Openflexure Stage Documentation

Release 0.2.1

Richard Bowman

Sep 05, 2019

Contents:

This module exposes the functions of the OpenFlexure Nano Motor Controller (AKA the "Sangaboard") in a friendly Python class. It allows the stage to be moved, as well as providing properties that allow it to be configured. Various context managers and generator functions are provided to simplify opening/closing the hardware, and common operations such as scanning through a list of points.

All of the functionality is accessed through the *openflexure_stage.stage.OpenFlexureStage* class, which optionally includes a *openflexure_stage.stage.LightSensor* module if the firmware and hardware are set up to include this and a *openflexure_stage.stage.Endstops* module if the firmware is compiled with endstop support

CHAPTER 1

Sangaboard hardware

This module is designed to work with the "sangaboard" motor controller, based on an Arduino Nano and some Darlington pair ICs. The PCB design is available on Github, and can be ordered through WaterScope or via Kitspace.

CHAPTER 2

Firmware

You will need to make sure that the Arduino is running the correct firmware "sketch". Assuming you are familiar with the Arduino IDE you can download the repository from github (either download a Zip file or clone the repository) and then look in the arduino_code folder for the sketch: *arduino_code/openflexure_nano_motor_controller/openflexure_nano_motor_controller.ino*.

More information is available in the README.md file in the arduino_code folder.

Once you've uploaded that sketch to your Arduino, the firmware should be done - you can test it in the Serial Monitor in the Arduino IDE, though make sure to set the baud rate to 115200 or it won't work!

CHAPTER 3

Getting started

To use the motor controller from Python, you will first need to install this module. It can be installed using *pip* in the usual way, which will also require the packages that it depends on (`future` and `pyserial`). The simplest way to use the module is this:

```
from openflexure_stage import OpenFlexureStage
with OpenFlexureStage() as stage:
    stage.move_rel([100,0,0])
    print(stage.position)
    stage.move_rel([-100,0,0])
```

By default, it will use the first available serial port - if you are using a Raspberry Pi and you don't have any other USB serial devices connected, this will usually work. If not, you need to specify the serial port in the constructor:

```
OpenFlexureStage("/dev/ttyUSB0")
```

The name of the serial port will depend on your operating system - Linux typically assigns names that look like `/dev/ttyUSB0` while Windows will often give it a name like `COM4`.

Make sure you close the stage after you're finished with it - the best way to do this is using a *with* block, but you can also call *close()* manually if required.

3.1 Moving the stage

The most basic thing you are likely to want to do with the stage. This is done with *move_rel()* most of the time, though it's also possible to make absolute moves. The Sangaboard keeps track of position in firmware, and will return its position if you query *position*.

3.2 Adjusting settings

There are a number of properties of your *OpenFlexureStage* object that can be used to change the way it works:

- *backlash*: software backlash compensation
- *ramp_time*: acceleration control
- *step_time*: define the maximum speed

3.3 Using optional features

If you compile support for it, you can add a light sensor to the stage, which is accessed as `light_sensor`. This returns a *LightSensor* which allows you to control the gain (if possible) and read the light intensity.

If you compile support for it, you can add mechanical endstops, which are accessed as `endstops`. This returns *Endstops* which allows you to home the axes, check endstop status, and control soft endstop position (if enabled).

CHAPTER 4

openflexure_stage

4.1 openflexure_stage package

4.1.1 Submodules

openflexure_stage.basic_serial_instrument module

This module defines a chopped-out class from nplab. It is a basic serial instrument class to simplify the process of interfacing with equipment that talks on a serial port. The idea is that your instrument can subclass `BasicSerialInstrument` and provide methods to control the hardware, which will mostly consist of *self.query()* commands.

The `QueriedProperty` class is a convenient shorthand to create a property that is read and/or set with a single serial query (i.e. a read followed by a write).

class openflexure_stage.basic_serial_instrument.**BasicSerialInstrument**(*port=None*, ***kwargs*)

Bases: `object`

An instrument that communicates by sending strings back and forth over serial

This base class provides commonly-used mechanisms that support the use of serial instruments. Most interactions with this class involve a call to the *query* method. This writes a message and returns the reply. This has been hacked together from the nplab MessageBusInstrument and SerialInstrument classes.

Threading Notes

The message bus protocol includes a property, *communications_lock*. All commands that use the communications bus should be protected by this lock. It's also permissible to use it to protect sequences of calls to the bus that must be atomic (e.g. a multi-part exchange of messages). However, try not to hold it too long - or odd things might happen if other threads are blocked for a long time. The lock is reentrant so there's no issue with acquiring it twice.

close()

Release the serial port

communications_lock
A lock object used to protect access to the communications bus

find_port()
Iterate through the available serial ports and query them to see if our instrument is there.

float_query(*query_string*, ***kwargs*)
Perform a query and return the result(s) as float(s) (see parsedQuery)

flush_input_buffer()
Make sure there's nothing waiting to be read, and clear the buffer if there is.

ignore_echo = False

int_query(*query_string*, ***kwargs*)
Perform a query and return the result(s) as integer(s) (see parsedQuery)

open(*port=None*, *quiet=True*)
Open communications with the serial port.

If no port is specified, it will attempt to autodetect. If quiet=True then we don't warn when ports are opened multiple times.

parsed_query(*query_string*, *response_string='%d'*, *re_flags=0*, *parse_function=None*, ***kwargs*)
Perform a query, returning a parsed form of the response.

First query the instrument with the given query string, then compare the response against a template. The template may contain text and placeholders (e.g. %i and %f for integer and floating point values respectively). Regular expressions are also allowed - each group is considered as one item to be parsed. However, currently it's not supported to use both % placeholders and regular expressions at the same time.

If placeholders %i, %f, etc. are used, the returned values are automatically converted to integer or floating point, otherwise you must specify a parsing function (applied to all groups) or a list of parsing functions (applied to each group in turn).

port_settings = {}

query(*queryString*, *multiline=False*, *termination_line=None*, *timeout=None*)
Write a string to the stage controller and return its response.

It will block until a response is received. The multiline and termination_line commands will keep reading until a termination phrase is reached.

read_multiline(*termination_line=None*, *timeout=None*)
Read one line from the underlying bus. Must be overriden.

This should not need to be reimplemented unless there's a more efficient way of reading multiple lines than multiple calls to readline().

readline(*timeout=None*)
Read one line from the serial port.

termination_character = '\n'
All messages to or from the instrument end with this character.

termination_line = None
If multi-line responses are recieved, they must end with this string

test_communications()
Check if the device is available on the current port.

This should be overridden by subclasses. Assume the port has been successfully opened and the settings are as defined by self.port_settings. Usually this function sends a command and checks for a known reply.

write(*query_string*)
Write a string to the serial port

class openflexure_stage.basic_serial_instrument.**OptionalModule**(*available*, *parent=None*, *module_type='Undefined'*, *model='Generic'*)

Bases: object

This allows a *BasicSerialInstrument* to have optional features.

OptionalModule is designed as a base class for interfacing with optional modules which may or may not be included with the serial instrument, and can be added or removed at run-time.

available

confirm_available()
Check if module is available, no return, will raise exception if not available!

describe()
Consistently spaced desciption for listing modules

class openflexure_stage.basic_serial_instrument.**QueriedProperty**(*get_cmd=None*, *set_cmd=None*, *validate=None*, *valrange=None*, *fdel=None*, *doc=None*, *response_string=None*, *ack_writes='no'*)

Bases: object

A Property interface that reads and writes from the instrument on the bus.

This returns a property-like (i.e. a descriptor) object. You can use it in a class definition just like a property. The property it creates will interact with the instrument over the communication bus to set and retrieve its value. It uses calls to *BasicSerialInstrument.parsed_query* to set or get the value of the property.

QueriedProperty can be used to define properties on a *BasicSerialInstrument* or an *OptionalModule* (in which case the *BasicSerialInstrument.parsed_query* method of the parent object will be used).

Arguments:

Get_cmd the string sent to the instrument to obtain the value

Set_cmd the string used to set the value (use {} or % placeholders)

Validate a list of allowable values

Valrange a maximum and minimum value

Fdel a function to call when it's deleted

Doc the docstring

Response_string supply a % code (as you would for response_string in a BasicSerialInstrument.parsed_query)

Ack_writes set to "readline" to discard a line of input after writing.

openflexure_stage.stage module

OpenFlexure Stage module

This Python code deals with the computer (Raspberry Pi) side of communicating with the OpenFlexure Motor Controller.

It is (c) Richard Bowman 2017 and released under GNU GPL v3

class openflexure_stage.stage.**Endstops**(*available*, *parent=None*, *model='min'*)
Bases: *openflexure_stage.basic_serial_instrument.OptionalModule*

An optional module for use with endstops.

If endstops are installed in the firmware the `openflexure_stage.OpenFlexureStage` will gain an optional module which is an instance of this class. It can be used to retrieve the type, state of the endstops, read and write maximum positions, and home.

home(*direction='min', axes=['x', 'y', 'z']*)
Home given/all axes in the given direction (min/max/both)

Parameters

- **direction** – one of {min,max,both}
- **axes** – list of axes e.g. ['x','y']

installed = []

maxima
Vector of maximum positions, homing to max endstops will measure this, can be set to a known value for use with max only and min+soft endstops

status
Get endstops status as {-1,0,1} for {min,no,max} endstop triggered for each axis

test_mode = False
List of installed endstop types (min, max, soft)

class openflexure_stage.stage.**LightSensor**(*available*, *parent=None*, *model='Generic'*)
Bases: *openflexure_stage.basic_serial_instrument.OptionalModule*

An optional module giving access to the light sensor.

If a light sensor is enabled in the motor controller's firmware, then the `openflexure_stage.OpenFlexureStage` will gain an optional module which is an instance of this class. It can be used to access the light sensor (usually via the I2C bus).

gain
"Get or set the current gain value of the light sensor.

Valid gain values are defined in the *valid_gains* property, and should be floating-point numbers.

integration_time
Get or set the integration time of the light sensor in milliseconds.

intensity
Read the current intensity measured by the light sensor (arbitrary units).

valid_gains = None

class openflexure_stage.stage.**OpenFlexureStage**(**args*, ***kwargs*)
Bases: *openflexure_stage.basic_serial_instrument.BasicSerialInstrument*

Class managing serial communications with an Openflexure Motor Controller

The *OpenFlexureStage* class handles setting up communications with the stage, wraps the various serial commands in Python methods, and provides iterators and context managers to simplify opening/closing the hardware connection and some other tasks like conducting a linear scan.

Arguments to the constructor are passed to the constructor of *openflexure_stage.basic_serial_instrument.BasicSerialInstrument*, most likely the only one necessary is *port* which should be set to the serial port you will use to communicate with the motor controller.

This class can be used as a context manager, i.e. it's encouraged to use it as:

```
with OpenFlexureStage() as stage:
    stage.move_rel([1000,0,0])
```

In that case, the serial port will automatically be closed at the end of the block, even if an error occurs. Otherwise, be sure to call the *close()* method to release the serial port.

axis_names = ('x', 'y', 'z')
The names of the stage's axes. NB this also defines the number of axes.

backlash
The distance used for backlash compensation.

Software backlash compensation is enabled by setting this property to a value other than *None*. The value can either be an array-like object (list, tuple, or numpy array) with one element for each axis, or a single integer if all axes are the same.

The property will always return an array with the same length as the number of axes.

The backlash compensation algorithm is fairly basic - it ensures that we always approach a point from the same direction. For each axis that's moving, the direction of motion is compared with `backlash`. If the direction is opposite, then the stage will overshoot by the amount in `-backlash[i]` and then move back by `backlash[i]`. This is computed per-axis, so if some axes are moving in the same direction as `backlash`, they won't do two moves.

board = None
Once initialised, *board* is a string that identifies the firmware version.

focus_rel(*z*)
Move the stage in the Z direction by z micro steps.

list_modules()
Return a list of strings detailing optional modules.

Each module will correspond to a string of the form `Module Name:  Model`

move_abs(*final*, ***kwargs*)
Make an absolute move to a position

NB the stage only accepts relative move commands, so this first queries the stage for its position, then instructs it to make about relative move.

move_rel(*displacement*, *axis=None*, *backlash=True*)
Make a relative move, optionally correcting for backlash.

displacement: integer or array/list of 3 integers axis: None (for 3-axis moves) or one of 'x','y','z' backlash: (default: True) whether to correct for backlash.

n_axes
The number of axes this stage has.

port_settings = {'baudrate': 115200, 'bytesize': 8, 'parity': 'N', 'stopbits': 1}
These are the settings for the stage's serial port, and can usually be left as default.

position
Get the position of the stage as a tuple of 3 integers.

print_help()
Print the stage's built-in help message.

query(*message*, **args*, ***kwargs*)
Send a message and read the response. See BasicSerialInstrument.query()

ramp_time
Get or set the acceleration time in microseconds.

The stage will accelerate/decelerate between stationary and maximum speed over *ramp_time* microseconds. Zero means the stage runs at full speed initially, with no accleration control. Small moves may last less than *2*ramp_time*, in which case the acceleration will be the same, but the stage will never reach full speed. It is saved to EEPROM on the Arduino, so it will be persistent even if the motor controller is turned off.

release_motors()
De-energise the stepper motor coils

scan_linear(*rel_positions*, *backlash=True*, *return_to_start=True*)
Scan through a list of (relative) positions (generator fn)

rel_positions should be an nx3-element array (or list of 3 element arrays). Positions should be relative to the starting position - not a list of relative moves.

backlash argument is passed to move_rel

if return_to_start is True (default) we return to the starting position after a successful scan. NB we always attempt to return to the starting position if an exception occurs during the scan..

scan_z(*dz*, ***kwargs*)
Scan through a list of (relative) z positions (generator fn)

This function takes a 1D numpy array of Z positions, relative to the position at the start of the scan, and converts it into an array of 3D positions with x=y=0. This, along with all the keyword arguments, is then passed to `scan_linear`.

step_time
Get or set the minimum time between steps of the motors in microseconds.

The step time is `1000000/max speed` in steps/second. It is saved to EEPROM on the Arduino, so it will be persistent even if the motor controller is turned off.

supported_light_sensors = ['TSL2591', 'ADS1115']
This is a list of the supported light sensor module types.

test_mode
Get or set test mode

In test mode

- Stage may return extra information
- When homing, the stage will remain at the 0 position
- Position will not be reset when an endstop is hit

4.1.2 Module contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

N

O

P

Q

R

S

T

V

W