wait_for_terascan_update()
Wait until a new terascan update is available

check_terascan_start_report()
Check report on terascan start.

Return "success" or "fail" if the operation is complete, or None if it is still in progress.

stop_terascan(*scan_type*, *sync=False*)
Stop terascan of the given type.

If reference cavity is disabled by setting use_cavity=False on creation and *scan_type* is "line", use "fine" instead. If sync==True, wait until the operation is complete.

get_terascan_status(*scan_type*, *web_status=True*)
Get status of a terascan of a given type.

Return a dictionary with 4 items: "current": current laser frequency (or None if no scan is in progress) "range": tuple with the fill scan range (or None if no frequency is available) "status": can be "stopped" (scan is not in progress), "scanning" (scan is in progress), or "stitching" (scan is in progress, but currently stitching) "web": whether scan is running in web interface (some failure modes still report "scanning" through the usual interface); only available if the laser web connection is on and if web_status==True.

If reference cavity is disabled by setting use_cavity=False on creation and *scan_type* is "line", use "fine" instead.

start_fast_scan(*scan_type*, *width*, *period*, *sync=False*, *setup_locks=True*)
Setup and start fast scan.

Parameters

- **scan_type** (*str*) – scan type. Can be "cavity_continuous", "cavity_single", "cavity_triangular", "etalon_continuous", "etalon_single", "resonator_continuous", "resonator_single", "resonator_ramp", "resonator_triangular", "ecd_continuous", "ecd_ramp", or "fringe_test" (see M2 Solstis JSON protocol manual for details)
- **width** (*float*) – scan width (in Hz).
- **period** (*float*) – scan time/period (in s).
- **sync** (*bool*) – if True, wait until the scan is set up (not until the whole scan is complete).
- **setup_locks** (*bool*) – if True, automatically setup etalon and reference cavity locks in the appropriate states for etalon, cavity, or resonator scans.

If reference cavity is disabled by setting use_cavity=False on creation, use resonator scans instead of cavity scans.

check_fast_scan_start_report()
Check fast scan start report.

Return "success" or "fail" if the operation is complete, or None if it is still in progress.

stop_fast_scan(*scan_type*, *return_to_start=True*, *sync=False*)
Stop fast scan of the given type.

If reference cavity is disabled by setting use_cavity=False on creation, use resonator scans instead of cavity scans. If return_to_start==True, return to the center frequency after stopping; otherwise, stay at the current instantaneous frequency. If sync==True, wait until the operation is complete.

get_fast_scan_status(*scan_type*)
Get status of a fast scan of a given type.

Return dictionary with 2 items: `"status"`: can be `"stopped"` (scan is not in progress), `"scanning"` (scan is in progress). `"value"`: current tuner value (in percent); does not necessary correspond to the scan progress.

If reference cavity is disabled by setting `use_cavity=False` on creation, use resonator scans instead of cavity scans.

stop_scan_web(*scan_type*)
Stop scan of the current type (terascan or fine scan) using web interface.

More reliable than native programming interface, but requires activated web interface. If reference cavity is disabled by setting `use_cavity=False` on creation, use resonator scans instead of cavity scans.

stop_all_operation(*repeated=True*, *attempt=0*)
Stop all laser operations (tuning and scanning).

More reliable than native programming interface, but requires activated web interface. If `repeated==True`, repeat trying to stop the operations until succeeded (more reliable, but takes more time). If `attempt>0`, it can supply the number of already tried attempts to stop (with `repeated=False`); the more attempts failed, the more drastic measures will be taken to stop (e.g., initialize short terascan or a fast scan, cycle wavemeter connection, etc.) Return `True` if the operation is success and `False` otherwise.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

Module contents

pylablib.devices.NI package

Submodules

pylablib.devices.NI.daq module

exception pylablib.devices.NI.daq.**NIError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic NI error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.NI.daq.**NIDAQmxError**(*exc*)
Bases: *pylablib.devices.NI.daq.NIError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

NI DAQmx backend operation error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.NI.daq.**TDeviceInfo**(*name*, *model*, *serial_number*)
Bases: tuple

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

model

name

serial_number

pylablib.devices.NI.daq.**get_device_info**(*name*)
Get device info.

Return tuple (name, model, serial).

pylablib.devices.NI.daq.**list_devices**()
List all connected NI DAQ devices

class pylablib.devices.NI.daq.**NIDAQ**(*dev_name='dev0'*, *rate=100.0*, *buffer_size=100000.0*, *reset=False*)
Bases: *pylablib.core.devio.interface.IDevice*

National Instruments DAQ device interface (wrapper around nidaqmx library).

Simplified interface to NI DAQ devices. Supports voltage, digital, and counter inputs (all synchronized to the same clock), and digital and voltage outputs (asynchronous).

Parameters

- **dev_name** (*str*) – root device name.
- **rate** (*float*) – analog input sampling rate (can be adjusted later).
- **buffer_size** (*int*) – size of the input buffer.
- **reset** (*int*) – if `True`, reset the device upon connection.

Error
alias of *NIError*

ReraiseError
alias of *NIDAQmxError*

BackendError
Used by autodoc_mock_imports.

open()
Open the connection

close()
Close the connection

is_opened()
Check if the device is connected

reset()
Reset the device. All channels will be removed

get_device_info()
Get device info.

Return tuple `(name, model, serial)`.

setup_clock(*rate*, *src=None*)
Setup analog input clock (which is the main system clock).

If `src==None`, use internal clock with the given rate; otherwise use *src* terminal as a clock source (in this case, *rate* should be higher than the expected source rate).

get_clock_parameters()
Get analog input clock configuration.

Return tuple `(rate, src)`.

export_clock(*terminal*)
Export system clock to the given terminal (`None` to disconnect all terminals)

Only terminal one can be active at a time.

get_export_clock_terminal()
Return terminal which outputs system clock (`None` if none is connected)

add_voltage_input(*name*, *channel*, *rng=(-10, 10)*, *terminal_cfg='default'*)
Add analog voltage input.

Readout is synchronized to the system clock.

Parameters

- **name** (*str*) – channel name to refer to it later.
- **channel** (*str*) – terminal name (e.g., `"ai0"`).
- **rng** – voltage range

- **terminal_cfg** – terminal configuration; can be "default", "rse" (single-ended, referenced to AI SENSE input), "nrse" (single-ended, referenced to AI GND), "diff" (differential), or "pseudodiff" (see NI DAQ manual for details).

add_counter_input (*name*, *counter*, *terminal*, *clk_src='ai/SampleClock'*, *output_format='rate'*)
Add counter input (value is related to the number of counts).

Readout is synchronized to the system clock.

Parameters

- **name** (*str*) – channel name.
- **counter** (*str*) – on-board counter name (e.g., "ctr0").
- **terminal** (*str*) – terminal name (e.g., "pfi0").
- **clk_src** (*str*) – source of the counter sampling clock. By default it is the analog input clock, which requires at least one voltage input channel (could be a dummy channel) to be set up first.
- **output_format** (*str*) – output format. Can be "acc" (return accumulated number of counts since the sampling start), "diff" (return number of counts passed between the two consecutive sampling points; essentially, a derivative of "acc"), or "rate" (return count rate based on the "diff" samples).

add_clock_period_input (*counter*, *clk_src='ai/SampleClock'*)
Add clock period counter.

Useful when using external sample clock with unknown period. The clock input can be returned during *read()* operation, and it is used to calculate counter inputs in "rate" mode. Readout is synchronized to the system clock.

Parameters

- **counter** (*str*) – on-board counter name (e.g., "ctr0") to be used for clock measure.
- **clk_src** (*str*) – source of the counter sampling clock. By default it is the analog input clock, which requires at least one voltage input channel (could be dummy channel) to operate.

add_digital_input (*name*, *channel*)
Add digital input.

Readout is synchronized to the system clock. :param name: channel name. :type name: str :param channel: terminal name (e.g., "port0/line12"). :type channel: str

get_input_channels (*include=('ai', 'ci', 'di')*)
Get names of all input channels (voltage input and counter input).

include specifies which channel types to include into the list ("ai" for voltage inputs, "ci" for counter inputs, "di" for digital inputs, "cpi" for clock period channel). The channels order is always fixed: first voltage inputs, then counter inputs, then digital inputs.

get_voltage_input_parameters ()
Get parameters (names, channels, output ranges, and terminal configurations) of all analog voltage input channels

get_counter_input_parameters ()
Get parameters (names, counters, terminals, clock sources, and output formats) of all counter input channels

get_digital_input_parameters()
Get parameters (names and channels) of all digital input channels

get_clock_period_input_parameters()
Get parameters (counter input) of the clock period input channel

start(*flush_read=0*, *finite=None*)
Start the sampling and output task.

flush_read specifies number of samples to read and discard after start. If *finite* is not `None`, it specifies finite number of sample to acquire before stopping.

If counter channels are used, the first sample is usually unreliable, so `flush_read=1` is recommended; however, if exactly *finite* pulses are required at the clock export channel, `flush_read=0` is needed (the total number of pulses is `flush_read+finite`).

stop()
Stop the sampling task

is_running()
Check if the task is running

available_samples()
Get number of available samples to read (return 0 if the task is not running)

get_buffer_size()
Get the sampling buffer size

wait_for_sample(*num=1*, *timeout=10.0*, *wait_time=0.001*)
Wait until at least *num* samples are available.

If they are not available immediately, loop while checking every *wait_time* interval until enough samples are accumulated. Return the number of available samples if successful, or 0 if the execution timed out.

read(*n=1*, *flush_read=0*, *timeout=10.0*, *include=('ai', 'ci', 'di')*)
Read *n* samples. If the task is not running, automatically start before reading and stop after.

Parameters

- **n** (*int*) – number of samples to read. If `n<=0`, read all available samples.
- **flush_read** (*int*) – number of initial samples to skip if the task is currently stopped and needs to be started. If counter channels are used, the first sample is usually unreliable, so `flush_read=1` is recommended; however, if exactly *n* pulses are required at the clock export channel, `flush_read=0` is needed.
- **include** (*tuple*) – specifies which channel types to include into the list (`"ai"` for voltage inputs, `"ci"` for counter inputs, `"di"` for digital inputs, `"cpi"` for clock period channel).

Returns 2D numpy array of values arranged according to `get_input_channels()` order with the given *include* parameter.

add_digital_output(*name*, *channel*)
Add digital output.

Parameters

- **name** (*str*) – channel name.
- **channel** (*str*) – terminal name (e.g., `"do0"`).

get_digital_output_channels()
Get names of all digital output channels

get_digital_output_parameters()
Get parameters (names and channels) of all digital output channels

set_digital_outputs(*names*, *values*)
Set values of one or several digital outputs.

Parameters

- **names** (*str or [str]*) – name or list of names of outputs.
- **values** – output value or list of values.

get_digital_outputs(*names=None*)
Get values of one or several digital outputs.

Parameters names (*str or [str] or None*) – name or list of names of outputs (None means all outputs).

Return list of values ordered by *names* (or by *get_digital_output_channels()* if names==None).

add_voltage_output(*name*, *channel*, *rng=(-10, 10)*, *initial_value=0.0*)
Add analog voltage output.

Parameters

- **name** (*str*) – channel name.
- **channel** (*str*) – terminal name (e.g., "ao0").
- **rng** – voltage range.
- **initial_value** (*float*) – initial output value (has to be initialized).

get_voltage_output_channels()
Get names of all analog voltage output channels

get_voltage_output_parameters()
Get parameters (names, channels and output ranges) of all analog voltage output channels

set_voltage_outputs(*names*, *values*)
Set values of one or several analog voltage outputs.

Parameters

- **names** (*str or [str]*) – name or list of names of outputs.
- **values** – output value or list values. These can be single numbers, or arrays if the output clock is setup (see *setup_voltage_output_clock()*). In the latter case it sets up the output waveforms; not that waveforms for all channels must have the same length (a single number signifying a constant output is also allowed) If the analog output is set up to the finite mode (continuous==False), the finite waveform output happens right away, with the number of samples determined by *samps_per_channel* parameter of *setup_voltage_output_clock()*. In this case, if the supplied waveform is shorter than the number of samples, it gets repeated; if it's longer, it gets cut off.

get_voltage_outputs(*names=None*)
Get values of one or several analog voltage outputs.

Parameters names (*str or [str] or None*) – name or list of names of outputs (None means all outputs).

Return list of values ordered by *names* (or by `get_voltage_output_channels()` if `names==None`). For continuous waveforms, return the array containing a single repetition of the waveform. For finite waveforms, repeat the array containing the last outputted waveform.

class NoParameterCaller(*device*, *kind*)

Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)

Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_device_variable(*key*)

Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)

Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)

Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)

Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

set_device_variable(*key*, *value*)

Set the value of a settings parameter

setup_voltage_output_clock(*rate=0*, *sync_with_ai=False*, *continuous=True*, *samps_per_chan=1000*)

Setup analog output clock configuration.

Parameters

- **rate** – clock rate; if 0, assume constant voltage output (default)
- **sync_with_ai** – if `True`, the clock is synchronized to the analog input clock (the main clock); note that in this case output changes only when the analog read task is running
- **continuous** – if `True`, any written waveform gets repeated continuously; otherwise, it outputs written waveform only once, and then latches the output on the last value
- **samps_per_chan** – if `continuous==False`, it determines number of samples to output before stopping

get_voltage_output_clock_parameters()

Get analog output clock configuration.

Return tuple (rate, sync_with_ai, continuous, samps_per_chan).

Module contents

pylablib.devices.Newport package

Submodules

pylablib.devices.Newport.base module

exception pylablib.devices.Newport.base.**NewportError**

Bases: *pylablib.core.devio.base.DeviceError*

Generic Newport device error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Newport.base.**NewportBackendError**(*exc*)

Bases: *pylablib.devices.Newport.base.NewportError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Newport backend communication error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

pylablib.devices.Newport.picomotor module

pylablib.devices.Newport.picomotor.**get_usb_devices_number**()

Get the number of controllers connected via USB

pylablib.devices.Newport.picomotor.**muxaddr**(**args*, ***kwargs*)

Multiplex the function over its addr argument

class pylablib.devices.Newport.picomotor.**TDeviceInfo**(*id*)

Bases: tuple

count()

Return number of occurrences of value.

id

index()

Return first index of value.

Raises ValueError if the value is not present.

class pylablib.devices.Newport.picomotor.**Picomotor8742**(*conn=0*, *backend='auto'*, *timeout=5.0*, *multiaddr=False*, *scan=True*)

Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*, *pylablib.devices.interface.stage.IMultiaxisStage*

Picomotor 8742 4-axis controller.

Parameters

- **conn** – connection parameters; can be an index (starting from 0) for USB devices, or an IP address (e.g., `"192.168.0.2"`) or host name (e.g., `"8742-12345"`) for Ethernet devices
- **backend** – communication backend; by default, try to determine from the communication parameters
- **timeout** (*float*) – default operation timeout
- **multiaddr** – if `True`, assume that there are several daisy-chained devices connected to the current one; in this case, `get_device_info` and related methods return dictionaries `{addr: value}` for all connected controllers instead of simply values for the given controller
- **scan** – if `True` and `multiaddr==True`, scan for all connected devices (call `scan_devices()`) upon connection

Error
alias of `pylablib.devices.Newport.base.NewportError`

query(*comm*, *axis=None*, *addr=None*, *read_reply=None*)

get_id(*addr=None*)
Get the device identification string

get_device_info(*addr=None*)
Get the device info of the controller board: `(id_string,)`

reset(*addr=None*)
Restart the device.

Reboots the CPU and restores all saved settings from the parameter memory.

save_parameters(*addr=None*)
Store current parameters to the non-volatile memory.

Affects axes speed and acceleration, motor types, and Ethernet parameters.

restore_parameters(*src='memory'*, *addr=None*)
Restore parameters from the non-volatile memory (if `src=="memory"`) for factory parameters (if `src=="factory"`).

Affects axes speed and acceleration, motor types, and Ethernet parameters.

scan_devices(*reassign='conflict'*, *sync=True*)
Scan for devices connected to the current host device via RS-485 daisy-chaining.

reassign controls how device addresses are assigned during the scan; can be `"none"` (keep current values; can lead to conflicts if several devices have the same address), `"conflict"` (change conflicting addresses), or `"all"` (assigned all new addresses in sequence starting from the host)

If `sync==True`, wait until the scan is done (might take several seconds).

get_addr_map()
Get address map for devices connected to the current host device via RS-485 daisy-chaining.

Return tuple `(addresses, conflict)`, where `addresses` is the list of all device addresses, and `conflict==True` if there address conflicts (several devices having the same address).

wait_for_scan(*timeout=10.0*)
Wait for the device connection scan to finish

get_addr(*addr=None*)
Get RS-485 address of the given device (host if *addr* is `None`)

set_addr(*new_addr*, *addr=None*)
Set RS-485 address of the given device (host if *addr* is `None`)

get_ethernet_parameters(*addr=None*)
Get Ethernet connection parameters.

Return tuple `(hostname, ipaddr, ipmode, gateway, netmask)`.

setup_ethernet(*hostname=None*, *ipmode=None*, *ipaddr=None*, *gateway=None*, *netmask=None*, *addr=None*)
Setup Ethernet connection parameters.

Any `None` value remains unchanged. Note that these settings only take effect after saving parameters to the memory (`save_parameters()`) and restarting the device (`reset()`). If the connection is made through Ethernet, then it will likely be invalidated, in which case a new device object with the updated parameters should be created after reset.

autodetect_motors(*addr=None*)
Autodetect connected motors.

The command involves sending single-step commands to the motors, so it requires all axes to be stopped, and it might slightly affect the current position. After the detection the types can be stored in the memory via `save_parameters()`.

get_motor_type(*axis='all'*, *addr=None*)
Get type of the given axis motor

set_motor_type(*axis='all'*, *motor_type='standard'*, *addr=None*)
Manually set type of the given axis motor

move_to(*axis*, *position*, *addr=None*)
Move to a given position

move_by(*axis*, *steps=1*, *addr=None*)
Move by a given number of steps

get_position(*axis='all'*, *addr=None*)
Get the current axis position

set_position_reference(*axis*, *position=0*, *addr=None*)
Set the current axis position as a reference (the actual motor position stays the same)

jog(*axis*, *direction*, *addr=0*)
Jog a given axis in a given direction.

direction can be either `"-"` (negative) or `"+"` (positive). The motion continues until it is explicitly stopped.

is_moving(*axis='all'*, *addr=None*)
Check if the axis is moving

wait_move(*axis='all'*, *addr=None*)
Wait until axis motion is done

stop(*axis='all'*, *immediate=False*, *addr=None*)
Stop motion of a given axis.

If `immediate==True` make an abrupt stop; otherwise, slow down gradually. Note that immediate stop has to stop all axes simultaneously, so it only takes `axis=="all"`.

get_velocity_parameters(*axis='all'*, *addr=None*)
Return velocity parameters `(speed, accel)` for the given axis and controller.

`speed` and `accel` denote, correspondingly, maximal (i.e., steady regime) moving speed and acceleration in steps/s and steps/s^2.

setup_velocity(*axis='all'*, *speed=None*, *accel=None*, *addr=None*)
Setup velocity parameters `(speed, accel)` for the given axis and controller.

`speed` and `accel` denote, correspondingly, maximal (i.e., steady regime) moving speed and acceleration in steps/s and steps/s^2. `None` values are left unchanged.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_all_axes()
Get the list of all available axes (taking mapping into account)

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

remap_axes (*mapping*, *accept_original=True*)
Rename axes to the new labels.

mapping is the new axes mapping, which can be a list of new axes name (corresponding to the old axes in order returned by `get_all_axes()`), or a dictionary `{alias: original}` of the new axes aliases.

set_device_variable (*key*, *value*)
Set the value of a settings parameter

unlock ()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.OZOptics package

Submodules

pylablib.devices.OZOptics.base module

exception pylablib.devices.OZOptics.base.**OZOpticsError**
Bases: `pylablib.core.devio.base.DeviceError`

Generic OZOptics devices error

args

with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.OZOptics.base.**OZOpticsBackendError** (*exc*)
Bases: `pylablib.devices.OZOptics.base.OZOpticsError`, `pylablib.core.devio.comm_backend.DeviceBackendError`

Generic OZOptics backend communication error

args

with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.OZOptics.base.**OZOpticsDevice** (*conn*, *timeout=20.0*)
Bases: `pylablib.core.devio.comm_backend.ICommBackendWrapper`

Generic OZOptics device.

Parameters conn – serial connection parameters (usually port or a tuple containing port and baudrate)

Error
alias of `OZOpticsError`

query (*comm*, *prefix=None*, *prefix_line=None*, *timeout=None*)
Query the device.

If *prefix* is not `None`, it can specify a string which should be at the beginning of the *prefix_line* line of the reply. If it is present, it is removed and the rest of that line is returned; otherwise, an error is raised. If *prefix_line* is `None`, return the first reply line beginning with the given prefix value (or raise an error if not such line is present).

restart()
: Restart the device

get_config()
: Get device configuration

class NoParameterCaller(*device*, *kind*)
: Bases: `object`

  Class to simplify calling functions without a parameter

apply_settings(*settings*)
: Apply the settings.

  settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
: Close the backend

get_device_variable(*key*)
: Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
: Get dict `{name: value}` containing full device information (including status and settings).

  include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
: Get dict `{name: value}` containing the device status (including settings).

  include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
: Get dict `{name: value}` containing all the device settings.

  include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
: Check if the device is connected

lock(*timeout=None*)
: Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
: Context manager for lock & unlock

open()
: Open the backend

set_device_variable(*key*, *value*)
: Set the value of a settings parameter

unlock()
: Unlock the access to the device from other threads/processes (isn't necessarily implemented)

class pylablib.devices.OZOptics.base.**TF100**(*conn*, *timeout=20.0*)
: Bases: *pylablib.devices.OZOptics.base.OZOpticsDevice*

OZOptics TF100 tunable filter.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

get_wavelength_correction()
Get the current wavelength correction parameters `(shift, scale)`.

The relation between the set/get wavelength and the wavelength set to the device is calculated as `device_wavelength = set_wavelength*scale + shift`

set_wavelength_correction(*shift=0.0*, *scale=1.0*)
Set the wavelength correction parameters.

The relation between the set/get wavelength and the wavelength set to the device is calculated as `device_wavelength = set_wavelength*scale + shift`

home()
Home the motor (needs to be called first after startup)

get_wavelength()
Get the currently set wavelength (or `None` if unknown / not homed)

set_wavelength(*wavelength*)
Set the current wavelength

Error
alias of *OZOpticsError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_config()
Get device configuration

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

query(*comm*, *prefix=None*, *prefix_line=None*, *timeout=None*)
Query the device.

If *prefix* is not `None`, it can specify a string which should be at the beginning of the *prefix_line* line of the reply. If it is present, it is removed and the rest of that line is returned; otherwise, an error is raised. If *prefix_line* is `None`, return the first reply line beginning with the given prefix value (or raise an error if not such line is present).

restart()
Restart the device

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

class pylablib.devices.OZOptics.base.**DD100**(*conn*, *timeout=20.0*)
Bases: *pylablib.devices.OZOptics.base.OZOpticsDevice*

OZOptics DD100 variable attenuator.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

home()
Home the motor (needs to be called first after startup)

get_min_attenuation()
Get the minimal possible attenuation (i.e., insertion loss)

get_max_attenuation()
Get the maximal possible possible attenuation in dB

get_attenuation()
Get the current attenuation in dB

set_attenuation(*att*)
Set the current attenuation in dB

Error
alias of *OZOpticsError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_config()
Get device configuration

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

query(*comm*, *prefix=None*, *prefix_line=None*, *timeout=None*)
Query the device.

If *prefix* is not `None`, it can specify a string which should be at the beginning of the *prefix_line* line of the reply. If it is present, it is removed and the rest of that line is returned; otherwise, an error is raised. If *prefix_line* is `None`, return the first reply line beginning with the given prefix value (or raise an error if not such line is present).

restart()
Restart the device

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

class pylablib.devices.OZOptics.base.**EPC04**(*conn*, *timeout=20.0*)
Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*

OZOptics EPC04 polarization controller.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

Error
alias of *OZOpticsError*

query(*comm*)

get_voltages()
Get all voltages

set_voltage(*channel*, *voltage*)
Set voltage at a given channel (0 through 3)

set_all_voltages(*voltages*)
Set all channel voltages.

voltages is a list of size 4 containing the voltage values.

step_voltage(*channel*, *step*)
Step voltage at the given channel by the given step

get_mode()
Get current operating mode.

Can be "dc" (constant voltage) or "ac" (scrambling).

set_mode(*mode='dc'*)
Set current operating mode.

Can be "dc" (constant voltage) or "ac" (scrambling).

get_frequencies()
Get all scrambling frequencies

set_frequency(*channel*, *frequency*)
Set scrambling frequency a given channel (0 through 3)

set_all_frequencies(*frequencies*)
Set all channel scrambling frequencies.

frequencies is a list of size 4 containing the frequency values.

get_waveform()
Get current scrambling waveform.

Can be "sin" (sine wave) or "tri" (triangle wave).

set_waveform(*waveform*)
Set current scrambling waveform.

Can be "sin" (sine wave) or "tri" (triangle wave).

save_preset()
Save current state as a power-up preset

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.Ophir package

Submodules

pylablib.devices.Ophir.base module

exception pylablib.devices.Ophir.base.**OphirError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Ophir device error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Ophir.base.**OphirBackendError**(*exc*)
Bases: *pylablib.devices.Ophir.base.OphirError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Generic Ophir backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.Ophir.base.**OphirDevice**(*conn*)
Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*

Generic Ophir device.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

Error
alias of *OphirError*

query(*comm*)
Send a query to the device and parse the reply

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

class pylablib.devices.Ophir.base.**THeadInfo**(*type*, *serial*, *name*, *capabilities*)
Bases: `tuple`

capabilities

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

name

serial

type

class pylablib.devices.Ophir.base.**TDeviceInfo**(*id*, *serial*, *name*, *rom_version*)
Bases: `tuple`

count()
Return number of occurrences of value.

id

index()
Return first index of value.

Raises ValueError if the value is not present.

name

rom_version

serial

class pylablib.devices.Ophir.base.**TWavelengthInfo**(*mode*, *rng*, *curr_idx*, *presets*, *curr_wavelength*)

Bases: tuple

count()

Return number of occurrences of value.

curr_idx

curr_wavelength

index()

Return first index of value.

Raises ValueError if the value is not present.

mode

presets

rng

class pylablib.devices.Ophir.base.**TRangeInfo**(*curr_idx*, *ranges*, *curr_range*)

Bases: tuple

count()

Return number of occurrences of value.

curr_idx

curr_range

index()

Return first index of value.

Raises ValueError if the value is not present.

ranges

class pylablib.devices.Ophir.base.**VegaPowerMeter**(*conn*)

Bases: *pylablib.devices.Ophir.base.OphirDevice*

Ophir Vega power meter.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

get_head_info()

Get head information.

Return tuple (type, serial, name, capabilities).

get_device_info()

Get device information.

Return tuple (id, serial, name, rom_version).

reset()

Reset the device

get_power()

Get the current power readings.

Return either measured power, or "over", if the power is overrange.

get_energy()
Get the current energy readings.

Return either measured energy, or `"over"`, if the energy is overrange.

get_frequency()
Get the current frequency readings.

Return either measured frequency, or `"over"`, if the power is overrange.

get_units()
Get device reading units

get_wavelength_info()
Get wavelength setting info.

Return tuple `(mode, rng, curr_idx, presets, curr_wavelength)`, where *mode* is the measurement mode (`"continuous"` or `"discrete"`), *rng* is a 2-tuple with the full wavelength range (in m) for continuous mode or a set of all wavelengths for discrete mode, *curr_idx* is the current wavelength preset index, *presets* is the list of all preset wavelengths (in m) for continuous mode or a set of all wavelengths for discrete mode, and *curr_wavelength* is the current measurement wavelength (in m) for continuous mode or the current wavelength name for discrete mode.

get_wavelength()
Get current wavelength

set_wavelength(*wavelength*)
Set current wavelength.

wavelength is either a wavelength (in m) for the continuous mode, or a wavelength preset (as a string) for a discrete mode.

get_range_info()
Get power range info.

Return tuple `(curr_idx, ranges, curr_range)`, where *curr_idx* is the current power range index, *ranges* is the list of ranges (in W) for all indices and *curr_range* is the current range (in W).

get_range()
Get current power range (maximal power in W)

get_range_idx()
Get current power range index

Index goes from 0 (highest) to maximal (lowest); auto-ranging is -1.

set_range_idx(*rng_idx*)
Set current range index.

rng_idx is the range index from 0 (highest) to maximal (lowest); auto-ranging is -1. The corresponding ranges are given by `get_range_info()`.

get_battery_condition()
Check if the batter is OK

get_baudrate()
Get current baud rate

get_supported_baudrates()
Get a list of all supported baud rates

set_baudrate(*baudrate*)
Set current baud rate.

If the baudrate is different from the current one, close the device connection. The device object will need to be re-created with the newly specified baud rate.

is_filter_in()
Check if the filter is set to be on at the power meter

set_filter(*filter_in=True*)
Change the filter setting at the power meter (on or off)

is_diffuser_in()
Check if the diffuser is set to be on at the power meter

set_diffuser(*diffuser_in=True*)
Change the diffuser setting at the power meter (on or off)

Error
alias of `OphirError`

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

query(*comm*)
Send a query to the device and parse the reply

set_device_variable(*key, value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.PCO package

Submodules

pylablib.devices.PCO.SC2 module

pylablib.devices.PCO.SC2.**list_cameras**(*cam_interface=None*)
List camera connections (interface kind and camera index).

If *cam_interface* is supplied, it defines one of camera interfaces to check (e.g., `"usb3"` or `"clhs"`). Otherwise, check all interfaces.

pylablib.devices.PCO.SC2.**get_cameras_number**(*cam_interface=None*)
Get the total number of connected PCOSC2 cameras.

If *cam_interface* is supplied, it defines one of camera interfaces to check (e.g., `"usb3"` or `"clhs"`). Otherwise, check all interfaces.

pylablib.devices.PCO.SC2.**reset_api**()
Reset API.

All cameras must be closed; otherwise, the prompt to reboot will appear.

class pylablib.devices.PCO.SC2.**TDeviceInfo**(*model, interface, sensor, serial_number*)
Bases: `tuple`

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

interface

model

sensor

serial_number

class pylablib.devices.PCO.SC2.**TCameraStatus**(*status, warnings, errors*)
Bases: `tuple`

count()
Return number of occurrences of value.

errors

index()
Return first index of value.

Raises ValueError if the value is not present.

status

warnings

class pylablib.devices.PCO.SC2.**TInternalBufferStatus**(*scheduled*, *scheduled_max*)
Bases: `tuple`

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

scheduled

scheduled_max

class pylablib.devices.PCO.SC2.**TFrameInfo**(*frame_index*, *raw_metadata*)
Bases: `tuple`

count()
Return number of occurrences of value.

frame_index

index()
Return first index of value.

Raises ValueError if the value is not present.

raw_metadata

class pylablib.devices.PCO.SC2.**PCOSC2Camera**(*idx=0*, *cam_interface=None*, *reboot_on_fail=True*)
Bases: `pylablib.devices.interface.camera.IBinROICamera`, `pylablib.devices.interface.camera.IExposureCamera`

PCO SC2 camera.

Parameters

- **idx** (*int*) – camera index (use `get_cameras_number()` to get the total number of connected cameras)
- **cam_interface** – camera interface; if it is `None`, get the first available connected camera (in this case *idx* is ignored); if not, then value of *idx* is used to connect to a particular camera (interfaces and indices can be obtain from `list_cameras()`)
- **reboot_on_fail** (*bool*) – if `True` and the camera raised an error during initialization (but after opening), reboot the camera and try to connect again useful when the camera is in a broken state (e.g., wrong ROI or pixel clock settings)

Error = <Mock name='mock.PCOSC2Error' id='140318697119248'>

TimeoutError = <Mock spec='str' id='140318678559376'>

open()
Open connection to the camera

close()
Close connection to the camera

is_opened()
Check if the device is connected

reboot(*wait=True*)
Reboot the camera.

If `wait==True`, wait for the recommended time (10 seconds) after reboot for the camera to fully restart; attempt to open the camera before that can lead to an error.

get_full_camera_data()
Get a dictionary the all camera data available through the SDK

update_full_data()
Update internal full camera data settings.

Takes some time (about 50ms), so more specific function are preferable for specific parameters.

get_device_info()
Get camera model data.

Return tuple `(model, interface, sensor, serial_number)`.

get_capabilities()
Get camera capabilities.

For description of the capabilities, see PCO SC2 manual.

get_camera_status(*full=False*)
Get camera status.

If `full==True`, return current camera status as a set of enabled status states; otherwise, return tuple `(status, warnings, errors)` with additional information about warnings and error.

get_temperature()
Get the current camera temperature

Return tuple `(CCD, cam, power)` with temperatures of the sensor, camera, and power supply respectively.

get_conversion_factor()
Get camera conversion factor (electrons per pixel value)

get_trigger_mode()
Get current trigger mode (see `set_trigger_mode()` for description)

set_trigger_mode(*mode*)
Set trigger mode.

Can be `"int"` (internal), `"software"` (software), `"ext"` (external+software), `"ext_exp"` (external exposure), `"ext_sync"` (external PLL sync), `"ext_exp_fast"` (fast external exposure), `"ext_cds"` (external CDS control), `"ext_exp_slow"` (slow external exposure)‘, or `"ext_sync_hdsdi"` (external synchronized SD/HDI).

For description, see PCO SDK manual.

send_software_trigger()
Send software trigger signal

class Buffer(*size*, *metadata_size=0*)
Bases: `object`

Single frame buffer object, which controls setup, cleanup, and synchronization

wait(*timeout*)

reset()

release()

get_internal_buffer_status()
Get the status of the internal smaller API buffer, showing the number of scheduled frames there, and the maximal number that can be scheduled

set_exposure(*exposure*)
Set camera exposure

get_exposure()
Get current exposure

set_frame_delay(*frame_delay*)
Set camera frame delay

get_frame_delay()
Get current frame delay

set_frame_period(*frame_time=0*, *adjust_exposure=False*)
Set frame time (frame acquisition period).

If the time can't be achieved even with zero frame delay and `adjust_exposure==True`, try to reduce the exposure to get the desired frame time; otherwise, keep the exposure the same.

get_frame_period()
Get current frame time (frame acquisition period)

get_frame_timings()
Get acquisition timing.

Return tuple `(exposure, frame_period)`.

get_pixel_rate()
Get camera pixel rate (in Hz)

get_available_pixel_rates()
Get all available pixel rates

set_pixel_rate(*rate=None*)
Set camera pixel rate (in Hz)

The rate is always rounded to the closest available. If *rate* is `None`, set the maximal possible rate.

setup_acquisition(*nframes=100*)
Setup acquisition.

nframes determines number of size of the ring buffer (by default, 100).

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:``setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition.

Clears buffers as well, so any readout afterwards is impossible.

acquisition_in_progress()
Check if the acquisition is in progress

clear_acquisition()
Clear acquisition settings

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend, hbin, vbin)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0), *hbin* and *vbin* specify binning.

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*, *hbin=1*, *vbin=1*, *symmetric=False*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0), *hbin* and *vbin* specify binning. By default, all non-supplied parameters take extreme values (0 for start, maximal for end, 1 for binning). If `symmetric==True` and camera requires symmetric ROI (see `requires_symmetric_roi()`), respect this symmetry in the resulting ROI; otherwise, try to use software ROI feature to set up the required ranges (note: while software ROI does affect the size of the read out frame, it does not change the readout time, which would be the same as with `symmetric==True`).

requires_symmetric_roi()
Check if the camera requires horizontally or vertically symmetric ROI.

Return a tuple `(horizontal, vertical)`. If `True`, one might still set up an asymmetric ROI for some cameras using the software ROI feature, but it does not affect camera readout rate

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning. In some cameras, the step and the minimal size depend on the binning, which can be supplied.

enable_pixel_correction(*enable=True*)
Enable or disable hotpixel correction

is_pixel_correction_enabled()
Check if hotpixel correction is enabled

get_noise_filter_mode()
Get the noise filter mode (for details, see `set_noise_filter_mode()`)

set_noise_filter_mode(*mode='on'*)
Set the noise filter mode.

Can be `"off"`, `"on"`, or `"on_hpc"` (on + hot pixel correction).

set_status_line_mode(*binary=True*, *text=False*)
Set status line mode.

binary determines if the binary line is present (it occupies first 14 pixels of the image). *text* determines if the text line is present (it is plane text timestamp, which takes first 8 rows and about 300 columns).

It is recommended to always have *binary* option on, since it is used to determine frame index for checking if there are any missing frames.

get_status_line_mode()

Get status line mode.

Return tuple (binary, text) (see *set_status_line_mode()* for description)

get_bit_alignment()

Get data bit alignment

Can be "LSB" (normal alignment) or "MSB" (if camera data is less than 16 bit, it is padded with zeros on the right to match 16 bit).

set_bit_alignment(*mode*)

Get data bit alignment

Can be "LSB" (normal alignment) or "MSB" (if camera data is less than 16 bit, it is padded with zeros on the right to match 16 bit).

set_metadata_mode(*mode=True*)

Set metadata mode

get_metadata_mode()

Get metadata mode.

Return tuple (enabled, size, version)

FrameTransferError

alias of *pylablib.devices.interface.camera.DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)

Bases: object

Class to simplify calling functions without a parameter

apply_settings(*settings*)

Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()

Get acquisition parameters.

Return dictionary {name: value}

get_data_dimensions()

Get readout data dimensions (in pixels) as a tuple (width, height); take indexing mode into account

get_device_variable(*key*)

Get the value of a settings, status, or full info parameter

get_frame_format()

Get format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array).

get_frame_info_fields()

Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is "list" or "array".

get_frame_info_format()

Get format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple (acquired, unread, skipped, size), where acquired is the total number of acquired frames, unread is the number of acquired but not read frames, skipped is the number of skipped (not read and then written over) frames, and buffer_size is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple (first, last) with images range (first inclusive). If no images are available, return None. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if None, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not None, it specifies the fields included for non-"tuple" formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"‘‘ (from the last read frame), "lastwait" (wait for the last successful wait_for_frame() call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

class pylablib.devices.PCO.SC2.**TStatusLine**(*framestamp*)
Bases: tuple

count()
Return number of occurrences of value.

framestamp

index()
Return first index of value.

Raises ValueError if the value is not present.

pylablib.devices.PCO.SC2.**get_status_line**(*frame*)
Get frame info from the binary status line.

Assume that the status line is present; if it isn't, the returned frame info will be a random noise.

Module contents

pylablib.devices.Pfeiffer package

Submodules

pylablib.devices.Pfeiffer.base module

exception pylablib.devices.Pfeiffer.base.**PfeifferError**

Bases: *pylablib.core.devio.base.DeviceError*

Generic Pfeiffer device error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Pfeiffer.base.**PfeifferBackendError**(*exc*)

Bases: *pylablib.devices.Pfeiffer.base.PfeifferError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Generic Pfeiffer backend communication error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.Pfeiffer.base.**TTPG260SwitchSettings**(*channel*, *low_thresh*, *high_thresh*)

Bases: tuple

channel

count()

Return number of occurrences of value.

high_thresh

index()

Return first index of value.

Raises ValueError if the value is not present.

low_thresh

class pylablib.devices.Pfeiffer.base.**TTPG260GaugeControlSettings**(*activation_control*, *deactivation_control*, *on_thresh*, *off_thresh*)

Bases: tuple

activation_control

count()

Return number of occurrences of value.

deactivation_control

index()

Return first index of value.

Raises ValueError if the value is not present.

off_thresh

on_thresh

class pylablib.devices.Pfeiffer.base.**TPG260**(*conn*)
Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*

TPG260 series (TPG261/262) pressure gauge.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

Error
alias of *PfeifferError*

comm(*msg*)
Send a command to the device

query(*msg*, *data_type='str'*)
Send a query to the device and return the reply

get_units()
Get device units for indication/reading ("mbar", "torr", or "pa")

set_units(*units*)
Set device units for indication/reading ("mbar", "torr", or "pa")

to_Pa(*value*, *units=None*)
Convert value in the given units to Pa.

If *units* is None, use the current display units.

from_Pa(*value*, *units=None*)
Convert value in the given units from Pa.

If *units* is None, use the current display units.

get_display_channel()
Get controller display channel

set_display_channel(*channel=1*)
Set controller display channel

get_display_resolution()
Get controller display resolution (number of digits)

set_display_resolution(*resolution=2*)
Set controller display resolution (number of digits)

is_enabled(*channel=1*)
Check if the gauge at the given channel is enabled.

If the gauge cannot be turned on/off (e.g., not connected), return None.

enable(*enable=True*, *channel=1*)
Enable or disable the gauge at the given channel

get_channel_status(*channel=1*)
Get channel status.

Can be "ok", "under" (underrange), "over" (overrange), "sensor_error", "sensor_off", "no_sensor", or "id_error".

get_pressure(*channel=1*, *display_units=False*, *status_error=True*)
Get pressure at a given channel.

If display_units==False, return result in Pa; otherwise, use display units obtained using *get_units()*. If status_error==True and the channel status is not "ok", raise and error; otherwise, return None.

get_gauge_kind(*channel=1*)

get_measurement_filter(*channel=1*)
Get gauge measurement filter ("fast", "medium", or "slow")

set_measurement_filter(*meas_filter*, *channel=1*)
Set gauge measurement filter ("fast", "medium", or "slow")

get_calibration_factor(*channel=1*)
Get gauge calibration factor

set_calibration_factor(*coefficient*, *channel=1*)
Set gauge calibration factor

get_switch_settings(*switch_function*)
Get settings for the given switch function (between 1 and 4).

Return tuple `(channel, low_thresh, high_thresh)`. The thresholds are given in Pa.

setup_switch(*switch_function*, *channel*, *low_thresh*, *high_thresh*)
Get settings for the given switch function (between 1 and 4).

Return tuple `(channel, low_thresh, high_thresh)`. The thresholds are given in Pa.

get_switch_status()
Return status of the 4 switch functions

get_gauge_control_settings(*channel*)
Get settings for the gauge control on the given channel.

Return tuple `(activation_control, deactivation_control, on_thresh, off_thresh)`. The thresholds are given in Pa.

setup_gauge_control(*channel*, *activation_control*, *deactivation_control*, *on_thresh*, *off_thresh*)
Setup gauge control on the given channel.

Return tuple `(activation_control, deactivation_control, on_thresh, off_thresh)`. The thresholds are given in Pa.

get_current_errors()
Get a list of all present error messages.

If there are no errors, return a single-element list `["no_error"]`.

reset_error()
Cancel currently active errors and return to measurement mode.

Return the list of currently present errors. If there are no errors, return a single-element list `["no_error"]`.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

class pylablib.devices.Pfeiffer.base.**DPG202**(*conn*)
Bases: `pylablib.core.devio.comm_backend.ICommBackendWrapper`

DPG202/TPG202 control unit.

Parameters conn – serial connection parameters (usually port or a tuple containing port and baudrate)

Error
alias of `PfeifferError`

query(*parameter*, *value='=?'*, *action=0*, *address=1*, *send_type=None*, *recv_type=None*)
Send a query to the device and parse the reply.

Parameters

- **parameter** – parameter number
- **value** – value to send (`"=?"` for a value request)
- **action** – request action (0 for value request, 1 for a command)
- **address** – unit address

- **send_type** – data type for the sent value (ignored for value requests)
- **recv_type** – data type for the received value (None means returning a raw string value)

get_value(*parameter*, *data_type*, *address=1*)
Send a data request to the device.

Parameters

- **parameter** – parameter number
- **data_type** – data type for the received value
- **address** – unit address

comm(*parameter*, *value*, *data_type*, *address=1*)
Send a control command to the device.

Parameters

- **parameter** – parameter number
- **value** – associated command value
- **data_type** – data type for the sent value
- **address** – unit address

get_pressure(*address=1*)
Get pressure at a given unit address

get_error_code(*address=1*)
Get the current error code at a given unit address

get_software_version(*address=1*)
Get the software version at a given unit address

get_device_name(*address=1*)
Get the name of the gauge at a given unit address

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key, value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.PhotonFocus package

Submodules

pylablib.devices.PhotonFocus.PhotonFocus module

class pylablib.devices.PhotonFocus.PhotonFocus.**LibraryController**(*lib*)
Bases: *pylablib.devices.utils.load_lib.LibraryController*

close(*opid*)
Mark device closing.

Return tuple `(close_result, uninit_result)` with the results of the closing and the shutdown. If library does not need to be shut down yet, set `uninit_result=None`

open()
Mark device opening.

Return tuple `(init_result, open_result, opid)` with the results of the initialization and the opening, and the opening ID which should afterwards be used for closing. If library is already initialized, set `init_result=None`

preinit()
Pre-initialize the library, if it hasn't been done already

shutdown()
Close all opened connections and shutdown the library

temp_open()
Context for temporarily opening a new device connection

pylablib.devices.PhotonFocus.PhotonFocus.**query_camera_name**(*port*)
Query cameras name at a given port in PFCam interface

class pylablib.devices.PhotonFocus.PhotonFocus.**TCameraInfo**(*manufacturer*, *port*, *version*, *type*)
Bases: `tuple`

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

manufacturer

port

type

version

pylablib.devices.PhotonFocus.PhotonFocus.**list_cameras**(*only_supported=True*)
List all cameras available through PFCam interface.

If `only_supported==True`, only return cameras which support PFCam protocol (this check only works if the camera is not currently accessed by some other software). Return a list `[(port, info)]`, where `port` is the pfcam port given to *`IPhotonFocusCamera`* and its subclasses, and `info` is the information returned by *`query_camera_name()`*.

pylablib.devices.PhotonFocus.PhotonFocus.**get_cameras_number**(*only_supported=True*)
Get the total number of connected PFCam cameras

pylablib.devices.PhotonFocus.PhotonFocus.**get_port_index**(*manufacturer*, *port*)
Find PhotonFocus port index based on the manufacturer and port

class pylablib.devices.PhotonFocus.PhotonFocus.**PFCamAttribute**(*port*, *name*)
Bases: `object`

PFCam camera attribute.

Allows to query and set values and get additional information. Usually created automatically by a PhotonFocus camera instance, but could also be created manually.

Parameters

- **sid** – camera session ID
- **name** – attribute text name

name
attribute name

readable
whether attribute is readable

Type bool

writable
whether attribute is writable

Type bool

is_command
whether attribute is a command

Type bool

min
minimal attribute value (if applicable)

Type float or int

max
maximal attribute value (if applicable)

Type float or int

values
list of possible attribute values (if applicable)

update_limits()
Update minimal and maximal attribute limits and return tuple `(min, max)`

truncate_value(*value*)
Truncate value to lie within attribute limits

get_value(*enum_as_str=True*)
Get attribute value.

If `enum_as_str==True`, return enum-style values as strings; otherwise, return corresponding integer values.

set_value(*value*, *truncate=True*)
Get attribute value.

If `truncate==True`, automatically truncate value to lie within allowed range.

call_command(*arg=0*)
If attribute is a command, call it with a given argument; otherwise, raise an error

class pylablib.devices.PhotonFocus.PhotonFocus.**TDeviceInfo**(*model*, *serial_number*, *grabber_info*)

Bases: `tuple`

count()
Return number of occurrences of value.

grabber_info

index()
Return first index of value.

Raises ValueError if the value is not present.

model

serial_number

class pylablib.devices.PhotonFocus.PhotonFocus.**IPhotonFocusCamera**(*pfcam_port=0*, ***kwargs*)

Bases: `pylablib.devices.interface.camera.IAttributeCamera`

Generic PFCam interface to a PhotonFocus camera. Does not handle frames acquisition, so needs to be mixed with a frame grabber class to be fully operational. In this mixing, the class attribute `GrabberClass` should be set to this frame grabber class.

Parameters

- **pfcam_port** – port number for pfcam interface (can be learned by *list_cameras()*; port number is the first element of the camera data tuple) can also be a tuple `(manufacturer, port)`, e.g., `("National Instruments", "port0")`.
- **kwargs** – keyword arguments passed to the frame grabber initialization

Error
alias of *pylablib.core.devio.base.DeviceError*

GrabberClass = None

setup_max_baudrate()
Setup the maximal available baudrate

get_baudrate()
Get the current baud rate

open()
Open connection to the camera

close()
Close connection to the camera

get_attribute_value(*name*, *error_on_missing=True*, *default=None*)
Get value of an attribute with the given name.

If the value doesn't exist or can not be read and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, assume that `error_on_missing==False`. If *name* points at a dictionary branch, return a dictionary with all values in this branch.

set_attribute_value(*name*, *value*, *truncate=True*, *error_on_missing=True*)
Set value of an attribute with the given name.

If the value doesn't exist or can not be written and `error_on_missing==True`, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). If `truncate==True`, truncate value to lie within attribute range.

get_all_attribute_values(*root=''*)
Get values of all attributes with the given *root*

set_all_attribute_values(*settings*, *root=''*, *truncate=True*)
Set values of all attributes with the given *root*.

If `truncate==True`, truncate value to lie within attribute range.

update_attribute_value(*name*, *value*, *error_on_missing=True*, *truncate=True*)
Set value of the attribute with a given name, but only if it's different from the current value.

Can take less time on some version of PFRemote (where single attribute setting is about 50ms). Arguments are the same as *set_attribute_value()*.

call_command(*name*, *arg=0*, *error_on_missing=True*)
Execute the given command with the given argument.

If the command doesn't exist and `error_on_missing==True`, raise error; otherwise, do nothing.

get_device_info()
Get camera model data.

Return tuple (model, serial_number, grabber_info).

get_detector_size()
Get camera detector size (in pixels) as a tuple (width, height)

get_roi()
Get current ROI.

Return tuple (hstart, hend, vstart, vend).

fast_shift_roi(*hstart=0*, *vstart=0*)
Shift ROI by only changing its origin, but keeping the shape the same.

Note that if the ROI is invalid, it won't be truncated (as is the standard behavior of set_roi()), which might lead to errors later.

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

By default, all non-supplied parameters take extreme values.

get_roi_limits(*hbin=1*, *vbin=1*)

get_exposure()
Get current exposure

set_exposure(*exposure*)
Set current exposure

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

set_frame_period(*frame_period*)
Set frame period (time between two consecutive frames in the internal trigger mode)

get_frame_timings()
Get acquisition timing.

Return tuple (exposure, frame_period).

is_CFR_enabled()
Check if the constant frame rate mode is enabled

enable_CFR(*enabled=True*)
Enable constant frame rate mode

get_trigger_interleave()
Check if the trigger interleave is on

set_trigger_interleave(*enabled*)
Set the trigger interleave option on or off

is_status_line_enabled()
Check if the status line is on

enable_status_line(*enabled=True*)
Enable or disable status line

get_black_level_offset()
Get the black level offset

set_black_level_offset(*offset*)
Set the black level offset

FrameTransferError
alias of *pylablib.devices.interface.camera.DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

TimeoutError
alias of *pylablib.core.devio.base.DeviceError*

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

clear_acquisition()
Clear acquisition settings

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary {name: value}

get_all_attributes(*copy=False*)
Return a dictionary of all available attributes.

If copy==True, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple (width, height); take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is "list" or "array".

get_frame_info_format()
Get format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff

readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

setup_acquisition(***kwargs*)
Setup acquisition.

Any non-specified acquisition parameters are assumed to be the same as previously set (or default, if not explicitly set before). Return the new acquisition parameters.

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is None). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be “lastread”‘‘ (from the last read frame), "lastwait" (wait for the last successful *wait_for_frame()* call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

class pylablib.devices.PhotonFocus.PhotonFocus.**PhotonFocusIMAQCamera**(*imaq_name='img0'*, *pfcam_port=0*)
Bases: *pylablib.devices.PhotonFocus.PhotonFocus.IPhotonFocusCamera*, *pylablib.devices.IMAQ.IMAQ.IMAQFrameGrabber*

IMAQ+PFCam interface to a PhotonFocus camera.

Parameters

- **imaq_name** – IMAQ interface name (can be learned by *IMAQ.list_cameras()*; usually, but not always, starts with "img")
- **pfcam_port** – port number for pfcam interface (can be learned by *list_cameras()*; port number is the first element of the camera data tuple) can

also be a tuple `(manufacturer, port)`, e.g., `("National Instruments", "port0")`.

Error
alias of *pylablib.core.devio.base.DeviceError*

GrabberClass
alias of *pylablib.devices.IMAQ.IMAQ.IMAQFrameGrabber*

open()
Open connection to the camera

FrameTransferError
alias of *pylablib.devices.interface.camera.DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

TimeoutError = <Mock spec='str' id='140318667689104'>

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

call_command(*name*, *arg=0*, *error_on_missing=True*)
Execute the given command with the given argument.

If the command doesn't exist and `error_on_missing==True`, raise error; otherwise, do nothing.

clear_acquisition()
Clear all acquisition details and free all buffers

clear_all_triggers(*reset_acquisition=True*)
Disable all triggers of the session

If the input triggers configuration has been changed, acquisition needs to be restart; if `reset_acquisition==True`, perform it automatically.

close()
Close connection to the camera

configure_trigger_in(*trig_type*, *trig_line=0*, *trig_pol='high'*, *trig_action='none'*, *timeout=None*, *reset_acquisition=True*)
Configure input trigger.

Parameters

- **trig_type** (*str*) – trigger source type; can be `"ext"`, `"rtsi"`, `"iso_in"`, or `"software"`
- **trig_line** (*int*) – trigger line number
- **trig_pol** (*str*) – trigger polarity; can be `"high"` or `"low"`
- **trig_action** (*str*) – trigger action; can be `"none"` (disable trigger), `"capture"` (start capturing), `"stop"` (stop capturing), `"buffer"` (capture a single frame), or `"bufflist"` (capture the whole buffer list once)
- **timeout** (*float*) – timeout in seconds; `None` means not timeout.

- **reset_acquisition** (*bool*) – if the input triggers configuration has been changed, acquisition needs to be restart; if `True`, perform it automatically

configure_trigger_out (*trig_type*, *trig_line=0*, *trig_pol='high'*, *trig_drive='disable'*)
Configure trigger output.

Parameters

- **trig_type** (*str*) – trigger drive destination type; can be `"ext"`, `"rtsi"`, or `"iso_out"`
- **trig_line** (*int*) – trigger line number
- **trig_pol** (*str*) – trigger polarity; can be `"high"` or `"low"`
- **trig_drive** (*str*) – trigger output signal; can be `"disable"` (disable drive), `"acq_in_progress"` (asserted when acquisition is started), `"acq_done"` (asserted when acquisition is done), `"unasserted"` (force unasserted level), `"asserted"` (force asserted level), `"hsync"` (asserted on start of a single line start), `"vsync"` (asserted on start of a frame scan), `"frame_start"` (asserted when a single frame is captured), or `"frame_done"` (asserted when a single frame is done)

enable_CFR (*enabled=True*)
Enable constant frame rate mode

enable_status_line (*enabled=True*)
Enable or disable status line

fast_shift_roi (*hstart=0*, *vstart=0*)
Shift ROI by only changing its origin, but keeping the shape the same.

Note that if the ROI is invalid, it won't be truncated (as is the standard behavior of `set_roi()`), which might lead to errors later.

get_acquisition_parameters ()
Get acquisition parameters.

Return dictionary `{name: value}`

get_all_attribute_values (*root=''*)
Get values of all attributes with the given *root*

get_all_attributes (*copy=False*)
Return a dictionary of all available attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_all_grabber_attribute_values ()
Get a dictionary of all readable attributes.

The attributes types are autodetected, and some of the types of uncommon attributes may be misrepresented.

get_attribute (*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_attribute_value (*name*, *error_on_missing=True*, *default=None*)
Get value of an attribute with the given name.

If the value doesn't exist or can not be read and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, assume that `error_on_missing==False`. If *name* points at a dictionary branch, return a dictionary with all values in this branch.

get_baudrate()
Get the current baud rate

get_black_level_offset()
Get the black level offset

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_info()
Get camera model data.

Return tuple `(model, serial_number, grabber_info)`.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_exposure()
Get current exposure

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

get_frame_timings()
Get acquisition timing.

Return tuple `(exposure, frame_period)`.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_grabber_attribute_value(*attr*, *default=None*, *kind='auto'*)
Get value of an attribute with a given name or index.

If *default* is not `None`, return *default* if the attribute is not supported; otherwise, raise an error. *kind* is the attribute kind, and it can be `"uint32"`, `"uint64"`, `"double"`, or `"auto"` (autodetect based on the stored list of attribute kinds).

get_grabber_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_grabber_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_grabber_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`.

get_roi_limits(*hbin=1*, *vbin=1*)

get_serial_params()
Return serial parameters as a tuple (write_term, datatype)

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_trigger_interleave()
Check if the trigger interleave is on

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if None, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them, while still keeping total returned frames number to *n*). If return_info==True, return tuple (frames, infos), where infos is a list of frame info tuples (camera-dependent); if some frames are missing and missing_frame!="skip", the corresponding frame info is None.

is_CFR_enabled()
Check if the constant frame rate mode is enabled

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return True.

is_opened()
Check if the device is connected

is_status_line_enabled()
Check if the status line is on

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of with block.

Useful for applying certain settings which can't be changed during the acquisition. If clear==True, clear acquisition in addition to pausing (by default, use the class default specified as _clear_pausing_acquisition attribute). Yields tuple (acq_in_progress, acq_params), which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*, *fastbuff=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple (first, last) with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return None. If peek==True, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them). If return_info==True, return tuple (frames, infos), where infos is a list of TFrameInfo single-element tuples containing frame index; if some frames are missing and missing_frame!="skip", the corresponding frame info is None. If fastbuff==False, return a list of individual frames (2D numpy arrays). Otherwise, return a list of 'chunks', which are 3D numpy arrays containing several frames; in this case, if *return_info* is True, then frame_info will automatically be in an "array" format, with the rows corresponding to the frames within the chunks, and the columns corresponding to the frames. Using `fastbuff` results in faster operation at high frame rates (>~1kFPS), at the expense of a more complicated frame processing in the following code.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

read_trigger(*trig_type*, *trig_line=0*, *trig_pol='high'*)
Read current value of a trigger (input or output).

Parameters

- **trig_type** (*str*) – trigger drive destination type; can be `"ext"`, `"rtsi"`, `"iso_in"`, or `"iso_out"`
- **trig_line** (*int*) – trigger line number
- **trig_pol** (*str*) – trigger polarity; can be `"high"` or `"low"`

reset()
Reset connection to the camera

send_software_trigger()
Send software trigger signal

serial_flush()
Flush CameraLink serial port

serial_read(*n*, *timeout=3.0*, *datatype=None*)
Read specified number of bytes from CameraLink serial port.

Parameters

- **n** – number of bytes to read
- **timeout** – operation timeout (in seconds)
- **datatype** – return datatype; can be `"bytes"` (return raw bytes), or `"str"` (convert into UTF-8 string) if `None`, use the value set up using *setup_serial_params()* (by default, `"bytes"`)

serial_readline(*timeout=3.0*, *datatype=None*, *maxn=1024*)
Read bytes from CameraLink serial port until the termination character (defined in camera file) is encountered.

Parameters

- **timeout** – operation timeout (in seconds)
- **datatype** – return datatype; can be `"bytes"` (return raw bytes), or `"str"` (convert into UTF-8 string) if `None`, use the value set up using *setup_serial_params()* (by default, `"bytes"`)
- **maxn** – maximal number of bytes to read

serial_write(*msg*, *timeout=3.0*, *term=None*)
Write message into CameraLink serial port.

Parameters

- **timeout** – operation timeout (in seconds)
- **term** – additional write terminator character to add to the message; if `None`, use the value set up using *setup_serial_params()* (by default, no additional terminator)

set_all_attribute_values(*settings*, *root=''*, *truncate=True*)
Set values of all attributes with the given *root*.

If `truncate==True`, truncate value to lie within attribute range.

set_attribute_value(*name*, *value*, *truncate=True*, *error_on_missing=True*)
Set value of an attribute with the given name.

If the value doesn't exist or can not be written and `error_on_missing==True`, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). If `truncate==True`, truncate value to lie within attribute range.

set_black_level_offset(*offset*)
Set the black level offset

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_exposure(*exposure*)
Set current exposure

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_frame_period(*frame_period*)
Set frame period (time between two consecutive frames in the internal trigger mode)

set_grabber_attribute_value(*attr*, *value*, *kind='int32'*)
Set value of an attribute with a given name or index.

kind is the attribute kind, and it can be "uint32", "uint64", "double", or "auto" (autodetect based on the stored list of attribute kinds).

set_grabber_roi (*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

set_image_indexing (*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

set_roi (*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

By default, all non-supplied parameters take extreme values.

set_trigger_interleave (*enabled*)
Set the trigger interleave option on or off

setup_acquisition (*mode='sequence'*, *nframes=100*)
Setup acquisition mode.

mode can be either "snap" (single frame or a fixed number of frames) or "sequence" (continuous acquisition). (note that *IMAQCamera.acquisition_in_progress()* would still return True in this case, even though new frames are no longer acquired). *nframes* sets up number of frame buffers.

setup_max_baudrate ()
Setup the maximal available baudrate

setup_serial_params (*write_term=''*, *datatype='bytes'*)
Setup default serial communication parameters.

Parameters

- **write_term** – default terminator character to be added to the sent messages
- **datatype** – type of the result of read commands; can be "bytes" (return raw bytes), or "str" (convert into UTF-8 string)

snap (*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition (**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:``setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is None). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition ()
Stop acquisition

update_attribute_value (*name*, *value*, *error_on_missing=True*, *truncate=True*)
Set value of the attribute with a given name, but only if it's different from the current value.

Can take less time on some version of PFRemote (where single attribute setting is about 50ms). Arguments are the same as *set_attribute_value()*.

wait_for_frame (*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"‘‘ (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

class pylablib.devices.PhotonFocus.PhotonFocus.**PhotonFocusSiSoCamera** (*siso_board*, *siso_applet*, *siso_port=0*, *pfcam_port=0*)

Bases: `pylablib.devices.PhotonFocus.PhotonFocus.IPhotonFocusCamera`, `pylablib.devices.SiliconSoftware.fgrab.SiliconSoftwareFrameGrabber`

IMAQ+PFCam interface to a PhotonFocus camera.

Parameters

- **siso_board** – Silicon Software board index, starting from 0; available boards can be learned by `fgrab.list_boards()`
- **siso_applet** – Silicon Software applet name, which can be learned by `fgrab.list_applets()`; usually, a simple applet like `"DualLineGray16"` or `"MediumLineGray16` are most appropriate; can be either an applet name, or a direct path to the applet DLL
- **siso_port** – Silicon Software port number, if several ports are supported by the camera and the applet
- **pfcam_port** – port number for pfcam interface (can be learned by `list_cameras()`; port number is the first element of the camera data tuple) can also be a tuple `(manufacturer, port)`, e.g., `("National Instruments", "port0")`.

Error
alias of `pylablib.core.devio.base.DeviceError`

GrabberClass
alias of `pylablib.devices.SiliconSoftware.fgrab.SiliconSoftwareFrameGrabber`

open ()
Open connection to the camera

FrameTransferError
alias of `pylablib.devices.interface.camera.DefaultFrameTransferError`

class NoParameterCaller (*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

TimeoutError = <Mock spec='str' id='140318667313232'>

acquisition_in_progress ()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

call_command(*name*, *arg=0*, *error_on_missing=True*)
Execute the given command with the given argument.

If the command doesn't exist and `error_on_missing==True`, raise error; otherwise, do nothing.

clear_acquisition()
Clear all acquisition details and free all buffers

close()
Close connection to the camera

enable_CFR(*enabled=True*)
Enable constant frame rate mode

enable_status_line(*enabled=True*)
Enable or disable status line

fast_shift_roi(*hstart=0*, *vstart=0*)
Shift ROI by only changing its origin, but keeping the shape the same.

Note that if the ROI is invalid, it won't be truncated (as is the standard behavior of `set_roi()`), which might lead to errors later.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_all_attribute_values(*root=''*)
Get values of all attributes with the given *root*

get_all_attributes(*copy=False*)
Return a dictionary of all available attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_all_grabber_attribute_values(*root=''*, ***kwargs*)
Get values of all frame grabber attributes with the given *root*.

Additional arguments are passed to `get_value` methods of individual attributes.

get_all_grabber_attributes(*copy=False*)
Return a dictionary of all available frame grabber grabber_attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_attribute_value(*name*, *error_on_missing=True*, *default=None*)
Get value of an attribute with the given name.

If the value doesn't exist or can not be read and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, assume that `error_on_missing==False`. If *name* points at a dictionary branch, return a dictionary with all values in this branch.

get_available_camlink_pixel_formats()
Get all available CamLink pixel formats and the output pixel formats as a tuple of 2 lists

get_baudrate()
Get the current baud rate

get_black_level_offset()
Get the black level offset

get_camlink_pixel_format()
Get CamLink pixel format and the output pixel format as a tuple

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_info()
Get camera model data.

Return tuple `(model, serial_number, grabber_info)`.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_exposure()
Get current exposure

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

get_frame_timings()
Get acquisition timing.

Return tuple `(exposure, frame_period)`.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_genicam_info_xml()
Get description in Genicam-compatible XML format

get_grabber_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_grabber_attribute_value(*name*, *error_on_missing=True*, *default=None*, ***kwargs*)
Get value of a frame grabber attribute with the given name.

If the value doesn't exist and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, automatically assume that `error_on_missing==False`. If *name* points at a dictionary branch, return a dictionary with all values in this branch. Additional arguments are passed to `get_value` methods of the individual attribute.

get_grabber_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_grabber_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_grabber_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`.

get_roi_limits(*hbin=1*, *vbin=1*)

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_system_info()
Get the dictionary with all system information parameters

get_trigger_interleave()
Check if the trigger interleave is on

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_CFR_enabled()
Check if the constant frame rate mode is enabled

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

is_status_line_enabled()
Check if the status line is on

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*, *fast-buff=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of `TFrameInfo` instances describing frame index, framestamp, and two timestamps (lower and higher precision); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`. Note that obtaining frame info takes about 100us, so `return_info="all"` should be avoided fro rates above 5-10kFPS. If `fastbuff==False`, return a list of individual frames (2D numpy arrays). Otherwise, return a list of 'chunks', which are 3D numpy arrays containing several frames; in this case, if *return_info* is `True`, then `frame_info` will automatically be in an `"array"` format, with the rows corresponding to the frames within the chunks, and the columns corresponding to the frames. Using `fastbuff` results in faster operation at high frame rates (>~1kFPS), at the expense of a more complicated frame processing in the following code.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

set_all_attribute_values(*settings*, *root=''*, *truncate=True*)
Set values of all attributes with the given *root*.

If `truncate==True`, truncate value to lie within attribute range.

set_all_grabber_attribute_values(*settings*, *root=''*, ***kwargs*)
Set values of all frame grabber attributes with the given *root*.

Additional arguments are passed to `set_value` methods of individual attributes.

set_attribute_value(*name*, *value*, *truncate=True*, *error_on_missing=True*)
Set value of an attribute with the given name.

If the value doesn't exist or can not be written and `error_on_missing==True`, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). If `truncate==True`, truncate value to lie within attribute range.

set_black_level_offset(*offset*)
Set the black level offset

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_exposure(*exposure*)
Set current exposure

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-"tuple" formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_frame_merge(*frame_merge=1*)

set_frame_period(*frame_period*)
Set frame period (time between two consecutive frames in the internal trigger mode)

set_grabber_attribute_value(*name*, *value*, *error_on_missing=True*, ***kwargs*)
Set value of a frame grabber attribute with the given name.

If the value doesn't exist and `error_on_missing==True`, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). Additional arguments are passed to `set_value` methods of the individual attribute.

set_grabber_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

By default, all non-supplied parameters take extreme values.

set_trigger_interleave(*enabled*)
Set the trigger interleave option on or off

setup_acquisition(*mode='sequence'*, *nframes=100*)
Setup acquisition mode.

mode can be either "snap" (single frame or a fixed number of frames) or "sequence" (continuous acquisition). (note that `IMAQCamera.acquisition_in_progress()` would still return `True` in this case, even though new frames are no longer acquired). *nframes* sets up number of frame buffers.

setup_camlink_pixel_format(*bits_per_pixel=8*, *taps=1*, *output_fmt=None*, *fmt=None*)
Set up CameraLink pixel format.

If *fmt* is `None`, use supplied *bits_per_pixel* (8, 10, 12, 14, or 16) and *taps* (1 or 2) to figure out the format; otherwise, *fmt* should be a numerical (e.g., `210`) or string (e.g., "FG_CL_MEDIUM_10_BIT") format.

output_fmt specifies the result frame format; if `None`, use grayscale with the given *bits_per_pixel* if *fmt* is `None`, or 16 bit grayscale otherwise.

setup_max_baudrate()
Setup the maximal available baudrate

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

update_attribute_value(*name*, *value*, *error_on_missing=True*, *truncate=True*)
Set value of the attribute with a given name, but only if it's different from the current value.

Can take less time on some version of PFRemote (where single attribute setting is about 50ms). Arguments are the same as `set_attribute_value()`.

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"‘‘ (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

`pylablib.devices.PhotonFocus.PhotonFocus.`**check_grabber_association**(*cam*)
Check if PhotonFocus camera has correct association between the frame grabber and the PFRemote interface.

cam should be an opened instance of `PhotonFocusIMAQCamera` or `PhotonFocusSiSoCamera`. Note that this function changes camera parameters such as exposure, frame period, ROI, trigger source, and status line.

`pylablib.devices.PhotonFocus.PhotonFocus.`**get_status_lines**(*frames*, *check_transposed=True*, *drop_magic=True*)
Extract status lines (up to first 6 entries) from the given frames.

frames can be 2D array (one frame), 3D array (stack of frames, first index is frame number), or list of array. Automatically check if the status line is present; return `None` if it's not. If `check_transposed==True`, check for the case where the image is transposed (i.e., line becomes a column). If `drop_magic==True`, remove the first status line entry, which is simply a special number marking the status line presence. Return a 1D or 2D numpy array, where the first axis (if present) is the frame number, and the last is the status line entry The entries after the magic are the frame index, timestamp (in us), missed trigger counters (up to 255), average frame value, and the integration time (in pixel clock cycles, which depend on the camera).

`pylablib.devices.PhotonFocus.PhotonFocus.`**get_status_line_position**(*frame*, *check_transposed=True*)
Check whether status line is present in the frame, and return its location.

Return tuple `(row, transposed)`, where *row* is the status line row (can be `-1` or `-2`) and *transposed* is `True` if the line is present in the transposed image. If no status line is found, return `None`. If `check_transposed==True`, check for the case where the image is transposed (i.e., line becomes a column).

`pylablib.devices.PhotonFocus.PhotonFocus.`**`remove_status_line`**`(`*frame, sl_pos='calculate', policy='duplicate', copy=True*`)`

Remove status line from the frame.

Parameters

- **`frame`** – a frame to process (2D or 3D numpy array; if 3D, the first axis is the frame number)
- **`sl_pos`** – status line position (returned by *`get_status_line_position()`*); if equal to `"calculate"`, calculate here; for a 3D array, assumed to be the same for all frames
- **`policy`** – determines way to deal with the status line; can be `"keep"` (keep as is), `"cut"` (cut off the status line row), `"zero"` (set it to zero), `"median"` (set it to the image median), or `"duplicate"` (set it equal to the previous row; default)
- **`copy`** – if `True`, make copy of the original frames; otherwise, attempt to remove the line in-place

`pylablib.devices.PhotonFocus.PhotonFocus.`**`find_skipped_frames`**`(`*lines, step=1*`)`

Check if there are skipped frames based on status line reading.

step specifies expected index step between neighboring frames.

Return list `[(idx, skipped)]`, where *idx* is the index after which *skipped* frames were skipped.

Module contents

pylablib.devices.SiliconSoftware package

Submodules

pylablib.devices.SiliconSoftware.fgrab module

`class` `pylablib.devices.SiliconSoftware.fgrab.`**`TBoardInfo`**`(`*name, full_name*`)`

Bases: `tuple`

`count``()`

Return number of occurrences of value.

`full_name`

`index``()`

Return first index of value.

Raises ValueError if the value is not present.

`name`

`pylablib.devices.SiliconSoftware.fgrab.`**`get_board_info`**`(`*board*`)`

Get board info for a given index (starting from 0) `pylablib.devices.SiliconSoftware.fgrab.`**`list_boards`**`()`

List all boards available through Silicon Software interface

`pylablib.devices.SiliconSoftware.fgrab.`**`get_boards_number`**`()`

List number of connected Silicon Software boards

class `pylablib.devices.SiliconSoftware.fgrab.`**`TAppletInfo`**`(`*name*, *file*`)`

Bases: `tuple`

`count``()`

Return number of occurrences of value.

`file`

`index``()`

Return first index of value.

Raises ValueError if the value is not present.

`name`

class `pylablib.devices.SiliconSoftware.fgrab.`**`TFullAppletInfo`**`(`*name*, *uid*, *desc*, *category*, *platform*, *tags*, *version*, *path*, *file*, *flags*, *info*`)`

Bases: `tuple`

`category`

`count``()`

Return number of occurrences of value.

`desc`

`file`

`flags`

`index``()`

Return first index of value.

Raises ValueError if the value is not present.

`info`

`name`

`path`

`platform`

`tags`

`uid`

`version`

`pylablib.devices.SiliconSoftware.fgrab.`**`list_applets`**`(`*board*, *full_desc=False*, *valid=True*, *on_board=False*`)`

List all applets available for this board.

board is the board index (starting from 0) given by its position in the list returned by `list_boards()`. If `full_desc==True`, return full description for each applet; otherwise, return only name and file name. If `valid==True`, list only valid and compatible applets; otherwise, list all applets. If `on_board==True`, list applets running on board; otherwise, list all applets contained in the system.

pylablib.devices.SiliconSoftware.fgrab.**get_applet_info**(*board*, ***kwargs*)
Return full information for an applet with the given parameters (e.g., name, or full path)

class pylablib.devices.SiliconSoftware.fgrab.**FGrabAttribute**(*fg*, *aid*, *port=0*, *system=False*)
Bases: `object`

Object representing an Silicon Software frame grabber parameter.

Allows to query and set values and get additional information. Usually created automatically by an :class:`` instance, but could be created manually.

Parameters

- **fg** – opened frame grabber handle
- **aid** – attribute ID
- **port** – camera port within the frame grabber
- **system** – if `True`, this is a system attribute; otherwise, it is a camera attribute

name
attribute name

min
minimal attribute value (if applicable)

Type float or int

max
maximal attribute value (if applicable)

Type float or int

inc
minimal attribute increment value (if applicable)

Type float or int

values
dictionary `{i: name}` of possible attribute values (if applicable)

update_limits()
Update minimal and maximal attribute limits and return tuple `(min, max, inc)`

truncate_value(*value*)
Truncate value to lie within attribute limits

get_value(*enum_as_str=True*)
Get attribute value.

If `enum_as_str==True`, return enum-style values as strings; otherwise, return corresponding integer values.

set_value(*value*, *truncate=True*)
Get attribute value.

If `truncate==True`, automatically truncate value to lie within allowed range.

class pylablib.devices.SiliconSoftware.fgrab.**TDeviceInfo**(*applet_info*, *system_info*, *software_version*)
Bases: `tuple`

applet_info

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

software_version

system_info

class pylablib.devices.SiliconSoftware.fgrab.**TFrameInfo**(*frame_index*, *framestamp*, *timestamp*, *timestamp_long*)

Bases: tuple

count()
Return number of occurrences of value.

frame_index

framestamp

index()
Return first index of value.

Raises ValueError if the value is not present.

timestamp

timestamp_long

class pylablib.devices.SiliconSoftware.fgrab.**SiliconSoftwareFrameGrabber**(*siso_board=0*, *siso_applet='DualAreaGr*, *siso_port=0*, *siso_detector_size=None*, *do_open=True*, ***kwargs*)

Bases: *pylablib.devices.interface.camera.IGrabberAttributeCamera*, *pylablib.devices.interface.camera.IROICamera*

Generic Silicon Software frame grabber interface.

Compared to *SiliconSoftwareCamera*, has more permissive initialization arguments, which simplifies its use as a base class for expanded cameras.

Parameters

- **siso_board** – board index, starting from 0; available boards can be learned by *list_boards()*
- **siso_applet** – applet name, which can be learned by *list_applets()*; usually, a simple applet like "DualLineGray16" or "MediumLineGray16 are most appropriate; can be either an applet name, or a direct path to the applet DLL
- **siso_port** – port number, if several ports are supported by the camera and the applet
- **siso_detector_size** – if not None, can specify the maximal detector size; by default, use the maximal available for the frame grabber (usually, 16384x16384)

Error = <Mock name='mock.SiliconSoftwareError' id='140318678604176'>

TimeoutError = <Mock spec='str' id='140318667313232'>

open()
    Open connection to the camera

close()
    Close connection to the camera

is_opened()
    Check if the device is connected

get_all_grabber_attribute_values(*root=''*, ***kwargs*)
    Get values of all frame grabber attributes with the given *root*.

    Additional arguments are passed to `get_value` methods of individual attributes.

set_all_grabber_attribute_values(*settings*, *root=''*, ***kwargs*)
    Set values of all frame grabber attributes with the given *root*.

    Additional arguments are passed to `set_value` methods of individual attributes.

get_system_info()
    Get the dictionary with all system information parameters

get_genicam_info_xml()
    Get description in Genicam-compatible XML format

get_device_info()
    Get camera model data.

    Return tuple `(applet_info, system_info, software_version)` with the board serial number and an the interface type (e.g., `"1430"` for NI PCIe-1430)

set_frame_merge(*frame_merge=1*)

get_detector_size()
    Get camera detector size (in pixels) as a tuple `(width, height)`

get_grabber_detector_size()
    Get camera detector size (in pixels) as a tuple `(width, height)`

get_roi()
    Get current ROI.

    Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_grabber_roi()
    Get current ROI.

    Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
    Setup camera ROI.

    hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

set_grabber_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
    Setup camera ROI.

    hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_grabber_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

setup_camlink_pixel_format(*bits_per_pixel=8*, *taps=1*, *output_fmt=None*, *fmt=None*)
Set up CameraLink pixel format.

If *fmt* is `None`, use supplied *bits_per_pixel* (8, 10, 12, 14, or 16) and *taps* (1 or 2) to figure out the format; otherwise, *fmt* should be a numerical (e.g., `210`) or string (e.g., `"FG_CL_MEDIUM_10_BIT"`) format. *output_fmt* specifies the result frame format; if `None`, use grayscale with the given *bits_per_pixel* if *fmt* is `None`, or 16 bit grayscale otherwise.

get_camlink_pixel_format()
Get CamLink pixel format and the output pixel format as a tuple

get_available_camlink_pixel_formats()
Get all available CamLink pixel formats and the output pixel formats as a tuple of 2 lists

setup_acquisition(*mode='sequence'*, *nframes=100*)
Setup acquisition mode.

mode can be either `"snap"` (single frame or a fixed number of frames) or `"sequence"` (continuous acquisition). (note that `IMAQCamera.acquisition_in_progress()` would still return `True` in this case, even though new frames are no longer acquired). *nframes* sets up number of frame buffers.

clear_acquisition()
Clear all acquisition details and free all buffers

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

acquisition_in_progress()
Check if acquisition is in progress

FrameTransferError
alias of `pylablib.devices.interface.camera.DefaultFrameTransferError`

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary {name: value}

get_all_grabber_attributes(*copy=False*)
Return a dictionary of all available frame grabber grabber_attributes.

If copy==True, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple (width, height); take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is "list" or "array".

get_frame_info_format()
Get format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple (acquired, unread, skipped, size), where acquired is the total number of acquired frames, unread is the number of acquired but not read frames, skipped is the number of skipped (not read and then written over) frames, and buffer_size is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_grabber_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_grabber_attribute_value(*name*, *error_on_missing=True*, *default=None*, ***kwargs*)
Get value of a frame grabber attribute with the given name.

If the value doesn't exist and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, automatically assume that `error_on_missing==False`. If *name* points at a dictionary branch, return a dictionary with all values in this branch. Additional arguments are passed to `get_value` methods of the individual attribute.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_grabber_attribute_value(*name*, *value*, *error_on_missing=True*, ***kwargs*)
Set value of a frame grabber attribute with the given name.

If the value doesn't exist and `error_on_missing==True`, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). Additional arguments are passed to `set_value` methods of the individual attribute.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"‘‘ (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*, *fastbuff=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of `TFrameInfo` instances describing frame index, framestamp, and two timestamps (lower and higher precision); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`. Note that obtaining frame info takes about 100us, so `return_info="all"` should be avoided fro rates above 5-10kFPS. If `fastbuff==False`, return a list of individual frames (2D numpy arrays). Otherwise, return a list of 'chunks', which are 3D numpy arrays containing several frames; in this case, if *return_info* is `True`, then `frame_info` will automatically be in an `"array"` format, with the rows corresponding to the frames within the chunks, and the columns corresponding to the frames. Using `fastbuff` results in faster operation at high frame rates (>~1kFPS), at the expense of a more complicated frame processing in the following code.

class pylablib.devices.SiliconSoftware.fgrab.**SiliconSoftwareCamera**(*board*, *applet*, *port=0*, *detector_size=None*)
Bases: `pylablib.devices.SiliconSoftware.fgrab.SiliconSoftwareFrameGrabber`

Generic Silicon Software frame grabber interface.

Parameters

- **board** – board index, starting from 0; available boards can be learned by `list_boards()`
- **applet** – applet name, which can be learned by `list_applets()`; usually, a simple applet like `"DualLineGray16"` or `"MediumLineGray16` are most appropriate; can be either an applet name, or a direct path to the applet DLL
- **port** – port number, if several ports are supported by the camera and the applet
- **detector_size** – if not `None`, can specify the maximal detector size; by default, use the maximal available for the frame grabber (usually, 16384x16384)

Error = <Mock name='mock.SiliconSoftwareError' id='140318678604176'>

FrameTransferError
alias of `pylablib.devices.interface.camera.DefaultFrameTransferError`

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

TimeoutError = <Mock spec='str' id='140318667313232'>

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

clear_acquisition()
Clear all acquisition details and free all buffers

close()
Close connection to the camera

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_all_grabber_attribute_values(*root=''*, ***kwargs*)
Get values of all frame grabber attributes with the given *root*.

Additional arguments are passed to `get_value` methods of individual attributes.

get_all_grabber_attributes(*copy=False*)
Return a dictionary of all available frame grabber grabber_attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_available_camlink_pixel_formats()
Get all available CamLink pixel formats and the output pixel formats as a tuple of 2 lists

get_camlink_pixel_format()
Get CamLink pixel format and the output pixel format as a tuple

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_info()
Get camera model data.

Return tuple `(applet_info, system_info, software_version)` with the board serial number and an the interface type (e.g., `"1430"` for NI PCIe-1430)

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is "list" or "array".

get_frame_info_format()
Get format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple (acquired, unread, skipped, size), where acquired is the total number of acquired frames, unread is the number of acquired but not read frames, skipped is the number of skipped (not read and then written over) frames, and buffer_size is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_genicam_info_xml()
Get description in Genicam-compatible XML format

get_grabber_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_grabber_attribute_value(*name*, *error_on_missing=True*, *default=None*, ***kwargs*)
Get value of a frame grabber attribute with the given name.

If the value doesn't exist and error_on_missing==True, raise error; otherwise, return *default*. If *default* is not None, automatically assume that error_on_missing==False. If *name* points at a dictionary branch, return a dictionary with all values in this branch. Additional arguments are passed to get_value methods of the individual attribute.

get_grabber_detector_size()
Get camera detector size (in pixels) as a tuple (width, height)

get_grabber_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_grabber_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_system_info()
Get the dictionary with all system information parameters

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

open()
Open connection to the camera

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*, *fastbuff=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of `TFrameInfo` instances describing frame index, framestamp, and two timestamps (lower and higher precision); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`. Note that obtaining frame info takes about 100us, so `return_info="all"` should be avoided fro rates above 5-10kFPS. If `fastbuff==False`, return a list of individual frames (2D numpy arrays). Otherwise, return a list of 'chunks', which are 3D numpy arrays containing several frames; in this case, if *return_info* is `True`, then `frame_info` will automatically be in an `"array"` format, with the rows corresponding to the frames within the chunks, and the columns corresponding to the frames. Using `fastbuff` results in faster operation at high frame rates (>~1kFPS), at the expense of a more complicated frame processing in the following code.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

set_all_grabber_attribute_values(*settings*, *root=''*, ***kwargs*)
Set values of all frame grabber attributes with the given *root*.

Additional arguments are passed to `set_value` methods of individual attributes.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array). Note that if the format is set to "array", the frame info format is also automatically set to "array".

set_frame_info_format (*fmt*, *include_fields=None*)
Set format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes) If *include_fields* is not None, it specifies the fields included for non-"tuple" formats.

set_frame_info_period (*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_frame_merge (*frame_merge=1*)

set_grabber_attribute_value (*name*, *value*, *error_on_missing=True*, ***kwargs*)
Set value of a frame grabber attribute with the given name.

If the value doesn't exist and error_on_missing==True, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). Additional arguments are passed to set_value methods of the individual attribute.

set_grabber_roi (*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

set_image_indexing (*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

set_roi (*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

setup_acquisition (*mode='sequence'*, *nframes=100*)
Setup acquisition mode.

mode can be either "snap" (single frame or a fixed number of frames) or "sequence" (continuous acquisition). (note that *IMAQCamera.acquisition_in_progress()* would still return True in this case, even though new frames are no longer acquired). *nframes* sets up number of frame buffers.

setup_camlink_pixel_format (*bits_per_pixel=8*, *taps=1*, *output_fmt=None*, *fmt=None*)
Set up CameraLink pixel format.

If *fmt* is `None`, use supplied *bits_per_pixel* (8, 10, 12, 14, or 16) and *taps* (1 or 2) to figure out the format; otherwise, *fmt* should be a numerical (e.g., `210`) or string (e.g., `"FG_CL_MEDIUM_10_BIT"`) format. *output_fmt* specifies the result frame format; if `None`, use grayscale with the given *bits_per_pixel* if *fmt* is `None`, or 16 bit grayscale otherwise.

snap (*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition (**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition ()
Stop acquisition

wait_for_frame (*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be “lastread”‘‘ (from the last read frame), `"lastwait"` (wait for the last successful *wait_for_frame()* call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

Module contents

pylablib.devices.SmarAct package

Submodules

pylablib.devices.SmarAct.scu3d module

class pylablib.devices.SmarAct.scu3d.**LibraryController** (*lib*)
Bases: *pylablib.devices.utils.load_lib.LibraryController*

close (*opid*)
Mark device closing.

Return tuple `(close_result, uninit_result)` with the results of the closing and the shutdown. If library does not need to be shut down yet, set `uninit_result=None`

open ()
Mark device opening.

Return tuple `(init_result, open_result, opid)` with the results of the initialization and the opening, and the opening ID which should afterwards be used for closing. If library is already initialized, set `init_result=None`

preinit ()
Pre-initialize the library, if it hasn’t been done already

shutdown()
Close all opened connections and shutdown the library

temp_open()
Context for temporarily opening a new device connection

class pylablib.devices.SmarAct.scu3d.**TDeviceInfo**(*device_id*, *firmware_version*, *dll_version*)
Bases: tuple

count()
Return number of occurrences of value.

device_id

dll_version

firmware_version

index()
Return first index of value.

Raises ValueError if the value is not present.

pylablib.devices.SmarAct.scu3d.**get_device_info**(*idx*)
Get info of the devices with the given index.

Return tuple (device_id, firmware_version, dll_version).

pylablib.devices.SmarAct.scu3d.**list_devices**()
List all connected devices

pylablib.devices.SmarAct.scu3d.**get_devices_number**()
Get number of connected SCU3D controller

class pylablib.devices.SmarAct.scu3d.**SCU3D**(*idx=0*, *axis_dir='+++'*)
Bases: *pylablib.devices.interface.stage.IMultiaxisStage*

SmarAct SCU3D translation stage controller.

Parameters

- **idx** (*int*) – stage index
- **axis_dir** (*str*) – 3-symbol string specifying default directions of the axes (each symbol be "+" or "-")

Error = <Mock name='mock.SmarActError' id='140318660914576'>

open()
Open the connection to the stage

close()
Close the connection to the stage

is_opened()
Check if the device is connected

get_device_info()
Get info of the devices with the given index.

Return tuple (device_id, firmware_version, dll_version).

get_axis_dir()
Get axis direction convention (a string of 3 symbols which are either "+" or "-" determining if the axis direction is flipped)

set_axis_dir(*axis_dir*)
Set axis direction convention (a string of 3 symbols which are either "+" or "-" determining if the axis direction is flipped)

move_macrostep(*axis*, *steps*, *voltage*, *frequency*)
Move along a given axis by a single "macrostep", which consists of several regular steps.

voltage (in Volts) and *frequency* (in Hz) specify the motion parameters. This simulates the controller operation, where one "step" at large step sizes consists of several small steps.

move_by(*axis*, *steps=1*, *stepsize=10*)
Move along a given axis with a given number of macrosteps using one of the predefined step size.

stepsize can range from 1 (smallest) to 20 (largest), and roughly corresponds to the handheld controller parameters.

get_status(*axis='all'*)
Get the axis status.

Can be "stopped" (default state), "setting_amplitude" (setting open-loop step amplitude), "moving" (open-loop movement), "targeting" (closed-loop movement), "holding" (closed-loop position holding), "calibrating" (sensor calibration), or "moving_to_reference" (calibrating position sensor).

wait_for_status(*axis*, *status='stopped'*, *timeout=30.0*)
Wait until the axis reaches a given status.

By default wait for "stopped" status (i.e., wait until the motion is finished).

wait_move(*axis*, *timeout=30.0*)
Wait for a given axis to stop moving

is_moving(*axis='all'*)
Check if a given axis is moving

stop(*axis='all'*)
Stop motion at a given axis

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

get_all_axes()
Get the list of all available axes (taking mapping into account)

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

remap_axes(*mapping*, *accept_original=True*)
Rename axes to the new labels.

mapping is the new axes mapping, which can be a list of new axes name (corresponding to the old axes in order returned by *get_all_axes()*), or a dictionary `{alias: original}` of the new axes aliases.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

Module contents

pylablib.devices.Tektronix package

Submodules

pylablib.devices.Tektronix.base module

exception pylablib.devices.Tektronix.base.**TektronixError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Tektronix devices error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Tektronix.base.**TektronixBackendError**(*exc*)
Bases: *pylablib.devices.Tektronix.base.TektronixError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Generic Tektronix backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

pylablib.devices.Tektronix.base.**muxchannel**(**args*, ***kwargs*)
Multiplex the function over its channel argument

class pylablib.devices.Tektronix.base.**TTriggerParameters**(*source*, *level*, *coupling*, *slope*)
Bases: `tuple`

count()
Return number of occurrences of value.

coupling

index()
Return first index of value.

Raises ValueError if the value is not present.

level

slope

source

class pylablib.devices.Tektronix.base.**ITektronixScope**(*addr*, *nchannels='auto'*)
Bases: `pylablib.core.devio.SCPI.SCPIDevice`

Generic Tektronix oscilloscope.

Parameters

- **addr** – device address; usually a VISA address string such as `"USB0::0x0699::0x0364::C000000::INSTR"`
- **nchannels** – can specify number of channels on the oscilloscope; by default, autodetect number of channels (might take several seconds on connection)

Error
alias of `TektronixError`

ReraiseError
alias of `TektronixBackendError`

get_channels_number()
Get the number of channels

get_channels(*only_main=False*)
Get the list of all input channels (if `only_main==True`) or all available channels (if `only_main==False`)

normalize_channel_name(*channel*)
Normalize channel name as represented by the oscilloscope

grab_single(*wait=True*, *software_trigger=False*, *wait_timeout=None*)
Set single waveform grabbing and wait for acquisition.

If `wait==True`, wait until the acquisition is complete; otherwise, return immediately. if `software_trigger==True`, send the software trigger after setup (i.e., the device triggers immediately regardless of the input).

wait_for_grabbing(*timeout=None*)
Wait until the acquisition is complete

grab_continuous(*enable=True*)
Start or stop continuous grabbing

stop_grabbing()
Stop grabbing or waiting (equivalent to `self.grab_continuous(False)`)

is_continuous()
Check if grabbing is continuous or single

is_grabbing()
Check if acquisition is in progress.

Return `True` if the oscilloscope is recording data, or if the trigger is armed/ready and waiting; return `False` if the acquisition is stopped. To check if the trigger has been triggered, use *get_trigger_state()*.

get_edge_trigger_source()
Get edge trigger source.

Can be an integer indicating channel number or a name of a special channel.

set_edge_trigger_source(*channel*)
Get edge trigger source.

Can be an integer indicating channel number or a name of a special channel.

get_edge_trigger_coupling()
Get edge trigger coupling ("ac" or "dc")

set_edge_trigger_coupling(*coupling*)
Set edge trigger coupling ("ac" or "dc")

get_edge_trigger_slope()
Get edge trigger slope ("fall" or "rise")

set_edge_trigger_slope(*slope*)
Set edge trigger slope ("fall" or "rise")

get_trigger_level()
Get edge trigger level (in Volts)

set_trigger_level(*level*)
Set edge trigger level (in Volts)

setup_edge_trigger(*source*, *level*, *coupling='dc'*, *slope='rise'*)
Setup edge trigger.

Set source, level, coupling and slope (see corresponding methods for details).

get_trigger_mode()
Get trigger mode.

Can be either "auto" or "norm".

set_trigger_mode(*trigger_mode='auto'*)
Set trigger mode.

Can be either "auto" or "norm".

get_trigger_state()
Get trigger state.

Can be "armed" (acquiring pretrigger), "ready" (pretrigger acquired, wait for trigger event), "trigger" (triggered, acquiring the rest of the waveform), "auto" ("auto" mode trigger is acquiring data in the absence of trigger), "save" (acquisition is stopped), or "scan" (oscilloscope in the scan mode)

force_trigger()
Force trigger event

get_horizontal_span()
Get horizontal span (in seconds)

set_horizontal_span(*span*)
Set horizontal span (in seconds)

get_horizontal_offset()
Get horizontal offset (position of the center of the sweep; in seconds)

set_horizontal_offset(*offset=0.0*)
Set horizontal offset (position of the center of the sweep; in seconds)

get_vertical_span(*channel*)
Get channel vertical span (in V)

set_vertical_span(*channel*, *span*)
Set channel vertical span (in V)

get_vertical_position(*channel*)
Get channel vertical position (offset of the zero volt line; in V)

set_vertical_position(*channel*, *position*)
Set channel vertical position (offset of the zero volt line; in V)

is_channel_enabled(*channel*)
Check if channel is enabled

enable_channel(*channel*, *enabled=True*)
Enable or disable given channel

get_selected_channel()
Get selected source channel.

Return number if it is a real channel, or a string name otherwise.

select_channel(*channel*)
Select a channel to read data.

Doesn't need to be called explicitly, if `read_multiple_sweeps()` or `read_sweep()` are used.

get_coupling(*channel*)
Get channel coupling.

Can be `"ac"`, `"dc"`, or `"gnd"`.

set_coupling(*channel*, *coupling='dc'*)
Set channel coupling.

Can be `"ac"`, `"dc"`, or `"gnd"`.

get_probe_attenuation(*channel*)
Get channel probe attenuation

set_probe_attenuation(*channel*, *attenuation*)
Set channel probe attenuation

get_points_number(*kind='send'*)
Get number of datapoints in various context.

`kind` defines the context. It can be `"acq"` (number of points acquired), `"trace"` (number of points in the source of the read-out trace; can be lower than `"acq"` if the data resolution is reduced, or if the source is not a channel data), or `"send"` (number of points in the sent waveform; can be lower than `"trace"` if `get_data_pts_range()` is used to specify and incomplete range). Not all kinds are defined for all scope model (e.g., `"trace"` is not defined for TDS2000 series oscilloscopes).

For length of read-out trace, see also `get_data_pts_range()`.

set_points_number(*pts_num*, *reset_limits=True*)
Set number of datapoints to record when acquiring a trace.

If `reset_limits==True`, reset the datapoints range (*set_data_pts_range()*) to the full range. The actual set value (returned by this method) can be different from the requested value.

get_data_pts_range()
Get range of data points to read.

The range is defined from 1 to the points number (returned by *get_points_number()*).

set_data_pts_range(*rng=None*)
Set range of data points to read.

The range is defined from 1 to the points number (returned by *get_points_number()* with `kind="acq"`). If `rng is None`, set the full range.

set_data_format(*fmt='default'*)
Set data transfer format.

fmt is a string describing the format; can be either `"ascii"`, or a numpy-style format string (e.g., `"<u2"`). If `"default"`, use the oscilloscope default format (usually binary with smallest appropriate byte size).

get_data_format()
Get data transfer format.

Return a string describing the format; can be either `"ascii"`, or a numpy-style format string (e.g., `"<u2"`).

get_wfmpre(*channel=None*, *enable=True*)
Get preamble dictionary describing all scaling and format data for the given channel or a list of channels.

Can be acquired once and used in subsequent multiple reads to save time on re-requesting. If *channel* is `None`, use the currently selected channel. If `enable==True`, make sure that the requested channel is enabled; getting preamble for disabled channels raises an error.

read_raw_data(*channel=None*, *fmt=None*, *timeout=None*)
Request, read and parse raw data at a given channel.

fmt is data format (e.g., `"i1"`, `"<i2"`, or `"ascii"`) or `"default"`, which uses the default oscilloscope format (usually binary with smallest appropriate byte size). If *fmt* is `None`, use the current format. If *channel* is `None`, use the currently selected channel.

Returned data is raw (i.e., not scaled and without x axis).

read_multiple_sweeps(*channels*, *wfmpres=None*, *ensure_fmt=False*, *timeout=None*, *return_wfmpres=None*)
Read data from a multiple channels channel.

Parameters

- **channels** – list of channel indices or names
- **wfmpres** – optional list or dictionary of preambles (obtained using *get_wfmpre()*); if it is `None`, obtain during reading, which slows down the data acquisition a bit
- **ensure_fmt** – if `True`, make sure that oscilloscope data format agrees with the one in *wfmpre*
- **timeout** – read timeout
- **return_wfmpres** – if `True`, return tuple `(sweeps, wfmpres)`, where `wfmpres` can be used for further sweep readouts.

read_sweep(*channel*, *wfmpre=None*, *ensure_fmt=True*, *timeout=None*)
Read data from a single channel.

Parameters

- **channel** – channel index or name
- **wfmpre** – optional preamble dictionary (obtained using *get_wfmpre()*); if it is None, obtain during reading, which slows down the data acquisition a bit
- **ensure_fmt** – if True, make sure that oscilloscope data format agrees with the one in *wfmpre*
- **timeout** – read timeout

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If read_echo==True, assume that the device first echoes the input and skip it.

close()
Close the backend

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If one_line==True, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

set_device_variable(*key*, *value*)
Set the value of a settings parameter

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive *write()* operations are bundled together with `;` delimiter. The actual write is performed at the *read()*/*ask()* operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform *wait_sync()*), `'dev'` (perform *wait_dev()*) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class pylablib.devices.Tektronix.base.**TDS2000**(*addr*, *nchannels='auto'*)
Bases: *pylablib.devices.Tektronix.base.ITektronixScope*

Tektronix TDS2000 series oscilloscope.

Parameters

- **addr** – device address; usually a VISA address string such as `"USB0::0x0699::0x0364::C000000::INSTR"`
- **nchannels** – can specify number of channels on the oscilloscope; by default, autodetect number of channels (might take several seconds on connection)

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *TektronixError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

ReraiseError
alias of *TektronixBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()
Close the backend

enable_channel(*channel*, *enabled=True*)
Enable or disable given channel

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

force_trigger()
Force trigger event

static get_arg_type(*arg*)
Autodetect argument type

get_channels(*only_main=False*)
Get the list of all input channels (if `only_main==True`) or all available channels (if `only_main==False`)

get_channels_number()
Get the number of channels

get_coupling(*channel*)
Get channel coupling.

Can be `"ac"`, `"dc"`, or `"gnd"`.

get_data_format()
Get data transfer format.

Return a string describing the format; can be either "ascii", or a numpy-style format string (e.g., "<u2").

get_data_pts_range()
Get range of data points to read.

The range is defined from 1 to the points number (returned by *get_points_number()*).

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_edge_trigger_coupling()
Get edge trigger coupling ("ac" or "dc")

get_edge_trigger_slope()
Get edge trigger slope ("fall" or "rise")

get_edge_trigger_source()
Get edge trigger source.

Can be an integer indicating channel number or a name of a special channel.

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_horizontal_offset()
Get horizontal offset (position of the center of the sweep; in seconds)

get_horizontal_span()
Get horizontal span (in seconds)

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_points_number(*kind='send'*)
Get number of datapoints in various context.

kind defines the context. It can be "acq" (number of points acquired), "trace" (number of points in the source of the read-out trace; can be lower than "acq" if the data resolution is reduced, or if the source is not a channel data), or "send" (number of points in the sent waveform; can be lower than "trace" if *get_data_pts_range()* is used to specify and incomplete range). Not all kinds are defined for all scope model (e.g., "trace" is not defined for TDS2000 series oscilloscopes).

For length of read-out trace, see also *get_data_pts_range()*.

get_probe_attenuation(*channel*)
Get channel probe attenuation

get_selected_channel()
Get selected source channel.

Return number if it is a real channel, or a string name otherwise.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_trigger_level()
Get edge trigger level (in Volts)

get_trigger_mode()
Get trigger mode.

Can be either `"auto"` or `"norm"`.

get_trigger_state()
Get trigger state.

Can be `"armed"` (acquiring pretrigger), `"ready"` (pretrigger acquired, wait for trigger event), `"trigger"` (triggered, acquiring the rest of the waveform), `"auto"` (`"auto"` mode trigger is acquiring data in the absence of trigger), `"save"` (acquisition is stopped), or `"scan"` (oscilloscope in the scan mode)

get_vertical_position(*channel*)
Get channel vertical position (offset of the zero volt line; in V)

get_vertical_span(*channel*)
Get channel vertical span (in V)

get_wfmpre(*channel=None*, *enable=True*)
Get preamble dictionary describing all scaling and format data for the given channel or a list of channels.

Can be acquired once and used in subsequent multiple reads to save time on re-requesting. If *channel* is `None`, use the currently selected channel. If `enable==True`, make sure that the requested channel is enabled; getting preamble for disabled channels raises an error.

grab_continuous(*enable=True*)
Start or stop continuous grabbing

grab_single(*wait=True*, *software_trigger=False*, *wait_timeout=None*)
Set single waveform grabbing and wait for acquisition.

If `wait==True`, wait until the acquisition is complete; otherwise, return immediately. if `software_trigger==True`, send the software trigger after setup (i.e., the device triggers immediately regardless of the input).

is_channel_enabled(*channel*)
Check if channel is enabled

is_continuous()
Check if grabbing is continuous or single

is_grabbing()
Check if acquisition is in progress.

Return `True` if the oscilloscope is recording data, or if the trigger is armed/ready and waiting; return `False` if the acquisition is stopped. To check if the trigger has been triggered, use `get_trigger_state()`.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

normalize_channel_name(*channel*)
Normalize channel name as represented by the oscilloscope

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

read_multiple_sweeps(*channels*, *wfmpres=None*, *ensure_fmt=False*, *timeout=None*, *return_wfmpres=None*)
Read data from a multiple channels channel.

Parameters

- **channels** – list of channel indices or names
- **wfmpres** – optional list or dictionary of preambles (obtained using `get_wfmpre()`); if it is `None`, obtain during reading, which slows down the data acquisition a bit
- **ensure_fmt** – if `True`, make sure that oscilloscope data format agrees with the one in *wfmpre*
- **timeout** – read timeout
- **return_wfmpres** – if `True`, return tuple `(sweeps, wfmpres)`, where `wfmpres` can be used for further sweep readouts.

read_raw_data (*channel=None*, *fmt=None*, *timeout=None*)
Request, read and parse raw data at a given channel.

fmt is data format (e.g., "i1", "<i2", or "ascii") or "default", which uses the default oscilloscope format (usually binary with smallest appropriate byte size). If *fmt* is None, use the current format. If *channel* is None, use the currently selected channel.

Returned data is raw (i.e., not scaled and without x axis).

read_sweep (*channel*, *wfmpre=None*, *ensure_fmt=True*, *timeout=None*)
Read data from a single channel.

Parameters

- **channel** – channel index or name
- **wfmpre** – optional preamble dictionary (obtained using get_wfmpre()); if it is None, obtain during reading, which slows down the data acquisition a bit
- **ensure_fmt** – if True, make sure that oscilloscope data format agrees with the one in *wfmpre*
- **timeout** – read timeout

reconnect (*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If new_instrument==True, create a new backend instance. If ignore_error==True, ignore errors on closing.

reset ()
Reset the device (by default, "*RST" command)

select_channel (*channel*)
Select a channel to read data.

Doesn't need to be called explicitly, if read_multiple_sweeps() or read_sweep() are used.

set_coupling (*channel*, *coupling='dc'*)
Set channel coupling.

Can be "ac", "dc", or "gnd".

set_data_format (*fmt='default'*)
Set data transfer format.

fmt is a string describing the format; can be either "ascii", or a numpy-style format string (e.g., "<u2"). If "default", use the oscilloscope default format (usually binary with smallest appropriate byte size).

set_data_pts_range (*rng=None*)
Set range of data points to read.

The range is defined from 1 to the points number (returned by get_points_number() with kind="acq"). If rng is None, set the full range.

set_device_variable (*key*, *value*)
Set the value of a settings parameter

set_edge_trigger_coupling (*coupling*)
Set edge trigger coupling ("ac" or "dc")

set_edge_trigger_slope (*slope*)
Set edge trigger slope ("fall" or "rise")

set_edge_trigger_source(*channel*)
Get edge trigger source.

Can be an integer indicating channel number or a name of a special channel.

set_horizontal_offset(*offset=0.0*)
Set horizontal offset (position of the center of the sweep; in seconds)

set_horizontal_span(*span*)
Set horizontal span (in seconds)

set_points_number(*pts_num*, *reset_limits=True*)
Set number of datapoints to record when acquiring a trace.

If `reset_limits==True`, reset the datapoints range (`set_data_pts_range()`) to the full range. The actual set value (returned by this method) can be different from the requested value.

set_probe_attenuation(*channel*, *attenuation*)
Set channel probe attenuation

set_trigger_level(*level*)
Set edge trigger level (in Volts)

set_trigger_mode(*trigger_mode='auto'*)
Set trigger mode.

Can be either `"auto"` or `"norm"`.

set_vertical_position(*channel*, *position*)
Set channel vertical position (offset of the zero volt line; in V)

set_vertical_span(*channel*, *span*)
Set channel vertical span (in V)

setup_edge_trigger(*source*, *level*, *coupling='dc'*, *slope='rise'*)
Setup edge trigger.

Set source, level, coupling and slope (see corresponding methods for details).

sleep(*delay*)
Wait for *delay* seconds

stop_grabbing()
Stop grabbing or waiting (equivalent to `self.grab_continuous(False)`)

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with **;** delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_for_grabbing(*timeout=None*)
Wait until the acquisition is complete

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class pylablib.devices.Tektronix.base.**DPO2000**(*addr*, *nchannels='auto'*)
Bases: *pylablib.devices.Tektronix.base.ITektronixScope*

Tektronix DPO2000 series oscilloscope.

Parameters

- **addr** – device address; usually a VISA address string such as `"USB0::0x0699::0x0364::C000000::INSTR"`
- **nchannels** – can specify number of channels on the oscilloscope; by default, autodetect number of channels (might take several seconds on connection)

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *TektronixError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

ReraiseError
alias of *TektronixBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If read_echo==True, assume that the device first echoes the input and skip it.

close()
Close the backend

enable_channel(*channel*, *enabled=True*)
Enable or disable given channel

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If one_line==True, read only a single line.

force_trigger()
Force trigger event

static get_arg_type(*arg*)
Autodetect argument type

get_channels(*only_main=False*)
Get the list of all input channels (if only_main==True) or all available channels (if only_main==False)

get_channels_number()
Get the number of channels

get_coupling(*channel*)
Get channel coupling.

Can be "ac", "dc", or "gnd".

get_data_format()
Get data transfer format.

Return a string describing the format; can be either "ascii", or a numpy-style format string (e.g., "<u2").

get_data_pts_range()
Get range of data points to read.

The range is defined from 1 to the points number (returned by *get_points_number()*).

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_edge_trigger_coupling()
Get edge trigger coupling ("ac" or "dc")

get_edge_trigger_slope()
Get edge trigger slope ("fall" or "rise")

get_edge_trigger_source()
Get edge trigger source.

Can be an integer indicating channel number or a name of a special channel.

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_horizontal_offset()
Get horizontal offset (position of the center of the sweep; in seconds)

get_horizontal_span()
Get horizontal span (in seconds)

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_points_number(*kind='send'*)
Get number of datapoints in various context.

kind defines the context. It can be "acq" (number of points acquired), "trace" (number of points in the source of the read-out trace; can be lower than "acq" if the data resolution is reduced, or if the source is not a channel data), or "send" (number of points in the sent waveform; can be lower than "trace" if *get_data_pts_range()* is used to specify and incomplete range). Not all kinds are defined for all scope model (e.g., "trace" is not defined for TDS2000 series oscilloscopes).

For length of read-out trace, see also *get_data_pts_range()*.

get_probe_attenuation(*channel*)
Get channel probe attenuation

get_selected_channel()
Get selected source channel.

Return number if it is a real channel, or a string name otherwise.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_trigger_level()
Get edge trigger level (in Volts)

get_trigger_mode()
Get trigger mode.

Can be either "auto" or "norm".

get_trigger_state()
Get trigger state.

Can be "armed" (acquiring pretrigger), "ready" (pretrigger acquired, wait for trigger event), "trigger" (triggered, acquiring the rest of the waveform), "auto" ("auto" mode trigger is acquiring data in the absence of trigger), "save" (acquisition is stopped), or "scan" (oscilloscope in the scan mode)

get_vertical_position(*channel*)
Get channel vertical position (offset of the zero volt line; in V)

get_vertical_span(*channel*)
Get channel vertical span (in V)

get_wfmpre(*channel=None*, *enable=True*)
Get preamble dictionary describing all scaling and format data for the given channel or a list of channels.

Can be acquired once and used in subsequent multiple reads to save time on re-requesting. If *channel* is None, use the currently selected channel. If enable==True, make sure that the requested channel is enabled; getting preamble for disabled channels raises an error.

grab_continuous(*enable=True*)
Start or stop continuous grabbing

grab_single(*wait=True*, *software_trigger=False*, *wait_timeout=None*)
Set single waveform grabbing and wait for acquisition.

If wait==True, wait until the acquisition is complete; otherwise, return immediately. if software_trigger==True, send the software trigger after setup (i.e., the device triggers immediately regardless of the input).

is_channel_enabled(*channel*)
Check if channel is enabled

is_continuous()
Check if grabbing is continuous or single

is_grabbing()
Check if acquisition is in progress.

Return True if the oscilloscope is recording data, or if the trigger is armed/ready and waiting; return False if the acquisition is stopped. To check if the trigger has been triggered, use *get_trigger_state()*.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

normalize_channel_name(*channel*)
Normalize channel name as represented by the oscilloscope

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is *DataFormat* description in numpy format (e.g., "<u2").

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

read_multiple_sweeps(*channels*, *wfmpres=None*, *ensure_fmt=False*, *timeout=None*, *return_wfmpres=None*)
Read data from a multiple channels channel.

Parameters

- **channels** – list of channel indices or names
- **wfmpres** – optional list or dictionary of preambles (obtained using *get_wfmpre()*); if it is `None`, obtain during reading, which slows down the data acquisition a bit
- **ensure_fmt** – if `True`, make sure that oscilloscope data format agrees with the one in *wfmpre*
- **timeout** – read timeout
- **return_wfmpres** – if `True`, return tuple `(sweeps, wfmpres)`, where `wfmpres` can be used for further sweep readouts.

read_raw_data(*channel=None*, *fmt=None*, *timeout=None*)
Request, read and parse raw data at a given channel.

fmt is data format (e.g., `"i1"`, `"<i2"`, or `"ascii"`) or `"default"`, which uses the default oscilloscope format (usually binary with smallest appropriate byte size). If *fmt* is `None`, use the current format. If *channel* is `None`, use the currently selected channel.

Returned data is raw (i.e., not scaled and without x axis).

read_sweep(*channel*, *wfmpre=None*, *ensure_fmt=True*, *timeout=None*)
Read data from a single channel.

Parameters

- **channel** – channel index or name
- **wfmpre** – optional preamble dictionary (obtained using *get_wfmpre()*); if it is `None`, obtain during reading, which slows down the data acquisition a bit

- **ensure_fmt** – if `True`, make sure that oscilloscope data format agrees with the one in *wfmpre*
- **timeout** – read timeout

reconnect (*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset ()
Reset the device (by default, `"*RST"` command)

select_channel (*channel*)
Select a channel to read data.

Doesn't need to be called explicitly, if `read_multiple_sweeps()` or `read_sweep()` are used.

set_coupling (*channel*, *coupling='dc'*)
Set channel coupling.

Can be `"ac"`, `"dc"`, or `"gnd"`.

set_data_format (*fmt='default'*)
Set data transfer format.

fmt is a string describing the format; can be either `"ascii"`, or a numpy-style format string (e.g., `"<u2"`). If `"default"`, use the oscilloscope default format (usually binary with smallest appropriate byte size).

set_data_pts_range (*rng=None*)
Set range of data points to read.

The range is defined from 1 to the points number (returned by `get_points_number()` with `kind="acq"`). If `rng is None`, set the full range.

set_device_variable (*key*, *value*)
Set the value of a settings parameter

set_edge_trigger_coupling (*coupling*)
Set edge trigger coupling (`"ac"` or `"dc"`)

set_edge_trigger_slope (*slope*)
Set edge trigger slope (`"fall"` or `"rise"`)

set_edge_trigger_source (*channel*)
Get edge trigger source.

Can be an integer indicating channel number or a name of a special channel.

set_horizontal_offset (*offset=0.0*)
Set horizontal offset (position of the center of the sweep; in seconds)

set_horizontal_span (*span*)
Set horizontal span (in seconds)

set_points_number (*pts_num*, *reset_limits=True*)
Set number of datapoints to record when acquiring a trace.

If `reset_limits==True`, reset the datapoints range (`set_data_pts_range()`) to the full range. The actual set value (returned by this method) can be different from the requested value.

set_probe_attenuation (*channel*, *attenuation*)
Set channel probe attenuation

set_trigger_level(*level*)
Set edge trigger level (in Volts)

set_trigger_mode(*trigger_mode='auto'*)
Set trigger mode.

Can be either `"auto"` or `"norm"`.

set_vertical_position(*channel*, *position*)
Set channel vertical position (offset of the zero volt line; in V)

set_vertical_span(*channel*, *span*)
Set channel vertical span (in V)

setup_edge_trigger(*source*, *level*, *coupling='dc'*, *slope='rise'*)
Setup edge trigger.

Set source, level, coupling and slope (see corresponding methods for details).

sleep(*delay*)
Wait for *delay* seconds

stop_grabbing()
Stop grabbing or waiting (equivalent to `self.grab_continuous(False)`)

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_for_grabbing(*timeout=None*)
Wait until the acquisition is complete

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (`str`) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.

- **arg_type** (*str*) – Argument type. Can be 'raw' (in which case data is sent raw), 'string', 'int', 'float', 'bool', a format string (such as '{:.3f}') or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., arg_type='{0};{1}' with arg=[1,2] will produce a string '1;2'); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with ",".
- **unit** (*str*) – If not None, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple (false_value, true_value) of two strings to represent bool argument; by default, use ._bool_selector attribute.
- **wait_sync** – if True, append the sync command (specified as ._wait_sync_comm attribute, "*OPC?" by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if None, use the class default ._default_write_sync attribute (False by default).
- **read_echo** (*bool*) – If True, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if read_echo==True.

Module contents

pylablib.devices.Thorlabs package

Submodules

pylablib.devices.Thorlabs.TLCamera module

class pylablib.devices.Thorlabs.TLCamera.**LibraryController**(*lib*)

Bases: *pylablib.devices.utils.load_lib.LibraryController*

close(*opid*)

Mark device closing.

Return tuple (close_result, uninit_result) with the results of the closing and the shutdown. If library does not need to be shut down yet, set uninit_result=None

open()

Mark device opening.

Return tuple (init_result, open_result, opid) with the results of the initialization and the opening, and the opening ID which should afterwards be used for closing. If library is already initialized, set init_result=None

preinit()

Pre-initialize the library, if it hasn't been done already

shutdown()

Close all opened connections and shutdown the library

temp_open()

Context for temporarily opening a new device connection

pylablib.devices.Thorlabs.TLCamera.**list_cameras**()
List connected TLCamera cameras

pylablib.devices.Thorlabs.TLCamera.**get_cameras_number**()
Get number of connected TLCamera cameras

class pylablib.devices.Thorlabs.TLCamera.**TDeviceInfo**(*model*, *name*, *serial_number*, *firmware_version*)
Bases: tuple

count()
Return number of occurrences of value.

firmware_version

index()
Return first index of value.

Raises ValueError if the value is not present.

model

name

serial_number

class pylablib.devices.Thorlabs.TLCamera.**TFrameInfo**(*frame_index*, *framestamp*, *pixelclock*, *pixeltype*, *offset*)
Bases: tuple

count()
Return number of occurrences of value.

frame_index

framestamp

index()
Return first index of value.

Raises ValueError if the value is not present.

offset

pixelclock

pixeltype

class pylablib.devices.Thorlabs.TLCamera.**ThorlabsTLCamera**(*serial=None*)
Bases: *pylablib.devices.interface.camera.IBinROICamera*, *pylablib.devices.interface.camera.IExposureCamera*

Thorlabs TSI camera.

Parameters serial (*str*) – camera serial number; can be either a string obtained using *list_cameras()* function, or None, which means connecting to the first available camera (not recommended unless only one camera is connected)

Error = <Mock name='mock.ThorlabsTLCameraError' id='140318670433168'>

TimeoutError = <Mock spec='str' id='140318678495312'>

open()
Open connection to the camera

close()
Close connection to the camera

is_opened()
Check if the device is connected

get_device_info()
Get camera model data.

Return tuple (model, name, serial_number, firmware_version).

class RingBuffer
Bases: object

Frames ring buffer.

Reacts to each new frame and stores it in the internal buffer.

reset()
Reset buffer and internal counters

setup(*buffsize*, *frame_dim*)
Setup a new buffer with the given maximal number of frames and frame dimensions

cleanup()
Cleanup the buffer

new_frame(*handle*, *buffer*, *idx*, *metadata*, *metadata_size*, *context*)
Callback for receiving a new frame

wait_for_frame(*idx=None*, *timeout=None*)
Wait for a new frame acquisition

get_frame(*idx*)
Get the frame with the given index (or None if it is outside the buffer range)

get_status()
Get buffer status (acquired, missed, stored)

get_frame_timings()
Get acquisition timing.

Return tuple (exposure, frame_period).

set_exposure(*exposure*)
Set camera exposure

get_trigger_mode()
Get trigger mode.

Can be "int" (internal/software), "ext" (external/hardware), or "bulb" (bulb trigger).

set_trigger_mode(*mode*)
Set trigger mode.

Can be "int" (internal/software), "ext" (external/hardware), or "bulb" (bulb trigger).

get_ext_trigger_parameters()
Return external trigger polarity

setup_ext_trigger(*polarity*)
Setup external trigger polarity ("rise" or "fall")

send_software_trigger()
Send software trigger signal

get_pixel_correction_parameters()
Return pixel correction parameters (enabled, threshold)

setup_pixel_correction(*enable=True*, *threshold=None*)
Enable or disable hotpixel correction and set its threshold (`None` means keep unchanged)

get_timestamp_clock_frequency()
Return frequency of the frame timestamp clock (in Hz)

setup_acquisition(*nframes=100*)
Setup acquisition.

nframes determines number of size of the ring buffer (by default, 100).

clear_acquisition()
Clear acquisition settings

start_acquisition(*frames_per_trigger='default'*, *auto_start=True*, *nframes=None*)
Start camera acquisition.

Parameters

- **frames_per_trigger** – number of frames to acquire per trigger (software of hardware); `None` means unlimited number; by default, set to `None` for software trigger (i.e., run until stopped), and 1 for hardware trigger (i.e., one frame per trigger pulse)
- **auto_start** – if `True` and the trigger is set into software mode, automatically start recording; otherwise, only start recording when *send_software_trigger()* is called explicitly; this value is meaningless in the hardware or bulb trigger mode
- **nframes** – number of frames in the ring buffer

stop_acquisition()
Stop acquisition

acquisition_in_progress()
Check if acquisition is in progress

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend, hbin, vbin)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0), *hbin* and *vbin* specify binning.

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*, *hbin=1*, *vbin=1*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0), *hbin* and *vbin* specify binning. By default, all non-supplied parameters take extreme values (0 for start, maximal for end, 1 for binning).

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning. In some cameras, the step and the minimal size depend on the binning, which can be supplied.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of `TFrameInfo` instances describing frame index and frame metadata, which contains framestamp, pixel clock, pixel format, and pixel offset; if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

FrameTransferError
alias of `pylablib.devices.interface.camera.DefaultFrameTransferError`

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_exposure()
Get current exposure

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by `set_frame_info_format()`) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_newest_image(*peek=False, return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_oldest_image(*peek=False, return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

set_device_variable(*key, value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt, include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

snap(*timeout=5.0, return_info=False*)
Snap a single frame

wait_for_frame (*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"`` (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

pylablib.devices.Thorlabs.base module

exception pylablib.devices.Thorlabs.base.**ThorlabsError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Thorlabs error

args

with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Thorlabs.base.**ThorlabsBackendError** (*exc*)
Bases: *pylablib.devices.Thorlabs.base.ThorlabsError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Thorlabs backend communication error

args

with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Thorlabs.base.**ThorlabsTimeoutError**
Bases: *pylablib.devices.Thorlabs.base.ThorlabsError*

Thorlabs timeout error

args

with_traceback ()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

pylablib.devices.Thorlabs.kinesis module

pylablib.devices.Thorlabs.kinesis.**list_kinesis_devices** (*filter_ids=True*)
List all Thorlabs APT/Kinesis devices connected to this PC.

Return list of tuples (conn, description). If `filter_ids==True`, only leave devices with Thorlabs-like IDs (8-digit numbers). Otherwise, show all devices (some of them might not be Thorlabs-related).

class pylablib.devices.Thorlabs.kinesis.**TDeviceInfo** (*serial_no*, *model_no*, *fw_ver*, *hw_type*, *hw_ver*, *mod_state*, *nchannels*, *notes*)
Bases: `tuple`

count()
Return number of occurrences of value.

fw_ver

hw_type

hw_ver

index()
Return first index of value.

Raises ValueError if the value is not present.

mod_state

model_no

nchannels

notes

serial_no

class pylablib.devices.Thorlabs.kinesis.**BasicKinesisDevice**(*conn*, *timeout=3.0*)
Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*

Generic Kinesis device.

Implements FTDI chip connectivity via pyft232 (virtual serial interface).

Parameters **conn** – serial connection parameters (usually an 8-digit device serial number).

Error
alias of *pylablib.devices.Thorlabs.base.ThorlabsError*

static list_devices(*filter_ids=True*)
List all connected devices.

Return list of tuples `(conn, description)`. If `filter_ids==True`, only leave devices with Thorlabs-like IDs (8-digit numbers). Otherwise, show all devices (some of them might not be Thorlabs-related).

send_comm(*messageID*, *param1=0*, *param2=0*, *source=1*, *dest=80*)
Send a message with no associated data.

For details, see APT communications protocol.

send_comm_data(*messageID*, *data*, *source=1*, *dest=80*)
Send a message with associated data.

For details, see APT communications protocol.

class CommShort(*messageID*, *param1*, *param2*, *source*, *dest*)
Bases: `tuple`

count()
Return number of occurrences of value.

dest

index()
Return first index of value.

Raises ValueError if the value is not present.

messageID

param1

param2

source

class CommData (*messageID*, *data*, *source*, *dest*)
Bases: `tuple`

count ()
Return number of occurrences of value.

data

dest

index ()
Return first index of value.

Raises ValueError if the value is not present.

messageID

source

recv_comm (*expected_id=None*)
Receive a message.

Return either `CommShort` or `CommData` depending on the message type (fixed length with two parameters, or variable length with associated data). If *expected_id* is not `None` and the received message ID is different from *expected_id*, raise an error. For details, see APT communications protocol.

query (*messageID*, *param1=0*, *param2=0*, *source=1*, *dest=80*, *replyID=-1*)
Send a query to the device and receive the reply.

A combination of `send_comm()` and `recv_comm()`. If *replyID* is not `None`, specifies the expected reply message ID; if -1 (default), set to te be `messageID+1` (the standard convention).

add_background_comm (*messageID*)
Mark given messageID as a 'background' message, which can be sent at any point without prompt (e.g., some operation confirmation).

If it is received instead during `recv_comm_` operations, it is ignored, and the corresponding counter is increased.

check_background_comm (*messageID*)
Return message counter and the last message value (`None` if not message received yet) of a given 'background' message

get_device_info (*dest=80*)
Get device info.

get_number_of_channels ()
Get number of channels on the device

blink (*channel=1*, *dest=80*)
Identify the physical device (by, e.g., blinking status LED or screen)

class NoParameterCaller (*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings (*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

class pylablib.devices.Thorlabs.kinesis.**TVelocityParams**(*min_velocity*, *acceleration*, *max_velocity*)

Bases: `tuple`

acceleration

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

max_velocity

min_velocity

class pylablib.devices.Thorlabs.kinesis.**TJogParams**(*mode*, *step_size*, *min_velocity*, *acceleration*, *max_velocity*, *stop_mode*)

Bases: tuple

acceleration

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

max_velocity

min_velocity

mode

step_size

stop_mode

class pylablib.devices.Thorlabs.kinesis.**TGenMoveParams**(*backlash_distance*)

Bases: tuple

backlash_distance

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

class pylablib.devices.Thorlabs.kinesis.**THomeParams**(*home_direction*, *limit_switch*, *velocity*, *offset_distance*)

Bases: tuple

count()
Return number of occurrences of value.

home_direction

index()
Return first index of value.

Raises ValueError if the value is not present.

limit_switch

offset_distance

velocity

class pylablib.devices.Thorlabs.kinesis.**TLimitSwitchParams**(*hw_kind_cw*, *hw_kind_ccw*, *hw_swapped*, *sw_position_cw*, *sw_position_ccw*, *sw_kind*)

Bases: tuple

count()
Return number of occurrences of value.

hw_kind_ccw

hw_kind_cw

hw_swapped

index()
Return first index of value.

Raises ValueError if the value is not present.

sw_kind

sw_position_ccw

sw_position_cw

class pylablib.devices.Thorlabs.kinesis.**KinesisDevice**(*conn*, *timeout=3.0*)
Bases: *pylablib.devices.Thorlabs.kinesis.BasicKinesisDevice*, *pylablib.devices.interface.stage.IStage*

status_bits = [(1, 'sw_bk_lim'), (2, 'sw_fw_lim'), (16, 'moving_bk'), (32, 'moving_fw'

class CommData(*messageID*, *data*, *source*, *dest*)
Bases: tuple

count()
Return number of occurrences of value.

data

dest

index()
Return first index of value.

Raises ValueError if the value is not present.

messageID

source

class CommShort(*messageID*, *param1*, *param2*, *source*, *dest*)
Bases: tuple

count()
Return number of occurrences of value.

dest

index()
Return first index of value.

Raises ValueError if the value is not present.

messageID

param1

param2

source

Error
alias of *pylablib.devices.Thorlabs.base.ThorlabsError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

add_background_comm(*messageID*)
Mark given messageID as a 'background' message, which can be sent at any point without prompt (e.g., some operation confirmation).

If it is received instead during `recv_comm_` operations, it is ignored, and the corresponding counter is increased.

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

blink(*channel=1*, *dest=80*)
Identify the physical device (by, e.g., blinking status LED or screen)

check_background_comm(*messageID*)
Return message counter and the last message value (`None` if not message received yet) of a given 'background' message

close()
Close the backend

get_device_info(*dest=80*)
Get device info.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_number_of_channels()
Get number of channels on the device

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

static list_devices(*filter_ids=True*)
List all connected devices.

Return list of tuples (conn, description). If filter_ids==True, only leave devices with Thorlabs-like IDs (8-digit numbers). Otherwise, show all devices (some of them might not be Thorlabs-related).

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

query(*messageID*, *param1=0*, *param2=0*, *source=1*, *dest=80*, *replyID=-1*)
Send a query to the device and receive the reply.

A combination of send_comm() and recv_comm(). If *replyID* is not None, specifies the expected reply message ID; if -1 (default), set to te be messageID+1 (the standard convention).

recv_comm(*expected_id=None*)
Receive a message.

Return either CommShort or CommData depending on the message type (fixed length with two parameters, or variable length with associated data). If *expected_id* is not None and the received message ID is different from *expected_id*, raise an error. For details, see APT communications protocol.

send_comm(*messageID*, *param1=0*, *param2=0*, *source=1*, *dest=80*)
Send a message with no associated data.

For details, see APT communications protocol.

send_comm_data(*messageID*, *data*, *source=1*, *dest=80*)
Send a message with associated data.

For details, see APT communications protocol.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

class pylablib.devices.Thorlabs.kinesis.**TFlipperParameters**(*transit_time*, *io1_oper_mode*, *io1_sig_mode*, *io1_pulse_width*, *io2_oper_mode*, *io2_sig_mode*, *io2_pulse_width*)

Bases: tuple

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

io1_oper_mode

io1_pulse_width

io1_sig_mode

io2_oper_mode

io2_pulse_width

io2_sig_mode

transit_time

class pylablib.devices.Thorlabs.kinesis.**MFF**(*conn*)

Bases: *pylablib.devices.Thorlabs.kinesis.KinesisDevice*

MFF (Motorized Filter Flip Mount) device.

Implements FTDI chip connectivity via pyft232 (virtual serial interface).

Parameters conn – serial connection parameters (usually 8-digit device serial number).

get_status_n(*channel=1*)

Get numerical status of the device.

For details, see APT communications protocol.

get_status(*channel=1*)

Get device status.

Return list of status strings, which can include `"sw_fw_lim"` (forward limit switch reached), `"sw_bk_lim"` (backward limit switch reached), `"moving_fw"` (moving forward), `"moving_bk"` (moving backward), `"jogging_fw"` (jogging forward), `"jogging_bk"` (jogging backward), `"homing"` (homing), `"homed"` (homing done), `"tracking"`, `"settled"`, `"motion_error"` (excessive position error), `"current_limit"` (motor current limit exceeded), or `"enabled"` (motor is enabled).

wait_for_status(*status*, *enabled*, *channel=1*, *timeout=None*, *period=0.05*)

Wait until the given status (or list of status bits) is in the desired state.

status is a string or a list of strings describing the status bits to monitor; for possible values, see *get_status()*. If `enabled==True`, wait until one of the given statuses is enabled; otherwise, wait until all given statuses are disabled. *period* specifies status checking period (in s).

move_to_state(*state*, *channel=0*)

Move to the given flip mount state (either 0 or 1)

get_state(*channel=0*)

Get the flip mount state (either 0 or 1).

Return `None` if the mount is current moving (i.e., the state os undefined)

get_flipper_parameters(*channel=1*)

Get current flipper parameters `(transit_time, io1_oper_mode, io1_sig_mode, io1_pulse_width, io2_oper_mode, io2_sig_mode, io2_pulse_width)`

`transit_time` specifies transit time (in seconds between 0.3 and 2.8); `io*_oper_mode` specifies operation mode (in vs. out and position vs. motion input/indication), `io*_sig_mode` specifies signal mode (button input, voltage edge input, edge output or pulse output). `io*_pulse_width` specifies output pulse width if the corresponding output mode is selected. For detailed mode description, see the flip mirror or APT manual.

setup_flipper(*transit_time=None*, *io1_oper_mode=None*, *io1_sig_mode=None*, *io1_pulse_width=None*, *io2_oper_mode=None*, *io2_sig_mode=None*, *io2_pulse_width=None*, *channel=1*)

Set flipper parameters.

`transit_time` specifies transit time (in seconds between 0.3 and 2.8); `io*_oper_mode` specifies operation mode (in vs. out and position vs. motion input/indication), `io*_sig_mode` specifies signal mode (button input, voltage edge input, edge output or pulse output). `io*_pulse_width` specifies output pulse width if the corresponding output mode is selected. If any parameter is `None`, use the current value. For detailed mode description, see the flip mirror or APT manual.

class CommData(*messageID*, *data*, *source*, *dest*)
Bases: `tuple`

count()
Return number of occurrences of value.

data

dest

index()
Return first index of value.

Raises ValueError if the value is not present.

messageID

source

class CommShort(*messageID*, *param1*, *param2*, *source*, *dest*)
Bases: `tuple`

count()
Return number of occurrences of value.

dest

index()
Return first index of value.

Raises ValueError if the value is not present.

messageID

param1

param2

source

Error
alias of *pylablib.devices.Thorlabs.base.ThorlabsError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

add_background_comm(*messageID*)
Mark given messageID as a 'background' message, which can be sent at any point without prompt (e.g., some operation confirmation).

If it is received instead during `recv_comm_` operations, it is ignored, and the corresponding counter is increased.

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

blink(*channel=1*, *dest=80*)
Identify the physical device (by, e.g., blinking status LED or screen)

check_background_comm(*messageID*)
Return message counter and the last message value (`None` if not message received yet) of a given 'background' message

close()
Close the backend

get_device_info(*dest=80*)
Get device info.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_number_of_channels()
Get number of channels on the device

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

static list_devices(*filter_ids=True*)
List all connected devices.

Return list of tuples `(conn, description)`. If `filter_ids==True`, only leave devices with Thorlabs-like IDs (8-digit numbers). Otherwise, show all devices (some of them might not be Thorlabs-related).

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

query(*messageID*, *param1=0*, *param2=0*, *source=1*, *dest=80*, *replyID=-1*)
Send a query to the device and receive the reply.

A combination of *send_comm()* and *recv_comm()*. If *replyID* is not `None`, specifies the expected reply message ID; if -1 (default), set to te be `messageID+1` (the standard convention).

recv_comm(*expected_id=None*)
Receive a message.

Return either *CommShort* or *CommData* depending on the message type (fixed length with two parameters, or variable length with associated data). If *expected_id* is not `None` and the received message ID is different from *expected_id*, raise an error. For details, see APT communications protocol.

send_comm(*messageID*, *param1=0*, *param2=0*, *source=1*, *dest=80*)
Send a message with no associated data.

For details, see APT communications protocol.

send_comm_data(*messageID*, *data*, *source=1*, *dest=80*)
Send a message with associated data.

For details, see APT communications protocol.

set_device_variable(*key*, *value*)
Set the value of a settings parameter

status_bits = [(1, 'sw_bk_lim'), (2, 'sw_fw_lim'), (16, 'moving_bk'), (32, 'moving_fw'

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

class pylablib.devices.Thorlabs.kinesis.**KinesisMotor**(*conn*, *scale='step'*)
Bases: *pylablib.devices.Thorlabs.kinesis.KinesisDevice*

Thorlabs motor controller.

Implements FTDI chip connectivity via pyft232 (virtual serial interface).

The physical units are encoder steps for position (ratio to m or degrees depends on the connected stage), steps/sec for velocity, and steps/sec^2 for acceleration.

Parameters

- **conn** (*str*) – serial connection parameters (usually an 8-digit device serial number).
- **scale** – scale of the position, velocity, and acceleration units to the internals units; can be `"stage"` (attempt to autodetect motor and stage parameters), a string with the name of the stage, e.g., `"MTS50-Z8"` or `"DDR100"` (use the stage name to extract the scale; determine velocity and acceleration from this scale and the motor model), `"step"` (use encoder/motor steps as units; determine velocity and acceleration from this scale and the motor model), a single number (use this as the ratio of internal steps to physical units; determine velocity and acceleration from this scale and the motor model), or a 3-tuple of numbers `(position_scale, velocity_scale, acceleration_scale)` which gives the ratio of internal units to physical units (useful for new or unrecognized controllers or stages, as no autodetection is required); in the case of unrecognized devices, use internal units (same as setting `scale=(1,1,1)`); if the scale can't be autodetected, it can be obtained from the APT manual knowing the device and the stage model

get_scale()
Get the scaling coefficients.

Return a tuple `(position_scale, velocity_scale, acceleration_scale)` for scaling of the physical units in terms of internal units. To get the coefficients source and physical units, use *get_scale_units()*.

get_scale_units()
Get units used for calculating scaled position, velocity and acceleration values.

Can be "deg" (autodetected rotational stage: deg, deg/s and deg/s^2), "m" (autodetected translational stage: m, m/sec and m/sec^2), "step" (autodetected driver but not detected step scale: steps, steps/sec and steps/sec^2) "user_step" (autodetected driver and user supplied step scale: user-supplied step scale for position, same units per sec or sec^2 for velocity and acceleration), 'user" (all three scales are supplied by user), or "internal" (no scales are supplied or detected, use device internal units)

get_stage()
Return the name of the stage which was supplied by the usr or autodetected.

If the stake is unknown, return None

get_status_n(*channel=1*)
Get numerical status of the device.

For details, see APT communications protocol.

get_status(*channel=1*)
Get device status.

Return list of status strings, which can include "sw_fw_lim" (forward limit switch reached), "sw_bk_lim" (backward limit switch reached), "moving_fw" (moving forward), "moving_bk" (moving backward), "jogging_fw" (jogging forward), "jogging_bk" (jogging backward), "homing" (homing), "homed" (homing done), "tracking", "settled", "motion_error" (excessive position error), "current_limit" (motor current limit exceeded), or "enabled" (motor is enabled).

wait_for_status(*status*, *enabled*, *channel=1*, *timeout=None*, *period=0.05*)
Wait until the given status (or list of status bits) is in the desired state.

status is a string or a list of strings describing the status bits to monitor; for possible values, see *get_status()*. If enabled==True, wait until one of the given statuses is enabled; otherwise, wait until all given statuses are disabled. *period* specifies status checking period (in s).

home(*sync=True*, *force=False*, *channel=1*, *timeout=None*)
Home the device.

If sync==True, wait until homing is done (with the given timeout). If force==False, only home if the device isn't homed already.

is_homing(*channel=1*)
Check if homing is in progress

is_homed(*channel=1*)
Check if the device is homed

wait_for_home(*channel=1*, *timeout=None*)
Wait until the device is homed

get_position(*channel=1*, *scale=True*)
Get current position.

If scale==True, return value in the physical units (see class description); otherwise, return it in the device internal units (steps).

set_position_reference(*position=0*, *channel=1*, *scale=True*)
Set position reference (actual motor position stays the same).

If scale==True, assume that the position is in the physical units (see class description); otherwise, assume it is in the device internal units (steps).

move_by(*distance=1*, *channel=1*, *scale=True*)
Move by a given amount (positive or negative) from the current position.

If `scale==True`, assume that the distance is in the physical units (see class description); otherwise, assume it is in the device internal units (steps).

move_to(*position*, *channel=1*, *scale=True*)
Move to *position* (positive or negative).

If `scale==True`, assume that the position is in the physical units (see class description); otherwise, assume it is in the device internal units (steps).

jog(*direction*, *channel=1*, *kind='continuous'*)
Jog in the given direction (`"+"` or `"-"`).

If `kind=="continuous"`, simply start motion in the given direction at the maximal speed until either the motor is stopped explicitly, or the limit is reached (this uses `MOT_MOVE_VELOCITY` command). If `kind=="builtin"`, use the built-in `MOT_MOVE_JOG` command, whose parameters are specified by `get_jog_parameters()`.

is_moving(*channel=1*)
Check if motion is in progress

wait_move(*channel=1*, *timeout=None*)
Wait until motion command is done

stop(*immediate=False*, *sync=True*, *channel=1*, *timeout=None*)
Stop the motion.

If `immediate==True` make an abrupt stop; otherwise, slow down gradually. If `sync==True`, wait until the motion is stopped.

wait_for_stop(*channel=1*, *timeout=None*)
Wait until motion or homing is done

get_velocity_parameters(*channel=1*, *scale=True*)
Get current velocity parameters `(min_velocity, acceleration, max_velocity)`

If `scale==True`, return values in the physical units (see class description); otherwise, return it in the device internal units.

setup_velocity(*min_velocity=None*, *acceleration=None*, *max_velocity=None*, *channel=1*, *scale=True*)
Set velocity parameters.

If any parameter is `None`, use the current value. If `scale==True`, assume that the specified values are in the physical units (see class description); otherwise, assume it is in the device internal units.

get_jog_parameters(*channel=1*, *scale=True*)
Get current jog parameters `(mode, step_size, min_velocity, acceleration, max_velocity, stop_mode)`

If `scale==True`, return values in the physical units (see class description); otherwise, return it in the device internal units.

setup_jog(*mode=None*, *step_size=None*, *min_velocity=None*, *acceleration=None*, *max_velocity=None*, *stop_mode=None*, *channel=1*, *scale=True*)
Set jog parameters.

If any parameter is `None`, use the current value. If `scale==True`, assume that the specified values are in the physical units (see class description); otherwise, assume it is in the device internal units.

get_homing_parameters(*channel=1*, *scale=True*)
Get current homing parameters (home_direction, limit_switch, velocity, offset_distance)

If scale==True, return values are in the physical units (see class description); otherwise, return it in the device internal units.

setup_homing(*home_direction=None*, *limit_switch=None*, *velocity=None*, *offset_distance=None*, *channel=1*, *scale=True*)
Set homing parameters.

If any parameter is None, use the current value. If scale==True, assume that the specified values are in the physical units (see class description); otherwise, assume it is in the device internal units.

get_gen_move_parameters(*channel=1*, *scale=True*)
Get general move parameters parameters (backlash_distance)

If scale==True, return values in the physical units (see class description); otherwise, return it in the device internal units.

setup_gen_move(*backlash_distance=None*, *channel=1*, *scale=True*)
Set jog parameters.

If any parameter is None, use the current value. If scale==True, assume that the specified value is in the physical units (see class description); otherwise, assume it is in the device internal units.

get_limit_switch_parameters(*channel=1*, *scale=True*)
Get current limit switch parameters (hw_kind_cw, hw_kind_ccw, hw_flipped, sw_position_cw, sw_position_ccw, sw_kind)

If scale==True, return values in the physical units (see class description); otherwise, return it in the device internal units (steps).

setup_limit_switch(*hw_kind_cw=None*, *hw_kind_ccw=None*, *hw_swapped=None*, *sw_position_cw=None*, *sw_position_ccw=None*, *sw_kind=None*, *channel=1*, *scale=True*)
Set home parameters.

If any parameter is None, use the current value. If scale==True, assume that the specified values are in the physical units (see class description); otherwise, assume it is in the device internal units (Steps).

class CommData(*messageID*, *data*, *source*, *dest*)
Bases: tuple

count()
Return number of occurrences of value.

data

dest

index()
Return first index of value.

Raises ValueError if the value is not present.

messageID

source

class CommShort(*messageID*, *param1*, *param2*, *source*, *dest*)
Bases: tuple

count()
Return number of occurrences of value.

dest

index()
Return first index of value.

Raises ValueError if the value is not present.

messageID

param1

param2

source

Error
alias of *pylablib.devices.Thorlabs.base.ThorlabsError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

add_background_comm(*messageID*)
Mark given messageID as a 'background' message, which can be sent at any point without prompt (e.g., some operation confirmation).

If it is received instead during `recv_comm_` operations, it is ignored, and the corresponding counter is increased.

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

blink(*channel=1*, *dest=80*)
Identify the physical device (by, e.g., blinking status LED or screen)

check_background_comm(*messageID*)
Return message counter and the last message value (`None` if not message received yet) of a given 'background' message

close()
Close the backend

get_device_info(*dest=80*)
Get device info.

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_number_of_channels()
Get number of channels on the device

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

is_opened()
Check if the device is connected

static list_devices(*filter_ids=True*)
List all connected devices.

Return list of tuples (conn, description). If filter_ids==True, only leave devices with Thorlabs-like IDs (8-digit numbers). Otherwise, show all devices (some of them might not be Thorlabs-related).

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

query(*messageID, param1=0, param2=0, source=1, dest=80, replyID=-1*)
Send a query to the device and receive the reply.

A combination of send_comm() and recv_comm(). If *replyID* is not None, specifies the expected reply message ID; if -1 (default), set to te be messageID+1 (the standard convention).

recv_comm(*expected_id=None*)
Receive a message.

Return either CommShort or CommData depending on the message type (fixed length with two parameters, or variable length with associated data). If *expected_id* is not None and the received message ID is different from *expected_id*, raise an error. For details, see APT communications protocol.

send_comm(*messageID, param1=0, param2=0, source=1, dest=80*)
Send a message with no associated data.

For details, see APT communications protocol.

send_comm_data(*messageID, data, source=1, dest=80*)
Send a message with associated data.

For details, see APT communications protocol.

set_device_variable(*key, value*)
Set the value of a settings parameter

status_bits = [(1, 'sw_bk_lim'), (2, 'sw_fw_lim'), (16, 'moving_bk'), (32, 'moving_fw'

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

pylablib.devices.Thorlabs.misc module

pylablib.devices.Thorlabs.serial module

class pylablib.devices.Thorlabs.serial.**ThorlabsSerialInterface**(*conn*)

Bases: `pylablib.core.devio.SCPI.SCPIDevice`

Generic Thorlabs device interface using Serial communication.

Parameters **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)

Error

alias of `pylablib.devices.Thorlabs.base.ThorlabsError`

ReraiseError

alias of `pylablib.devices.Thorlabs.base.ThorlabsBackendError`

open()

Open the backend

BackendError

alias of `pylablib.core.devio.comm_backend.DeviceBackendError`

class NoParameterCaller(*device*, *kind*)

Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)

Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)

Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in `read()`. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()

Close the backend

flush(*one_line=False*)

Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

static get_arg_type(*arg*)

Autodetect argument type

get_device_variable(*key*)

Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)

Get the device status register (by default, `"*ESR?"` command)

get_full_info(*include=0*)

Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)

Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

static parse_array_data(*data, fmt, include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string', timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False, timeout=None, flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True, ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

set_device_variable(*key, value*)
Set the value of a settings parameter

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (`str`) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (`str`) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (`str`) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (`tuple`) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (`bool`) – If `True`, read a single line after write.
- **read_echo_delay** (`float`) – The delay between write and read if `read_echo==True`.

class pylablib.devices.Thorlabs.serial.**FW**(*conn*, *respect_bound=True*)

Bases: *pylablib.devices.Thorlabs.serial.ThorlabsSerialInterface*

Thorlabs FW102/212 motorized filter wheels.

Parameters

- **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)
- **respect_bound** (*bool*) – if True, avoid crossing the boundary between the first and the last position in the wheel

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)

Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If read_echo==True, assume that the device first echoes the input and skip it.

get_position()

Get the wheel position (starting from 1)

set_position(*pos*)

Set the wheel position (starting from 1)

get_pcount()

Get the number of wheel positions (6 or 12)

set_pcount(*pcount*)

Set the number of wheel positions (6 or 12)

get_speed_mode()

Get the motion speed mode ("low" or "high")

set_speed_mode(*speed_mode*)

Set the motion speed mode ("low" or "high")

get_trigger_mode()

Get the trigger mode ("in" to input external trigger, "out" to output trigger)

set_trigger_mode(*trigger_mode*)

Set the trigger mode ("in" to input external trigger, "out" to output trigger)

get_sensor_mode()

Get the sensor mode ("off" to turn off when idle to eliminate stray light, "on" to remain on)

set_sensor_mode(*sensor_mode*)

Set the sensor mode ("off" to turn off when idle to eliminate stray light, "on" to remain on)

store_settings()

Store current settings as default

BackendError

alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error

alias of *pylablib.devices.Thorlabs.base.ThorlabsError*

class NoParameterCaller(*device*, *kind*)

Bases: object

Class to simplify calling functions without a parameter

ReraiseError

alias of *pylablib.devices.Thorlabs.base.ThorlabsBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

close()
Close the backend

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)
Get the device status register (by default, `"*ESR?"` command)

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data, fmt, include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

set_device_variable(*key*, *value*)
Set the value of a settings parameter

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive `write()` operations are bundled together with `;` delimiter. The actual write is performed at the `read()`/`ask()` operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform `wait_sync()`), `'dev'` (perform `wait_dev()`) or `'none'` (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class pylablib.devices.Thorlabs.serial.**FWv1**(*conn*, *pcount=6*, *respect_bound=True*)
Bases: *pylablib.devices.Thorlabs.serial.ThorlabsSerialInterface*

Thorlabs FW102/212 v1.0 (older version) motorized filter wheels.

Parameters

- **conn** – serial connection parameters (usually port or a tuple containing port and baudrate)
- **pcount** – number of positions in the wheel
- **respect_bound** (*bool*) – if `True`, avoid crossing the boundary between the first and the last position in the wheel

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If `read_echo==True`, assume that the device first echoes the input and skip it.

get_position()
Get the wheel position (starting from 1)

set_position(*pos*)
Set the wheel position (starting from 1)

get_pcount()
Get the number of wheel positions (6 or 12)

get_trigger_mode()
Get the trigger mode ("in" to input external trigger, "out" to output trigger)

set_trigger_mode(*trigger_mode*)
Set the trigger mode ("in" to input external trigger, "out" to output trigger)

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *pylablib.devices.Thorlabs.base.ThorlabsError*

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

ReraiseError
alias of *pylablib.devices.Thorlabs.base.ThorlabsBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict {name: value} of the available device settings. Non-applicable settings are ignored.

close()
Close the backend

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If one_line==True, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)
Get the device status register (by default, "*ESR?" command)

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI '*IDN?' command)

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, `"*RST"` command)

set_device_variable(*key*, *value*)
Set the value of a settings parameter

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()

Context manager for using a write buffer.

While it's active, all the consecutive *write()* operations are bundled together with `;` delimiter. The actual write is performed at the *read()*/*ask()* operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)

Pause execution until device overlapped commands are complete.

wait_type is either `'sync'` (perform *wait_sync()*), `'dev'` (perform *wait_dev()*) or `'none'` (do nothing).

wait_dev()

Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)

Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)

Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with `","`.
- **arg_type** (*str*) – Argument type. Can be `'raw'` (in which case data is sent raw), `'string'`, `'int'`, `'float'`, `'bool'`, a format string (such as `'{:.3f}'`) or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., `arg_type='{0};{1}'` with `arg=[1,2]` will produce a string `'1;2'`); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with `","`.
- **unit** (*str*) – If not `None`, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple `(false_value, true_value)` of two strings to represent bool argument; by default, use `._bool_selector` attribute.
- **wait_sync** – if `True`, append the sync command (specified as `._wait_sync_comm` attribute, `"*OPC?"` by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if `None`, use the class default `._default_write_sync` attribute (`False` by default).
- **read_echo** (*bool*) – If `True`, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if `read_echo==True`.

class pylablib.devices.Thorlabs.serial.**MDT69xA**(*conn*)

Bases: *pylablib.devices.Thorlabs.serial.ThorlabsSerialInterface*

Thorlabs MDT693A/4A high-voltage source.

Uses MDT693A program interface, so should be compatible with both A and B versions (though it doesn't support all functions of MDT693B/4B)

Parameters `conn` – serial connection parameters (usually port or a tuple containing port and baudrate)

get_voltage(*channel='x'*)
Get the output voltage in Volts at a given channel

set_voltage(*voltage*, *channel='x'*)
Set the output voltage in Volts at a given channel

get_voltage_range()
Get the selected voltage range in Volts (75, 100 or 150)

BackendError
alias of *pylablib.core.devio.comm_backend.DeviceBackendError*

Error
alias of *pylablib.devices.Thorlabs.base.ThorlabsError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

ReraiseError
alias of *pylablib.devices.Thorlabs.base.ThorlabsBackendError*

apply_settings(*settings*)
Apply the settings.

settings is a dict `{name: value}` of the available device settings. Non-applicable settings are ignored.

ask(*msg*, *data_type='string'*, *delay=0.0*, *timeout=None*, *read_echo=False*)
Write a message and read a reply.

msg is the query message, *delay* is the delay between write and read. Other parameters are the same as in *read()*. If `read_echo==True`, assume that the device first echoes the input and skip it.

close()
Close the backend

flush(*one_line=False*)
Flush the read buffer (read all the available data and return the number of bytes read).

If `one_line==True`, read only a single line.

static get_arg_type(*arg*)
Autodetect argument type

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_esr(*timeout=None*)
Get the device status register (by default, `"*ESR?"` command)

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_id(*timeout=None*)
Get the device IDN. (query SCPI `'*IDN?'` command)

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

open()
Open the backend

static parse_array_data(*data*, *fmt*, *include_header=False*)
Parse the data returned by the device. *fmt* is `DataFormat` description in numpy format (e.g., `"<u2"`).

If `include_header==True`, the data is assumed to be in a (somewhat) standard SCPI format: `b'#'`, then a single digit `s` denoting length of the size block, then `s` digits denoting length of the data (in bytes) followed by the actual data. Otherwise (`include_header==False`), assume that the header is already removed.

read(*data_type='string'*, *timeout=None*)
Read data from the device.

data_type determines the type of the data. Can be `'raw'` (just raw data), `'string'` (with trailing and leading spaces stripped), `'int'`, `'float'`, `'bool'` (interprets `0` or `'off'` as `False`, anything else as `True`), `'value'` (returns tuple `(value, unit)`, where *value* is float), a callable (return the result of this callable applied to the string value), a dictionary (return the stored value corresponding to the string value, or to the value converted into integer if the string value is not present), or a list of data types (the result is treated as a list of values with the given types separated by commas). *timeout* overrides the default value.

read_binary_array_data(*include_header=False*, *timeout=None*, *flush_term=True*)
Read a binary data in the from the device.

The data assumes the standard binary transfer header consisting of `"#"` symbol, then a single digit with the size of the length string, then the length string containing the length of the binary data (in bytes). If `include_header==True`, return the data with the header; otherwise, return only the content. If `flush_term==True`, flush the following line to skip terminator characters after the binary data, which are added by some devices. *timeout* overrides the default value.

reconnect(*new_instrument=True*, *ignore_error=True*)
Remake the connection.

If `new_instrument==True`, create a new backend instance. If `ignore_error==True`, ignore errors on closing.

reset()
Reset the device (by default, "*RST" command)

set_device_variable(*key*, *value*)
Set the value of a settings parameter

sleep(*delay*)
Wait for *delay* seconds

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

using_write_buffer()
Context manager for using a write buffer.

While it's active, all the consecutive *write()* operations are bundled together with ; delimiter. The actual write is performed at the *read()*/*ask()* operation or at the end of the block.

wait(*wait_type='sync'*, *timeout=None*, *wait_callback=None*)
Pause execution until device overlapped commands are complete.

wait_type is either 'sync' (perform *wait_sync()*), 'dev' (perform *wait_dev()*) or 'none' (do nothing).

wait_dev()
Pause execution of the device commands until device overlapped commands (e.g., taking sweeps) are complete.

Note that the code execution is not paused.

wait_sync(*timeout=None*, *wait_callback=None*)
Pause execution of the script until device overlapped commands (e.g., taking sweeps) are complete.

timeout and *wait_callback* override default constructor parameters.

write(*msg*, *arg=None*, *arg_type=None*, *unit=None*, *bool_selector=None*, *wait_sync=None*, *read_echo=False*, *read_echo_delay=0.0*)
Send a command.

Parameters

- **msg** (*str*) – Text message.
- **arg** – Optional argument to append in the end. If a list of arguments is supplied, the result is joined with ",".
- **arg_type** (*str*) – Argument type. Can be 'raw' (in which case data is sent raw), 'string', 'int', 'float', 'bool', a format string (such as '{:.3f}') or a list of argument types (for an iterable argument); if format string is used and the argument is a list or a tuple, then it is expanded as a list of arguments (e.g., arg_type='{0};{1}' with arg=[1,2] will produce a string '1;2'); if a list of types is used, each element of *arg* is converted using the corresponding type, and the result is joined with ",".
- **unit** (*str*) – If not None, use it as a unit to append after the value.
- **bool_selector** (*tuple*) – A tuple (false_value, true_value) of two strings to represent bool argument; by default, use ._bool_selector attribute.
- **wait_sync** – if True, append the sync command (specified as ._wait_sync_comm attribute, "*OPC?" by default) after the message and pause the execution command is complete; useful in long set operations, where the device might ignore later inputs until the current command is complete; if None, use the class default ._default_write_sync attribute (False by default).

- **read_echo** (*bool*) – If True, read a single line after write.
- **read_echo_delay** (*float*) – The delay between write and read if read_echo==True.

Module contents

pylablib.devices.Toptica package

Submodules

pylablib.devices.Toptica.base module

exception pylablib.devices.Toptica.base.**TopticaError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Toptica device error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Toptica.base.**TopticaBackendError**(*exc*)
Bases: *pylablib.devices.Toptica.base.TopticaError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Toptica backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

pylablib.devices.Toptica.ibeam module

pylablib.devices.Toptica.ibeam.**muxchan**(**args*, ***kwargs*)
Multiplex the function over its addr argument

class pylablib.devices.Toptica.ibeam.**TDeviceInfo**(*serial*, *version*)
Bases: tuple

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

serial

version

class pylablib.devices.Toptica.ibeam.**TWorkHours**(*power_up*, *laser_on*)
Bases: tuple

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

laser_on

power_up

class pylablib.devices.Toptica.ibeam.**TTemperatures**(*diode*, *baseplate*)
Bases: tuple

baseplate

count()
Return number of occurrences of value.

diode

index()
Return first index of value.

Raises ValueError if the value is not present.

class pylablib.devices.Toptica.ibeam.**TopticaIBeam**(*conn='COM1'*)
Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*

Toptica iBeam smart laser controller.

Parameters

- **conn** – connection parameters - index of the Attocube ANC350 in the system (for a single controller leave 0)
- **timeout** (*float*) – default operation timeout

Error
alias of *pylablib.devices.Toptica.base.TopticaError*

open()
Open the backend

query(*comm*, *multiline=False*, *keep_whitespace=False*, *check_error='FEW'*)

reboot()
Reboot the laser system

get_device_info()
Get the device info of the laser system: (serial, version)

get_full_data(*formatted=False*)
Return the comprehensive device data

get_work_hours()
Get the work hours (power on time and laser on time)

get_channels_number()
Get number of supported laser channels

is_enabled()
Check if the output is enabled

enable(*enabled=True*)
Turn the output on or off

is_channel_enabled(*channel='all'*)
Check if the specific channel is enabled

enable_channel(*channel*, *enabled=True*)
Turn the specific channel on or off

get_channel_power(*channel='all'*)
Get specified channel power (in W)

set_channel_power(*channel*, *power*)
Set channel power (in W)

get_output_power()
Get current output power (in W)

get_drive_current()
Get current diode drive current (in A)

get_current_limits()
Get settings of all current limits (in A) as a dictionary

get_temperatures()
Get settings of all current limits (in A) as a dictionary

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

set_device_variable(*key, value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.Trinamic package

Submodules

pylablib.devices.Trinamic.base module

exception pylablib.devices.Trinamic.base.**TrinamicError**
Bases: *pylablib.core.devio.base.DeviceError*

Generic Trinamic error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pylablib.devices.Trinamic.base.**TrinamicBackendError**(*exc*)
Bases: *pylablib.devices.Trinamic.base.TrinamicError*, *pylablib.core.devio.comm_backend.DeviceBackendError*

Generic Trinamic backend communication error

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.Trinamic.base.**TMCM1110**(*conn*)
Bases: *pylablib.core.devio.comm_backend.ICommBackendWrapper*, *pylablib.devices.interface.stage.IStage*

Trinamic stepper motor controller TMCM-1110 controlled using TMCL Firmware.

Parameters conn – serial connection parameters (usually port or a tuple containing port and baudrate)

Error
alias of *TrinamicError*

open()
Open the backend

class ReplyData(*comm, status, value, addr, module*)
Bases: tuple

addr

comm

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

module

status

value

query(*comm*, *comm_type*, *value*, *result_format='i'*, *bank=0*, *addr=0*)
Send a query to the stage and return the reply.

For details, see TMCM-1110 firmware manual.

get_axis_parameter(*parameter*, *result_format='i'*, *addr=0*)
Get a given axis parameter

set_axis_parameter(*parameter*, *value*, *addr=0*)
Set a given axis parameter (volatile; resets on power cycling)

store_axis_parameter(*parameter*, *value=None*, *addr=0*)
Store a given axis parameter in EEPROM (by default, value is the current value)

get_global_parameter(*parameter*, *result_format='i'*, *bank=0*, *addr=0*)
Get a given global parameter

set_global_parameter(*parameter*, *value*, *bank=0*, *addr=0*)
Set a given global parameter

get_general_input(*port=0*, *bank=0*, *addr=0*)
Get value of an input at a given bank (0-2) and port.

Bank 0 is digital input (7 ports), bank 1 is analog input (1 port, value from 0 to 2**16-1), bank 2 is digital output (8 ports). For port assignments, see TMCM-1110 firmware manual.

set_general_output(*value*, *port=0*, *bank=2*, *addr=0*)
Set value of a digital input at a given bank (only bank 2 is available) and port.

For port assignments, see TMCM-1110 firmware manual.

move_to(*position*, *addr=0*)
Move to a given position

move_by(*steps=1*, *addr=0*)
Move by a given number of steps

get_position(*addr=0*)
Get the current axis position

set_position_reference(*pos=0*, *addr=0*)
Set the current axis position as a reference (the actual motor position stays the same)

jog(*direction*, *speed=None*, *addr=0*)
Jog in a given direction with a given speed.

direction can be either `"-"` (negative, left) or `"+"` (positive, right). The motion continues until it is explicitly stopped, or until a limit is hit. If `speed is None`, use the standard speed value.

stop(*addr=0*)
Stop motion

get_microstep_resolution(*addr=0*)
Get the number of microsteps per full step (always a power of 2)

set_microstep_resolution(*resolution*, *addr=0*)
Set the number of microsteps per full step (rounded to a nearest power of 2)

get_current_parameters(*addr=0*)
Return diving current parameter `(drive_current, standby_current)`.

`drive_current` is the maximal drive current, which is given as a fraction of the maximal generated current current (which is either 1A or 2.8A depending on the hardware jumper). `standby_current` is given as a fraction of `drive_current`.

setup_current(*drive_current=None*, *standby_current=None*, *addr=0*)
Set drive and standby currents.

WARNING: too high of a setting might damage the motor. `drive_current` is the maximal drive current, which is given as a fraction of the maximal generated current current (which is either 1A or 2.8A depending on the hardware jumper). `standby_current` is given as a fraction of `drive_current`. Any `None` parameters are left unchanged.

get_limit_switches_parameters(*addr=0*)
Return limit switch parameters `(left_enable, right_enable)`

setup_limit_switches(*left_enable=None*, *right_enable=None*, *addr=0*)
Setup limit switch parameters

get_velocity_parameters(*addr=0*)
Return velocity parameters `(speed, accel, pulse_divisor, ramp_divisor)`.

`speed` and `accel` denote, correspondingly, maximal (i.e., steady regime) moving speed and acceleration in *internal* units. `pulse_divisor` is the driver pulse divisor, which defines how internal velocity units translate into microsteps/s (see `get_velocity_factor()`); can only be a power of 2, higher values mean slower motion. `ramp_divisor` is the driver ramp divisor, which, together with the pulse divisor, defines how internal acceleration units translate into microsteps/s^2 (see `get_acceleration_factor()`); rounded to the nearest power of 2, higher values mean slower acceleration.

setup_velocity(*speed=None*, *accel=None*, *pulse_divisor=None*, *ramp_divisor=None*, *addr=0*)
Setup velocity parameters `(speed, accel, pulse_divisor, ramp_divisor)`.

`speed` and `accel` denote, correspondingly, maximal (i.e., steady regime) moving speed and acceleration in *internal* units. `pulse_divisor` is the driver pulse divisor, which defines how internal velocity units translate into microsteps/s (see `get_velocity_factor()`); rounded to the nearest power of 2, higher values mean slower motion. `ramp_divisor` is the driver ramp divisor, which, together with the pulse divisor, defines how internal acceleration units translate into microsteps/s^2 (see `get_acceleration_factor()`); rounded to the nearest power of 2, higher values mean slower acceleration. `None` values are left unchanged.

get_velocity_factor(*addr=0*)
Get the ratio between the real speed (in microsteps/s) and the internal units

get_acceleration_factor(*addr=0*)
Get the ratio between the real acceleration (in microsteps/s^2) and the internal units

get_current_speed(*addr=0*)
Get the instantaneous speed in internal units

is_moving(*addr=0*)
Check if the motor is moving

wait_move(*addr=0*)
Wait until motion is done

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the backend

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

lock(*timeout=None*)
Lock the access to the device from other threads/processes (isn't necessarily implemented)

locking(*timeout=None*)
Context manager for lock & unlock

set_device_variable(*key*, *value*)
Set the value of a settings parameter

unlock()
Unlock the access to the device from other threads/processes (isn't necessarily implemented)

Module contents

pylablib.devices.interface package

Submodules

pylablib.devices.interface.camera module

exception pylablib.devices.interface.camera.**DefaultFrameTransferError**

Bases: *pylablib.core.devio.base.DeviceError*

Generic frame transfer error

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class pylablib.devices.interface.camera.**TFramesStatus**(*acquired*, *unread*, *skipped*, *buffer_size*)

Bases: tuple

acquired

buffer_size

count()

Return number of occurrences of value.

index()

Return first index of value.

Raises ValueError if the value is not present.

skipped

unread

class pylablib.devices.interface.camera.**TFrameSize**(*width*, *height*)

Bases: tuple

count()

Return number of occurrences of value.

height

index()

Return first index of value.

Raises ValueError if the value is not present.

width

class pylablib.devices.interface.camera.**TFramePosition**(*left*, *top*)

Bases: tuple

count()

Return number of occurrences of value.

index()

Return first index of value.

Raises ValueError if the value is not present.

left

top

class pylablib.devices.interface.camera.**TFrameInfo**(*frame_index*)
Bases: `tuple`

count()
Return number of occurrences of value.

frame_index

index()
Return first index of value.

Raises ValueError if the value is not present.

class pylablib.devices.interface.camera.**ICamera**(**args*, ***kwargs*)
Bases: *pylablib.core.devio.interface.IDevice*

Generic camera class.

Provides a consistent common interface for the most frequently encountered camera functions.

Error
alias of *pylablib.core.devio.base.DeviceError*

TimeoutError
alias of *pylablib.core.devio.base.DeviceError*

FrameTransferError
alias of *DefaultFrameTransferError*

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

setup_acquisition(***kwargs*)
Setup acquisition.

Any non-specified acquisition parameters are assumed to be the same as previously set (or default, if not explicitly set before). Return the new acquisition parameters.

clear_acquisition()
Clear acquisition settings

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

acquisition_in_progress()
Check if acquisition is in progress

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"`` (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes)

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-"tuple" formats.

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by `set_frame_info_format()`) is "list" or "array".

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be "none" (replacing them with `None`), "zero" (replacing them with zero-filled frame), or "skip" (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the connection

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

open()
Open the connection

set_device_variable(*key*, *value*)
Set the value of a settings parameter

pylablib.devices.interface.camera.**acqstopped**(**args*, ***kwargs*)
Decorator which temporarily stops acquisition for the function call

pylablib.devices.interface.camera.**acqcleared**(**args*, ***kwargs*)
Decorator which temporarily clears acquisition for the function call

class pylablib.devices.interface.camera.**FrameCounter**
Bases: `object`

Frame counter.

Keeps track of the buffer occupation, acquired/missed frames, last read and wait buffers, etc.

reset(*buffer_size=None*)
Reset the counters.

If `buffer_size is None`, assume the the buffer is deallocated. Otherwise, it specifies the frame buffer size (in frames).

update_acquired_frames(*acquired_frames*)
Update the counter of acquired frames (needs to be called by the camera whenever necessary)

wait_start(*acquired_frames*)
Set up waiting routine (called in the beginning of `ICamera.wait_for_frame()`)

is_wait_done(*acquired_frames=None*, *since='lastread'*, *nframes=1*)
Check if the waiting condition is satisfied based on the counter values:

If not `None`, *acquired_frames* specifies the most recent number of acquired frames (the internal counters is automatically updated). *since* and *nframes* have the same meaning as in `ICamera.wait_for_frame()`.

wait_done()
Clean up waiting routine (called in the end of `ICamera.wait_for_frame()`)

get_frames_status(*acquired_frames=None*)
Get status of the internal counters.

Return tuple `(acquired, unread, skipped, buffer_size)`. If the buffer is not allocated, all counters are 0.

get_new_frames_range(*acquired_frames=None*)
Get the range of the new frames (acquired but not read)

trim_frames_range(*rng*)
Trim the given frames range to only contains frames which are still in the buffer (i.e., remove the frames which are too old and have been overwritten)

advance_read_frames(*rng*)
Mark the specified frames range as read and advance the last read counter

set_first_valid_frame(*first_valid_frame*)
Set the first valid frame; all frames older than it are considered invalid when calculating skipped frames and trimming ranges

class pylablib.devices.interface.camera.**FrameNotifier**(*strict=False*)
Bases: `object`

Notifier for a new available frame.

Used when the camera runs a separate polling thread or a callback, which needs to notify the main thread that a new frame has been acquired.

Parameters **strict** – determines whether *wait()* waits for a specified frame index, or just for any new frame (which is checked later)

reset()
Reset the internal frame counter

inc()
Increment the internal frame counter, notify the waiting threads, and return the counter value

wait(*idx=None*, *timeout=None*)
Wait for a new frame with a given index (if `None`, for the next acquired frame)

class pylablib.devices.interface.camera.**ChunkBufferManager**(*chunk_size=1048576*)
Bases: `object`

Buffer manager, which takes care of creating and removing the buffer chunks, and reading out some parts of them.

Parameters **chunk_size** – the minimal size of a single buffer chunk (continuous memory segment potentially containing several frames).

get_ctypes_frames_list(*ctype=<class 'ctypes.c_char_p'>*)
Get stored buffers as a ctypes array with pointer of the given type

get_frames_data(*idx*, *nframes=1*)
Get frames data starting from *idx* and spanning *nframes* frames.

Return a list of tuples `(nread, chunk_data)`, where `nread` is the number of frames in the chunk, and `chunk_data` is the raw buffer pointer as a `ctypes.c_char_p` object.

allocate(*nframes*, *frame_size*)
Allocate buffers for the given number of frames and frame size (in bytes)

deallocate()
Deallocate the buffers

class pylablib.devices.interface.camera.**IAttributeCamera**(**args*, ***kwargs*)
Bases: *pylablib.devices.interface.camera.ICamera*

Camera class which supports camera attributes.

The method `_list_attributes` must be defined in a subclass; it should produce a list of camera attributes, which have `name` attribute for placing them into a dictionary. Attributes can also have `readable` and `writable` attributes, which are used in *get_all_attribute_values()* and *set_all_attribute_values()* to determine if the attribute values should be collected or set. Method `_update_attributes` should be called on opening to populate the dictionary of available attributes.

One can also define `_normalize_attribute_name`, which normalizes the attribute name into a dictionary name (e.g., replaces separators, removes spaces, or normalizes case).

get_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_all_attributes(*copy=False*)
Return a dictionary of all available attributes.

If `copy==True`, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_attribute_value(*name*, *error_on_missing=True*, *default=None*, ***kwargs*)
Get value of an attribute with the given name.

If the value doesn't exist and `error_on_missing==True`, raise error; otherwise, return *default*. If *default* is not `None`, automatically assume that `error_on_missing==False`. If *name* points at a dictionary branch, return a dictionary with all values in this branch. Additional arguments are passed to `get_value` methods of the individual attribute.

set_attribute_value(*name*, *value*, *error_on_missing=True*, ***kwargs*)
Set value of an attribute with the given name.

If the value doesn't exist and `error_on_missing==True`, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). Additional arguments are passed to `set_value` methods of the individual attribute.

get_all_attribute_values(*root=''*, ***kwargs*)
Get values of all attributes with the given *root*.

Additional arguments are passed to `get_value` methods of individual attributes.

set_all_attribute_values(*settings*, *root=''*, ***kwargs*)
Set values of all attributes with the given *root*.

Additional arguments are passed to `set_value` methods of individual attributes.

Error
alias of *pylablib.core.devio.base.DeviceError*

FrameTransferError
alias of *DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

TimeoutError
alias of *pylablib.core.devio.base.DeviceError*

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

clear_acquisition()
Clear acquisition settings

close()
Close the connection

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

open()
Open the connection

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

`set_device_variable`(*key*, *value*)
Set the value of a settings parameter

`set_frame_format`(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

`set_frame_info_format`(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by `get_frame_info_fields()`; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

`set_frame_info_period`(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

`set_image_indexing`(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

`setup_acquisition`(***kwargs*)
Setup acquisition.

Any non-specified acquisition parameters are assumed to be the same as previously set (or default, if not explicitly set before). Return the new acquisition parameters.

`snap`(*timeout=5.0*, *return_info=False*)
Snap a single frame

`start_acquisition`(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

`stop_acquisition`()
Stop acquisition

`wait_for_frame`(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"`` (from the last read frame), "lastwait" (wait for the last successful *wait_for_frame()* call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

class pylablib.devices.interface.camera.**IGrabberAttributeCamera**(**args*, ***kwargs*)

Bases: *pylablib.devices.interface.camera.ICamera*

Camera class which supports frame grabber attributes.

Essentially the same as *IAttributeCamera*, but with relevant methods and attributes renamed to support both frame grabber and camera attributes handling simultaneously.

The method _list_grabber_attributes must be defined in a subclass; it should produce a list of camera attributes, which have name attribute for placing them into a dictionary. Attributes can also have readable and writable attributes, which are used in *get_all_grabber_attribute_values()* and *set_all_grabber_attribute_values()* to determine if the attribute values should be collected or set. Method _update_grabber_attributes should be called on opening to populate the dictionary of available attributes.

One can also define _normalize_grabber_attribute_name, which normalizes the attribute name into a dictionary name (e.g., replaces separators, removes spaces, or normalizes case).

get_grabber_attribute(*name*, *error_on_missing=True*)
Get the camera attribute with the given name

get_all_grabber_attributes(*copy=False*)
Return a dictionary of all available frame grabber grabber_attributes.

If copy==True, copy the dictionary; otherwise, return the internal dictionary structure (should not be modified).

get_grabber_attribute_value(*name*, *error_on_missing=True*, *default=None*, ***kwargs*)
Get value of a frame grabber attribute with the given name.

If the value doesn't exist and error_on_missing==True, raise error; otherwise, return *default*. If *default* is not None, automatically assume that error_on_missing==False. If *name* points at a dictionary branch, return a dictionary with all values in this branch. Additional arguments are passed to get_value methods of the individual attribute.

set_grabber_attribute_value(*name*, *value*, *error_on_missing=True*, ***kwargs*)
Set value of a frame grabber attribute with the given name.

If the value doesn't exist and error_on_missing==True, raise error; otherwise, do nothing. If *name* points at a dictionary branch, set all values in this branch (in this case *value* must be a dictionary). Additional arguments are passed to set_value methods of the individual attribute.

get_all_grabber_attribute_values(*root="*, ***kwargs*)
Get values of all frame grabber attributes with the given *root*.

Additional arguments are passed to get_value methods of individual attributes.

set_all_grabber_attribute_values(*settings*, *root="*, ***kwargs*)
Set values of all frame grabber attributes with the given *root*.

Additional arguments are passed to set_value methods of individual attributes.

Error
alias of *pylablib.core.devio.base.DeviceError*

FrameTransferError
alias of *DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

TimeoutError
alias of *pylablib.core.devio.base.DeviceError*

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

clear_acquisition()
Clear acquisition settings

close()
Close the connection

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

open()
Open the connection

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

setup_acquisition(***kwargs*)
Setup acquisition.

Any non-specified acquisition parameters are assumed to be the same as previously set (or default, if not explicitly set before). Return the new acquisition parameters.

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be “lastread”‘‘ (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

class `pylablib.devices.interface.camera.`**TAcqTimings**(*exposure*, *frame_period*)
Bases: `tuple`

count()
Return number of occurrences of value.

exposure

frame_period

index()
Return first index of value.

Raises ValueError if the value is not present.

class pylablib.devices.interface.camera.**IExposureCamera**(**args*, ***kwargs*)
Bases: *pylablib.devices.interface.camera.ICamera*

get_exposure()
Get current exposure

set_exposure(*exposure*)
Set camera exposure

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

get_frame_timings()
Get acquisition timing.

Return tuple `(exposure, frame_period)`.

Error
alias of *pylablib.core.devio.base.DeviceError*

FrameTransferError
alias of *DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

TimeoutError
alias of *pylablib.core.devio.base.DeviceError*

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

clear_acquisition()
Clear acquisition settings

close()
Close the connection

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is "list" or "array".

get_frame_info_format()
Get format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple (acquired, unread, skipped, size), where acquired is the total number of acquired frames, unread is the number of acquired but not read frames, skipped is the number of skipped (not read and then written over) frames, and buffer_size is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple (first, last) with images range (first inclusive). If no images are available, return None. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

open()
Open the connection

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see `read_multiple_images()`).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array). Note that if the format is set to "array", the frame info format is also automatically set to "array".

set_frame_info_format (*fmt*, *include_fields=None*)
Set format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes) If *include_fields* is not None, it specifies the fields included for non-"tuple" formats.

set_frame_info_period (*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing (*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

setup_acquisition (***kwargs*)
Setup acquisition.

Any non-specified acquisition parameters are assumed to be the same as previously set (or default, if not explicitly set before). Return the new acquisition parameters.

snap (*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition (**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of *setup_acquisition()*,if the parameter is None). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition ()
Stop acquisition

wait_for_frame (*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be “lastread”‘‘ (from the last read frame), "lastwait" (wait for the last successful *wait_for_frame()* call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

class pylablib.devices.interface.camera.**TAxisROILimit**(*min*, *max*, *pstep*, *sstep*, *maxbin*)

Bases: `tuple`

count()

Return number of occurrences of value.

index()

Return first index of value.

Raises ValueError if the value is not present.

max

maxbin

min

pstep

sstep

pylablib.devices.interface.camera.**truncate_roi_axis**(*roi*, *lim*, *symmetric=False*)

Truncate ROI to conform to the given ROI limits.

roi is a tuple `(start, stop, bin)`, and *lim* is a tuple `(min, max, pstep, sstep, maxbin)`. Assume that `pstep` and `sstep` divide `min` and `max`, and that either `pstep` divides `sstep` or the other way around. If `symmetric==True`, then `max` should be even.

class pylablib.devices.interface.camera.**IROICamera**(**args*, ***kwargs*)

Bases: *pylablib.devices.interface.camera.ICamera*

get_roi()

Get current ROI.

Return tuple `(hstart, hend, vstart, vend)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0).

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*)

Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0). By default, all non-supplied parameters take extreme values (0 for start, maximal for end).

get_roi_limits(*hbin=1*, *vbin=1*)

Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning (fixed to 1 if not binning is allowed). In some cameras, the step and the minimal size depend on the binning, which can be supplied.

Error

alias of *pylablib.core.devio.base.DeviceError*

FrameTransferError

alias of *DefaultFrameTransferError*

class NoParameterCaller(*device*, *kind*)

Bases: `object`

Class to simplify calling functions without a parameter

TimeoutError
alias of *pylablib.core.devio.base.DeviceError*

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

clear_acquisition()
Clear acquisition settings

close()
Close the connection

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

open()
Open the connection

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *`read_multiple_images()`*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *`read_multiple_images()`*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *`get_frame_info_fields()`*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

setup_acquisition(***kwargs*)
Setup acquisition.

Any non-specified acquisition parameters are assumed to be the same as previously set (or default, if not explicitly set before). Return the new acquisition parameters.

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"‘‘ (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

class pylablib.devices.interface.camera.**IBinROICamera**(**args*, ***kwargs*)
Bases: `pylablib.devices.interface.camera.ICamera`

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend, hbin, vbin)`. *hstart* and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0), *hbin* and *vbin* specify binning.

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*, *hbin=1*, *vbin=1*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start is inclusive, stop is exclusive, starting from 0), *hbin* and *vbin* specify binning. By default, all non-supplied parameters take extreme values (0 for start, maximal for end, 1 for binning).

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning. In some cameras, the step and the minimal size depend on the binning, which can be supplied.

Error
alias of *pylablib.core.devio.base.DeviceError*

FrameTransferError
alias of *DefaultFrameTransferError*

class **NoParameterCaller**(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

TimeoutError
alias of *pylablib.core.devio.base.DeviceError*

acquisition_in_progress()
Check if acquisition is in progress

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

clear_acquisition()
Clear acquisition settings

close()
Close the connection

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_frame_format()
Get format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is `"list"` or `"array"`.

get_frame_info_format()
Get format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple `(first, last)` with images range (first inclusive). If no images are available, return `None`. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if `None`, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them, while still keeping total returned frames number to *n*). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return `True`.

is_opened()
Check if the device is connected

open()
Open the connection

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of `with` block.

Useful for applying certain settings which can't be changed during the acquisition. If `clear==True`, clear acquisition in addition to pausing (by default, use the class default specified as `_clear_pausing_acquisition` attribute). Yields tuple `(acq_in_progress, acq_params)`, which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple `(first, last)` with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return `None`. If `peek==True`, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be `"none"` (replacing them with `None`), `"zero"` (replacing them with zero-filled frame), or `"skip"` (skipping them). If `return_info==True`, return tuple `(frames, infos)`, where `infos` is a list of frame info tuples (camera-dependent, by default, only the frame index); if some frames are missing and `missing_frame!="skip"`, the corresponding frame info is `None`.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return `None`. If `peek==True`, return the image but not mark it as read. If `return_info==True`, return tuple `(frame, info)`, where `info` is an info tuples (camera-dependent, see *read_multiple_images()*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be `"list"` (list of 2D arrays, or 3D array for some `fastbuff` cameras), `"array"` (a single 3D array). Note that if the format is set to `"array"`, the frame info format is also automatically set to `"array"`.

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be `"namedtuple"` (potentially nested named tuples; convenient to get particular values), `"list"` (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), `"array"` (same as `"list"`, but with a numpy array, which is easier to use for `fastbuff` readout supported by some cameras), or `"dict"` (flat dictionary with the same fields as the `"list"` format; more resilient to future format changes) If *include_fields* is not `None`, it specifies the fields included for non-`"tuple"` formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to `None`) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be `"rct"` (first index row, second index column, rows counted from the top), `"rcb"` (same as `"rc"`, rows counted from the bottom), `"xyt"` (first index column, second index row, rows counted from the top), or `"xyb"` (same as `"xyt"`, rows counted from the bottom)

setup_acquisition(***kwargs*)
Setup acquisition.

Any non-specified acquisition parameters are assumed to be the same as previously set (or default, if not explicitly set before). Return the new acquisition parameters.

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:‘‘setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be “lastread”‘‘ (from the last read frame), `"lastwait"` (wait for the last successful `wait_for_frame()` call), `"now"` (from the start of the current call), or `"start"` (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, `None` (infinite timeout), or a tuple `(timeout, frame_timeout)`, in which case the call times out if the total time exceeds `timeout`, or a single frame wait exceeds `frame_timeout`. If the call times out, raise `TimeoutError`. If `error_on_stopped==True` and the acquisition is not running, raise `Error`; otherwise, simply return `False` without waiting.

pylablib.devices.interface.camera.**remove_status_line**(*frame*, *status_line*, *policy='duplicate'*, *copy=True*, *value=0*)

Remove status line, if present.

Parameters

- **frame** – a frame to process (2D or 3D numpy array; if 3D, the first axis is the frame number)
- **status_line** – status line descriptor (from the frames message)
- **policy** – determines way to deal with the status line; can be `"keep"` (keep as is), `"cut"` (cut off the status-line-containing row/column), `"zero"` (set it to zero), `"value"` (set it to a given value), `"median"` (set it to the image median), or

"duplicate" (set it equal to the previous row; default) "cut" is only possible of the status line is on the edge of the image.

- **copy** – if True, make copy of the original frames; otherwise, attempt to remove the line in-place

pylablib.devices.interface.camera.**extract_status_line**(*frame*, *status_line*, *copy=True*)
Extract status line, if present.

Parameters

- **frame** – a frame to process (2D or 3D numpy array; if 3D, the first axis is the frame number)
- **status_line** – status line descriptor (from the frames message)
- **copy** – if True, make copy of the original status line data.

pylablib.devices.interface.camera.**insert_status_line**(*frame*, *status_line*, *value*, *copy=True*)
Insert status line, if present.

Parameters

- **frame** – a frame to process (2D or 3D numpy array; if 3D, the first axis is the frame number)
- **status_line** – status line descriptor (from the frames message)
- **value** – status line value
- **copy** – if True, make copy of the original status line data.

pylablib.devices.interface.camera.**get_status_line_roi**(*frame*, *status_line*)
Return ROI taken by the status line in the given frame

pylablib.devices.interface.stage module

class pylablib.devices.interface.stage.**IStage**
Bases: *pylablib.core.devio.interface.IDevice*

Generic stage class

class NoParameterCaller(*device*, *kind*)
Bases: object

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict {name: value} of the device available settings. Non-applicable settings are ignored.

close()
Close the connection

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

open()
Open the connection

set_device_variable(*key*, *value*)
Set the value of a settings parameter

pylablib.devices.interface.stage.**muxaxis**(**args*, ***kwargs*)
Multiplex the function over its axis argument

class pylablib.devices.interface.stage.**IMultiaxisStage**
Bases: *pylablib.devices.interface.stage.IStage*

Generic multiaxis stage class.

Has methods to assign and map axes and the axis device parameter.

get_all_axes()
Get the list of all available axes (taking mapping into account)

remap_axes(*mapping*, *accept_original=True*)
Rename axes to the new labels.

mapping is the new axes mapping, which can be a list of new axes name (corresponding to the old axes in order returned by *get_all_axes()*), or a dictionary `{alias: original}` of the new axes aliases.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

close()
Close the connection

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_full_info(*include=0*)
Get dict `{name: value}` containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_full_status(*include=0*)
Get dict `{name: value}` containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

get_settings(*include=0*)
Get dict `{name: value}` containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting `include=-10` queries all available variables.

is_opened()
Check if the device is connected

open()
Open the connection

set_device_variable(*key*, *value*)
Set the value of a settings parameter

Module contents

pylablib.devices.uc480 package

Submodules

pylablib.devices.uc480.uc480 module

class pylablib.devices.uc480.uc480.**TCameraInfo**(*cam_id*, *dev_id*, *sens_id*, *model*, *serial_number*, *in_use*, *status*)

Bases: `tuple`

cam_id

count()
Return number of occurrences of value.

dev_id

in_use

index()
Return first index of value.

Raises ValueError if the value is not present.

model

sens_id

serial_number

status

pylablib.devices.uc480.uc480.**list_cameras**()
List camera connections (interface kind and camera index)

pylablib.devices.uc480.uc480.**get_cameras_number**()
Get the total number of connected uc480 cameras

pylablib.devices.uc480.uc480.**find_by_serial**(*serial_number*)
Find device ID using its serial number

class pylablib.devices.uc480.uc480.**TDeviceInfo**(*cam_id*, *model*, *manufacturer*, *serial_number*, *usb_version*, *date*, *dll_version*, *camera_type*)

Bases: tuple

cam_id

camera_type

count()
Return number of occurrences of value.

date

dll_version

index()
Return first index of value.

Raises ValueError if the value is not present.

manufacturer

model

serial_number

usb_version

class pylablib.devices.uc480.uc480.**TAcquiredFramesStatus**(*acquired*, *transfer_missed*, *frameskip_events*)

Bases: tuple

acquired

count()
Return number of occurrences of value.

frameskip_events

index()
Return first index of value.

Raises ValueError if the value is not present.

transfer_missed

class pylablib.devices.uc480.uc480.**TTimestamp**(*year*, *month*, *day*, *hour*, *minute*, *second*, *millisecond*)

Bases: tuple

count()
Return number of occurrences of value.

day

hour

index()
Return first index of value.

Raises ValueError if the value is not present.

millisecond

minute

month

second

year

class pylablib.devices.uc480.uc480.**TFrameInfo**(*frame_index*, *framestamp*, *timestamp*, *timestamp_dev*, *size*, *io_status*, *flags*)

Bases: tuple

count()
Return number of occurrences of value.

flags

frame_index

framestamp

index()
Return first index of value.

Raises ValueError if the value is not present.

io_status

size

timestamp

timestamp_dev

class pylablib.devices.uc480.uc480.**UC480Camera**(*cam_id=0*, *roi_binning_mode='auto'*, *dev_id=None*)

Bases: *pylablib.devices.interface.camera.IBinROICamera*, *pylablib.devices.interface.camera.IExposureCamera*

Thorlabs uc480 camera.

Parameters

- **cam_id** (*int*) – camera ID; use 0 to get the first available camera
- **roi_binning_mode** – determines whether binning in ROI refers to binning or subsampling; can be "bin", "subsample", or "auto" (since most cameras only support one, it will pick the one which has non-trivial value, or "bin" if both are available).
- **dev_id** (*int*) – if None use *cam_id* as a camera id (cam_id field of the camera info returned by *list_cameras()*); otherwise, ignore value of *cam_id* and use *dev_id* as device id (dev_id field of the camera info). The first method requires assigning camera IDs beforehand (otherwise IDs might overlap, in which case only one camera can be accessed), but the assigned IDs are permanent; the second method always has unique IDs, but they might change if the cameras are disconnected and reconnected. For a more reliable assignment, one can use *find_by_serial()* function to find device ID based on the camera serial number.

Error = <Mock name='mock.uc480Error' id='140318655904208'>

TimeoutError = <Mock spec='str' id='140318650256464'>

FrameTransferError = <Mock spec='str' id='140318650254544'>

open()
Open connection to the camera

close()
Close connection to the camera

is_opened()
Check if the device is connected

get_device_info()
Get camera model data.

Return tuple `(model, manufacturer, serial_number, usb_version, date, dll_version, camera_type)`.

get_camera_id()
Get the current camera id

set_camera_id(*cam_id*)
Set the new camera id (stored in non-volatile memory, i.e., survives power cycling)

get_frame_timings()
Get acquisition timing.

Return tuple `(exposure, frame_period)`.

set_exposure(*exposure*)
Set camera exposure

set_frame_period(*frame_time*)
Set frame period (time between two consecutive frames in the internal trigger mode)

get_pixel_rate()
Get camera pixel rate (in Hz)

get_available_pixel_rates()
Get all available pixel rates (in Hz)

get_pixel_rates_range()
Get range of allowed pixel rates (in Hz).

Return tuple `(min, max, step)` if minimal and maximal value, and a step.

set_pixel_rate(*rate=None*)
Set camera pixel rate (in Hz)

The rate is always rounded to the closest available. If *rate* is `None`, set the maximal possible rate.

get_all_color_modes()
Get a list of all available color modes

get_color_mode()
Get current color mode.

For possible modes, see *get_all_color_modes()*.

set_color_mode(*mode*)
Set current color mode.

For possible modes, see *get_all_color_modes()*.

get_gains()
Get current gains.

Return tuple `(master, red, green, blue)` of corresponding gain factors.

get_max_gains()
Get maximal gains.

Return tuple `(master, red, green, blue)` of corresponding maximal gain factors.

set_gains(*master=None*, *red=None*, *green=None*, *blue=None*)
Set current gains.

If supplied value is `None`, keep it unchanged.

get_gain_boost()
Check if gain boost is enabled

set_gain_boost(*enabled*)
Enable or disable gain boost

setup_acquisition(*nframes=100*)
Setup acquisition.

nframes determines number of size of the ring buffer (by default, 100).

clear_acquisition()
Clear acquisition settings

start_acquisition(**args*, ***kwargs*)
Start acquisition.

Can take the same keyword parameters as *:meth:``setup_acquisition*. If the acquisition is not set up yet, set it up using the supplied parameters (use default of `setup_acquisition()`,if the parameter is `None`). Otherwise, if any supplied parameters are different from the current ones, change them and reset the acquisition.

stop_acquisition()
Stop acquisition

acquisition_in_progress()
Check if acquisition is in progress

get_frames_status()
Get acquisition and buffer status.

Return tuple `(acquired, unread, skipped, size)`, where `acquired` is the total number of acquired frames, `unread` is the number of acquired but not read frames, `skipped` is the number of skipped (not read and then written over) frames, and `buffer_size` is the total buffer size (in frames).

get_acquired_frame_status()

set_frameskip_behavior(*behavior*)
Choose the camera behavior if frame skip event is encountered when waiting for a new frame, reading frames, getting buffer status, etc.

Can be `"error"` (raise `uc480FrameTransferError`), `"ignore"` (continue acquisition, ignore the gap), or `"skip"` (mark some number of frames as skipped, but keep the frame counters consistent).

get_supported_subsampling_modes()
Get all supported subsampling modes.

Return tuple `(horizontal, vertical)` of lists with all possible supported subsampling factors.

get_subsampling()
Get current subsampling

set_subsampling(*hsub=1*, *vsub=1*)
Set subsampling.

If values are not supported, get the closest value below the requested. Automatically turns off binning.

get_supported_binning_modes()
Get all supported binning modes.

Return tuple `(horizontal, vertical)` of lists with all possible supported binning factors.

get_binning()
Get current binning

set_binning(*hbin=1*, *vbin=1*)
Set binning.

If values are not supported, get the closest value below the requested. Automatically turns off subsampling.

get_detector_size()
Get camera detector size (in pixels) as a tuple `(width, height)`

get_roi()
Get current ROI.

Return tuple `(hstart, hend, vstart, vend, hbin, vbin)`.

set_roi(*hstart=0*, *hend=None*, *vstart=0*, *vend=None*, *hbin=1*, *vbin=1*)
Setup camera ROI.

hstart and *hend* specify horizontal image extent, *vstart* and *vend* specify vertical image extent (start are inclusive, stop are exclusive, starting from 0), *hbin* and *vbin* specify binning. By default, all non-supplied parameters take extreme values.

get_roi_limits(*hbin=1*, *vbin=1*)
Get the minimal and maximal ROI parameters.

Return tuple `(hlim, vlim)`, where each element is in turn a limit 5-tuple `(min, max, pstep, sstep, maxbin)` with, correspondingly, minimal and maximal size, position and size step, and the maximal binning. In some cameras, the step and the minimal size depend on the binning, which can be supplied.

class NoParameterCaller(*device*, *kind*)
Bases: `object`

Class to simplify calling functions without a parameter

apply_settings(*settings*)
Apply the settings.

settings is the dict `{name: value}` of the device available settings. Non-applicable settings are ignored.

get_acquisition_parameters()
Get acquisition parameters.

Return dictionary `{name: value}`

get_data_dimensions()
Get readout data dimensions (in pixels) as a tuple `(width, height)`; take indexing mode into account

get_device_variable(*key*)
Get the value of a settings, status, or full info parameter

get_exposure()
Get current exposure

get_frame_format()
Get format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array).

get_frame_info_fields()
Get the names of frame info fields.

Applicable when frame info format (set by *set_frame_info_format()*) is "list" or "array".

get_frame_info_format()
Get format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes)

get_frame_info_period()
Get period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

get_frame_period()
Get frame period (time between two consecutive frames in the internal trigger mode)

get_full_info(*include=0*)
Get dict {name: value} containing full device information (including status and settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_full_status(*include=0*)
Get dict {name: value} containing the device status (including settings).

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

get_image_indexing()
Get indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

get_new_images_range()
Get the range of the new images.

Return tuple (first, last) with images range (first inclusive). If no images are available, return None. If some images were in the buffer were overwritten, exclude them from the range.

get_settings(*include=0*)
Get dict {name: value} containing all the device settings.

include specifies either a list of variables (only these variables are returned), or a priority threshold (only values with the priority equal or higher are returned). Since the lowest priority is -10, setting include=-10 queries all available variables.

grab(*nframes=1*, *frame_timeout=5.0*, *missing_frame='none'*, *return_info=False*, *buff_size=None*)
Snap *nframes* images (with preset image read mode parameters)

buff_size determines buffer size (if None, use the default size). Timeout is specified for a single-frame acquisition, not for the whole acquisition time. *missing_frame* determines what to do with frames which have been lost: can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them, while still keeping total returned frames number to *n*). If return_info==True, return tuple (frames, infos), where infos is a list of frame info tuples (camera-dependent); if some frames are missing and missing_frame!="skip", the corresponding frame info is None.

is_acquisition_setup()
Check if acquisition is set up.

If the camera does not support separate acquisition setup, always return True.

pausing_acquisition(*clear=None*)
Context manager which temporarily pauses acquisition during execution of with block.

Useful for applying certain settings which can't be changed during the acquisition. If clear==True, clear acquisition in addition to pausing (by default, use the class default specified as _clear_pausing_acquisition attribute). Yields tuple (acq_in_progress, acq_params), which indicates whether acquisition is currently in progress, and what are the current acquisition parameters.

read_newest_image(*peek=False*, *return_info=False*)
Read the newest un-read image.

If no un-read frames are available, return None. If peek==True, return the image but not mark it as read. If return_info==True, return tuple (frame, info), where info is an info tuples (camera-dependent, see *read_multiple_images()*).

read_oldest_image(*peek=False*, *return_info=False*)
Read the oldest un-read image.

If no un-read frames are available, return None. If peek==True, return the image but not mark it as read. If return_info==True, return tuple (frame, info), where info is an info tuples (camera-dependent, see *read_multiple_images()*).

set_device_variable(*key*, *value*)
Set the value of a settings parameter

set_frame_format(*fmt*)
Set format for the returned images.

Can be "list" (list of 2D arrays, or 3D array for some fastbuff cameras), "array" (a single 3D array). Note that if the format is set to "array", the frame info format is also automatically set to "array".

set_frame_info_format(*fmt*, *include_fields=None*)
Set format of the frame info.

Can be "namedtuple" (potentially nested named tuples; convenient to get particular values), "list" (flat list of values, with field names are given by *get_frame_info_fields()*; convenient for building a table), "array" (same as "list", but with a numpy array, which is easier to use for fastbuff

readout supported by some cameras), or "dict" (flat dictionary with the same fields as the "list" format; more resilient to future format changes) If *include_fields* is not None, it specifies the fields included for non-"tuple" formats.

set_frame_info_period(*period=1*)
Set period of frame info acquisition.

Frame info might be skipped (set to None) except for frames which indices are divisible by *period*. Useful for certain cameras where acquiring frame info takes a lot of time and can reduce performance at higher frame rates. Note that this parameter can still be ignored (i.e., always set to 1) if the performance is not an issue for a given camera class.

set_image_indexing(*indexing*)
Set up indexing for the returned images.

Can be "rct" (first index row, second index column, rows counted from the top), "rcb" (same as "rc", rows counted from the bottom), "xyt" (first index column, second index row, rows counted from the top), or "xyb" (same as "xyt", rows counted from the bottom)

snap(*timeout=5.0*, *return_info=False*)
Snap a single frame

wait_for_frame(*since='lastread'*, *nframes=1*, *timeout=20.0*, *error_on_stopped=False*)
Wait for one or several new camera frames.

since specifies the reference point for waiting to acquire *nframes* frames; can be "lastread"‘‘ (from the last read frame), "lastwait" (wait for the last successful wait_for_frame() call), "now" (from the start of the current call), or "start" (from the acquisition start, i.e., wait until *nframes* frames have been acquired). *timeout* can be either a number, None (infinite timeout), or a tuple (timeout, frame_timeout), in which case the call times out if the total time exceeds timeout, or a single frame wait exceeds frame_timeout. If the call times out, raise TimeoutError. If error_on_stopped==True and the acquisition is not running, raise Error; otherwise, simply return False without waiting.

read_multiple_images(*rng=None*, *peek=False*, *missing_frame='skip'*, *return_info=False*)
Read multiple images specified by *rng* (by default, all un-read images).

If *rng* is specified, it is a tuple (first, last) with images range (first inclusive). If no new frames are available, return an empty list; if no acquisition is running, return None. If peek==True, return images but not mark them as read. *missing_frame* determines what to do with frames which are out of range (missing or lost): can be "none" (replacing them with None), "zero" (replacing them with zero-filled frame), or "skip" (skipping them). If return_info==True, return tuple (frames, infos), where infos is a list of TFrameInfo instances describing frame index, framestamp, global timestamp (real time), device timestamp (time from camera restart, in 0.1us steps), frame size, digital input state, and additional flags; if some frames are missing and missing_frame!="skip", the corresponding frame info is None. Note that obtaining frame info might take about 2ms, so at high frame rates it will become a limiting factor.

Module contents

pylablib.devices.utils package

Submodules

pylablib.devices.utils.load_lib module

pylablib.devices.utils.load_lib.**get_os_lib_folder**()

Get default Windows DLL folder (`System32` or `SysWOW64`, depending on Python and Windows bitness)

pylablib.devices.utils.load_lib.**get_program_files_folder**(*subfolder=”*, *arch=None*)

Get default Windows Program Files folder or a subfolder within it.

If *arch* is `None`, use the current Python architecture to determine the folder; otherwise, it specifies the architecture (`"32bit"` for `Program Files (x86)`, `"64bit"` for `Program Files`)

pylablib.devices.utils.load_lib.**load_lib**(*name*, *locations=('global',)*, *call_conv='cdecl'*, *locally=False*, *depends=None*, *depends_required=True*, *error_message=None*, *check_order='location'*, *return_location=False*)

Load DLL.

Parameters

- **name** – name or path of the library (can also be a list or a tuple with several names, which are tried in that order).
- **locations** – list or tuple of locations to search for a library; the function tries locations in order and returns the first successfully loaded library a location is a string which can be a path to the containing folder, `"parameter/*"` (the remaining part is a subpath inside `"devices/dlls"` library parameters; if this parameter is defined, it names folder or file for the dll), or `"global"` (load path as is; also searches in the standard OS specified locations determined by `PATH` variable, e.g., `System32` folder).
- **depends** – if specified, it is a list of dependency libraries which need to be loaded first before the main DLL; they are assumed to be in the same location as the main file
- **depends_required** – if `False`, ignore errors during dependency loads
- **locally** (*bool*) – if `True`, prepend path to the DLL containing folder to the environment `PATH` folders; this is usually required, if the loaded DLL imports other DLLs in the same folder
- **call_conv** (*str*) – DLL call convention; can be either `"cdecl"` (corresponds to `ctypes.cdll`) or `"stdcall"` (corresponds to `ctypes.windll`)
- **error_message** (*str*) – error message to add in addition to the default error message shown when the DLL is not found
- **check_order** (*str*) – determines the order in which possible combinations of names and locations are looped over; can be `"location"` (loop over locations, and for each location loop over names), `"name"` (loop over names, and for each name loop over locations), or a list of tuples `[(loc,name)]` specifying order of checking (in the latter case, *name* and *location* arguments are ignored, except for generating error message).
- **return_location** (*bool*) – if `True`, return a tuple `(dll, location, folder)` instead of a single dll.

class pylablib.devices.utils.load_lib.**TLibraryOpenResult**(*init_result*, *open_result*, *opid*)

Bases: `tuple`

count()

Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

init_result

open_result

opid

class pylablib.devices.utils.load_lib.**TLibraryCloseResult**(*close_result*, *uninit_result*)
Bases: tuple

close_result

count()
Return number of occurrences of value.

index()
Return first index of value.

Raises ValueError if the value is not present.

uninit_result

class pylablib.devices.utils.load_lib.**LibraryController**(*lib*)
Bases: object

Simple wrapper to control libraries which require initialization when a new device is opened or shutdown when all devices are closed.

Parameters lib – controlled library

preinit()
Pre-initialize the library, if it hasn't been done already

open()
Mark device opening.

Return tuple (init_result, open_result, opid) with the results of the initialization and the opening, and the opening ID which should afterwards be used for closing. If library is already initialized, set init_result=None

close(*opid*)
Mark device closing.

Return tuple (close_result, uninit_result) with the results of the closing and the shutdown. If library does not need to be shut down yet, set uninit_result=None

temp_open()
Context for temporarily opening a new device connection

shutdown()
Close all opened connections and shutdown the library

Module contents

Module contents

Submodules

pylablib.widgets module

Module contents

`pylablib.reload_all(from_load_path=True, keep_parameters=True)`
Reload all loaded modules.

If `keep_parameters==True`, keep the current library parameters (`pylablib.par`); otherwise, reset them to default.

`pylablib.unload_all()`
Reload all loaded modules.

`pylablib.load_par(path)`
Load library parameters from a file

`pylablib.setbp()`

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z