Experimentor Documentation

Release 0.3.0rc1

Aquiles Carattino

Jul 22, 2021

Contents:

A flexible package for experiment control and automation

Experimentor is a Python package aimed at simplifying the task of controlling experiments in various fields. The starting point of the development was a nano photonics setup and therefore the examples and the bulk of the code makes references to optical microscopes, but by no means this project is limited to them.

The documentation will cover from the basics of installation in a Python virtual environment to more complex tasks such as adding new features to the package.

CHAPTER 1

Installing

The code of this program is the repository that can be found at https://github.com/aquilesC/experimentor.

If you need further assistance with the installation of the code, please check *Installation instructions*

1.1 Setting up a Python working environment

This guide is thought for users on Windows willing to either use python 2.7 or 3+

1. Download the version of python you want from https://www.python.org/downloads/windows/ and install it
2. It may be that after installing, python is not added to the path. Don't worry, things are going to be sorted out later.
3. Get pip from:

   ```
   bootstrap.pypa.io/get-pip.py
   ```

4. Run:

   ```
   path/to/python/python.exe get-pip.py
   ```

5. Go to `path/to/python/Scripts`
6. Run:

   ```
   pip.exe install virtualenv
   pip.exe install virtualenvwrapper-powershell
   ```

   At this point you have a working installation of virtual environment that will allow you to isolate your development from your computer, ensuring no mistakes on versions will happen. Let's create a new working environment called Testing

7. Run:

```
virtualenv.exe Testing --python=path\to\python\python.exe
```

The last piece is important, because it will allow you to select the exact version of python you want to run, it can be either `python2` or `python 3` and also it can be Python 64 or 32 bit. You will also create a folder called Testing, in which all the packages you are going to install are going to be kept.

8. Go to the folder `Testing\Scripts`. Try to run `activate` If an error happens (most likely) follow the instructions below. Windows has a weird way of handling execution policies and we are going to change that. Open PowerShell with administrator rights (normally, just right click on it and select run as administrator) Run the following command:

```
Set-ExecutionPolicy RemoteSigned
```

This will allow to run local scripts. Go back to the PowerShell without administrative rights and run again the script activate

9. Now you are working on a safe development environment. If you just type python you will see that you are running the exact version you wanted. The same goes for packages, you can download specific versions, completely isolated from what is happening in the computer or in other virtual environments. Imagine there is more than one user and one decides to use numpy 64-bit but you need numpy 32-bit, you both can work isolated from each other. Moreover, if you run:

```
pip freeze > requirements.txt
```

You are going to generate a file (requirements.txt) with all the installed packages at that given time

10. For developing GUI's, most likely we are going to use PyQt. Since there is no official repository to install it through pip, we need to download the appropriate wheel from:

```
http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyqt4
```

Afterwards, just run (replacing the last part of the command by the wheel you have just downloaded):

```
pip install PyQt4-4.11.4-cp36-cp36m-win32.whl
```

11. For saving data, specially when dealing with big datasets, there is almost nothing better than using HDF5 (https://support.hdfgroup.org/HDF5/). For installing, follow the same procedures than with PyQt, you can find the wheel here: `http://www.lfd.uci.edu/~gohlke/pythonlibs/#h5py`

Note: *h5py* requires to have some Visual Basic distributables. Go to `http://landinghub.visualstudio.com/visual-cpp-build-tools` to download and install. HDF5 is particularly useful when the dataset is bigger than the memory available, since it writes/reads to disk but to the user everything is presented as an array. For example saving to disk is just assigning a value to a variable:

```
dset[:,:,i] = img
```

This line would be writing to disk the 2D array img. When reading:

```
img = dset[:,:,1]
```

Would load to memory only one 2D array. For the documentation and understanding of how HDF5 works, I highly suggest reading the website:

```
http://docs.h5py.org/en/latest/quick.html
```

1.2 Installation instructions

To install Experimentor it is important to be inside of a virtual environment. If you want to set up a working environment, I suggest you to check *Setting up a Python working environment*. From the command line you can run the following command:

```
pip install -U https://github.com/aquilesC/experimentor/archive/master.zip
```

Remember that in this case master refers to the branch you are installing. In case you want to work with specific branches of the code, you should change it.

If you are planning to develop code (you need to change, correct a bug, or anything else), you need to install the package in an editable way. Just run:

```
pip install -e git+git@github.com:aquilesC/experimentor.git#egg=experimentor
```

This will install the package inside of your virtual environment and will generate a copy of the repository in virtualenv/src/experimentor that you can edit and push to the repository of your choice. This is very handy when you want to test new features, etc. It is also possible to work with different branches, making it very easy to keep track of the changes in the upstream code.

After you have installed the program, you can check how to *Starting to use the Experimentor*

1.3 Starting to use the Experimentor

The package provides the basic classes and functions to communicate with devices and plan experiments. The package was designed to impose a workflow that guarantees the user will be able to plan an experiment from start to finish.

Experimentor is packaged with the folder containing the main code, i.e. the package itself, and a folder of examples that show some cases of how the program can be used.

The logic behind Experimentor is that the parameters that a user has to set for performing an experiment are layed out in YAML files. So, for example, the port at which a photodiode is plugged is defined in a yaml file. The steps of the experiment are also layed out in a YAML file, in order to make clear what needs to be set, changed, scanned, etc. These files are then read and passed to a special class of Experimentor.

Experimentor defines only general classes and methods that enable the user to define a common approach to the problems, however every experiment is different and has to be developed by each user. The examples folder are a good starting point for lerning, as well as the explanations found hereafter.

The steps needed to make an experiment using the Experimentor are as follow:

> First one has to define which devices are going to be used and make a YAML file for them. Then, the steps of the experiment are layed out in another YAML file. A class based on Experimentor is written, with methods for every step layed out in the YAML file. In principle a GUI can be built to modify the parameters passed to the experimentor class.

Each step is thought in order to force the user to be in control of his/her experiment and not to rely on preconceived concepts that can be far from how the setup actually works.

1.3.1 Defining devices, sensors and actuators

Every experiment should start by defining what devices, sensors and actuators are going to be used. The general structure is that devices communicate with the computer, while sensors and actuators are plugged into devices. For

example an acquisition card such as an oscilloscope is a *device*. A photodiode connected to it is a *sensor*, while a piezo stage connected to the output of a function generator is an *actuator*.

In the same fashion, a tunable laser is a device and the wavelength is an actuator. In this way the same structure of programs can be preserved throughout different projects. This is specially handy when developing GUIs, since it enables the iteration through all sensors or all actuators of a given device.

Devices, sensors and actuators are defined through YAML files. The examples folder contains some general files that show how to do it. Generally speaking it would look like something like this:

```
NI-DAQ:
  name: NI-DAQ
  type: daq
  model: ni
  number: 2
  driver: experimentor.models.daq.ni/ni
  connection:
    type: daq
    port: 2
  trigger: external
  trigger_source: PFI0
```

No field is required a priory, but giving a name is highly recommended. All the other fields are self explanatory. Sensors and actuators are defined in a similar way:

```
NI-DAQ:
  Stage 1:
    port: 1
    type: analog
    mode: output
    description: Example analog Out
    calibration:
      units: um  # Target units, starting from volts. The calibration thus would be:␣
↪device_value (true units) = slope*volts+offset
      slope: 1
      offset: 0
    limits:
      min: 0um
      max: 10um
    default: 5um
```

Note that the first key is the device to which the actuator is plugged. If defining actuators or sensors in the same file, they should be nested according to the device to which they are plugged. The first key afterwards is the name of the device and should be unique; if not, it will be overriden by the latest sensor/actuator loaded. The two more important pieces of information are the calibration and limits. The first explains the program how to convert from volts (the natural units of any ADQ) to the units of the actuator/sensor. The latter is for safety purposes and to maximize the convertion resolution of the DAQ that support setting variable gains.

None of the keys specified here are mandatory, but common sense dictates that the ones shown in the example are the minimum required ones for an experiment.

1.3.2 Defining the Experiment

The experiment, following the scheme developed for devices, sensors and actuators, is layed out in a YAML file. When writing it, the user has to keep in mind what are the parameters that need to be used, the kind of measurements that have to be achieved, etc. At this stage it is only a matter of thinking, the real logic and communication with devices will come later.

For example, if we want to scan the wavelength of a laser, we would write something like this:

```
init:
  devices: 'config/devices.yml'
  sensors: 'config/sensors.yml'
  actuators: 'config/actuators.yml'

scan:
  laser:
    name: Santec Laser
    params:
      start_wavelength: 1491 nm
      stop_wavelength:  1510 nm
      wavelength_speed:  10 nm/s
      interval_trigger: 0.01 nm
      sweep_mode: ContOne
      wavelength_sweeps: 1
  detectors:
    NI-DAQ:
      - Photodiode Test
      - Photodiode 2

finish:
  laser:
    shutter: False
```

You see that by laying down the experiment like this, it is easier to decie what we should do 'under the hood'. Of course, this example was already iterated; normally, you would write down fewer parameters, and while developing the code, you'll realized you forgot to declare an important variable, then you go back and you added, etc.

Block by block:

```
init:
  devices: 'config/devices.yml'
  sensors: 'config/sensors.yml'
  actuators: 'config/actuators.yml'
```

Of course the first step is to established where the config files are. Having it explicitly stated enables the user to keep several config files for different experiments, but with the same underlying logic. May not be initially apparent why at the beginning, but it becomes clearer with time.

The second block is where we actually define what scan we want to do:

```
scan:
  laser:
    name: Santec Laser
    params:
      start_wavelength: 1491 nm
      stop_wavelength:  1510 nm
      wavelength_speed:  10 nm/s
      interval_trigger: 0.01 nm
      sweep_mode: ContOne
      wavelength_sweeps: 1
  detectors:
    NI-DAQ:
      - Photodiode Test
      - Photodiode 2
```

The first key, *laser* establishes what device we are going to scan. The name here, as you may have guessed, is the

name we gave to the device when we defined it in *devices.yml*. The block of *parmas* sets all the parameters we need to make a scan, i.e., the starting wavelength, the stop wavelength, etc. At this moment I won't enter into the details of the chosen names, but they are closely related to properties in the driver of the laser.

Finally, the *detectors* block determines what detectors are going to be monitored while the laser is scanning. They are nested according to the device to which they are plugged to.

There are few things worth noting before moving forward. First, there is no need of any more information for performing a scan. However you see also that we are not establishing any logic to the measurement, meaning, for example, when and how we trigger it.

Experimentor is thought in such a way that the logic should be hardcoded into the Python code. However, if it is important to have the flexibility of altering a trigger behavior, etc. one could add an extra parameter in the scan block that later will be interpreted by the Python code.

Finally, we have to do something when the experiment finishes, in our case we only want to close the shutter:

```
finish:
  laser:
    shutter: False
```

The overall structure of the yaml file may look a bit more involved than needed by simple experiments; for example we explicitly state which laser we use, while we could have hard coded this (there is only one laser plugged into the experiment). However keeping a more flexible approach enables users to re utilize code more easily. Scanning a laser today may be scanning a stepper motor tomorrow.

1.4 Experimentor Reference

1.4.1 Models

Actions

Action

An action is an event that gets triggered on a device. For example, a camera can have an action `acquire` or `read`. They should normally be associated with the pressing of a button. Action is a handy decorator to register methods on a model and have quick access to them when building a user interface. They are multi-threaded by default, however, they share the same executor, defined at the model-level. Therefore, if a device is able to run several actions simultaneously, different executors can be defined at the moment of Action instantiation.

To extend Actions, the best is to sub class it and re implement the `get_executor` method, or any other method relevant to change the expected behavior.

Examples

A general purpose model can implement two methods: `initialize` and `auto_calibrate`, we can use the Actions to increment their usability:

```
class TestModel:
    @Action
    def initialize(self):
        print('Initializing')

    @Action
```

(continues on next page)

(continued from previous page)

```
    def auto_calibrate(self):
        print('Auto Calibrating')

tm = TestModel()
tm.initialize()
tm.auto_calibrate()
print(tm.get_actions())
```

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.action.**Action**(*method=None*, ***kwargs*)

Decorator for methods in models. Actions are useful when working with methods that run once, and are normally associated with pressing of a button. Actions are multi-threaded by default, using a single executor that returns a future.

Even though Actions (intended as the method in a model) can take arguments, it may be a better approach to store the parameters as attributes before triggering an action. In this way, triggering an action would be equivalent to pressing a button. In the same way, actions can store return values as attribute in the model itself, avoiding the need to keep track of the future returned by the action. Be aware of potential racing conditions that may arise when using shared memory to exchange information.

Todo: Define a clear protocol for exchanging information with models. Should it be state-based (i.e. storing parameters as attributes in the class) or statement based (i.e. passing parameters as arguments of methods).

---

get_executor()

Gets the executor either explicitly defined as an argument when instantiating the Action, or grabs it from the parent instance, and thus is shared between all action in a model.

To change the behavior, subclass Action and overwrite this method.

get_lock()

Gets the lock specified in the keyword arguments while creating the Action, or defaults to the lock stored in the instance and thus shared between all actions in the model.

Deprecated since version 0.3.0: Since v0.3.0 we are favoring concurrent.futures instead of lower-level threading for Actions.

get_run()

Generates the run function that will be applied to the method. It looks a big convoluted, but it is one of the best approaches to make it easy to extend the Actions in the longer run. The return callable grabs the executor from the method `self.get_executor()`.

Returns A function that takes two arguments: method and instance and that submits them to an executor

Return type callable

set_action(*method*)

Wrapper that returns this own class but initializes it with a method and a previously stored dict of kwargs. This method is what happens when the Action itself is defined with arguments.

Parameters **method** (*callable*) – The method that is decorated by the Action

Returns Returns an instance of the Action using the previously stored kwargs but adding the method

Return type *Action*

Features

Features

Features in a model are those parameters that can be read, set, or both. They were modeled after Lantz Feat objects, and the idea is that they can encapsulate common patterns in device control. They are similar to `Settings` in behavior, except for the absence of a cache. Features do communicate with the device when reading a value.

For example, a feature could be the value of an analog input on a DAQ, or the temperature of a camera. They are meant to be part of a measurement, their values can change in loops in order to make a scan. Features can be used as decorators in pretty much the same way @propery can be used. The only difference is that they register themselves in the models properties object, so it is possible to update values either by submitting a value directly to the Feature or by sending a dictionary to the properties and updating all at once.

It is possible to mark a feature as a setting. In this case, the value will not be read from the device, but it will be cached. In case it is needed to refresh a value from the device, it is possible to use a specific argument, such as `None`. For example:

```
@Feature(setting=True, force_update_arg=0)
def exposure(self):
    self.driver.get_exposure()

@exposure.setter
def exposure(self, exposure_time):
    self.driver.set_exposure(exposure_time)
```

Todo: It is possible to define complex behavior such as unit conversion, limit checking, etc. We should narrow down what is appropriate for a model and what should go into the Controller.

Todo: A useful pattern is to catch the exception raised by the controllers if a value is out of range, or with the wrong units.

---

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.feature.**Feature**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*, ***kwargs*)

Properties that belong to models. It makes easier the setting and getting of attributes, while at the same time it keeps track of the properties of each model. A Feature is, fundamentally, a descriptor, that extends some functionality by accepting keyword arguments when defining.

Todo: There is a lot of functionality that can be implemented, but that hasn't yet, such as checking limits, unit conversion, etc.

---

name

The name of the feature, it must be unique since it will be used as a key in a dictionary.

Type str

kwargs

If the feature is initialized with arguments, they will be stored here. Only keyword arguments are allowed.

Type dict

deleter (*fdel*)

getter (*fget*)

kwargs = None

name = ''

setter (*fset*)

Properties

Properties

Every model in Experimentor has a set of properties that define their state. A camera has, for example, an exposure time, a DAQ card has a delay between data points, and an Experiment holds global parameters, such as the number of repetitions a measurement should take.

In many situations, the parameters are stored as a dictionary, mainly because they are easy to retrieve from a file on the hard drive and to access from within the class. We want to keep that same approach, but adding extra features.

Features of Properties

Each parameter stored on a property will have three values: new_value, value, old_value, which represent the value which will be set, the value that is currently set and the value that was there before. In this way it is possible to just update on the device those values that need updating, it is also possible to revert back to the previously known value.

Each value will also be marked with a flag to_update in case the value was changed, but not yet transmitted to the device. This allows us to collect all the values we need, for example looping through a user interface, reading a config file, and applying only those needed whenever desired.

The Properties have also another smart feature, achieved through linking. Linking means building a relationship between the parameters stored within the class and the methods that need to be executed in order to get or set those values. In the linking procedure, we can set only getter methods for read-only properties, or both methods. A general apply function then allows to use the known methods to set the values that need to be updated to the device.

Future Roadmap

We can consider forcing methods to always act on properties defined as new/known/old in order to use that information as a form of cache and validation strategy.

license MIT, see LICENSE for more details

copyright 2021 Aquiles Carattino

class experimentor.models.properties.**Properties**(*parent: experimentor.models.models.BaseModel*, ***kwargs*)

Class to store the properties of models. It keeps track of changes in order to monitor whether a specific value needs to be updated. It also allows to keep track of what method should be triggered for each update.

all()

Returns a dictionary with all the known values.

Returns **properties** – All the known values

Return type dict

apply(*property*, *force=False*)
Applies the new value to the property. This is provided that the property is marked as to_update, or forced to be updated.

Parameters

- **property** (*str*) – The string identifying the property
- **force** (*bool (default: False)*) – If set to true it will update the propery on the device, regardless of whether it is marked as to_update or not.

apply_all()
Applies all changes marked as 'to_update', using the links to methods generated with :meth:~link

autolink()
Links the properties defined as `ModelProp` in the models using their setters and getters.

fetch(*prop*)
Fetches the desired property from the device, provided that a link is available.

fetch_all()
Fetches all the properties for which a link has been established and updates the value. This method does not alter the to_update flag, new_value, nor old_value.

classmethod from_dict(*parent*, *data*)
Create a Properties object from a dictionary, including the linking information for methods. The data has to be passed in the following form: {property: [value, getter, setter]}, where *getter* and *setter* are the methods used by :meth:~link.

Parameters

- **parent** – class to which the properties are attached
- **data** (*dict*) – Information on the values, getter and setter for each property

get_property(*prop*)
Get the information of a given property, including the new value, value, old value and if it is marked as to be updated.

Returns **prop** – The requested property as a dictionary

Return type dict

link(*linking*)
Link properties to methods for update and retrieve them.

Parameters **linking** (*dict*) – Dictionary in where information is stored as parameter=>[getter, setter], for example:

```
linking = {'exposure_time': [self.get_exposure, self.set_exposure]}
```

In this case, `exposure_time` is the property stored, while `get_exposure` is the method that will be called for getting the latest value, and set_exposure will be called to set the value. In case set_exposure returns something different from None, no extra call to get_exposure will be made.

to_update()
Returns a dictionary containing all the properties marked to be updated.

Returns **props** – all the properties that still need to be updated

Return type dict

unlink (*unlink_list*)
Unlinks the properties and the methods. This is just to prevent overwriting linkings under the hood and forcing the user to actively unlink before linking again.

Parameters **unlink_list** (*list*) – List containing the names of the properties to be unlinked.

update (*values: dict*)
Updates the values in the same way the update method of a dictionary works. It, however, stores the values as a new value, it does not alter the values stored. For updating the proper values use `self.upgrade()`.

After updating the values, use `self.apply_all()` to send the new values to the device.

upgrade (*values*, *force=False*)
This method actually overwrites the values stored in the properties. This method should be used only when the real values generated by a device are known. It will change the new values to None, it will set the value to value, and it will set the `to_update` flag to false.

Parameters

- **values** (*dict*) – Dictionary in the form {property: new_value}
- **force** (*bool*) – If force is set to True, it will create the missing properties instead of raising an exception.

Models

Models

Models are a buffer between user interactions and real devices. Models should define at least some basic common properties, for example how to read a value from a sensor and how to apply a value to an actuator. Models can also take care of manipulating data, for example calculating an FFT and returning it to the user.

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.models.**BaseModel**
All models should inherit from this base model. It defines some basic methods and checks that prevent errors later at runtime.

_features
Dictionary-like object to store the properties of the model

Type *ExpDict*

_actions
List-like object to store the available actions. It also stores a lock to prevent multiple actions to be triggered at the same time

Type *ExpList*

_settings
Dictionary-like object where the settings are stored. This dictionary is also used to retrieve the latest known value of the setting.

Type *ExpDict*

_signals
Dictionary-like object to store the signals of the model

Type *ExpDict*

_subscribers
Dictionary-like object storing the subscribers to different signals arising from this model

Type *ExpDict*

classmethod as_process(**args*, ***kwargs*)
Instantiate the model as a `ProxyObject` that will run on a separate process.

> **Warning:** This is WORK IN PROGRESS and will remain so for the foreseeable future.

clean_up_threads()
Keep only the threads that are alive.

create_context()
Creates the ZMQ context. In case of wanting to use a specific context (perhaps globally defined), overwrite this method in the child classes. This method is called during the model instantiation.

create_publisher()
Creates a ZMQ publisher. It will be used by signals to broadcast their information. There is a delay before returning the publisher to guarantee that it was properly initialized before actually using it.

Returns

- *zmq.Publisher* – Returns the initialized publisher
- *.. todo:: This method has a high chance of being converted to an Action in order to let it run in parallel*

emit(*signal_name*, *payload*, ***kwargs*)
Emits a signal using the publisher bound to the model. It uses the method `BaseModel.get_publisher()` to get the publisher to use. You can override that method in order to use a different publisher (for example, an experiment-based publisher instead of a model-based one.

Notes

If subscribers are too slow, a queue will build up on the publisher, which may lead to the model itself crashing. It is important to be sure subscribers can keep up.

Parameters

- **signal_name** (*str*) – The name of the signal is used as a topic for the publisher. Remember that in PyZMQ, topics are filtered on the subscriber side, therefore everything is always broadcasted broadly, which can be a bottleneck for performance in case there are many subscribers.
- **payload** – It will be sent by the publisher. In case it is a `numpy` array, it will use a zero-copy strategy. For the rest, it will send using `send_pyobj`, which serializes the payload using pickle. This can be a *slow* process for complex objects.
- **kwargs** – Optional keyword arguments to make the method future-proof. Rigth now, the only supported keyword argument is `meta`, which will append to the current meta_data being broadcast. For numpy arrays, metadata is a dictionary with the following keys: `numpy`, `dtype`, `shape`. For non-numpy objects, the only key is `numpy`. The submitted metadata is appended to the internal metadata, therefore be careful not to overwrite its keys unless you know what you are doing.

finalize()
Finalizes the model. It only takes care of closing the publisher. Child classes should implement their

own finalize methods (they get called automatically), and either close the publisher explicitly or use this method.

classmethod get_actions()
Returns the list of actions stored in the model. In case this behavior needs to be extended, the method can be overwritten in any child class.

get_context()
Gets the context. By default it is stored as a 'private' attribute of the model. Overwrite this method in child classes if there is need to extend functionality.

Returns The context created with `self.create_context()`

Return type zmq.Context

classmethod get_features()
Returns the dict-like features of the model. If this behavior needs to be extended, the method can be overwritten by any child class.

get_publisher()
Returns the publisher stored as a private attribute, and initialized during instantiation of the model. Consider overwriting it in order to extend functionality.

get_publisher_port()
ZMQ allows to create publishers that bind to an available port without specifying which one. This flexibility means that we should check to which port the publisher was bound if we want to use it. See `self.create_publisher()` for more details.

Returns The port to which the publisher is bound. A string of integers

Return type str

get_publisher_url()
Each publisher can run on a different computer. This method should return the URL in which to connect to the publisher.

Todo: Right now it only returns localhost, this MUST be improved

---

initialize()

classmethod set_actions(*actions*)
Method to store actions in the model. It is a convenience method that can be overwritten by child classes.

subscribers

class experimentor.models.models.**ExpDict**

class experimentor.models.models.**ExpList**

lock = <Lock(owner=None)>

class experimentor.models.models.**ProxyObject**(*cls*, **args*, ***kwargs*)
Creates an object that can run on a separate process. It uses pipes to exchange information in and out. This is experimental and not meant to be used in a real application. It is here as a way of documenting one of the possible directions.

Note: Right now we are using the multiprocessing pipes to exchange information, it would be useful to use the zmq options in order to have a consistent interface through the project.

Exceptions

Model Exceptions

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

exception experimentor.models.exceptions.**ExperimentorException**
Base exception for all experimentor modules

exception experimentor.models.exceptions.**LinkException**

exception experimentor.models.exceptions.**ModelException**

exception experimentor.models.exceptions.**PropertyException**

exception experimentor.models.exceptions.**SignalException**

Meta

Meta Models

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.meta.**MetaModel**(*name*, *bases*, *attrs*)
Meta Model type which will be responsible for keeping track of all the created models in the program. This is very useful for things like automatically building a GUI, initializing/finishing all the devices, etc. and also to perform checks at the beginning of the runtime, by doing introspection on all the defined models, regardless of whether they are instantiated later on or no.

One of the tasks is to generate a list of signals available in each model. Signals are specified as class attributes and therefore they can be accounted for before instantiating the class. Once the class is being instantiated, each object will re-instantiate the signals in order to keep its own copy, and establishing the proper owner of the signal.

get_instances(*recursive=False*)
Get all instances of this class in the registry.

Parameters **recursive** (*bool*) – Search for instances recursively through inherited objects

get_models(*recursive=False*)
Gets all the models which share the MetaModel origin.

Parameters **recursive** (*bool*) – Search recurisvely in sub classes of the model

Models for Devices

Base Device

class experimentor.models.devices.base_device.**ModelDevice**
All models for devices should inherit from this class.

Meta Devices

class experimentor.models.devices.meta.**MetaDevice**(*name*, *bases*, *attrs*)
This is a Meta Class that should be used only by devices and not by the experiment itself. It is only to give more granularity to the program when wanting to perform operations on all the devices or on different possible measurements.

Device Exceptions

exception experimentor.models.devices.exceptions.**DeviceException**

Models for Cameras

Base Camera Model

Base Camera Model

Camera class with the base methods. Having a base class exposes the general API for working with cameras. This file is important to keep track of the methods which are exposed to the View. The class BaseCamera should be subclassed when developing new Models for other cameras. This ensures that all the methods are automatically inherited and there are no breaks downstream.

Conventions

Images are 0-indexed. Therefore, a camera with (1024px X 1024px) will be used as img[0:1024, 0:1024] (remember Python leaves out the last value in the slice.

Region of Interest is specified with the coordinates of the corners. A full-frame with the example above would be given by X=[0,1023], Y=[0,1023]. Be careful, since the maximum width (or height) of the camera is 1024.

The camera keeps track of the coordinates of the initial pixel. For full-frame, this will always be [0,0]. When croping, the corner-pixel will change. It is very important to keep track of this value when building a GUI, since after the first crop, if the user wants to crop even further, the information has to be referenced to the already cropped area.

Notes

IMPORTANT Whatever new function is implemented in a specific model, it should be first declared in the BaseCamera class. In this way the other models will have access to the method and the program will keep running (perhaps with the wrong behavior though).

class experimentor.models.devices.cameras.base_camera.**BaseCamera**(*camera*, *initial_config=None*)
Base Camera model. All camera models should inherit from this model in order to extend functionality. There are some assumptions regarding how to update different settings such as exposure, gain, region of interest.

Parameters **camera** (*str or int*) – Parameter to identify the camera when loading or initializing it.

AQUISITION_MODE
Different acquisition modes: Continuous, Single, Keep last.

Type dict

cam_num
This parameter will be used to identify the camera when loading or initializing it.

Type str or int

running
Whether the camera is running or not

Type bool

max_width
Maximum width, in pixels

Type int

max_height
Maximum height, in pixels

Type int

data_type
The data type of the images generated by the camera. This can be used to allocate the correct amount of memory in buffers, or to reduce data before displaying it. For example, `np.uint16`.

Type np data type

temp_image
It stores the last image acquired by the camera. Useful for user interfaces that need to display images at a rate different than the acquisition rate.

Type np.array

ACQUISITION_MODE = {0: 'Single', 1: 'Continuous', 2: 'Keep Last'}

MODE_CONTINUOUS = 1

MODE_LAST = 2

MODE_SINGLE_SHOT = 0

ROI
Sets up the ROI. Not all cameras are 0-indexed, so this is an important place to define the proper ROI.

vals [list or tuple] Organized as (X, Y), where the coordinates for the ROI would be X[0], X[1], Y[0], Y[1]

acquisition_mode
Single or continuous. :param int mode: One of self.MODE_CONTINUOUS, self.MODE_SINGLE_SHOT

Type Set the readout mode of the camera

acquisition_ready()
Checks if the acquisition in the camera is over.

binning
The binning of the camera if supported. Has to check if binning in X/Y can be different or not, etc.

The binning is specified as a list or tuple like: [X, Y], with the information of the binning in the X or Y direction.

camera = 'Base Camera Model'

ccd_height
Returns the CCD height in pixels this is equivalent to the *max_height*

ccd_width
Returns the CCD width in pixels this is equivalent to the *max_width*

clear_ROI()
Clears the ROI by setting it to the maximum available area.

clear_binning()
Clears the binning of the camera to its default value.

configure(*properties: dict*)
Configure the camera based on a dictionary of properties.

Deprecated since version 0.3.0: By implementing features, this method is no longer required

exposure
Sets the exposure of the camera.

gain
Sets the gain on the camera, if possible

gain [float] The gain, depending on the camera it can be an integer, it can be specified in dB, etc.

initialize()
Initializes the camera.

read_camera()
Reads the camera and stores the image in the temp_image attribute

serial_number
Returns the serial number of the camera, or a way of identifying the camera in an experiment.

stop_acquisition()
Stops the acquisition without closing the connection to the camera.

stop_camera()
Stops the acquisition and closes the connection with the camera.

trigger_camera()
Triggers the camera.

Camera Model Exceptions

exception experimentor.models.devices.cameras.exceptions.**CameraException**

exception experimentor.models.devices.cameras.exceptions.**CameraNotFound**

exception experimentor.models.devices.cameras.exceptions.**CameraTimeout**

exception experimentor.models.devices.cameras.exceptions.**WrongCameraState**

Basler

class experimentor.models.devices.cameras.basler.basler.**BaslerCamera**(*camera*, *initial_config=None*)

ROI

acquisition_mode

auto_exposure
Off, Once, Continuous

Type Auto exposure can take one of three values

auto_gain
Off, Once, Continuous

Type Auto Gain must be one of three values

binning_x

binning_y

buffer_size

ccd_height

ccd_width

continuous_reads()

exposure
The exposure of the camera, defined in units of time

finalize()
Finalizes the model. It only takes care of closing the publisher. Child classes should implement their own finalize methods (they get called automatically), and either close the publisher explicitly or use this method.

frame_rate

gain
Gain is a float

height

initialize
Decorator for methods in models. Actions are useful when working with methods that run once, and are normally associated with pressing of a button. Actions are multi-threaded by default, using a single executor that returns a future.

Even though Actions (intended as the method in a model) can take arguments, it may be a better approach to store the parameters as attributes before triggering an action. In this way, triggering an action would be equivalent to pressing a button. In the same way, actions can store return values as attribute in the model itself, avoiding the need to keep track of the future returned by the action. Be aware of potential racing conditions that may arise when using shared memory to exchange information.

Todo: Define a clear protocol for exchanging information with models. Should it be state-based (i.e. storing parameters as attributes in the class) or statement based (i.e. passing parameters as arguments of methods).

---

new_image
Base signal which implements the common pattern for defining, emitting and connecting a signal

pixel_format
Pixel format must be one of Mono8, Mono12, Mono12p

read_camera() → list
Reads the camera and stores the image in the temp_image attribute

start_free_run()
Starts a free run from the camera. It will preserve only the latest image. It depends on how quickly the experiment reads from the camera whether all the images will be available or only some.

stop_camera
Decorator for methods in models. Actions are useful when working with methods that run once, and are normally associated with pressing of a button. Actions are multi-threaded by default, using a single executor that returns a future.

Even though Actions (intended as the method in a model) can take arguments, it may be a better approach to store the parameters as attributes before triggering an action. In this way, triggering an action would be equivalent to pressing a button. In the same way, actions can store return values as attribute in the model itself, avoiding the need to keep track of the future returned by the action. Be aware of potential racing conditions that may arise when using shared memory to exchange information.

Todo: Define a clear protocol for exchanging information with models. Should it be state-based (i.e. storing parameters as attributes in the class) or statement based (i.e. passing parameters as arguments of methods).

---

stop_continuous_reads()

stop_free_run
Decorator for methods in models. Actions are useful when working with methods that run once, and are normally associated with pressing of a button. Actions are multi-threaded by default, using a single executor that returns a future.

Even though Actions (intended as the method in a model) can take arguments, it may be a better approach to store the parameters as attributes before triggering an action. In this way, triggering an action would be equivalent to pressing a button. In the same way, actions can store return values as attribute in the model itself, avoiding the need to keep track of the future returned by the action. Be aware of potential racing conditions that may arise when using shared memory to exchange information.

Todo: Define a clear protocol for exchanging information with models. Should it be state-based (i.e. storing parameters as attributes in the class) or statement based (i.e. passing parameters as arguments of methods).

---

trigger_camera()
Triggers the camera.

width

Models for Experiments

Base Experiment Model

Base class for the experiments. `BaseExperiment` defines the common patterns that every experiment should have. Importantly, it starts an independent process called publisher, that will be responsible for broadcasting messages that are appended to a queue. The messages rely on the pyZMQ library and should be tested further in order to assess their limitations. The general pattern is that of the PUB/SUB, with one publisher and several subscribers.

The messages should include a *topic* and data. For this, the elements in the queue should be dictionaries with two keywords: **data** and **topic**. `data['data']` will be serialized through the use of cPickle, and is handled automatically by pyZQM through the use of `send_pyobj`. The subscribers should be aware of this and use either unpickle or `recv_pyobj`.

In order to stop the publisher process, the string `'stop'` should be placed in `data['data']`. The message will be broadcast and can be used to stop other processes, such as subscribers.

Todo: Check whether the serialization of objects with cPickle may be a bottleneck for performance.

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.experiments.base_experiment.**BaseExperiment**

class experimentor.models.experiments.base_experiment.**Experiment**(*filename=None*)
Base class to define experiments. Should keep track of the basic methods needed regardless of the experiment to be performed. For instance, a way to start and a way to finalize a measurement. This class is not meant to be instantiated directly, but should be subclassed in each project.

Parameters **filename** (*str or None*) – Path to the config file that will be loaded. Ideally it should be an absolute path, to prevent problems. If you submit a relative path, it will depend on how you are running the program if the file will be found or not.

config
Properties object to store the values of the parameters of the experiments. See *experimentor.models.properties* to understand the options and how it works

Type *Properties*

logger
The logger of the experiment, this is for internal use only

Type logger

alive_threads

connect(*method*, *topic*, **args*, ***kwargs*)
Async method that connects the running publisher to the given method on a specific topic.

Parameters

- **method** – method that will be connected on a given topic
- **topic** (*str*) – the topic that will be used by the subscriber to discriminate what information to collect.
- **args** – extra arguments will be passed to the subscriber, which in turn will pass them to the function
- **kwargs** – extra keyword arguments will be passed to the subscriber, which in turn will pass them to the function

connections

finalize()
Needs to be overridden by child classes.

list_alive_threads

load_configuration(*filename*, *loader=<class 'yaml.loader.SafeLoader'>*)
Loads the configuration file in YAML format.

Parameters **filename** (*str*) – full path to where the configuration file is located.

Raises **FileNotFoundError** – if the file does not exist.

static make_filename(*folder: Union[str, tuple], filename: str*)
This routine will check if the folder to store data exists, and create it if not. It will also check if the file exists, if it does, it will increase by 1 a counter until an available name appears, and return both the directory and the filename.

Parameters

- **filename** – if it contains a '{i}' or similar in its specification, it will use it as a counter, if not, the number will be prepended to the filename
- **folder** – either a string with the full path to the folder (bear in mind differences of folder separators) or a tuple that will be joined using os.path.join

num_threads

set_up()
Needs to be overridden by child classes.

start
Base signal which implements the common pattern for defining, emitting and connecting a signal

stop_subscribers()
Puts the proper data into every alive subscriber in order to stop it.

update_config(***kwargs*)

class experimentor.models.experiments.base_experiment.**FormatDict**
Simple solution to do partial formatting of strings. For example:

```
>>> a = 'fiber_end_{cartridge}_{i:04}.npy'
>>> cartridge = 123
>>> a.format_map(FormatDict(cartridge=cartridge))
'fiber_end_123_{i:04}.npy'
```

class experimentor.models.experiments.base_experiment.**FormatPlaceholder**(*key*)

class experimentor.models.experiments.base_experiment.**MetaExperiment**(*name*, *bases*, *attrs*)
Meta Model type which will be responsible for keeping track of all the created experiments. It will also be responsible for registering the publisher, in order to have only one throughout the program and accessible from other parts of the program. This meta class may be overkill, since in principle every program will be only one experiment, but this is left as an effort to be future-proof.

Note: Defining meta classes may generate a feeling of obscurantism in the code. It may be wise to remove it and find a simpler/straightforward approach.

1.4.2 Views

class experimentor.views.data_view_widget.**DataViewWidget**(*parent=None*)
Base class that defines some common patterns for views which are meant to display data.

default_Layout
method get_layout

Type By default, views will have a QHBoxLayout, it can be overriden when subclassing, or by changing the

data
of what specific type of data it is.

Type This is the data being represented by the widget. This allows to define abstract methods for saving, regardless

default_layout = 'horizontal'

get_layout()
Returns the layout specified as the class attribute default_layout. Override this method to provide more complex behavior.

set_layout()

experimentor.views.decorators.**try_except_dialog**(*func*)
Decorator to add to methods used in user interfaces. If there is a chance of an error appearing because of devices in the wrong state, etc. but the logic is not fail proof, you can use this decorator to display an error message with the stack trace instead of crashing the program.

exception experimentor.views.exceptions.**ViewException**

Camera View

class experimentor.views.camera.**CameraViewerWidget**(*parent=None*)
The Camera Viewer Widget is a wrapper around PyQtGraph ImageView. It adds some common methods for getting extra mouse interactions, such as performing an auto-range through right-clicking, it allows to drag and drop horizontal and vertical lines to define a ROI, and it allows to draw on top of the image. The core idea is to make these options explicit, in order to systematize them in one place.

clicked_on_image: Emits [float, float] with the coordinates where the mouse was clicked on the image. Does not distinguish between left/right clicks. Any further processing must be done downstream.

layout

Type QHBoxLayout, in case extra elements must be added

viewport

Type GraphicsLayoutWidget

view

Type ViewBox

img

Type ImageItem

imv

Type ImageView

auto_levels

Type Whether to actualize the levels of the image every time they are refreshed

add_actions_to_menu()
Adds actions to the contextual menu. If you want to have control on which actions appear, consider subclassing this widget and overriding this method.

clicked_on_image

classmethod connect_to_camera(*camera*, *refresh_time=50*, *parent=None*)
Instantiate the viewer using connect_to_camera in order to get some functionality out of the box. It will create a timer to automatically update the image

do_auto_range()
Sets the levels of the image based on the maximum and minimum. This is useful when auto-levels are off (the default behavior), and one needs to quickly adapt the histogram.

draw_target_pointer(*locations*)
gets an image and draws a circle around the target locations.

Parameters **locations** (*DataFrame*) – DataFrame generated by trackpy's locate method. It only requires columns *x* and *y* with coordinates.

get_roi_values()
Get's the ROI values in camera-space. It keeps track of the top left corner in order to update the values before returning. :return: Position of the corners of the ROI region assuming 0-indexed cameras.

keyPressEvent(*key*)
Triggered when there is a key press with some modifier. Shift+C: Removes the cross hair from the screen These last two events have to be handeled in the mainWindow that implemented this widget.

mouseMoved(*arg*)
Updates the position of the cross hair. The mouse has to be moved while pressing down the Ctrl button.

mouse_clicked(*evnt*)

scene()
Shortcut to getting the image scene

set_roi_lines(*X*, *Y*)

setup_cross_cut(*max_size*)
Set ups the horizontal line for the cross cut.

setup_cross_hair(*max_size*)
Sets up a cross hair.

setup_mouse_tracking()

setup_roi_lines(*max_size=None*)
Sets up the ROI lines surrounding the image.

Parameters **max_size** (*list*) – List containing the maximum size of the image to avoid ROIs bigger than the CCD.

update_image(*image*, *auto_range=False*, *auto_histogram_range=False*)
Updates the image being displayed with some sensitive defaults, which can be over written if needed.

Camera Viewer Widget

Wrapper around PyQtGraph ImageView.

class experimentor.views.camera.camera_viewer_widget.**CameraViewerWidget**(*parent=None*)
The Camera Viewer Widget is a wrapper around PyQtGraph ImageView. It adds some common methods for getting extra mouse interactions, such as performing an auto-range through right-clicking, it allows to drag and drop horizontal and vertical lines to define a ROI, and it allows to draw on top of the image. The core idea is to make these options explicit, in order to systematize them in one place.

clicked_on_image: Emits [float, float] with the coordinates where the mouse was clicked on the image. Does not distinguish between left/right clicks. Any further processing must be done downstream.

layout

Type QHBoxLayout, in case extra elements must be added

viewport

Type GraphicsLayoutWidget

view

Type ViewBox

img

Type ImageItem

imv

Type ImageView

auto_levels

Type Whether to actualize the levels of the image every time they are refreshed

add_actions_to_menu()
Adds actions to the contextual menu. If you want to have control on which actions appear, consider subclassing this widget and overriding this method.

clicked_on_image

classmethod connect_to_camera(*camera*, *refresh_time=50*, *parent=None*)
Instantiate the viewer using connect_to_camera in order to get some functionality out of the box. It will create a timer to automatically update the image

do_auto_range()
Sets the levels of the image based on the maximum and minimum. This is useful when auto-levels are off (the default behavior), and one needs to quickly adapt the histogram.

draw_target_pointer(*locations*)
gets an image and draws a circle around the target locations.

Parameters **locations** (*DataFrame*) – DataFrame generated by trackpy's locate method. It only requires columns *x* and *y* with coordinates.

get_roi_values()
Get's the ROI values in camera-space. It keeps track of the top left corner in order to update the values before returning. :return: Position of the corners of the ROI region assuming 0-indexed cameras.

keyPressEvent(*key*)
Triggered when there is a key press with some modifier. Shift+C: Removes the cross hair from the screen These last two events have to be handeled in the mainWindow that implemented this widget.

mouseMoved(*arg*)
Updates the position of the cross hair. The mouse has to be moved while pressing down the Ctrl button.

mouse_clicked(*evnt*)

scene()
Shortcut to getting the image scene

set_roi_lines(*X*, *Y*)

setup_cross_cut(*max_size*)
Set ups the horizontal line for the cross cut.

setup_cross_hair(*max_size*)
Sets up a cross hair.

setup_mouse_tracking()

setup_roi_lines(*max_size=None*)
Sets up the ROI lines surrounding the image.

Parameters **max_size** (*list*) – List containing the maximum size of the image to avoid ROIs bigger than the CCD.

update_image(*image*, *auto_range=False*, *auto_histogram_range=False*)
Updates the image being displayed with some sensitive defaults, which can be over written if needed.

Model View

class experimentor.views.model_view.model_view.**ModelViewWidget**(*model: experimentor.models.devices.base_device.ModelDe parent=None*)

get_layout()

model_to_layout()

set_layout()

Widgets

class experimentor.views.widgets.**ToggableButton**(**args*, ***kwargs*)

toggle(*self*)

class experimentor.views.widgets.toggable_button.**ToggableButton**(**args*, ***kwargs*)

toggle(*self*)

1.4.3 Drivers

Analog Discovery

See the digilent.constants.

class experimentor.drivers.digilent.**AnalogDiscovery**
Bases: object

analog_in_acquisition_mode_get()

Returns Current mode

Return type AcquisitionMode

analog_in_acquisition_mode_info()
Returns the supported AnalogIn acquisition modes. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the ACQMODE constants in dwf.h. The acquisition mode selects one of the following modes, ACQMODE:

Returns Bitfield of modes, needs to be parsed

Return type int

analog_in_bits_info()

analog_in_buffer_size_get()

analog_in_buffer_size_info()

analog_in_buffer_size_set(*buffer_size*)

analog_in_channel_attenuation_get(*channel*)

analog_in_channel_attenuation_set(*channel*, *attenuation*)
Configures the attenuation for each channel. When channel index is specified as -1, each enabled AnalogIn channel attenuation will be configured to the same level. The attenuation does not change the attenuation on the device, just informs the library about the externally applied attenuation. :param channel: :type channel: int :param attenuation: :type attenuation: float

analog_in_channel_count()

analog_in_channel_disable(*channel*)
Disables the specified channel. See *analog_in_channel_enable()*

Parameters **channel** (*int*) –

analog_in_channel_enable(*channel*)
Enables the specified channel. See *analog_in_channel_disable()*

Parameters **channel** (*int*) –

analog_in_channel_enable_get(*channel*)

analog_in_channel_filter_get(*channel*)

analog_in_channel_filter_info()

analog_in_channel_filter_set(*channel*, *filter*)

analog_in_channel_offset_get(*channel*)

analog_in_channel_offset_info()

analog_in_channel_offset_set(*channel*, *offset*)

analog_in_channel_range_get(*channel*)

analog_in_channel_range_info()

Returns

- **volts_min** (*float*)
- **volts_max** (*float*)
- **volts_steps** (*float*)

analog_in_channel_range_set(*channel*, *channel_range*)

analog_in_configure(*reconfigure=1*, *start=1*)

analog_in_frequency_get()

analog_in_frequency_info()
Retrieves the minimum and maximum (ADC frequency) settable sample frequency.

Returns

- **min_freq** (*float*) – Minimum allowed frequency
- **max_freq** (*float*) – Maximum allowed frequency

analog_in_frequency_set(*frequency*)

analog_in_noise_size_info()

analog_in_record_length_get()

analog_in_record_length_set(*length*)

analog_in_reset()

analog_in_samples_left()

Retrieves the number of samples left in the acquisition.

Returns Number of samples remaining

Return type int

analog_in_samples_valid()

analog_in_sampling_delay_get()

analog_in_sampling_delay_set(*delay*)

analog_in_sampling_slope_get()

analog_in_sampling_slope_set(*slope*)

Parameters **slope** (*TriggerSlope*) –

analog_in_sampling_source_get()

analog_in_sampling_source_set(*source*)

Parameters **source** (*TriggerSource*) –

analog_in_status(*read_data=0*)

Checks the status of the acquisition

Parameters **read_data** (*int*) – 0 or 1, to indicate whether data should be read from the device

Returns The instrument state

Return type InstrumentState

analog_in_status_auto_trigger()

Verifies if the acquisition is auto triggered.

Returns I guess it returns 1 if the acquisition was auto triggered

Return type int

analog_in_status_data(*channel*, *samples*, *buffer=None*)

Retrieves the acquired data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer.

Parameters

- **channel** (*int*) –

- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type np.array

analog_in_status_data_16(*channel*, *first*, *samples*, *buffer=None*)

Retrieves the acquired raw data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer or creates a new one. This is the **raw** data, as opposed to what *analog_in_status_data()* returns.

Parameters

- **channel** (*int*) –
- **first** (*int*) –
- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type numpy.array

analog_in_status_data_2(*channel*, *first*, *samples*, *buffer=None*)

Retrieves the acquired data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer or creates a new buffer. This method allows to specify which data will be copied. To retrieve all data see *analog_in_status_data()*.

Parameters

- **channel** (*int*) –
- **first** (*int*) –
- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type numpy.array

analog_in_status_index()

Retrieves the buffer write pointer which is needed in ScanScreen acquisition mode to display the scan bar. :returns: Variable to receive the position of the acquisition. :rtype: int

analog_in_status_noise(*channel*, *samples*)

Retrieves the acquired noise samples from the specified idxChannel on the AnalogIn instrument.

Parameters

- **channel** (*int*) –
- **samples** (*int*) –

Returns minimum noise data, maximum noise data

Return type 2-colum numpy.array

analog_in_status_record()
Retrieves information about the recording process. The data loss occurs when the device acquisition is faster than the read process to PC. In this case, the device recording buffer is filled and data samples are overwritten. Corrupt samples indicate that the samples have been overwritten by the acquisition process during the previous read. In this case, try optimizing the loop process for faster execution or reduce the acquisition frequency or record length to be less than or equal to the device buffer size (record length <= buffer size/frequency).

Returns

- **data_available** (*int*) – Available number of samples
- **data_lost** (*int*) – Lost samples after the last check
- **data_corrupt** (*int*) – Number of samples that can be corrupt

analog_in_status_sample(*channel*)
Gets the last ADC conversion sample from the specified idxChannel on the AnalogIn instrument.

Parameters **channel** (*int*) –

Returns Sample value

Return type float

analog_in_trigger_auto_timeout_get()

analog_in_trigger_auto_timeout_info()

analog_in_trigger_auto_timeout_set(*timeout=0*)

analog_in_trigger_channel_get()

analog_in_trigger_channel_info()

analog_in_trigger_channel_set(*channel*)
Sets the trigger channel.

analog_in_trigger_condition_get()

analog_in_trigger_condition_info()
Returns the supported trigger type options for the instrument. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the DwfTriggerSlope constants in dwf.h. These trigger condition options are:

- **DwfTriggerSlopeRise (This is the default setting):**
  - For edge and transition trigger on rising edge.
  - For pulse trigger on positive pulse; For window exiting.
- **DwfTriggerSlopeFall**
  - For edge and transition trigger on falling edge.
  - For pulse trigger on negative pulse; For window entering.
- **DwfTriggerSlopeEither**
  - For edge and transition trigger on either edge.
  - For pulse trigger on either positive or negative pulse.

Returns **info**

Return type int

analog_in_trigger_condition_set(*condition*)

analog_in_trigger_filter_get()

analog_in_trigger_filter_info()
Returns the supported trigger filters. They are returned (by reference) as a bit field which can be parsed using the IsBitSet Macro. Individual bits are defined using the FILTER constants in DWF.h. Select trigger detector sample source, FILTER:

- filterDecimate: Looks for trigger in each ADC conversion, can detect glitches.
- filterAverage: Looks for trigger only in average of N samples, given by *analog_in_frequency_set()*.

analog_in_trigger_filter_set(*trig_filter*)

analog_in_trigger_holdoff_get()

analog_in_trigger_holdoff_info()
Returns the supported range of the trigger Hold-Off time in Seconds. The trigger hold-off is an adjustable period of time during which the acquisition will not trigger. This feature is used when you are triggering on burst waveform shapes, so the oscilloscope triggers only on the first eligible trigger point.

Returns

- **min_holdoff** (*float*)
- **max_holdoff** (*float*)
- **steps** (*float*)

analog_in_trigger_holdoff_set(*holdoff*)

analog_in_trigger_hysteresis_get()

analog_in_trigger_hysteresis_info()
Retrieves the range of valid trigger hysteresis voltage levels for the AnalogIn instrument in Volts. The trigger detector uses two levels: low level (TriggerLevel - Hysteresis) and high level (TriggerLevel + Hysteresis). Trigger hysteresis can be used to filter noise for Edge or Pulse trigger. The low and high levels are used in transition time triggering.

analog_in_trigger_hysteresis_set(*level*)

analog_in_trigger_length_condition_get()

analog_in_trigger_length_condition_hysteresis_get()

analog_in_trigger_length_condition_info()
Returns the supported trigger length condition options for the AnalogIn instrument. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the TRIGLEN constants in DWF.h. These trigger length condition options are:

- triglenLess: Trigger immediately when a shorter pulse or transition time is detected.
- triglenTimeout: Trigger immediately as the pulse length or transition time is reached.
- triglenMore: Trigger when the length/time is reached, and pulse or transition has ended.

Returns

Return type supported trigger length conditions

analog_in_trigger_length_condition_set(*length*)

analog_in_trigger_length_info()
Returns the supported range of trigger length for the instrument in Seconds. The trigger length specifies the minimal or maximal pulse length or transition time.

analog_in_trigger_length_set(*length*)

analog_in_trigger_level_get()

analog_in_trigger_level_info()

analog_in_trigger_level_set(*level*)

analog_in_trigger_position_get()

analog_in_trigger_position_info()
Returns the minimum and maximum values of the trigger position in seconds. For Single/Repeated acquisition mode the horizontal trigger position is used is relative to the buffer middle point. For Record mode the position is relative to the start of the capture.

Todo: The documentation specifies steps as double, but it makes more sense for it to be an integer. Other methods like *analog_in_trigger_auto_timeout_info()* use an integer

---

Returns

- **min_trigger** (*float*)
- **max_trigger** (*float*)
- **steps** (*float*)

analog_in_trigger_position_set(*position*)

analog_in_trigger_source_get()

analog_in_trigger_source_set(*source*)

analog_in_trigger_type_get()

analog_in_trigger_type_set(*trig_type*)

analog_out_count()
The number of analog output channels available on this board.

Returns The number of analog channels available

Return type int

analogin_noise_size_get()
Returns the used AnalogIn instrument noise buffer size. This is automatically adjusted according to the sample buffer size. For instance, having maximum buffer size of 8192 and noise buffer size of 512, setting the sample buffer size to 4096 the noise buffer size will be 256.

Returns Current noise buffer size

Return type int

analong_in_acquisition_mode_set(*mode*)

Parameters **mode** (*AcquisitionMode*) –

digital_out_configure(*status*)

digital_out_count()
Returns the number of Digital Out channels by the device specified by hdwf.

digital_out_counter_get (*channel*)

digital_out_counter_info (*channel*)

digital_out_counter_init_get (*channel*)

digital_out_counter_init_set (*channel*, *start_high*, *divider*)
Sets the initial state and counter value of the specified channel.

digital_out_counter_set (*channel*, *low*, *high*)
Sets the counter low and high values for the specified channel..

digital_out_data_info (*channel*)
Returns the maximum buffers size, the number of custom data bits.

digital_out_data_set (*channel*, *data_array*, *num_bits*)

digital_out_divider_get (*channel*)

digital_out_divider_info (*channel*)

digital_out_divider_init_get (*channel*)

digital_out_divider_init_set (*channel*, *divider*)

digital_out_divider_set (*channel*, *divider*)

digital_out_enable_get (*channel*)

digital_out_enable_set (*channel*, *enable*)

digital_out_idle_get (*channel*)

digital_out_idle_info (*channel*)
Returns the supported idle output types of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using `DigitalOutIdle`:

- DwfDigitalOutIdleInit: Output initial value.
- DwfDigitalOutIdleLow: Low level.
- DwfDigitalOutIdleHigh: High level.
- DwfDigitalOutIdleZet: Three state.

digital_out_idle_set (*channel*, *idle*)

digital_out_internal_clock_info ()

digital_out_output_get (*channel*)

digital_out_output_info (*channel*)
Returns the supported output modes of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the `DigitalOutOutput`:

- DwfDigitalOutOutputPushPull: Default setting.
- DwfDigitalOutOutputOpenDrain: External pull needed.
- DwfDigitalOutOutputOpenSource: External pull needed.
- DwfDigitalOutOutputThreeState: Available with custom and random types.

digital_out_output_set (*channel*, *output*)

digital_out_play_data_set (*bits*, *bits_per_sample*, *count*)

digital_out_play_rate_set (*rate*)

digital_out_repeat_get()

digital_out_repeat_info()

digital_out_repeat_set(*repeat*)

digital_out_repeat_status()

digital_out_repeat_trigger_get()

digital_out_repeat_trigger_set(*trigger*)
Sets the repeat trigger option. To include the trigger in wait-run repeat cycles, set fRepeatTrigger to TRUE. It is disabled by default.

digital_out_reset()

digital_out_run_get()

digital_out_run_info()

digital_out_run_set(*run_len*)

digital_out_run_status()
Reads the remaining run length. It returns data from the last *digital_out_status()* call.

digital_out_status()

digital_out_trigger_slope_get()

digital_out_trigger_slope_set(*slope*)

digital_out_trigger_source_get()

digital_out_trigger_source_set(*source*)

digital_out_type_get(*channel*)

digital_out_type_info(*channel*)
Returns the supported types of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using `DigitalOutType`:

- DwfDigitalOutTypePulse: Frequency = internal frequency/divider/(low + high counter).
- DwfDigitalOutTypeCustom: Sample rate = internal frequency / divider.
- DwfDigitalOutTypeRandom: Random update rate = internal frequency/divider/counter alternating between

low and high values. - DwfDigitalOutTypeROM: ROM logic, the DIO input value is used as address for output value - DwfDigitalOutTypePlay: Supported with Digital Discovery.

digital_out_type_set(*channel*, *out_type*)

digital_out_wait_get()

digital_out_wait_info()
Returns the supported wait length range in seconds. The wait length is how long the instrument waits after being triggered to generate the signal. Default value is zero.

digital_out_wait_set(*wait*)

initialize(*dev_num=-1*)
Initialize the communication with a device identified by its order

Parameters **dev_num** (*int*) – The device number to open, by default it opens the last device

Raises `DriverException` – If the device can't be opened

1.5 experimentor

1.5.1 experimentor package

Subpackages

experimentor.config package

Submodules

experimentor.config.global_settings module

Global Settings

Settings that should be available to any experimentor project. If you are starting a new project, you can use the settings below as an example, and override the ones you think need to be overwritten. Especially things like:

- EXPERIMENT_MODEL
- EXPERIMENT_MODEL_INIT
- START_WINDOW

The only variables that will be considered are those written all in CAPITAL LETTERS.

Module contents

Settings

Experimentor relies on some general settings in order to run. For example, one can specify the port at which the publisher or pusher connects, or the window which is the starting point for the user interface. We specify some global parameres at `experimentor.config.global_settings`, that can be overriden at runtime by specifying the environmental variable *EXPERIMENTOR_SETTINGS_MODULE*.

Only variables written in ALL CAPITAL LETTERS will be taken into account.

Note: The inspiration for this flow comes from **'Django's Settings module<https://github.com/django/django/blob/c574bec0929cd2527268c96a492d25223a9fd576/django/conf/__init__.py>'_**

---

class experimentor.config.**Settings**(*settings_module*)

Bases: object

Loads the global parameters and overrides them with those specified in the settings module of the project.

experimentor.core package

Submodules

experimentor.core.app module

experimentor.core.data_source module

Data Source

These objects are defined in models and are meant to be used to broadcast information across different objects, either on different threads, processes, or computers. In their core, they are ZMQ Publishers and hold the necessary information in order to create a subscriber based on them.

class experimentor.core.data_source.**DataSource**
Bases: object

connect()

finalize()

initialize()

experimentor.core.exceptions module

exception experimentor.core.exceptions.**DuplicatedParameter**
Bases: *experimentor.core.exceptions.ExperimentorException*

exception experimentor.core.exceptions.**ExperimentDefinitionException**
Bases: *experimentor.core.exceptions.ExperimentorException*

exception experimentor.core.exceptions.**ExperimentorException**
Bases: Exception

exception experimentor.core.exceptions.**ModelDefinitionException**
Bases: *experimentor.core.exceptions.ExperimentorException*

experimentor.core.measurement_parameters module

Measurement Classes

Collection of classes useful for developing the logic of a measurement

Section author: Aquiles Carattino

class experimentor.core.measurement_parameters.**Parameter**(*units=None*, *ui_class=None*)
Bases: object

Parameters that belong to a measurement. They allow to define units, limits and ui_classes.

name = ''

experimentor.core.measurement_procedure module

class experimentor.core.measurement_procedure.**Procedure**(*procedure*)
Bases: object

Decorator to check the validity of a procedure before performing a measurement

check_parameters(*cls*, **args*, ***kwargs*)

experimentor.core.meta module

class experimentor.core.meta.**ExperimentorProcess**(**args*, ***kwargs*)
Bases: multiprocessing.context.Process

class experimentor.core.meta.**ExperimentorThread**(**args*, ***kwargs*)
Bases: threading.Thread

class experimentor.core.meta.**MetaProcess**(*name*, *bases*, *attrs*)
Bases: type

Meta Class that should be shared by all processes in order to be sure they all switch off nicely when done.

get_instances(*recursive=False*)
Get all instances of this class in the registry. If recursive=True search subclasses recursively

experimentor.core.publisher module

Publisher

Publishers are responsible for broadcasting the message over the ZMQ PUB/SUB architecture. The publisher runs continuously on a separated process and grabs elements from a queue, which in turn are sent through a socket to any other processes listening.

Todo: In the current implementation, data is serialized for being added to a Queue, then deserialized by the publisher and serialized again to be sent. These three steps could be simplify into one if, for example, one assumes that objects where pickled. There is also a possibility of assuming numpy arrays and using a zero-copy strategy.

---

copyright Aquiles Carattino

license MIT, see LICENSE for more details

class experimentor.core.publisher.**Publisher**(*event*, *name=None*)
Bases: *experimentor.core.meta.ExperimentorProcess*

Publisher class in which the queue for publishing messages is defined and also a separated process is started. It is important to have a new process, since the serialization/deserialization of messages from the QUEUE may be a bottleneck for performance.

run()
Start a new process that will be responsible for broadcasting the messages.

Todo: Find a way to start the publisher on a different port if the one specified is in use.

---

stop()

experimentor.core.publisher.**start_publisher**()
Wrapper function to start the publisher. It takes care of checking that there is only one publisher running by storing it in the settings.

Todo: Find a good way of starting a publisher once per measurement cycle.

---

experimentor.core.pusher module

Pusher

New in version 0.2.0.

Half the ZMQ implementation is abut broadcasting information from a publisher to different subscribers. However, the other half is giving information to the publisher to broadcast. We are doing this with a PUSH/PULL pattern. The pusher is therefore able to send information to the Publisher to then broadcast. There can be many instances of pushers, but only one publisher. In other words, this is a fan-in type of architecture.

class experimentor.core.pusher.**Pusher**(*port=None*)
Bases: object

The Pusher is class that wraps some common methods of the ZMQ PUSH/PULL architecture.

> **Warning:** The main problem with this pattern is that if there is not PULL on the other side, a queue will build up on the PUSH side. This happens if, for example, we close the publisher but we keep generating data. Eventually the queue will outgrow the memory and the computer will crash.

Parameters **port** (*int*) – The port on which to connect the PUSH end. If not specified, it will grab the default value from settings

pusher
The socket where the communication happens

Type socket

i
The number of messages that were pushed from a given

Type int

topic_i
Number of data frames sent on each topic. For example: topic_i['topic']

Type dict

lock
In case the same pusher is shared between different threads, this ensures the messages are sent in the proper block

Type RLOCK

finish()

publish(*data*, *topic=''*)
Publish data on a given topic. This is the core of the Pusher object.

Parameters

- **data** – Data can be any Python object, provided that it is serializable
- **topic** (*str*) – The topic on which the data is being transmitted. If nothing is specified, it will be a broad transmission, meaning that every subscriber will receive it.

experimentor.core.signal module

class experimentor.core.signal.**Signal**
Bases: object

Base signal which implements the common pattern for defining, emitting and connecting a signal

emit(*payload=None*, ***kwargs*)
Emitting a signal relies on the owner's publisher or whatever method the owner specifies for broadcasting. In principle this allows for some flexibility in case owners use different ways of broadcasting information. For example, the owner could be a QObject and it could use the internals of Qt to emitting signals.

url

experimentor.core.subscriber module

Subscriber

Example script on how to run separate processes to process the data coming from a publisher like the one on `publisher.py`. The first process just grabs the frame and puts it in a Queue. The Queue is then used by another process in order to analyse, process, save, etc. It has to be noted that on UNIX systems, getting from a queue with `Queue.get()` is particularly slow, much slower than serializing a numpy array with cPickle.

class experimentor.core.subscriber.**Subscriber**(*func*, *url*, *topic*)
Bases: threading.Thread

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()

experimentor.core.subscriber_process module

Subscriber

Example script on how to run separate processes to process the data coming from a publisher like the one on `publisher.py`. The first process just grabs the frame and puts it in a Queue. The Queue is then used by another process in order to analyse, process, save, etc. It has to be noted that on UNIX systems, getting from a queue with `Queue.get()` is particularly slow, much slower than serializing a numpy array with cPickle.

> **Warning:** This is work in process. On Windows, since processes are spawned, the subscriber would not work as expected. That is why we work with Threads instead.

class experimentor.core.subscriber_process.**Subscriber**(*func*, *topic*, *publish_topic=None*, *args=None*, *kwargs=None*)
Bases: *experimentor.core.meta.ExperimentorProcess*

run()
Method to be run in sub-process; can be overridden in sub-class

stop()

Module contents

experimentor.drivers package

Subpackages

experimentor.drivers.PhotonicScience package

Submodules

experimentor.drivers.PhotonicScience.scmoscam module

UUTrack.Controller.devices.PhotonicScience.scmoscam.py

A wrapper class originally written by Perceval Guillou, perceval@photonic-science.com in Py2 and has been tested successfully with scmoscontrol.dll SCMOS Pleora (GEV) control dll (x86)v5.6.0.0 (date modified 10/2/2013)

SaFa @nanoLINX has adapted the wrapper class for a camera control program.

v1.0, 24 feb. 2015

Section author: SaFa <S.Faez@uu.nl>

class experimentor.drivers.PhotonicScience.scmoscam.**GEVSCMOS**(*cwd_path*, *name*)

Bases: object

AbortSnap()

AutoBinningFilter(*enable*)

Close()

Demangle(*image_pointer*, *Nx*, *Ny*)

EnableAutoLevel(*enable*)

EnableBestFit(*enable*)

EnableBinningFilter(*enable*)

EnableBrightPixel(*enable*)

EnableClip(*enable*)

EnableFlatField(*enable*)

EnableGamma(*enable*)

EnableOffset(*enable*)

EnableRemapping(*enable*)

EnableSharpening(*enable*)

EnableSmooth(*enable*)

EnableStreaming(*enable*)

FreeSequence()

GetDLL()

GetDLLName()

GetImage(*imp=None*)

GetImagePointer()

GetMode()

GetName()

GetOptions()

GetPedestal()

GetRawImage()

GetRemapSize()

GetSequencePointer(*id*)

GetSize()

GetSizeMax()

GetState()

GetStatus()

GetTemperature()

Has8bitGainModes()

HasBinning()

HasClockSpeedLimit()

HasHPMapping()

HasIntensifier()

HasTemperature()

InitFunctions()

InitSequence(*imnum*)

IsFlipped()

IsInCamCor()

IsIntensifier()

LoadCamDLL()

MakeFlatField()

Open()

OpenMap(*file_name='distortion.map'*)

Remap(*image_pointer*, *Nx*, *Ny*)

ResetOptions()

SaveSequence()

SelectIportDevice()

SetALCMaxExp(*maxexp*)

SetALCWin(*l*, *t*, *r*, *b*)

SetBFPeek(*peek*)

SetChipGain(*gain*)

SetClockSpeed(*mode*)

SetExposure(*expo*, *unit*)

SetFlatAverage(*average_number*)

SetFlickerMode(*value*)

SetGainMode(*mode*)

SetGammaBright(*value*)

SetGammaPeak(*value*)

SetIFDelay(*delay*)

SetIntensifierGain(*gain*)

SetPowerSavingMode(*mode*)

SetSoftBin(*Sx*, *Sy*)

SetSubArea(*left*, *top*, *right*, *bottom*)

SetTemperature(*temp*)

SetTrigger(*mode*)

SetVideoGain(*gain*)

Snap()

SnapAndReturn()

SnapSequence()

SoftBinImage(*image_pointer*, *Nx*, *Ny*)

UnloadCamDLL()

UpdateSize()

UpdateSizeMax()

Module contents

UUTrack.Controller.devices.PhotonicScience

company Photonic Science.

experimentor.drivers.hamamatsu package

Submodules

experimentor.drivers.hamamatsu.hamamatsu_camera module

Module contents

UUTrack.Controller.devices.hamamatsu

company Hamamatsu.

experimentor.drivers.keysight package

Submodules

experimentor.drivers.keysight.inifiniivision module

Module contents

UUTrack.Controller.devices.keysight

company Keysight.

experimentor.drivers.santec package

Submodules

experimentor.drivers.santec.tsl710 module

Module contents

experimentor.drivers.thorlabs package

Submodules

experimentor.drivers.thorlabs.data_types module

experimentor.drivers.thorlabs.mabuchi module

experimentor.drivers.thorlabs.stepper_motor module

experimentor.drivers.thorlabs.tdc001 module

Module contents

experimentor.drivers.digilent package

Submodules

Digilent

class experimentor.drivers.digilent.**AnalogDiscovery**
Bases: object

analog_in_acquisition_mode_get()

Returns Current mode

Return type AcquisitionMode

analog_in_acquisition_mode_info()
Returns the supported AnalogIn acquisition modes. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the ACQMODE constants in dwf.h. The acquisition mode selects one of the following modes, ACQMODE:

Returns Bitfield of modes, needs to be parsed

Return type int

analog_in_bits_info()

analog_in_buffer_size_get()

analog_in_buffer_size_info()

analog_in_buffer_size_set(*buffer_size*)

analog_in_channel_attenuation_get(*channel*)

analog_in_channel_attenuation_set(*channel*, *attenuation*)
Configures the attenuation for each channel. When channel index is specified as -1, each enabled AnalogIn channel attenuation will be configured to the same level. The attenuation does not change the attenuation on the device, just informs the library about the externally applied attenuation. :param channel: :type channel: int :param attenuation: :type attenuation: float

analog_in_channel_count()

analog_in_channel_disable(*channel*)
Disables the specified channel. See *analog_in_channel_enable()*

Parameters **channel** (*int*) –

analog_in_channel_enable(*channel*)
Enables the specified channel. See *analog_in_channel_disable()*

Parameters **channel** (*int*) –

analog_in_channel_enable_get(*channel*)

analog_in_channel_filter_get(*channel*)

analog_in_channel_filter_info()

analog_in_channel_filter_set(*channel*, *filter*)

analog_in_channel_offset_get(*channel*)

analog_in_channel_offset_info()

analog_in_channel_offset_set(*channel*, *offset*)

analog_in_channel_range_get(*channel*)

analog_in_channel_range_info()

Returns

- **volts_min** (*float*)
- **volts_max** (*float*)
- **volts_steps** (*float*)

analog_in_channel_range_set(*channel*, *channel_range*)

analog_in_configure(*reconfigure=1*, *start=1*)

analog_in_frequency_get()

analog_in_frequency_info()
Retrieves the minimum and maximum (ADC frequency) settable sample frequency.

Returns

- **min_freq** (*float*) – Minimum allowed frequency
- **max_freq** (*float*) – Maximum allowed frequency

analog_in_frequency_set(*frequency*)

analog_in_noise_size_info()

analog_in_record_length_get()

analog_in_record_length_set(*length*)

analog_in_reset()

analog_in_samples_left()

Retrieves the number of samples left in the acquisition.

Returns Number of samples remaining

Return type int

analog_in_samples_valid()

analog_in_sampling_delay_get()

analog_in_sampling_delay_set(*delay*)

analog_in_sampling_slope_get()

analog_in_sampling_slope_set(*slope*)

Parameters **slope** (*TriggerSlope*) –

analog_in_sampling_source_get()

analog_in_sampling_source_set(*source*)

Parameters **source** (*TriggerSource*) –

analog_in_status(*read_data=0*)
Checks the status of the acquisition

Parameters **read_data** (*int*) – 0 or 1, to indicate whether data should be read from the device

Returns The instrument state

Return type InstrumentState

analog_in_status_auto_trigger()

Verifies if the acquisition is auto triggered.

Returns I guess it returns 1 if the acquisition was auto triggered

Return type int

analog_in_status_data(*channel*, *samples*, *buffer=None*)

Retrieves the acquired data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer.

Parameters

- **channel** (*int*) –
- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type np.array

analog_in_status_data_16(*channel*, *first*, *samples*, *buffer=None*)

Retrieves the acquired raw data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer or creates a new one. This is the **raw** data, as opposed to what *analog_in_status_data()* returns.

Parameters

- **channel** (*int*) –
- **first** (*int*) –
- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type numpy.array

analog_in_status_data_2(*channel*, *first*, *samples*, *buffer=None*)

Retrieves the acquired data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer or creates a new buffer. This method allows to specify which data will be copied. To retrieve all data see *analog_in_status_data()*.

Parameters

- **channel** (*int*) –
- **first** (*int*) –
- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type numpy.array

analog_in_status_index()

Retrieves the buffer write pointer which is needed in ScanScreen acquisition mode to display the scan bar. :returns: Variable to receive the position of the acquisition. :rtype: int

analog_in_status_noise(*channel*, *samples*)
Retrieves the acquired noise samples from the specified idxChannel on the AnalogIn instrument.

Parameters

- **channel** (*int*) –
- **samples** (*int*) –

Returns minimum noise data, maximum noise data

Return type 2-colum numpy.array

analog_in_status_record()
Retrieves information about the recording process. The data loss occurs when the device acquisition is faster than the read process to PC. In this case, the device recording buffer is filled and data samples are overwritten. Corrupt samples indicate that the samples have been overwritten by the acquisition process during the previous read. In this case, try optimizing the loop process for faster execution or reduce the acquisition frequency or record length to be less than or equal to the device buffer size (record length <= buffer size/frequency).

Returns

- **data_available** (*int*) – Available number of samples
- **data_lost** (*int*) – Lost samples after the last check
- **data_corrupt** (*int*) – Number of samples that can be corrupt

analog_in_status_sample(*channel*)
Gets the last ADC conversion sample from the specified idxChannel on the AnalogIn instrument.

Parameters **channel** (*int*) –

Returns Sample value

Return type float

analog_in_trigger_auto_timeout_get()

analog_in_trigger_auto_timeout_info()

analog_in_trigger_auto_timeout_set(*timeout=0*)

analog_in_trigger_channel_get()

analog_in_trigger_channel_info()

analog_in_trigger_channel_set(*channel*)
Sets the trigger channel.

analog_in_trigger_condition_get()

analog_in_trigger_condition_info()
Returns the supported trigger type options for the instrument. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the DwfTriggerSlope constants in dwf.h. These trigger condition options are:

- **DwfTriggerSlopeRise (This is the default setting):**
  - For edge and transition trigger on rising edge.
  - For pulse trigger on positive pulse; For window exiting.
- **DwfTriggerSlopeFall**
  - For edge and transition trigger on falling edge.

  - For pulse trigger on negative pulse; For window entering.
- **DwfTriggerSlopeEither**
  - For edge and transition trigger on either edge.
  - For pulse trigger on either positive or negative pulse.

Returns **info**

Return type int

analog_in_trigger_condition_set(*condition*)

analog_in_trigger_filter_get()

analog_in_trigger_filter_info()
Returns the supported trigger filters. They are returned (by reference) as a bit field which can be parsed using the IsBitSet Macro. Individual bits are defined using the FILTER constants in DWF.h. Select trigger detector sample source, FILTER:

- filterDecimate: Looks for trigger in each ADC conversion, can detect glitches.
- filterAverage: Looks for trigger only in average of N samples, given by *analog_in_frequency_set()*.

analog_in_trigger_filter_set(*trig_filter*)

analog_in_trigger_holdoff_get()

analog_in_trigger_holdoff_info()
Returns the supported range of the trigger Hold-Off time in Seconds. The trigger hold-off is an adjustable period of time during which the acquisition will not trigger. This feature is used when you are triggering on burst waveform shapes, so the oscilloscope triggers only on the first eligible trigger point.

Returns

- **min_holdoff** (*float*)
- **max_holdoff** (*float*)
- **steps** (*float*)

analog_in_trigger_holdoff_set(*holdoff*)

analog_in_trigger_hysteresis_get()

analog_in_trigger_hysteresis_info()
Retrieves the range of valid trigger hysteresis voltage levels for the AnalogIn instrument in Volts. The trigger detector uses two levels: low level (TriggerLevel - Hysteresis) and high level (TriggerLevel + Hysteresis). Trigger hysteresis can be used to filter noise for Edge or Pulse trigger. The low and high levels are used in transition time triggering.

analog_in_trigger_hysteresis_set(*level*)

analog_in_trigger_length_condition_get()

analog_in_trigger_length_condition_hysteresis_get()

analog_in_trigger_length_condition_info()
Returns the supported trigger length condition options for the AnalogIn instrument. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the TRIGLEN constants in DWF.h. These trigger length condition options are:

- triglenLess: Trigger immediately when a shorter pulse or transition time is detected.

- triglenTimeout: Trigger immediately as the pulse length or transition time is reached.
- triglenMore: Trigger when the length/time is reached, and pulse or transition has ended.

Returns

Return type supported trigger length conditions

analog_in_trigger_length_condition_set(*length*)

analog_in_trigger_length_info()
Returns the supported range of trigger length for the instrument in Seconds. The trigger length specifies the minimal or maximal pulse length or transition time.

analog_in_trigger_length_set(*length*)

analog_in_trigger_level_get()

analog_in_trigger_level_info()

analog_in_trigger_level_set(*level*)

analog_in_trigger_position_get()

analog_in_trigger_position_info()
Returns the minimum and maximum values of the trigger position in seconds. For Single/Repeated acquisition mode the horizontal trigger position is used is relative to the buffer middle point. For Record mode the position is relative to the start of the capture.

Todo: The documentation specifies steps as double, but it makes more sense for it to be an integer. Other methods like *analog_in_trigger_auto_timeout_info()* use an integer

---

Returns

- **min_trigger** (*float*)
- **max_trigger** (*float*)
- **steps** (*float*)

analog_in_trigger_position_set(*position*)

analog_in_trigger_source_get()

analog_in_trigger_source_set(*source*)

analog_in_trigger_type_get()

analog_in_trigger_type_set(*trig_type*)

analog_out_count()
The number of analog output channels available on this board.

Returns The number of analog channels available

Return type int

analogin_noise_size_get()
Returns the used AnalogIn instrument noise buffer size. This is automatically adjusted according to the sample buffer size. For instance, having maximum buffer size of 8192 and noise buffer size of 512, setting the sample buffer size to 4096 the noise buffer size will be 256.

Returns Current noise buffer size

Return type int

analong_in_acquisition_mode_set(*mode*)

Parameters **mode** (*AcquisitionMode*) –

digital_out_configure(*status*)

digital_out_count()
Returns the number of Digital Out channels by the device specified by hdwf.

digital_out_counter_get(*channel*)

digital_out_counter_info(*channel*)

digital_out_counter_init_get(*channel*)

digital_out_counter_init_set(*channel*, *start_high*, *divider*)
Sets the initial state and counter value of the specified channel.

digital_out_counter_set(*channel*, *low*, *high*)
Sets the counter low and high values for the specified channel..

digital_out_data_info(*channel*)
Returns the maximum buffers size, the number of custom data bits.

digital_out_data_set(*channel*, *data_array*, *num_bits*)

digital_out_divider_get(*channel*)

digital_out_divider_info(*channel*)

digital_out_divider_init_get(*channel*)

digital_out_divider_init_set(*channel*, *divider*)

digital_out_divider_set(*channel*, *divider*)

digital_out_enable_get(*channel*)

digital_out_enable_set(*channel*, *enable*)

digital_out_idle_get(*channel*)

digital_out_idle_info(*channel*)
Returns the supported idle output types of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using `DigitalOutIdle`:

- DwfDigitalOutIdleInit: Output initial value.
- DwfDigitalOutIdleLow: Low level.
- DwfDigitalOutIdleHigh: High level.
- DwfDigitalOutIdleZet: Three state.

digital_out_idle_set(*channel*, *idle*)

digital_out_internal_clock_info()

digital_out_output_get(*channel*)

digital_out_output_info(*channel*)
Returns the supported output modes of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the `DigitalOutOutput`:

- DwfDigitalOutOutputPushPull: Default setting.
- DwfDigitalOutOutputOpenDrain: External pull needed.

- DwfDigitalOutOutputOpenSource: External pull needed.
- DwfDigitalOutOutputThreeState: Available with custom and random types.

digital_out_output_set(*channel*, *output*)

digital_out_play_data_set(*bits*, *bits_per_sample*, *count*)

digital_out_play_rate_set(*rate*)

digital_out_repeat_get()

digital_out_repeat_info()

digital_out_repeat_set(*repeat*)

digital_out_repeat_status()

digital_out_repeat_trigger_get()

digital_out_repeat_trigger_set(*trigger*)
Sets the repeat trigger option. To include the trigger in wait-run repeat cycles, set fRepeatTrigger to TRUE. It is disabled by default.

digital_out_reset()

digital_out_run_get()

digital_out_run_info()

digital_out_run_set(*run_len*)

digital_out_run_status()
Reads the remaining run length. It returns data from the last *digital_out_status()* call.

digital_out_status()

digital_out_trigger_slope_get()

digital_out_trigger_slope_set(*slope*)

digital_out_trigger_source_get()

digital_out_trigger_source_set(*source*)

digital_out_type_get(*channel*)

digital_out_type_info(*channel*)
Returns the supported types of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using `DigitalOutType`:

- DwfDigitalOutTypePulse: Frequency = internal frequency/divider/(low + high counter).
- DwfDigitalOutTypeCustom: Sample rate = internal frequency / divider.
- DwfDigitalOutTypeRandom: Random update rate = internal frequency/divider/counter alternating between

low and high values. - DwfDigitalOutTypeROM: ROM logic, the DIO input value is used as address for output value - DwfDigitalOutTypePlay: Supported with Digital Discovery.

digital_out_type_set(*channel*, *out_type*)

digital_out_wait_get()

digital_out_wait_info()
Returns the supported wait length range in seconds. The wait length is how long the instrument waits after being triggered to generate the signal. Default value is zero.

digital_out_wait_set(*wait*)

initialize(*dev_num=-1*)
Initialize the communication with a device identified by its order

Parameters **dev_num** (*int*) – The device number to open, by default it opens the last device

Raises DriverException – If the device can't be opened

Module contents

class experimentor.drivers.digilent.**AnalogDiscovery**
Bases: object

analog_in_acquisition_mode_get()

Returns Current mode

Return type AcquisitionMode

analog_in_acquisition_mode_info()
Returns the supported AnalogIn acquisition modes. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the ACQMODE constants in dwf.h. The acquisition mode selects one of the following modes, ACQMODE:

Returns Bitfield of modes, needs to be parsed

Return type int

analog_in_bits_info()

analog_in_buffer_size_get()

analog_in_buffer_size_info()

analog_in_buffer_size_set(*buffer_size*)

analog_in_channel_attenuation_get(*channel*)

analog_in_channel_attenuation_set(*channel*, *attenuation*)
Configures the attenuation for each channel. When channel index is specified as -1, each enabled AnalogIn channel attenuation will be configured to the same level. The attenuation does not change the attenuation on the device, just informs the library about the externally applied attenuation. :param channel: :type channel: int :param attenuation: :type attenuation: float

analog_in_channel_count()

analog_in_channel_disable(*channel*)
Disables the specified channel. See *analog_in_channel_enable()*

Parameters **channel** (*int*) –

analog_in_channel_enable(*channel*)
Enables the specified channel. See *analog_in_channel_disable()*

Parameters **channel** (*int*) –

analog_in_channel_enable_get(*channel*)

analog_in_channel_filter_get(*channel*)

analog_in_channel_filter_info()

analog_in_channel_filter_set(*channel*, *filter*)

analog_in_channel_offset_get(*channel*)

analog_in_channel_offset_info()

analog_in_channel_offset_set(*channel*, *offset*)

analog_in_channel_range_get(*channel*)

analog_in_channel_range_info()

Returns

- **volts_min** (*float*)
- **volts_max** (*float*)
- **volts_steps** (*float*)

analog_in_channel_range_set(*channel*, *channel_range*)

analog_in_configure(*reconfigure=1*, *start=1*)

analog_in_frequency_get()

analog_in_frequency_info()

Retrieves the minimum and maximum (ADC frequency) settable sample frequency.

Returns

- **min_freq** (*float*) – Minimum allowed frequency
- **max_freq** (*float*) – Maximum allowed frequency

analog_in_frequency_set(*frequency*)

analog_in_noise_size_info()

analog_in_record_length_get()

analog_in_record_length_set(*length*)

analog_in_reset()

analog_in_samples_left()

Retrieves the number of samples left in the acquisition.

Returns Number of samples remaining

Return type int

analog_in_samples_valid()

analog_in_sampling_delay_get()

analog_in_sampling_delay_set(*delay*)

analog_in_sampling_slope_get()

analog_in_sampling_slope_set(*slope*)

Parameters **slope** (*TriggerSlope*) –

analog_in_sampling_source_get()

analog_in_sampling_source_set(*source*)

Parameters **source** (*TriggerSource*) –

analog_in_status(*read_data=0*)
Checks the status of the acquisition

Parameters **read_data** (*int*) – 0 or 1, to indicate whether data should be read from the device

Returns The instrument state

Return type InstrumentState

analog_in_status_auto_trigger()

Verifies if the acquisition is auto triggered.

Returns I guess it returns 1 if the acquisition was auto triggered

Return type int

analog_in_status_data(*channel*, *samples*, *buffer=None*)

Retrieves the acquired data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer.

Parameters

- **channel** (*int*) –
- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type np.array

analog_in_status_data_16(*channel*, *first*, *samples*, *buffer=None*)
Retrieves the acquired raw data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer or creates a new one. This is the **raw** data, as opposed to what *analog_in_status_data()* returns.

Parameters

- **channel** (*int*) –
- **first** (*int*) –
- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type numpy.array

analog_in_status_data_2(*channel*, *first*, *samples*, *buffer=None*)

Retrieves the acquired data samples from the specified idxChannel on the AnalogIn instrument. It copies the data samples to the provided buffer or creates a new buffer. This method allows to specify which data will be copied. To retrieve all data see *analog_in_status_data()*.

Parameters

- **channel** (*int*) –
- **first** (*int*) –

- **samples** (*int*) –
- **buffer** (*c_double array, optional*) –

Returns Array with the data

Return type numpy.array

analog_in_status_index()
Retrieves the buffer write pointer which is needed in ScanScreen acquisition mode to display the scan bar. :returns: Variable to receive the position of the acquisition. :rtype: int

analog_in_status_noise(*channel*, *samples*)
Retrieves the acquired noise samples from the specified idxChannel on the AnalogIn instrument.

Parameters

- **channel** (*int*) –
- **samples** (*int*) –

Returns minimum noise data, maximum noise data

Return type 2-colum numpy.array

analog_in_status_record()
Retrieves information about the recording process. The data loss occurs when the device acquisition is faster than the read process to PC. In this case, the device recording buffer is filled and data samples are overwritten. Corrupt samples indicate that the samples have been overwritten by the acquisition process during the previous read. In this case, try optimizing the loop process for faster execution or reduce the acquisition frequency or record length to be less than or equal to the device buffer size (record length <= buffer size/frequency).

Returns

- **data_available** (*int*) – Available number of samples
- **data_lost** (*int*) – Lost samples after the last check
- **data_corrupt** (*int*) – Number of samples that can be corrupt

analog_in_status_sample(*channel*)
Gets the last ADC conversion sample from the specified idxChannel on the AnalogIn instrument.

Parameters **channel** (*int*) –

Returns Sample value

Return type float

analog_in_trigger_auto_timeout_get()

analog_in_trigger_auto_timeout_info()

analog_in_trigger_auto_timeout_set(*timeout=0*)

analog_in_trigger_channel_get()

analog_in_trigger_channel_info()

analog_in_trigger_channel_set(*channel*)
Sets the trigger channel.

analog_in_trigger_condition_get()

analog_in_trigger_condition_info()
Returns the supported trigger type options for the instrument. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the DwfTriggerSlope constants in dwf.h. These trigger condition options are:

- **DwfTriggerSlopeRise (This is the default setting):**
  - For edge and transition trigger on rising edge.
  - For pulse trigger on positive pulse; For window exiting.
- **DwfTriggerSlopeFall**
  - For edge and transition trigger on falling edge.
  - For pulse trigger on negative pulse; For window entering.
- **DwfTriggerSlopeEither**
  - For edge and transition trigger on either edge.
  - For pulse trigger on either positive or negative pulse.

Returns **info**

Return type int

analog_in_trigger_condition_set(*condition*)

analog_in_trigger_filter_get()

analog_in_trigger_filter_info()
Returns the supported trigger filters. They are returned (by reference) as a bit field which can be parsed using the IsBitSet Macro. Individual bits are defined using the FILTER constants in DWF.h. Select trigger detector sample source, FILTER:

- filterDecimate: Looks for trigger in each ADC conversion, can detect glitches.
- filterAverage: Looks for trigger only in average of N samples, given by *analog_in_frequency_set()*.

analog_in_trigger_filter_set(*trig_filter*)

analog_in_trigger_holdoff_get()

analog_in_trigger_holdoff_info()
Returns the supported range of the trigger Hold-Off time in Seconds. The trigger hold-off is an adjustable period of time during which the acquisition will not trigger. This feature is used when you are triggering on burst waveform shapes, so the oscilloscope triggers only on the first eligible trigger point.

Returns

- **min_holdoff** (*float*)
- **max_holdoff** (*float*)
- **steps** (*float*)

analog_in_trigger_holdoff_set(*holdoff*)

analog_in_trigger_hysteresis_get()

analog_in_trigger_hysteresis_info()
Retrieves the range of valid trigger hysteresis voltage levels for the AnalogIn instrument in Volts. The trigger detector uses two levels: low level (TriggerLevel - Hysteresis) and high level (TriggerLevel +

Hysteresis). Trigger hysteresis can be used to filter noise for Edge or Pulse trigger. The low and high levels are used in transition time triggering.

analog_in_trigger_hysteresis_set(*level*)

analog_in_trigger_length_condition_get()

analog_in_trigger_length_condition_hysteresis_get()

analog_in_trigger_length_condition_info()
Returns the supported trigger length condition options for the AnalogIn instrument. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the TRIGLEN constants in DWF.h. These trigger length condition options are:

- triglenLess: Trigger immediately when a shorter pulse or transition time is detected.
- triglenTimeout: Trigger immediately as the pulse length or transition time is reached.
- triglenMore: Trigger when the length/time is reached, and pulse or transition has ended.

Returns

Return type supported trigger length conditions

analog_in_trigger_length_condition_set(*length*)

analog_in_trigger_length_info()
Returns the supported range of trigger length for the instrument in Seconds. The trigger length specifies the minimal or maximal pulse length or transition time.

analog_in_trigger_length_set(*length*)

analog_in_trigger_level_get()

analog_in_trigger_level_info()

analog_in_trigger_level_set(*level*)

analog_in_trigger_position_get()

analog_in_trigger_position_info()
Returns the minimum and maximum values of the trigger position in seconds. For Single/Repeated acquisition mode the horizontal trigger position is used is relative to the buffer middle point. For Record mode the position is relative to the start of the capture.

Todo: The documentation specifies steps as double, but it makes more sense for it to be an integer. Other methods like *analog_in_trigger_auto_timeout_info()* use an integer

---

Returns

- **min_trigger** (*float*)
- **max_trigger** (*float*)
- **steps** (*float*)

analog_in_trigger_position_set(*position*)

analog_in_trigger_source_get()

analog_in_trigger_source_set(*source*)

analog_in_trigger_type_get()

analog_in_trigger_type_set(*trig_type*)

analog_out_count()
The number of analog output channels available on this board.

Returns The number of analog channels available

Return type int

analogin_noise_size_get()
Returns the used AnalogIn instrument noise buffer size. This is automatically adjusted according to the sample buffer size. For instance, having maximum buffer size of 8192 and noise buffer size of 512, setting the sample buffer size to 4096 the noise buffer size will be 256.

Returns Current noise buffer size

Return type int

analong_in_acquisition_mode_set(*mode*)

Parameters **mode** (*AcquisitionMode*) –

digital_out_configure(*status*)

digital_out_count()
Returns the number of Digital Out channels by the device specified by hdwf.

digital_out_counter_get(*channel*)

digital_out_counter_info(*channel*)

digital_out_counter_init_get(*channel*)

digital_out_counter_init_set(*channel*, *start_high*, *divider*)
Sets the initial state and counter value of the specified channel.

digital_out_counter_set(*channel*, *low*, *high*)
Sets the counter low and high values for the specified channel..

digital_out_data_info(*channel*)
Returns the maximum buffers size, the number of custom data bits.

digital_out_data_set(*channel*, *data_array*, *num_bits*)

digital_out_divider_get(*channel*)

digital_out_divider_info(*channel*)

digital_out_divider_init_get(*channel*)

digital_out_divider_init_set(*channel*, *divider*)

digital_out_divider_set(*channel*, *divider*)

digital_out_enable_get(*channel*)

digital_out_enable_set(*channel*, *enable*)

digital_out_idle_get(*channel*)

digital_out_idle_info(*channel*)
Returns the supported idle output types of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using `DigitalOutIdle`:

- DwfDigitalOutIdleInit: Output initial value.
- DwfDigitalOutIdleLow: Low level.

- DwfDigitalOutIdleHigh: High level.
- DwfDigitalOutIdleZet: Three state.

digital_out_idle_set(*channel*, *idle*)

digital_out_internal_clock_info()

digital_out_output_get(*channel*)

digital_out_output_info(*channel*)
Returns the supported output modes of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using the `DigitalOutOutput`:

- DwfDigitalOutOutputPushPull: Default setting.
- DwfDigitalOutOutputOpenDrain: External pull needed.
- DwfDigitalOutOutputOpenSource: External pull needed.
- DwfDigitalOutOutputThreeState: Available with custom and random types.

digital_out_output_set(*channel*, *output*)

digital_out_play_data_set(*bits*, *bits_per_sample*, *count*)

digital_out_play_rate_set(*rate*)

digital_out_repeat_get()

digital_out_repeat_info()

digital_out_repeat_set(*repeat*)

digital_out_repeat_status()

digital_out_repeat_trigger_get()

digital_out_repeat_trigger_set(*trigger*)
Sets the repeat trigger option. To include the trigger in wait-run repeat cycles, set fRepeatTrigger to TRUE. It is disabled by default.

digital_out_reset()

digital_out_run_get()

digital_out_run_info()

digital_out_run_set(*run_len*)

digital_out_run_status()
Reads the remaining run length. It returns data from the last *digital_out_status()* call.

digital_out_status()

digital_out_trigger_slope_get()

digital_out_trigger_slope_set(*slope*)

digital_out_trigger_source_get()

digital_out_trigger_source_set(*source*)

digital_out_type_get(*channel*)

digital_out_type_info(*channel*)
Returns the supported types of the channel. They are returned (by reference) as a bit field. This bit field can be parsed using the IsBitSet Macro. Individual bits are defined using `DigitalOutType`:

- DwfDigitalOutTypePulse: Frequency = internal frequency/divider/(low + high counter).
- DwfDigitalOutTypeCustom: Sample rate = internal frequency / divider.
- DwfDigitalOutTypeRandom: Random update rate = internal frequency/divider/counter alternating between

low and high values. - DwfDigitalOutTypeROM: ROM logic, the DIO input value is used as address for output value - DwfDigitalOutTypePlay: Supported with Digital Discovery.

digital_out_type_set(*channel*, *out_type*)

digital_out_wait_get()

digital_out_wait_info()
Returns the supported wait length range in seconds. The wait length is how long the instrument waits after being triggered to generate the signal. Default value is zero.

digital_out_wait_set(*wait*)

initialize(*dev_num=-1*)
Initialize the communication with a device identified by its order

Parameters **dev_num** (*int*) – The device number to open, by default it opens the last device

Raises DriverException – If the device can't be opened

Module contents

experimentor.lib package

Submodules

experimentor.lib.actuator module

actuator.py

Actuators are all the devices able to modify the experiment. For example a piezo stage is an actuator. The properties of the actuators are read-only; in principle one cannot change the port at which a specific sensor is plugged without re-generating the object. The actuator has a property called value, that can be accessed directly like so:

```
`python prop = {'name':  'Actuator 1'} a = Actuator(prop) a.value =
Q_('1nm') print(a.value) `
```

Bear in mind that setting the value of an actuator triggers a communication with a real device. You have to be careful if there is something connected to it.

class experimentor.lib.actuator.**Actuator**(*properties*)
Bases: object

device

make_ramp(*ramp_properties*)
Sets the actuator to make a ramp if it is in its capabilities. Properties established all the properties that are needed for the ramp.

properties

value
The value of the device.

experimentor.lib.device module

device.py

Devices are connected to the computer. They control sensors and actuators. A device has to be able to set and read values. Setting complex devices such as a laser would require to define it as a device and its properties as sensors or actuators respectively.

Warning: If problems arise when adding new devices, tt is important to check :meth:initialize_driver . It was hardcoded which parameters are passed when initializing each device type.

Todo: Make flexible parameters when initializing the driver of the devices.

Section author: Aquiles Carattino

class experimentor.lib.device.**Device**(*properties*)

Bases: object

Device is responsible for the communication with real devices. Device takes only one argument, a dictionary of properties, including the driver. Device has two properties, one called _properties that stores the initial properties passed to the device and is read-only. _params stores the parameters passed during execution; it doesn't store a history, just the latest one.

add_driver(*driver*)

Adds the driver of the device. It has to be initialized() :param driver: driver of any class. :return: Null

apply_value(*actuator*, *value*)

Applies a given value to an actuator through the driver of the device. It is only a relay function left here to keep the hierarchical structure of the program, i.e. actuators communicate with devices, devices communicate with models and models with drivers.

Parameters

- **actuator** – instance of Actuator
- **value** – A value to be set. Ideally a Quantity.

apply_values(*values*)

Iterates over all values of a dictionary and sets the values of the driver to it. It is kept for legacy support but it is very important to switch to apply_value, passing an actuator.

Warning: This method can misbehave with the new standards of sensors and actuators in place since version 0.1.

Parameters **values** – a dictionary of parameters and desired values for those parameters. The parameters should have units.

initialize_driver()

Initializes the driver. There are 4 types of possible connections:

- GPIB
- USB
- serial

- daq

The first 3 are based on Lantz and its initialization routine, while daq was inherited from previous code and has a different initialization routine.

params

properties

read_value(*sensor*)
Reads a value from a sensor. This method is just a relay to a model, in order to keep the structure of the program tidy.

experimentor.lib.fitgaussian module

experimentor.lib.fitgaussian.**fitgaussian**(*data*)
Returns (height, x, y, width_x, width_y) the gaussian parameters of a 2D distribution found by a fit

experimentor.lib.fitgaussian.**gaussian**(*height*, *center_x*, *center_y*, *width_x*, *width_y*)
Returns a gaussian function with the given parameters

experimentor.lib.fitgaussian.**moments**(*data*)
Returns (height, x, y, width_x, width_y) the gaussian parameters of a 2D distribution by calculating its moments

experimentor.lib.log module

Logging Options

Standardizing logging options for experimentor.

copyright Aquiles Carattino

license MIT, see LICENSE for more details

experimentor.lib.log.**get_logger**(*name='experimentor'*, *level=10*)

experimentor.lib.log.**get_mp_logger**(*level=10*)

experimentor.lib.log.**log_to_file**(*filename*, *level=20*, *fmt=None*)

experimentor.lib.log.**log_to_screen**(*logger*, *level=20*, *fmt=None*)

experimentor.lib.recursive_attributes module

Functions to get and set attributes of nested objects. These functions allow to do things like:

```
>>> rgetattr(obj, 'sub1.sub2.attr')
```

Taken from: https://stackoverflow.com/a/31174427/4467480

experimentor.lib.recursive_attributes.**rgetattr**(*obj*, *attr*, **args*)
Recursive get attribute of objects.

experimentor.lib.recursive_attributes.**rsetattr**(*obj*, *attr*, *val*)
Iteratively gets attributes of objects until the last level and then sets its value.

experimentor.lib.sensor module

Sensor

Sensors are all the devices able to get a value from the experiment. For example a thermocouple is a sensor. The properties of the sensor are read-only; in principle one cannot change the port at which a specific sensor is plugged without re-generating the object.

Section author: Aquiles Carattino

class experimentor.lib.sensor.**Sensor**(*properties*)

Bases: object

add_device(*device*)

Adds the driver to the current sensor. In this context a driver is a class able to read the value from the device.

properties

value

Module contents

experimentor.models package

Subpackages

experimentor.models.daq package

Module contents

experimentor.models.devices package

Subpackages

experimentor.models.devices.cameras package

Subpackages

experimentor.models.devices.cameras.basler package

Submodules

experimentor.models.devices.cameras.basler.basler module

class experimentor.models.devices.cameras.basler.basler.**BaslerCamera**(*camera*, *initial_config=None*)

Bases: *experimentor.models.devices.cameras.base_camera.BaseCamera*

ROI

acquisition_mode

auto_exposure
Off, Once, Continuous

Type Auto exposure can take one of three values

auto_gain
Off, Once, Continuous

Type Auto Gain must be one of three values

binning_x

binning_y

buffer_size

ccd_height

ccd_width

continuous_reads()

exposure
The exposure of the camera, defined in units of time

finalize()
Finalizes the model. It only takes care of closing the publisher. Child classes should implement their own finalize methods (they get called automatically), and either close the publisher explicitly or use this method.

frame_rate

gain
Gain is a float

height

initialize
Decorator for methods in models. Actions are useful when working with methods that run once, and are normally associated with pressing of a button. Actions are multi-threaded by default, using a single executor that returns a future.

Even though Actions (intended as the method in a model) can take arguments, it may be a better approach to store the parameters as attributes before triggering an action. In this way, triggering an action would be equivalent to pressing a button. In the same way, actions can store return values as attribute in the model itself, avoiding the need to keep track of the future returned by the action. Be aware of potential racing conditions that may arise when using shared memory to exchange information.

Todo: Define a clear protocol for exchanging information with models. Should it be state-based (i.e. storing parameters as attributes in the class) or statement based (i.e. passing parameters as arguments of methods).

---

new_image
Base signal which implements the common pattern for defining, emitting and connecting a signal

pixel_format
Pixel format must be one of Mono8, Mono12, Mono12p

read_camera() → list
Reads the camera and stores the image in the temp_image attribute

start_free_run()
Starts a free run from the camera. It will preserve only the latest image. It depends on how quickly the experiment reads from the camera whether all the images will be available or only some.

stop_camera
Decorator for methods in models. Actions are useful when working with methods that run once, and are normally associated with pressing of a button. Actions are multi-threaded by default, using a single executor that returns a future.

Even though Actions (intended as the method in a model) can take arguments, it may be a better approach to store the parameters as attributes before triggering an action. In this way, triggering an action would be equivalent to pressing a button. In the same way, actions can store return values as attribute in the model itself, avoiding the need to keep track of the future returned by the action. Be aware of potential racing conditions that may arise when using shared memory to exchange information.

Todo: Define a clear protocol for exchanging information with models. Should it be state-based (i.e. storing parameters as attributes in the class) or statement based (i.e. passing parameters as arguments of methods).

---

stop_continuous_reads()

stop_free_run
Decorator for methods in models. Actions are useful when working with methods that run once, and are normally associated with pressing of a button. Actions are multi-threaded by default, using a single executor that returns a future.

Even though Actions (intended as the method in a model) can take arguments, it may be a better approach to store the parameters as attributes before triggering an action. In this way, triggering an action would be equivalent to pressing a button. In the same way, actions can store return values as attribute in the model itself, avoiding the need to keep track of the future returned by the action. Be aware of potential racing conditions that may arise when using shared memory to exchange information.

Todo: Define a clear protocol for exchanging information with models. Should it be state-based (i.e. storing parameters as attributes in the class) or statement based (i.e. passing parameters as arguments of methods).

---

trigger_camera()
Triggers the camera.

width

Module contents

Submodules

experimentor.models.devices.cameras.base_camera module

Base Camera Model

Camera class with the base methods. Having a base class exposes the general API for working with cameras. This file is important to keep track of the methods which are exposed to the View. The class BaseCamera should be subclassed

when developing new Models for other cameras. This ensures that all the methods are automatically inherited and there are no breaks downstream.

Conventions

Images are 0-indexed. Therefore, a camera with (1024px X 1024px) will be used as img[0:1024, 0:1024] (remember Python leaves out the last value in the slice.

Region of Interest is specified with the coordinates of the corners. A full-frame with the example above would be given by X=[0,1023], Y=[0,1023]. Be careful, since the maximum width (or height) of the camera is 1024.

The camera keeps track of the coordinates of the initial pixel. For full-frame, this will always be [0,0]. When croping, the corner-pixel will change. It is very important to keep track of this value when building a GUI, since after the first crop, if the user wants to crop even further, the information has to be referenced to the already cropped area.

Notes

IMPORTANT Whatever new function is implemented in a specific model, it should be first declared in the BaseCamera class. In this way the other models will have access to the method and the program will keep running (perhaps with the wrong behavior though).

class experimentor.models.devices.cameras.base_camera.**BaseCamera**(*camera*, *initial_config=None*)

Bases: *experimentor.models.devices.base_device.ModelDevice*

Base Camera model. All camera models should inherit from this model in order to extend functionality. There are some assumptions regarding how to update different settings such as exposure, gain, region of interest.

Parameters **camera** (*str or int*) – Parameter to identify the camera when loading or initializing it.

AQUISITION_MODE
Different acquisition modes: Continuous, Single, Keep last.

Type dict

cam_num
This parameter will be used to identify the camera when loading or initializing it.

Type str or int

running
Whether the camera is running or not

Type bool

max_width
Maximum width, in pixels

Type int

max_height
Maximum height, in pixels

Type int

data_type
The data type of the images generated by the camera. This can be used to allocate the correct amount of memory in buffers, or to reduce data before displaying it. For example, `np.uint16`.

Type np data type

temp_image
It stores the last image acquired by the camera. Useful for user interfaces that need to display images at a rate different than the acquisition rate.

Type np.array

ACQUISITION_MODE = {0: 'Single', 1: 'Continuous', 2: 'Keep Last'}

MODE_CONTINUOUS = 1

MODE_LAST = 2

MODE_SINGLE_SHOT = 0

ROI
Sets up the ROI. Not all cameras are 0-indexed, so this is an important place to define the proper ROI.

vals [list or tuple] Organized as (X, Y), where the coordinates for the ROI would be X[0], X[1], Y[0], Y[1]

acquisition_mode
Single or continuous. :param int mode: One of self.MODE_CONTINUOUS, self.MODE_SINGLE_SHOT

Type Set the readout mode of the camera

acquisition_ready()
Checks if the acquisition in the camera is over.

binning
The binning of the camera if supported. Has to check if binning in X/Y can be different or not, etc.

The binning is specified as a list or tuple like: [X, Y], with the information of the binning in the X or Y direction.

camera = 'Base Camera Model'

ccd_height
Returns the CCD height in pixels this is equivalent to the *max_height*

ccd_width
Returns the CCD width in pixels this is equivalent to the *max_width*

clear_ROI()
Clears the ROI by setting it to the maximum available area.

clear_binning()
Clears the binning of the camera to its default value.

configure(*properties: dict*)
Configure the camera based on a dictionary of properties.

Deprecated since version 0.3.0: By implementing features, this method is no longer required

exposure
Sets the exposure of the camera.

gain
Sets the gain on the camera, if possible

gain [float] The gain, depending on the camera it can be an integer, it can be specified in dB, etc.

initialize()
Initializes the camera.

read_camera()
Reads the camera and stores the image in the temp_image attribute

serial_number
Returns the serial number of the camera, or a way of identifying the camera in an experiment.

stop_acquisition()
Stops the acquisition without closing the connection to the camera.

stop_camera()
Stops the acquisition and closes the connection with the camera.

trigger_camera()
Triggers the camera.

experimentor.models.devices.cameras.exceptions module

exception experimentor.models.devices.cameras.exceptions.**CameraException**
Bases: *experimentor.models.devices.exceptions.DeviceException*

exception experimentor.models.devices.cameras.exceptions.**CameraNotFound**
Bases: *experimentor.models.devices.cameras.exceptions.CameraException*

exception experimentor.models.devices.cameras.exceptions.**CameraTimeout**
Bases: *experimentor.models.devices.cameras.exceptions.CameraException*

exception experimentor.models.devices.cameras.exceptions.**WrongCameraState**
Bases: *experimentor.models.devices.cameras.exceptions.CameraException*

Module contents

Submodules

experimentor.models.devices.base_device module

class experimentor.models.devices.base_device.**ModelDevice**
Bases: *experimentor.models.models.BaseModel*

All models for devices should inherit from this class.

experimentor.models.devices.exceptions module

exception experimentor.models.devices.exceptions.**DeviceException**
Bases: Exception

experimentor.models.devices.meta module

class experimentor.models.devices.meta.**MetaDevice**(*name*, *bases*, *attrs*)
Bases: *experimentor.models.meta.MetaModel*

This is a Meta Class that should be used only by devices and not by the experiment itself. It is only to give more granularity to the program when wanting to perform operations on all the devices or on different possible measurements.

Module contents

experimentor.models.experiments package

Submodules

experimentor.models.experiments.base_experiment module

Base Experiment Model

Base class for the experiments. `BaseExperiment` defines the common patterns that every experiment should have. Importantly, it starts an independent process called publisher, that will be responsible for broadcasting messages that are appended to a queue. The messages rely on the pyZMQ library and should be tested further in order to assess their limitations. The general pattern is that of the PUB/SUB, with one publisher and several subscribers.

The messages should include a *topic* and data. For this, the elements in the queue should be dictionaries with two keywords: **data** and **topic**. `data['data']` will be serialized through the use of cPickle, and is handled automatically by pyZQM through the use of `send_pyobj`. The subscribers should be aware of this and use either unpickle or `recv_pyobj`.

In order to stop the publisher process, the string `'stop'` should be placed in `data['data']`. The message will be broadcast and can be used to stop other processes, such as subscribers.

Todo: Check whether the serialization of objects with cPickle may be a bottleneck for performance.

---

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.experiments.base_experiment.**BaseExperiment**
Bases: *experimentor.models.models.BaseModel*

class experimentor.models.experiments.base_experiment.**Experiment**(*filename=None*)
Bases: *experimentor.models.experiments.base_experiment.BaseExperiment*

Base class to define experiments. Should keep track of the basic methods needed regardless of the experiment to be performed. For instance, a way to start and a way to finalize a measurement. This class is not meant to be instantiated directly, but should be subclassed in each project.

Parameters **filename** (*str or None*) – Path to the config file that will be loaded. Ideally it should be an absolute path, to prevent problems. If you submit a relative path, it will depend on how you are running the program if the file will be found or not.

config
Properties object to store the values of the parameters of the experiments. See *experimentor.models.properties* to understand the options and how it works

Type *Properties*

logger
The logger of the experiment, this is for internal use only

Type logger

alive_threads

connect(*method*, *topic*, **args*, ***kwargs*)
Async method that connects the running publisher to the given method on a specific topic.

Parameters

- **method** – method that will be connected on a given topic
- **topic** (*str*) – the topic that will be used by the subscriber to discriminate what information to collect.
- **args** – extra arguments will be passed to the subscriber, which in turn will pass them to the function
- **kwargs** – extra keyword arguments will be passed to the subscriber, which in turn will pass them to the function

connections

finalize()
Needs to be overridden by child classes.

list_alive_threads

load_configuration(*filename*, *loader=<class 'yaml.loader.SafeLoader'>*)
Loads the configuration file in YAML format.

Parameters **filename** (*str*) – full path to where the configuration file is located.

Raises **FileNotFoundError** – if the file does not exist.

static make_filename(*folder: Union[str, tuple]*, *filename: str*)
This routine will check if the folder to store data exists, and create it if not. It will also check if the file exists, if it does, it will increase by 1 a counter until an available name appears, and return both the directory and the filename.

Parameters

- **filename** – if it contains a '{i}' or similar in its specification, it will use it as a counter, if not, the number will be prepended to the filename
- **folder** – either a string with the full path to the folder (bear in mind differences of folder separators) or a tuple that will be joined using os.path.join

num_threads

set_up()
Needs to be overridden by child classes.

start
Base signal which implements the common pattern for defining, emitting and connecting a signal

stop_subscribers()
Puts the proper data into every alive subscriber in order to stop it.

update_config(***kwargs*)

class experimentor.models.experiments.base_experiment.**FormatDict**
Bases: dict

Simple solution to do partial formatting of strings. For example:

```
>>> a = 'fiber_end_{cartridge}_{i:04}.npy'
>>> cartridge = 123
>>> a.format_map(FormatDict(cartridge=cartridge))
'fiber_end_123_{i:04}.npy'
```

class experimentor.models.experiments.base_experiment.**FormatPlaceholder**(*key*)
Bases: object

class experimentor.models.experiments.base_experiment.**MetaExperiment**(*name*, *bases*, *attrs*)

Bases: *experimentor.models.meta.MetaModel*

Meta Model type which will be responsible for keeping track of all the created experiments. It will also be responsible for registering the publisher, in order to have only one throughout the program and accessible from other parts of the program. This meta class may be overkill, since in principle every program will be only one experiment, but this is left as an effort to be future-proof.

Note: Defining meta classes may generate a feeling of obscurantism in the code. It may be wise to remove it and find a simpler/straightforward approach.

Module contents

experimentor.models.laser package

Module contents

experimentor.models.procedures package

Submodules

experimentor.models.procedures.procedure module

Module contents

Submodules

experimentor.models.action module

Action

An action is an event that gets triggered on a device. For example, a camera can have an action `acquire` or `read`. They should normally be associated with the pressing of a button. Action is a handy decorator to register methods on a model and have quick access to them when building a user interface. They are multi-threaded by default, however, they share the same executor, defined at the model-level. Therefore, if a device is able to run several actions simultaneously, different executors can be defined at the moment of Action instantiation.

To extend Actions, the best is to sub class it and re implement the `get_executor` method, or any other method relevant to change the expected behavior.

Examples

A general purpose model can implement two methods: `initialize` and `auto_calibrate`, we can use the Actions to increment their usability:

```
class TestModel:
    @Action
    def initialize(self):
        print('Initializing')

    @Action
    def auto_calibrate(self):
        print('Auto Calibrating')

tm = TestModel()
tm.initialize()
tm.auto_calibrate()
print(tm.get_actions())
```

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.action.**Action**(*method=None*, ***kwargs*)
Bases: object

Decorator for methods in models. Actions are useful when working with methods that run once, and are normally associated with pressing of a button. Actions are multi-threaded by default, using a single executor that returns a future.

Even though Actions (intended as the method in a model) can take arguments, it may be a better approach to store the parameters as attributes before triggering an action. In this way, triggering an action would be equivalent to pressing a button. In the same way, actions can store return values as attribute in the model itself, avoiding the need to keep track of the future returned by the action. Be aware of potential racing conditions that may arise when using shared memory to exchange information.

Todo: Define a clear protocol for exchanging information with models. Should it be state-based (i.e. storing parameters as attributes in the class) or statement based (i.e. passing parameters as arguments of methods).

---

get_executor()
Gets the executor either explicitly defined as an argument when instantiating the Action, or grabs it from the parent instance, and thus is shared between all action in a model.

To change the behavior, subclass Action and overwrite this method.

get_lock()
Gets the lock specified in the keyword arguments while creating the Action, or defaults to the lock stored in the instance and thus shared between all actions in the model.

Deprecated since version 0.3.0: Since v0.3.0 we are favoring concurrent.futures instead of lower-level threading for Actions.

get_run()
Generates the run function that will be applied to the method. It looks a big convoluted, but it is one of the best approaches to make it easy to extend the Actions in the longer run. The return callable grabs the executor from the method `self.get_executor()`.

Returns A function that takes two arguments: method and instance and that submits them to an executor

Return type callable

set_action(*method*)
Wrapper that returns this own class but initializes it with a method and a previously stored dict of kwargs. This method is what happens when the Action itself is defined with arguments.

Parameters **method** (*callable*) – The method that is decorated by the Action

Returns Returns an instance of the Action using the previously stored kwargs but adding the method

Return type *Action*

experimentor.models.decorators module

Decorators

Useful decorators for models.

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

experimentor.models.decorators.**avoid_repeat**(*func*)

experimentor.models.decorators.**make_async_thread**(*func*)
Simple decorator to make a method run on a separated thread. This decorator will not work on simple functions, since it requires the first argument to be an instantiated class (self). It will store the method in an attribute of the class, called *_threads*', or it will create it if it does not exist yet.

TODO: Check what happens with the _thread list and inherited classes. Is there a risk that the list will be shared? If the list is defined as a class attribute instead of an object attribute, most likely it will. If it is defined outside of the scope and then linked to the class, also.

Warning: In complex scenarios, this simple decorator can give raise to mistakes, i.e. objects having access to other objects threads.

experimentor.models.decorators.**not_implemented**(*func*)
Raises a warning in the logger in case the method was not implemented by child classes, but it does not prevent the program from running.

experimentor.models.exceptions module

Model Exceptions

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

exception experimentor.models.exceptions.**ExperimentorException**
Bases: Exception

Base exception for all experimentor modules

exception experimentor.models.exceptions.**LinkException**
Bases: *experimentor.models.exceptions.PropertyException*

exception experimentor.models.exceptions.**ModelException**
Bases: *experimentor.models.exceptions.ExperimentorException*

exception experimentor.models.exceptions.**PropertyException**
Bases: *experimentor.models.exceptions.ModelException*

exception experimentor.models.exceptions.**SignalException**
Bases: *experimentor.models.exceptions.ExperimentorException*

experimentor.models.feature module

Features

Features in a model are those parameters that can be read, set, or both. They were modeled after Lantz Feat objects, and the idea is that they can encapsulate common patterns in device control. They are similar to `Settings` in behavior, except for the absence of a cache. Features do communicate with the device when reading a value.

For example, a feature could be the value of an analog input on a DAQ, or the temperature of a camera. They are meant to be part of a measurement, their values can change in loops in order to make a scan. Features can be used as decorators in pretty much the same way @propery can be used. The only difference is that they register themselves in the models properties object, so it is possible to update values either by submitting a value directly to the Feature or by sending a dictionary to the properties and updating all at once.

It is possible to mark a feature as a setting. In this case, the value will not be read from the device, but it will be cached. In case it is needed to refresh a value from the device, it is possible to use a specific argument, such as `None`. For example:

```
@Feature(setting=True, force_update_arg=0)
def exposure(self):
    self.driver.get_exposure()

@exposure.setter
def exposure(self, exposure_time):
    self.driver.set_exposure(exposure_time)
```

Todo: It is possible to define complex behavior such as unit conversion, limit checking, etc. We should narrow down what is appropriate for a model and what should go into the Controller.

Todo: A useful pattern is to catch the exception raised by the controllers if a value is out of range, or with the wrong units.

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.feature.**Feature**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*, ***kwargs*)
Bases: `object`

Properties that belong to models. It makes easier the setting and getting of attributes, while at the same time it keeps track of the properties of each model. A Feature is, fundamentally, a descriptor, that extends some functionality by accepting keyword arguments when defining.

Todo: There is a lot of functionality that can be implemented, but that hasn't yet, such as checking limits, unit conversion, etc.

name
The name of the feature, it must be unique since it will be used as a key in a dictionary.

Type str

kwargs
If the feature is initialized with arguments, they will be stored here. Only keyword arguments are allowed.

Type dict

deleter(*fdel*)

getter(*fget*)

kwargs = None

name = ''

setter(*fset*)

experimentor.models.meta module

Meta Models

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.meta.**MetaModel**(*name*, *bases*, *attrs*)
Bases: type

Meta Model type which will be responsible for keeping track of all the created models in the program. This is very useful for things like automatically building a GUI, initializing/finishing all the devices, etc. and also to perform checks at the beginning of the runtime, by doing introspection on all the defined models, regardless of whether they are instantiated later on or no.

One of the tasks is to generate a list of signals available in each model. Signals are specified as class attributes and therefore they can be accounted for before instantiating the class. Once the class is being instantiated, each object will re-instantiate the signals in order to keep its own copy, and establishing the proper owner of the signal.

get_instances(*recursive=False*)
Get all instances of this class in the registry.

Parameters **recursive** (*bool*) – Search for instances recursively through inherited objects

get_models(*recursive=False*)
Gets all the models which share the MetaModel origin.

Parameters **recursive** (*bool*) – Search recurisvely in sub classes of the model

experimentor.models.models module

Models

Models are a buffer between user interactions and real devices. Models should define at least some basic common properties, for example how to read a value from a sensor and how to apply a value to an actuator. Models can also take care of manipulating data, for example calculating an FFT and returning it to the user.

license MIT, see LICENSE for more details

copyright 2020 Aquiles Carattino

class experimentor.models.models.**BaseModel**
Bases: object

All models should inherit from this base model. It defines some basic methods and checks that prevent errors later at runtime.

_features
Dictionary-like object to store the properties of the model

Type *ExpDict*

_actions
List-like object to store the available actions. It also stores a lock to prevent multiple actions to be triggered at the same time

Type *ExpList*

_settings
Dictionary-like object where the settings are stored. This dictionary is also used to retrieve the latest known value of the setting.

Type *ExpDict*

_signals
Dictionary-like object to store the signals of the model

Type *ExpDict*

_subscribers
Dictionary-like object storing the subscribers to different signals arising from this model

Type *ExpDict*

classmethod as_process(**args*, ***kwargs*)
Instantiate the model as a *ProxyObject* that will run on a separate process.

> **Warning:** This is WORK IN PROGRESS and will remain so for the foreseeable future.

clean_up_threads()
Keep only the threads that are alive.

create_context()
Creates the ZMQ context. In case of wanting to use a specific context (perhaps globally defined), overwrite this method in the child classes. This method is called during the model instantiation.

create_publisher()
Creates a ZMQ publisher. It will be used by signals to broadcast their information. There is a delay before returning the publisher to guarantee that it was properly initialized before actually using it.

Returns

- *zmq.Publisher* – Returns the initialized publisher

- *.. todo:: This method has a high chance of being converted to an Action in order to let it run in parallel*

emit(*signal_name*, *payload*, ***kwargs*)

Emits a signal using the publisher bound to the model. It uses the method *BaseModel.get_publisher()* to get the publisher to use. You can override that method in order to use a different publisher (for example, an experiment-based publisher instead of a model-based one.

Notes

If subscribers are too slow, a queue will build up on the publisher, which may lead to the model itself crashing. It is important to be sure subscribers can keep up.

Parameters

- **signal_name** (*str*) – The name of the signal is used as a topic for the publisher. Remember that in PyZMQ, topics are filtered on the subscriber side, therefore everything is always broadcasted broadly, which can be a bottleneck for performance in case there are many subscribers.
- **payload** – It will be sent by the publisher. In case it is a `numpy` array, it will use a zero-copy strategy. For the rest, it will send using `send_pyobj`, which serializes the payload using pickle. This can be a *slow* process for complex objects.
- **kwargs** – Optional keyword arguments to make the method future-proof. Rigth now, the only supported keyword argument is `meta`, which will append to the current meta_data being broadcast. For numpy arrays, metadata is a dictionary with the following keys: `numpy`, `dtype`, `shape`. For non-numpy objects, the only key is `numpy`. The submitted metadata is appended to the internal metadata, therefore be careful not to overwrite its keys unless you know what you are doing.

finalize()

Finalizes the model. It only takes care of closing the publisher. Child classes should implement their own finalize methods (they get called automatically), and either close the publisher explicitly or use this method.

classmethod get_actions()

Returns the list of actions stored in the model. In case this behavior needs to be extended, the method can be overwritten in any child class.

get_context()

Gets the context. By default it is stored as a 'private' attribute of the model. Overwrite this method in child classes if there is need to extend functionality.

Returns The context created with `self.create_context()`

Return type zmq.Context

classmethod get_features()

Returns the dict-like features of the model. If this behavior needs to be extended, the method can be overwritten by any child class.

get_publisher()

Returns the publisher stored as a private attribute, and initialized during instantiation of the model. Consider overwriting it in order to extend functionality.

get_publisher_port()

ZMQ allows to create publishers that bind to an available port without specifying which one. This flexibility means that we should check to which port the publisher was bound if we want to use it. See `self.create_publisher()` for more details.

Returns The port to which the publisher is bound. A string of integers

Return type str

get_publisher_url()
Each publisher can run on a different computer. This method should return the URL in which to connect to the publisher.

Todo: Right now it only returns localhost, this MUST be improved

initialize()

classmethod set_actions(*actions*)
Method to store actions in the model. It is a convenience method that can be overwritten by child classes.

subscribers

class experimentor.models.models.**ExpDict**
Bases: dict

class experimentor.models.models.**ExpList**
Bases: list

lock = <Lock(owner=None)>

class experimentor.models.models.**ProxyObject**(*cls*, **args*, ***kwargs*)
Bases: object

Creates an object that can run on a separate process. It uses pipes to exchange information in and out. This is experimental and not meant to be used in a real application. It is here as a way of documenting one of the possible directions.

Note: Right now we are using the multiprocessing pipes to exchange information, it would be useful to use the zmq options in order to have a consistent interface through the project.

experimentor.models.properties module

Properties

Every model in Experimentor has a set of properties that define their state. A camera has, for example, an exposure time, a DAQ card has a delay between data points, and an Experiment holds global parameters, such as the number of repetitions a measurement should take.

In many situations, the parameters are stored as a dictionary, mainly because they are easy to retrieve from a file on the hard drive and to access from within the class. We want to keep that same approach, but adding extra features.

Features of Properties

Each parameter stored on a property will have three values: new_value, value, old_value, which represent the value which will be set, the value that is currently set and the value that was there before. In this way it is possible to just update on the device those values that need updating, it is also possible to revert back to the previously known value.

Each value will also be marked with a flag to_update in case the value was changed, but not yet transmitted to the device. This allows us to collect all the values we need, for example looping through a user interface, reading a config file, and applying only those needed whenever desired.

The Properties have also another smart feature, achieved through linking. Linking means building a relationship between the parameters stored within the class and the methods that need to be executed in order to get or set those values. In the linking procedure, we can set only getter methods for read-only properties, or both methods. A general apply function then allows to use the known methods to set the values that need to be updated to the device.

Future Roadmap

We can consider forcing methods to always act on properties defined as new/known/old in order to use that information as a form of cache and validation strategy.

license MIT, see LICENSE for more details

copyright 2021 Aquiles Carattino

class experimentor.models.properties.**Properties**(*parent: experimentor.models.models.BaseModel*, ***kwargs*)

Bases: object

Class to store the properties of models. It keeps track of changes in order to monitor whether a specific value needs to be updated. It also allows to keep track of what method should be triggered for each update.

all()
Returns a dictionary with all the known values.

Returns properties – All the known values

Return type dict

apply(*property*, *force=False*)
Applies the new value to the property. This is provided that the property is marked as to_update, or forced to be updated.

Parameters

- **property** (*str*) – The string identifying the property
- **force** (*bool (default: False)*) – If set to true it will update the propery on the device, regardless of whether it is marked as to_update or not.

apply_all()
Applies all changes marked as 'to_update', using the links to methods generated with :meth:~link

autolink()
Links the properties defined as ModelProp in the models using their setters and getters.

fetch(*prop*)
Fetches the desired property from the device, provided that a link is available.

fetch_all()
Fetches all the properties for which a link has been established and updates the value. This method does not alter the to_update flag, new_value, nor old_value.

classmethod from_dict(*parent*, *data*)
Create a Properties object from a dictionary, including the linking information for methods. The data has to be passed in the following form: {property: [value, getter, setter]}, where *getter* and *setter* are the methods used by :meth:~link.

Parameters

- **parent** – class to which the properties are attached
- **data** (*dict*) – Information on the values, getter and setter for each property

get_property(*prop*)
Get the information of a given property, including the new value, value, old value and if it is marked as to be updated.

Returns **prop** – The requested property as a dictionary

Return type dict

link(*linking*)
Link properties to methods for update and retrieve them.

Parameters **linking** (*dict*) – Dictionary in where information is stored as parameter=>[getter, setter], for example:

```
linking = {'exposure_time': [self.get_exposure, self.set_exposure]}
```

In this case, `exposure_time` is the property stored, while `get_exposure` is the method that will be called for getting the latest value, and set_exposure will be called to set the value. In case set_exposure returns something different from None, no extra call to get_exposure will be made.

to_update()
Returns a dictionary containing all the properties marked to be updated.

Returns **props** – all the properties that still need to be updated

Return type dict

unlink(*unlink_list*)
Unlinks the properties and the methods. This is just to prevent overwriting linkings under the hood and forcing the user to actively unlink before linking again.

Parameters **unlink_list** (*list*) – List containing the names of the properties to be unlinked.

update(*values: dict*)
Updates the values in the same way the update method of a dictionary works. It, however, stores the values as a new value, it does not alter the values stored. For updating the proper values use `self.upgrade()`.

After updating the values, use `self.apply_all()` to send the new values to the device.

upgrade(*values*, *force=False*)
This method actually overwrites the values stored in the properties. This method should be used only when the real values generated by a device are known. It will change the new values to None, it will set the value to value, and it will set the `to_update` flag to false.

Parameters

- **values** (*dict*) – Dictionary in the form {property: new_value}
- **force** (*bool*) – If force is set to True, it will create the missing properties instead of raising an exception.

Module contents

experimentor.views package

Subpackages

experimentor.views.camera package

Submodules

experimentor.views.camera.camera_viewer_widget module

Camera Viewer Widget

Wrapper around PyQtGraph ImageView.

class experimentor.views.camera.camera_viewer_widget.**CameraViewerWidget**(*parent=None*)
Bases: *experimentor.views.data_view_widget.DataViewWidget*

The Camera Viewer Widget is a wrapper around PyQtGraph ImageView. It adds some common methods for getting extra mouse interactions, such as performing an auto-range through right-clicking, it allows to drag and drop horizontal and vertical lines to define a ROI, and it allows to draw on top of the image. The core idea is to make these options explicit, in order to systematize them in one place.

clicked_on_image: Emits [float, float] with the coordinates where the mouse was clicked on the image. Does not distinguish between left/right clicks. Any further processing must be done downstream.

layout

Type QHBoxLayout, in case extra elements must be added

viewport

Type GraphicsLayoutWidget

view

Type ViewBox

img

Type ImageItem

imv

Type ImageView

auto_levels

Type Whether to actualize the levels of the image every time they are refreshed

add_actions_to_menu()
Adds actions to the contextual menu. If you want to have control on which actions appear, consider subclassing this widget and overriding this method.

clicked_on_image

classmethod connect_to_camera(*camera*, *refresh_time=50*, *parent=None*)
Instantiate the viewer using connect_to_camera in order to get some functionality out of the box. It will create a timer to automatically update the image

do_auto_range()
Sets the levels of the image based on the maximum and minimum. This is useful when auto-levels are off (the default behavior), and one needs to quickly adapt the histogram.

draw_target_pointer(*locations*)
gets an image and draws a circle around the target locations.

Parameters **locations** (*DataFrame*) – DataFrame generated by trackpy's locate method. It only requires columns *x* and *y* with coordinates.

get_roi_values()
Get's the ROI values in camera-space. It keeps track of the top left corner in order to update the values before returning. :return: Position of the corners of the ROI region assuming 0-indexed cameras.

keyPressEvent(*key*)
Triggered when there is a key press with some modifier. Shift+C: Removes the cross hair from the screen These last two events have to be handeled in the mainWindow that implemented this widget.

mouseMoved(*arg*)
Updates the position of the cross hair. The mouse has to be moved while pressing down the Ctrl button.

mouse_clicked(*evnt*)

scene()
Shortcut to getting the image scene

set_roi_lines(*X*, *Y*)

setup_cross_cut(*max_size*)
Set ups the horizontal line for the cross cut.

setup_cross_hair(*max_size*)
Sets up a cross hair.

setup_mouse_tracking()

setup_roi_lines(*max_size=None*)
Sets up the ROI lines surrounding the image.

Parameters **max_size** (*list*) – List containing the maximum size of the image to avoid ROIs bigger than the CCD.

update_image(*image*, *auto_range=False*, *auto_histogram_range=False*)
Updates the image being displayed with some sensitive defaults, which can be over written if needed.

Module contents

class experimentor.views.camera.**CameraViewerWidget**(*parent=None*)
Bases: *experimentor.views.data_view_widget.DataViewWidget*

The Camera Viewer Widget is a wrapper around PyQtGraph ImageView. It adds some common methods for getting extra mouse interactions, such as performing an auto-range through right-clicking, it allows to drag and drop horizontal and vertical lines to define a ROI, and it allows to draw on top of the image. The core idea is to make these options explicit, in order to systematize them in one place.

clicked_on_image: Emits [float, float] with the coordinates where the mouse was clicked on the image. Does not distinguish between left/right clicks. Any further processing must be done downstream.

layout

Type QHBoxLayout, in case extra elements must be added

viewport

Type GraphicsLayoutWidget

view

Type ViewBox

img

Type ImageItem

imv

Type ImageView

auto_levels

Type Whether to actualize the levels of the image every time they are refreshed

add_actions_to_menu()
Adds actions to the contextual menu. If you want to have control on which actions appear, consider subclassing this widget and overriding this method.

clicked_on_image

classmethod connect_to_camera(*camera*, *refresh_time=50*, *parent=None*)
Instantiate the viewer using connect_to_camera in order to get some functionality out of the box. It will create a timer to automatically update the image

do_auto_range()
Sets the levels of the image based on the maximum and minimum. This is useful when auto-levels are off (the default behavior), and one needs to quickly adapt the histogram.

draw_target_pointer(*locations*)
gets an image and draws a circle around the target locations.

Parameters **locations** (*DataFrame*) – DataFrame generated by trackpy's locate method. It only requires columns *x* and *y* with coordinates.

get_roi_values()
Get's the ROI values in camera-space. It keeps track of the top left corner in order to update the values before returning. :return: Position of the corners of the ROI region assuming 0-indexed cameras.

keyPressEvent(*key*)
Triggered when there is a key press with some modifier. Shift+C: Removes the cross hair from the screen These last two events have to be handeled in the mainWindow that implemented this widget.

mouseMoved(*arg*)
Updates the position of the cross hair. The mouse has to be moved while pressing down the Ctrl button.

mouse_clicked(*evnt*)

scene()
Shortcut to getting the image scene

set_roi_lines(*X*, *Y*)

setup_cross_cut(*max_size*)
Set ups the horizontal line for the cross cut.

setup_cross_hair(*max_size*)
Sets up a cross hair.

setup_mouse_tracking()

setup_roi_lines(*max_size=None*)
Sets up the ROI lines surrounding the image.

Parameters **max_size** (*list*) – List containing the maximum size of the image to avoid ROIs bigger than the CCD.

update_image(*image*, *auto_range=False*, *auto_histogram_range=False*)
Updates the image being displayed with some sensitive defaults, which can be over written if needed.

experimentor.views.model_view package

Submodules

experimentor.views.model_view.model_view module

class experimentor.views.model_view.model_view.**ModelViewWidget**(*model: experimentor.models.devices.base_device.ModelDe parent=None*)

Bases: PyQt5.QtWidgets.QWidget

get_layout()

model_to_layout()

set_layout()

Module contents

class experimentor.views.model_view.**ModelViewWidget**(*model: experimentor.models.devices.base_device.ModelDevice, parent=None*)

Bases: PyQt5.QtWidgets.QWidget

get_layout()

model_to_layout()

set_layout()

experimentor.views.widgets package

Submodules

experimentor.views.widgets.toggable_button module

class experimentor.views.widgets.toggable_button.**ToggableButton**(**args*, ***kwargs*)

Bases: PyQt5.QtWidgets.QPushButton

toggle(*self*)

Module contents

class experimentor.views.widgets.**ToggableButton**(**args*, ***kwargs*)
Bases: PyQt5.QtWidgets.QPushButton

toggle(*self*)

Submodules

experimentor.views.data_view_widget module

class experimentor.views.data_view_widget.**DataViewWidget**(*parent=None*)
Bases: PyQt5.QtWidgets.QWidget

Base class that defines some common patterns for views which are meant to display data.

default_Layout
method get_layout

Type By default, views will have a QHBoxLayout, it can be overriden when subclassing, or by changing the

data
of what specific type of data it is.

Type This is the data being represented by the widget. This allows to define abstract methods for saving, regardless

default_layout = 'horizontal'

get_layout()
Returns the layout specified as the class attribute default_layout. Override this method to provide more complex behavior.

set_layout()

experimentor.views.decorators module

experimentor.views.decorators.**try_except_dialog**(*func*)
Decorator to add to methods used in user interfaces. If there is a chance of an error appearing because of devices in the wrong state, etc. but the logic is not fail proof, you can use this decorator to display an error message with the stack trace instead of crashing the program.

experimentor.views.exceptions module

exception experimentor.views.exceptions.**ViewException**
Bases: Exception

Module contents

Submodules

experimentor.management module

Module contents

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W