Helmut Hörner

COMICS:
A Modular, Configurable
Confocal Microscope Control Software

Project Report

TU Wien
Institute of Atomic and Subatomic Physics

Vienna, January 2021

Abstract

This paper presents the 'Confocal Microsccope Control Software', or COMICS for short. It was developed by the author in course of a technical physics master studies project at the Vienna Institute of Atomic and Subatomic Physics.

COMICS is based on the Qudi software framework, and is separated into a User Interface Layer, a Logic Layer, and a Hardware Layer. Especially the separated Hardware Layer allows for easy integration of new hardware components, if required. This paper gives a brief introduction into the confocal microscope hardware currently controlled by the COMICS software, then a detailed description of the software architecture and the software interfaces, followed by installation instructions, configuration instructions and a user's guide.

Contents

1 Introduction **9**

2 The Confocal Microscope Hardware Design **10**
2.1 Axes Orientation . . . 13

3 COMICS Software Design **14**
3.1 Introduction to the Qudi Framework . . . 14
3.2 COMICS software architecture overview . . . 15

4 Interfaces for Hardware Layer Classes **17**
4.1 ati_positioner_interface . . . 17
4.1.1 get_name() . . . 17
4.1.2 set_name(new_name) . . . 17
4.1.3 get_pos_ch_count() . . . 18
4.1.4 get_pos_axis_count(channel_cnf) . . . 18
4.1.5 get_pos_device_name(channel_cnf) . . . 18
4.1.6 get_pos_range(channel_cnf) . . . 19
4.1.7 get_pos_axes(channel_cnf) . . . 19
4.1.8 set_pos_position(channel_cnf, x=None, y=None, z=None, a=None) 20
4.1.9 get_pos_position(channel_cnf, extra_precision=1) . . . 21
4.1.10 get_pos_target_position(channel_cnf) . . . 22
4.1.11 pos_matches_target_position(self, channel_cnf, xyz, pos) . . . 22
4.1.12 get_pos_ini_position(channel_cnf, xyz) . . . 23
4.1.13 pos_provides_cur_pos_info() . . . 23
4.1.14 pos_is_moving(channel_cnf) . . . 24
4.1.15 pos_is_resetting(channel_cnf) . . . 24
4.1.16 pos_allows_homing(channel_cnf) . . . 24
4.1.17 pos_home(channel_cnf) . . . 25
4.1.18 pos_trigger_queue(channel_cnf) . . . 25
4.1.19 get_pos_default_unit(channel_cnf) . . . 25
4.1.20 get_pos_precision(channel_cnf) . . . 26
4.1.21 get_pos_steps_default(channel_cnf, axis_cnf) . . . 26
4.1.22 get_pos_steps_min(channel_cnf, axis_cnf) . . . 27
4.1.23 get_pos_steps_max(channel_cnf, axis_cnf) . . . 27
4.1.24 close_positioner() . . . 27
4.2 ati_counter_interface . . . 28
4.2.1 get_name() . . . 28
4.2.2 set_name(new_name) . . . 28
4.2.3 get_pcnt_sources_count() . . . 28
4.2.4 get_pcnt_channel_count(source_cnf) . . . 29
4.2.5 get_pcnt_channel_names(source_cnf) . . . 29
4.2.6 get_pcnt_rate(source_cnf) . . . 30

4.2.7 update_cur_count_rate(output_source_cnf, channel_idx) 30
4.2.8 get_last_cnt_rates() 31
4.2.9 close_counter() 31
4.3 ati_camera_interface 32
4.3.1 get_name() 32
4.3.2 is_connected() 32
4.3.3 get_size() 32
4.3.4 get_resolutions() 33
4.3.5 set_resolution(res_index) 33
4.3.6 start_single_acquisition(wait_for_acquisition) 34
4.3.7 is_acquiring() 34
4.3.8 stop_acquisition() 34
4.3.9 get_acquired_data() 35
4.3.10 get_ready_state() 35
4.3.11 set_cooler(state) 35
4.3.12 is_cooler_on() 35
4.3.13 set_temperature(temp) 36
4.3.14 provides_temperature_information() 36
4.3.15 get_temperature() 36
4.3.16 get_temperature_precision() 37
4.3.17 set_exposure(exposure) 37
4.3.18 get_exposure(get_actual_val_from_hardware = True) 37
4.3.19 get_max_exposure() 37
4.3.20 has_shutter() 38
4.3.21 get_shutter_settings() 38
4.3.22 get_default_shutter_setting() 38
4.3.23 set_shutter(index) 38
4.3.24 get_additional_settings() 39
4.3.25 set_additional_settings(ID, value) 41
4.4 ati_actuator_interface 42
4.4.1 get_name() 42
4.4.2 set_name(new_name) 42
4.4.3 get_act_ch_count() 42
4.4.4 get_act_ch_name(ch_cnf) 43
4.4.5 get_act_ch_output(ch_cnf) 43
4.4.6 get_act_ch_is_digital(ch_cnf) 44
4.4.7 set_act_output(ch_cnf, value) 44

5 Logic Layer Class Interface **45**
5.1 Remarks on Absolute and Relative Coordinates 45
5.2 General Scan Methods 45
5.2.1 get_scan_type_count() 45
5.2.2 get_scan_type_max_index() 46
5.2.3 get_scan_type_name(scan_type) 46

5.2.4 set_cur_scan_type_index(scan_type) 46
5.2.5 get_cur_scan_type_index() 46
5.2.6 set_scan_center_position(scan_type, x=None, y=None, z=None, a=None, absolute = False) 47
5.2.7 set_scan_X_range(scan_type, new_range, absolute = False) . . . 48
5.2.8 set_scan_Y_range(scan_type, new_range, absolute = False) . . . 49
5.2.9 set_scan_Z_range(scan_type, new_range, absolute = False) . . . 50
5.2.10 get_scan_range(scan_type, xyz, absolute = False) 51
5.2.11 matches_target_position(scan_type, xyz, pos, absolute = False) . 52
5.2.12 set_scan_resolution(scan_type, xyz, new_res) 53
5.2.13 same_scan_resolution(scan_type, xyz_1, xyz_2) 54
5.2.14 get_scan_resolution(scan_type, xyz) 55
5.2.15 get_min_resolution(scan_type, xyz) 56
5.2.16 get_max_resolution(scan_type, xyz) 57
5.2.17 start_scanning(scan_type, depth_scan = False, tag = 'logic') . . 57
5.2.18 stop_scanning() 58
5.2.19 continue_scanning(depth_scan, tag = 'logic') 59
5.2.20 cur_scan_is_depth_scan() 59
5.2.21 get_cur_main_scan_continuable() 60
5.2.22 get_cur_depthscan_continuable() 60
5.2.23 get_main_scan_image(scan_type) 61
5.2.24 get_depth_scan_image(scan_type) 62
5.2.25 get_scan_counter() 63
5.2.26 get_ax_index_from_xyz_index(d, scan_type, xyz) 64
5.2.27 get_third_scan_axis_xyz_index(scan_type, d) 65
5.2.28 allows_depth_scan(scan_type) 65
5.2.29 save_file(scan_type, file_name, main_image) 66
5.2.30 load_file(scan_type, file_name) 67
5.3 Main Positioning Device Control Methods 68
5.3.1 set_position(tag, scan_type=-1, x=None, y=None, z=None, a=None, absolute = False) 68
5.3.2 get_position(scan_type = -1, absolute = False) 69
5.3.3 get_target_position(scan_type, absolute = False) 70
5.3.4 pos_provides_cur_pos_info(scan_type) 71
5.3.5 get_pos_range(scan_type, xyz, absolute = False) 72
5.3.6 set_pos_ini_position(scan_type, main_axes, absolute_center) . . 73
5.3.7 get_pos_ini_position(scan_type, xyz, absolute, absolute_center) 73
5.3.8 get_pos_is_centered(scan_type, main_axes) 74
5.3.9 absolute_pos_coords_available(scan_type) 74
5.3.10 pos_abs_to_rel(scan_type, xyz, pos_abs) 75
5.3.11 pos_rel_to_abs(scan_type, xyz, pos_rel) 76
5.3.12 get_pos_length_unit(scan_type) 76
5.3.13 get_pos_precision(scan_type) 77
5.3.14 get_pos_axes(scan_type = -1) 77

5.3.15 get_pos_ch_device_name(scan_type, first_char_upper) 78
5.4 Pre-Positioning Device Control Methods 78
5.4.1 set_pre_position(scan_type, x=None, y=None, z=None, a=None, absolute = False) 79
5.4.2 get_pre_pos_position(scan_type, absolute = False) 80
5.4.3 pre_pos_allows_homing(scan_type) 81
5.4.4 pre_pos_home(scan_type) 81
5.4.5 get_pre_pos_ini_position(scan_type, xyz, absolute = False) . . . 82
5.4.6 absolute_pre_pos_coords_available(scan_type) 82
5.4.7 pre_pos_abs_to_rel(scan_type, xyz, pos_abs) 83
5.4.8 pre_pos_rel_to_abs(scan_type, xyz, pos_rel) 83
5.4.9 pre_pos_trigger_queue(scan_type) 84
5.4.10 get_pre_pos_length_unit(scan_type) 84
5.4.11 get_pre_pos_precision(scan_type) 85
5.4.12 matches_pre_pos_target_position(scan_type, xyz, pos, absolute = False) 85
5.4.13 get_pre_pos_range(scan_type, xyz, absolute = False) 86
5.4.14 get_pre_pos_axes(scan_type = -1) 87
5.4.15 get_pre_pos_ch_device_name(scan_type, first_char_upper) . . 87
5.5 Camera Control Methods 88
5.5.1 is_camera(scan_type) 88
5.5.2 cam_is_connected(scan_type) 88
5.5.3 get_cam_resolutions(scan_type) 89
5.5.4 set_cam_resolution(scan_type, res_index) 89
5.5.5 cam_has_shutter(scan_type) 90
5.5.6 get_cam_shutter_settings(scan_type) 90
5.5.7 get_cam_default_shutter_setting(scan_type) 91
5.5.8 set_cam_shutter(scan_type, mode) 91
5.5.9 set_cam_exposure(time, scan_type=-1) 92
5.5.10 get_cam_exposure(scan_type) 92
5.5.11 get_cam_max_exposure(scan_type) 93
5.5.12 cam_provides_temperature_information(scan_type) 93
5.5.13 cam_get_temperature_precision(scan_type) 94
5.5.14 cam_get_temperature(scan_type) 94
5.5.15 cam_start_acquistion(scan_type, first_call) 95
5.5.16 cam_wait_for_acquisition_end(scan_type) 96
5.5.17 cam_is_acquiring(scan_type) 97
5.5.18 cam_fetch_image(scan_type) 98
5.5.19 get_last_cam_image() 99
5.5.20 cam_stop_acquistion(scan_type) 99
5.5.21 get_additional_cam_settings(scan_type) 100
5.5.22 set_additional_cam_setting(scan_type, id, value) 102
5.6 Photon Counter Control Methods 103
5.6.1 get_pcounter_channel_names(scan_type, default="") 103

5.6.2 get_cur_pcounter_channel_names() 103
5.6.3 get_cur_pcounter_rate(scan_type, ch_idx) 104
5.7 Signals 104
5.7.1 signal_xy_image_updated(scan_type) 104
5.7.2 signal_depth_image_updated(scan_type) 104
5.7.3 signal_cam_update_display(scan_type) 104
5.7.4 signal_cam_acquisition_finished() 104
5.7.5 signal_change_position(tag) 105

6 Software Installation and Configuration 105
6.1 Installation of all Components and Drivers 105
6.1.1 Install Git for Windows 105
6.1.2 Install Anaconda 105
6.1.3 Install PyCharm 106
6.1.4 Copy COMICS Source Code and Install Python Modules 106
6.1.5 First Start and Configuration of PyCharm 107
6.1.6 Install Python Modules in Python 3.6 Environment 109
6.1.7 Install NI PyDAQ Driver Software 110
6.1.8 Install PI Driver and Configuration Software 110
6.1.9 Install Thorlabs Kinesis Driver Software 111
6.1.10 Install Swabian Time Tagger Driver Software 112
6.1.11 Install Andor SDK Software 113
6.1.12 Create Desktop Shortcut Icon 115
6.2 Software Configuration 116
6.2.1 Kinesis Positioner 116
6.2.2 PI Positioner 118
6.2.3 NI Card 120
6.2.4 Swabian Time Tagger 124
6.2.5 Andor Camera 127
6.2.6 Kinesis Actuators 129
6.2.7 COMICS Logic Module 130
6.2.8 COMICS User Interface 137

7 User's Guide 138
7.1 Start Qudi and COMICS 138
7.2 Camera Scan Mode 140
7.2.1 Image Settings 140
7.2.2 Extended Camera Settings 140
7.2.3 Pre-Positioning 141
7.2.4 Temperature 142
7.2.5 Color Scale Setting 142
7.2.6 Display Settings 142
7.2.7 Crosshair Position Controls 143
7.2.8 Taking a Snapshot 144

7.2.9 Live Video Acquisition 144
7.3 Confocal Scan With Two-Axes Positioning-Device (e.g. Mirror Scan) . . . 145
7.3.1 Scan Resolution Settings 145
7.3.2 Scan Range Settings 146
7.3.3 Pre-Positioning 146
7.3.4 Count Rate 146
7.3.5 Color Scale Setting 146
7.3.6 Display Settings 147
7.3.7 Crosshair Position Controls 147
7.3.8 Start a Scan 148
7.3.9 Stop and Resume a Scan 149
7.4 Confocal Scan With Three-Axes Positioning-Device (e.g. Stage Scan) . . 149
7.4.1 Main Scan Pre-Positioning 150
7.4.2 Depth Scan Pre-Positioning 151
7.4.3 Cross Hair Position Controls 151
7.4.4 Starting and Stopping Main-Scan and Depth-Scan 152
7.5 Saving and Loading Images 152
7.6 Saving Images 152
7.7 Loading Images 152

1 Introduction

This paper presents the COMICS software package. COMICS is an acronym for 'Confocal Microscope Control Software'. The software was developed by the author in course of a technical physics master studies project at the Institute of Atomic and Subatomic Physics in Vienna.

Whereas the primary design goal for the COMICS software package was to meet the immediate requirements of controlling a confocal microscope newly designed and built at the Institute of Atomic and Subatomic Physics, it was an equally important design-goal to ensure a high level of flexibity and modularity so that potential future modifications or enhancements of the hardware can be easily accommodated.

As we show in this paper, the COMICS software is strictly separated into a User Interface Layer, a Logic Layer, and a Hardware Layer. Especially the separated Hardware Layer allows for easy integration of new hardware components in the future, if required. For example, there is a defined software interface between Logic Layer and Hardware Layer for controlling any type of positioning device, like, for example, laser mirrors, piezo stages or stepper motors.

If any of these positioning hardware components will change in the future, it will only be required to add a matching hardware module for the newly added single hardware component into the otherwise unchanged software package. The same goes for photon counter hardware, camera hardware or actuators (like flip mounts), for which also reference hardware modules have been implemented.

A configuration file, defining which hardware components should be accessed and used for which type of scan (or camera imaging) adds an extra level of flexibility.

The following chapter gives a brief introduction into the confocal microscope hardware at the Institute of Atomic and Subatomic Physics in Vienna for which the COMICS software has been primarily developed. After that, we give a detailed description of the software architecture and the software interfaces, followed by installation instructions, configuration instructions and a user's guide.

2 The Confocal Microscope Hardware Design

The confocal microscope for which the COMICS software has been developed for immediate application has the following features:

The sample is to be placed on a sample holder on top of a PI piezo stage, which is controlled by an E-727 piezo microcontroller from Physik Instrumente GbmH. The controller is connected to the computer via USB. The stage has a range of 100 µm in x, y, and z direction. To allow pre-positioning, the whole stage itself can be moved in the millimeters range along all three axes with three Thorlab Kinesis stepper motors (see figure 1).

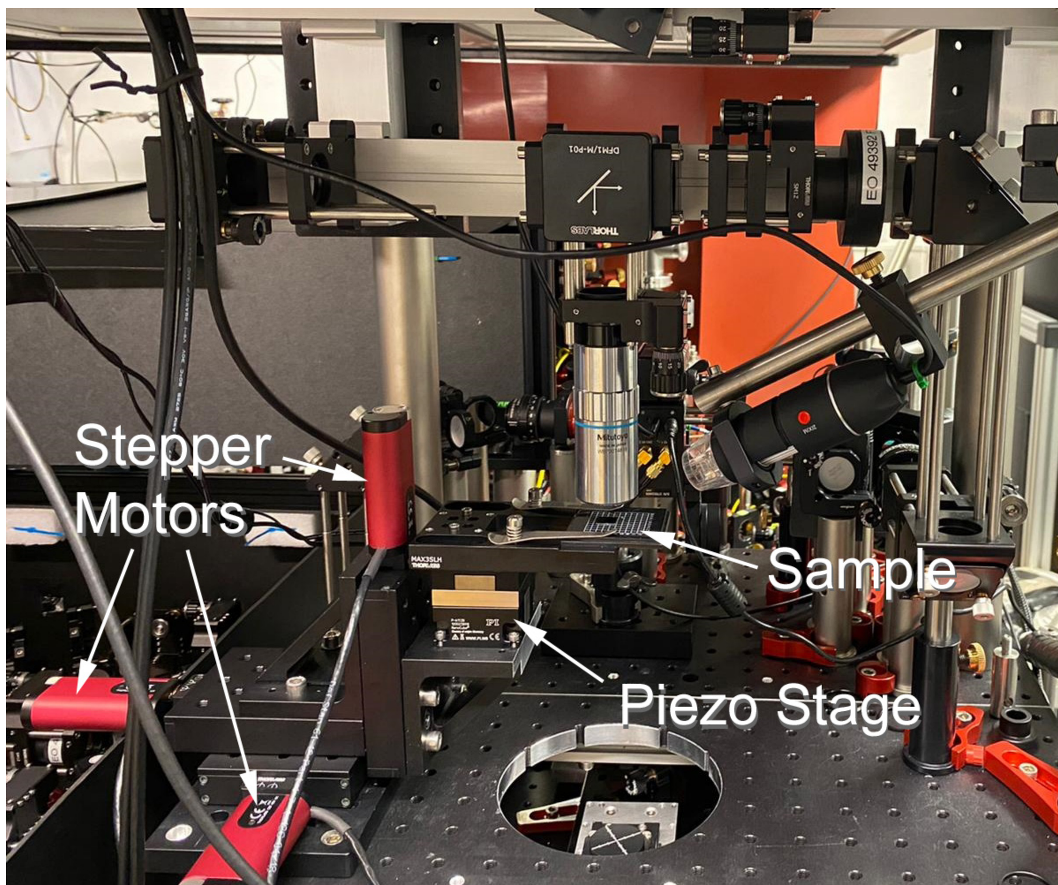


Figure 1: Stepper motors, piezo stage and sample holder

Each of the stepper motors is controlled by a K-Cube Stepper Motor Controller. These controllers all connect to the PC via a Thorlab K-Cube USB Controller Hub (see figure 2).

Figure 2: K-Cube Stepper Motor Controllers

In its final configuration, the confocal microscope will include two main beam paths for an objective facing the sample from top, and another objective facing the sample from below. As of completion of this paper, only the objective facing the sample from above is included and operable. In this upper beam path, the beam is guided from the objective ($50\times$) via an total reflective mirror through a 200 mm focal lens and a scan lens towards a laser scan mirror.

This laser scan mirror offers an alternative method for scanning the sample. It is controlled by a piezo controller, which reacts to analog voltage signals from a NI (National Instruments) IO card plugged directly into the personal computer. Also, analog voltage signals indicating the current *actual* position are fed back into the NI IO card from the controller.

For imaging, the beam path leads through a dichroic mirror which separates the fluorescence from the excitation beam. For first applications it reflects light with wavelength $\lambda < 550\,\text{nm}$ and transmits light of wavelength $\lambda > 550\,\text{nm}$. This allows to filter out and (further down) detect light coming from molecular fluorescence emitters.

There are two ways of imaging: Firstly, the confocal microscope allows to take wide-field images with an Andor iXon Ultra 897 camera from Oxford Instruments, connected to the computer via USB. The corresponding beam path is depicted with a red dotted line

in figure 3, and must be switched active with a flipmount mirror (red movable mirror in figure 3).

In wide-field mode, a 125 mm lens is moved into the wide-field illumination path (right red movable lens in figure 3), so that there is a focus on the back focal plane of the objective, and consequently a collimated light beam comes out of the objective. An equivalent lens must also be added to the imaging path in wide field mode (left red movable lens in figure 3).

Secondly, there is the confocal mode, depicted with the green dotted line in figure 3. The confocal imaging beam path leads through a pinhole between two lenses (see figure 3). Then outcoming collimated beam finds its way to a *COUNT 50N* Single Photon Counting Module (SPCM) from Laser Components, connected to a Time Tagger (Swabian Instruments) device. The Time Tagger device is connected to the computer via USB.

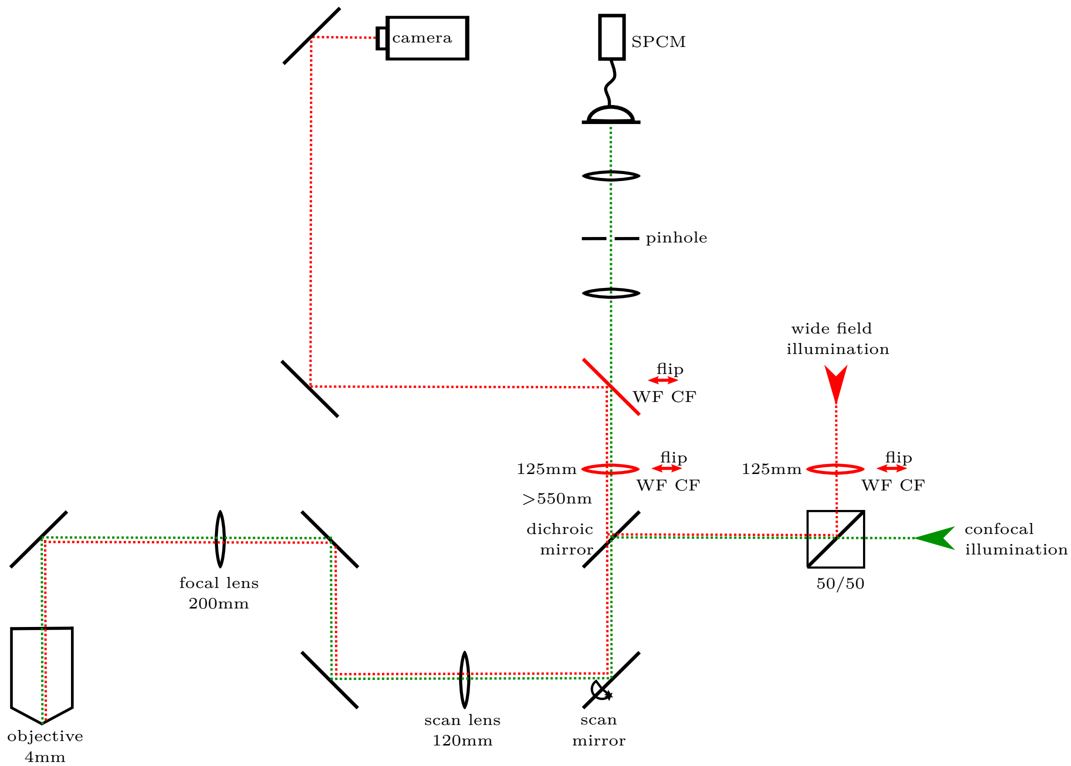


Figure 3: Beam paths in the confocal microscope. The confocal beam path is displayed in green, the wide-field beam-path in red. The two movable lenses and the movable mirror displayed in red color are depicted in the wide-field imaging position, and must be flipped out of the beam-path for confocal imaging.

2.1 Axes Orientation

The orientation of the scan axes is generally subject to both the specific hardware as well as to the specific configuration.

Having said that, Fig. 4 shows the orientation of axes of the confocal microscope hardware at the Institute of Atomic and Subatomic Physics in Vienna according to the software configuration as of January 2021.

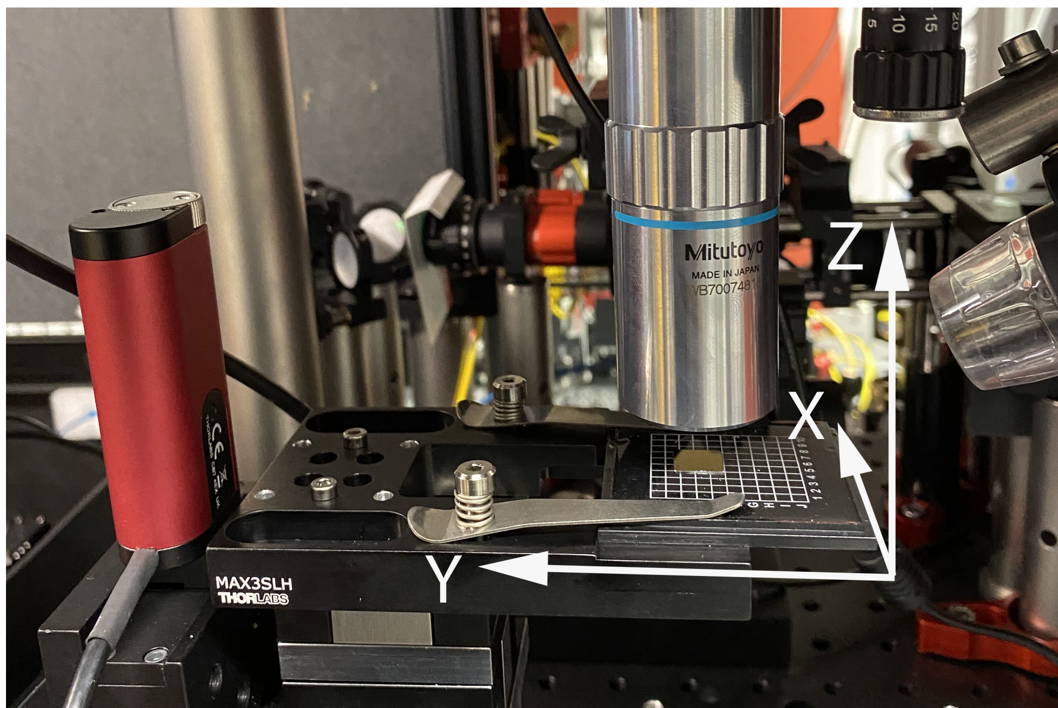


Figure 4: orientation of axes according to current configuration (Jan. 2021)

Please note that the x-axis and y-axis directions refer to features on the sample. This means, for example, that features more on the left side of the sample are located at ever larger values of the y-axis. However, the according stage movement during a stage-scan has to be necessarily towards the opposite direction: The stage must move *right* (towards negative y-direction) so that features on the left side of the sample come into the scan focus. The same is analogously true for the x-axis.

3 COMICS Software Design

3.1 Introduction to the Qudi Framework

The COMICS software package is based on the Qudi framework, which defines itself to be "a Python software suite for controlling complex experiments and managing the acquisition and processing of measurement data" (see [Binder et al, 2017, p. 86]).

The Qudi framework allows a collection of software modules to be loaded and connected to one another by a manager component according to the settings specified in a configuration file.

A Qudi session proceeds as follows: A user starts the `start.py` file. This loads the core components that take care of configuration parsing, module management, and error logging. The module manager reads a configuration file to determine which modules should be launched and configured for a specific experiment. (see Fig. 5a).

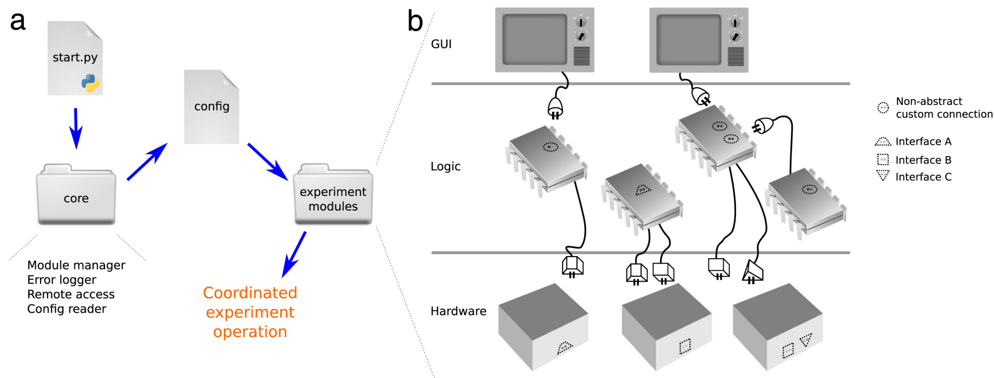


Figure 5: Qudi functional and structural design. From: [Binder et al, 2017, p. 87]

Experiment software modules in Qudi are divided into three categories: hardware modules, experiment "logic", and user interface. These categories and the relationships between them are illustrated in Figure 5b.

All logic modules connect down to hardware modules via well-defined interfaces. This means that a logic module does not care about specific hardware details, as long as a matching hardware module implementing the required interface is available.

It is noteworthy that a hardware module may implement multiple interfaces; e.g. an analog/digital IO hardware module may implement an interface for positioning an laser-mirror via it's analog output channels, as well as another interface for registering counts from an SPCM via its digital input lines.

3.2 COMICS software architecture overview

Figure 6 gives an overview of the COMICS architecture:

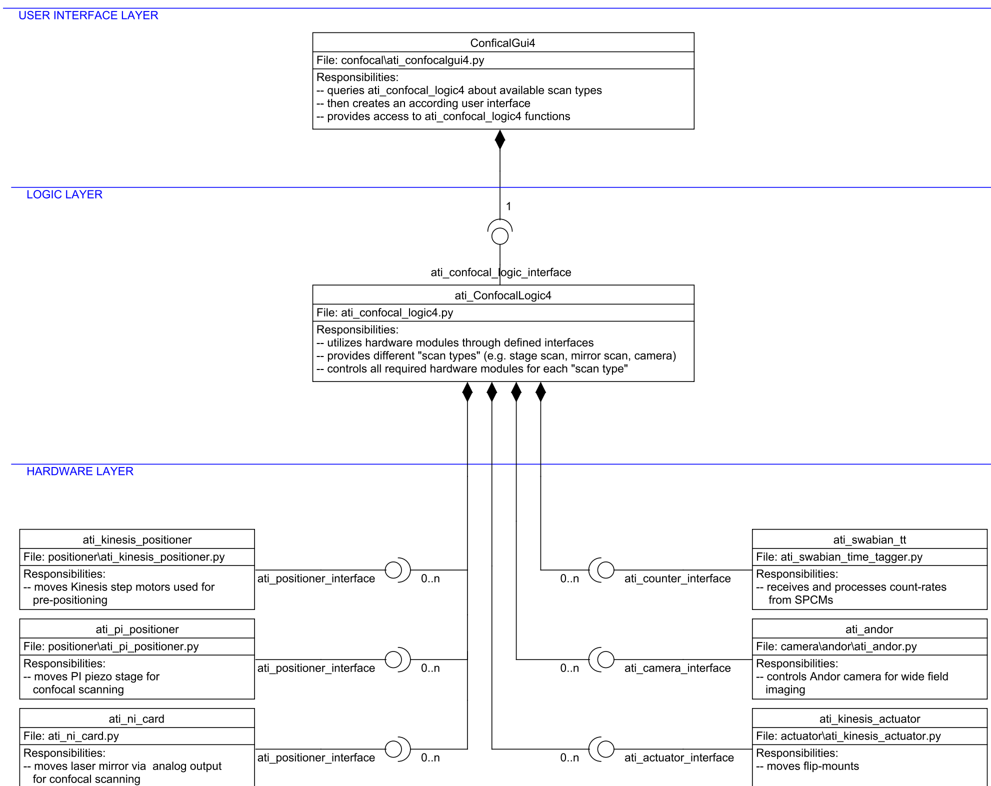


Figure 6: COMICS software architecture

When starting the user interface from the Qudi module manager, an instance of class `ConfocalGui4` is created. On instancing the class determines from the corresponding section of the Qudi configuration file to which logic module it should connect to.

The logic module class implementing the `ati_confocal_logic_interface` is `ati_ConfocalLogic4`. After instancing, this class retrieves all so-called *scan types* from the corresponding section in the Qudi configuration file. Each *scan types* has a unique name (e.g. "Mirror Scan" or "Stage Scan"), and the configuration file determines which positioning hardware should be used for each of these *scan types*.

Also, the configuration file determines for each *scan type* the hardware modules through which count-rates should be received, and (optionally) which positioning hardware should be used for pre-positioning. Further, a control sequence can be stored in the configuration file for each *scan type*, specifying actions for actuators to be executed before a scan

is started.

Wide field imaging through a camera module is also supported and can be integrated as a separate *scan type*. In such case, instead of positioning and photon-counting hardware classes, a camera hardware module class is referenced.

On startup, the GUI class instance of `ConfocalGui4` queries the connected `ati_ConfocalLogic4` class instance about number, name and features of all *scan types*, and builds up the graphical user interface correspondingly (with a tab for each *scan type* and all required user controls).

Depending on the hardware component classes referenced in the corresponding configuration-file section of each *scan type*, the `ati_ConfocalLogic4` class instance will connect to various hardware modules by means of well-defined class interfaces.

Every positioning hardware is accessed through a class implementing the `ati_positioner_interface`. Currently, the following module classes implement this interface:

- The `ati_kinesis_positioner` class for controlling Kinesis stepper motors;
- the `ati_PI_positioner` class for controlling a PI piezo stage; and
- the `ati_ni_card` class for controlling a piezo mirror by means of analog IO channels.

The following hardware module classes have been additionally implemented:

- The `ati_sqbian_tt` class with an `ati_counter_interface`, allowing access to the Swabian time-tagger hardware for retreiving and processing SPCM count-rates;
- the `ati_andor` class with an `ati_camera_interface`, providing access to Andor cameras;
- and the `ati_kinesis_actuator` class with an `ati_actuator_interface`, allowing to control Kinesis flip-mounts.

4 Interfaces for Hardware Layer Classes

In this chapter we describe the above-mentioned interfaces for the various hardware module classes.

4.1 ati_positioner_interface

The `ati_positioner_interface` needs to be implemented by all hardware module classes used for controlling positioning devices like stages, mirrors, stepper motors, etc.

This chapter describes the required class methods comprising the interface.

4.1.1 get_name()

Returns a string with the name of the positioning device. This method is expected to return a meaningful name already after the class has been instanced.

Return value:

- `str`: name of device.

See also:

- `set_name(new_name)`
- `get_pos_device_name(channel_cnf)`

4.1.2 set_name(new_name)

This method can be used to change the name of the device.

Parameter:

- `str new_name`: new name of the device.

See also:

- `get_name()`

4.1.3 get_pos_ch_count()

A single instance of a class implementing this `ati_positoner_interface` can not only be connected to a single positioning device, but (potentially) also to a controller that actually controls several positioning devices simultaneously. An example for that would be an analog IO card controlling two separate laser mirrors (each with two axes).

Each device is accessed through a separate "channel", numbered from 1 to n (with n being the number of devices). The method `get_pos_ch_count()` returns the number of channels (hence, the number of controllable devices):

Return value:

- `int`: number of channels, i.e.: number of controllable positioning devices.

4.1.4 get_pos_axis_count(channel_cnf)

This function returns the number of axes of the device identified by channel index `channel_cnf`.

Parameter:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `int`: number of axis.

4.1.5 get_pos_device_name(channel_cnf)

In case an instance of a class implementing this interface is representing multiple positioning devices simultaneously, the `get_name()` method is not sufficient, as it returns only one name.

In such case, method `get_pos_device_name(channel_cnf)` can be used to get the specific name of the individual positioning device identified by channel index `channel_cnf`.

Parameter:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `str`: name of positioning device.

See also:

- `get_name()`

4.1.6 get_pos_range(channel_cnf)

This method returns the physical range of all axes of the positioning device identified by channel index `channel_cnf`.

Parameter:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `list[n][2]`: A list with n entries (where n is the number of axes of the positioning device identified by channel index `channel_cnf`). Each entry consists of another list with two float values, representing the lower and upper bound of the axis' physical range in meters.

4.1.7 get_pos_axes(channel_cnf)

This function returns a list containing the names of all axes of the positioning device identified by channel index `channel_cnf`.

Parameter:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `list[n]`: A list with n entries (where n is the number of axes of the positioning device identified by channel index `channel_cnf`). Each entry consists of a string value with the name of the corresponding axis (e.g. `['X','Y','Z']`).

4.1.8 set_pos_position(channel_cnf, x=None, y=None, z=None, a=None)

This method moves the positioner referenced by channel index `channel_cnf` along one or more axes. The target positions of the axes to be moved must be passed on to the method via the four optional parameters `x`, `y`, `z`, and `a`; where `x` stands for the devices' first axes, `y` stands for the devices' second axes, `z` stands for the devices' third axes (if available) and `a` stands for any potentially available fourth axes (if available). Please note that the actual naming of the axes as returned by `get_pos_axes(channel_cnf)` may differ from "`x`", "`y`", and "`z`".

By default, the device should start to move immediately after this method has been called. However, it may also happen that the device is a slow-moving positioning device (like, stepper motors) and is currently re-setting (e.g. homing). Then, the movement is delayed and can be triggered at a later point in time by calling the method `pos_trigger_queue(channel_cnf)`.

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.
- `optional float x` : Target position of the first axis (in meters). Must be within the range returned by `get_pos_range(channel_cnf)[0]`.
- `optional float y` : Target position of the second axis (in meters). Must be within the range returned by `get_pos_range(channel_cnf)[1]`.
- `optional float z` : Target position of the third axis, if available (in meters). Must be within the range returned by `get_pos_range(channel_cnf)[2]`.
- `optional float a` : Target position of a potential fourth axes, if available (in meters). Must be within the range returned by `get_pos_range(channel_cnf)[3]`.

See also:

- `get_pos_position(channel_cnf, extra_precision=1)`
- `get_pos_target_position(channel_cnf)`
- `pos_matches_target_position(self, channel_cnf, xyz, pos)`
- `pos_is_moving(channel_cnf)`
- `pos_is_resetting(channel_cnf)`
- `pos_trigger_queue(channel_cnf)`

4.1.9 get_pos_position(channel_cnf, extra_precision=1)

Provided the positioner referenced by channel index `channel_cnf` has sensors to determine the current *actual* position (which can be queried by calling `pos_provides_cur_pos_info()`), this class method returns:

- information about the current position of all axes,
- information whether or not the current position of each axes currently matches the target position (withing tolerances), and
- for each axis information about how long the position already matches the target position within tolerances (or, zero, if not matching).

If the device does not have senors to determine the current actual position, this class method will just return the target position and assume that we are already on target.

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.
- `optional float extra_precision=1`: Any value $n > 1$ handed over here will cause the "within tolerance" information to be n times more "picky". E.g. a value of 2 means that a current axis position will be considered to be within tolerance only if it is within 0.5 times the default tolerance.

Return values (tuple):

- `float list[n]`: A list with n entries, where n stand for the number of axes. The list contains the current positions of each axis in meters. If the device has no sensors to determine the actual positions, the list will just contain the target positions. For each axis, the list contains the boolean
- `bool list[n]`: A list with n entries, where n stand for the number of axes. For each axis, the list contains `True`, if the current axis' position is on target (within tolerance), otherwise `False`. If the device has no sensors to determine the actual positions, the list will just contain `True` values.
- `float list[n]`: A list with n entries, where n stand for the number of axes. For each axis, the list contains the duration in seconds for which the actual position is already matching the target position (within tolerance). If, for an axis, the current position is *not* matching the target position, a value of 0 is returned. If the device has no sensors to determine the actual positions, the list will just contain the duration in seconds since calling of `set_pos_position(channel_cnf, x=None, y=None, z=None, a=None)` .

See also:

- `set_pos_position(channel_cnf, x=None, y=None, z=None, a=None)`
- `pos_provides_cur_pos_info()`
- `get_pos_target_position(channel_cnf)`
- `pos_matches_target_position(self, channel_cnf, xyz, pos)`

4.1.10 get_pos_target_position(channel_cnf)

This class method returns the target positions of the positioner referenced by channel index `channel_cnf`.

Parameter:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `float list[n]`: A list with n entries, where n stand for the number of axes. The list contains the target positions of each axis in meters. If no target position has been defined for an axis, the list will contain an `None` entry for this axis.

See also:

- `set_pos_position(channel_cnf, x=None, y=None, z=None, a=None)`

4.1.11 pos_matches_target_position(self, channel_cnf, xyz, pos)

This class method can be used to determine whether a given position `pos` matches the target position of a specific axis of a specific device (plus/minus minor rounding deviations).

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.
- `int xyz`: Index identifying the axis. Must be larger or equal 0 and smaller the number of available axis as returned by `len(get_pos_axes(channel_cnf))`.
- `float pos`: Position in meters to which the target position should be compared to.

Return value:

- `bool`: `True`, if the given position `pos` matches the target position of the axis referenced by `xyz_index` of the device referenced by `channel_cnf` (plus/minus minor rounding deviations). `False` otherwise, or if `pos` is `None`, or if no target position has yet been set.

See also:

- `set_pos_position(channel_cnf, x=None, y=None, z=None, a=None)`

4.1.12 get_pos_ini_position(channel_cnf, xyz)

This class method can be used to determine the initial "neutral" position for a specific axis of a specific device.

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

- `int xyz`: Index identifying the axis. Must be larger or equal 0 and smaller the number of available axis as returned by `len(get_pos_axes(channel_cnf))`.

Return value:

- `float`: Initial position for this axis in meters.

4.1.13 pos_provides_cur_pos_info()

Return value:

- `bool`: `True` if the positioners controlled by this device can provide information about the actual current position, `False` otherwise.

See also:

- `get_pos_position(channel_cnf, extra_precision=1)`

- `pos_matches_target_position(self, channel_cnf, xyz, pos)`

4.1.14 pos_is_moving(channel_cnf)

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `bool list[n]`: A list with n entries, where n stand for the number of axes. For each axis, the list contains `True` if the positioner is still moving along the axis, or `False` otherwise. If the hardware does not provide this information, all list entries are always `False`.

See also:

- `set_pos_position(channel_cnf, x=None, y=None, z=None, a=None)`
- `get_pos_position(channel_cnf, extra_precision=1)`

4.1.15 pos_is_resetting(channel_cnf)

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `bool list[n]`: A list with n entries, where n stand for the number of axes. For each axis, the list contains `True` if the positioner is resetting along the axis, or `False` otherwise.

See also:

- `pos_home(channel_cnf)`

4.1.16 pos_allows_homing(channel_cnf)

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `bool`: `True` if the positioning device has a callable homing procedure, which moves all axis to a "home" position, or `False` otherwise.

See also:

- `pos_home(channel_cnf)`

4.1.17 pos_home(channel_cnf)

This class method triggers a "homing" procedure, which moves all axis to a "home" position, provided the device controller has such a procedure.

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

See also:

- `pos_allows_homing(channel_cnf)`
- `pos_is_resetting(channel_cnf)`

4.1.18 pos_trigger_queue(channel_cnf)

By default, the device should start to move immediately after the class method `set_pos_position(channel_cnf, x=None, y=None, z=None, a=None)` has been called. However, it may also happen that the device is a slow-moving positioning device (like, stepper motors) and is currently re-setting (e.g. homing). Then, the movement is delayed and can be triggered at a later point in time by calling the method `pos_trigger_queue(channel_cnf)`.

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

See also:

- `set_pos_position(channel_cnf, x=None, y=None, z=None, a=None)`
- `pos_is_resetting(channel_cnf)`

4.1.19 get_pos_default_unit(channel_cnf)

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `string`: The length unit to be used in an user interface ('m'... millimeters, 'u'... micrometers, 'n'... nanometers).

See also:

- `get_pos_precision(channel_cnf)`

4.1.20 get_pos_precision(channel_cnf)

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.

Return value:

- `int`: Number of decimal points to be display in an user interface when using the length unit as returned by `get_pos_default_unit(channel_cnf)`.

See also:

- `get_pos_default_unit(channel_cnf)`

4.1.21 get_pos_steps_default(channel_cnf, axis_cnf)

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.
- `int axis_cnf`: Index identifying the axis. Must be larger or equal 1 and smaller or equal the number of available axis as returned by `len(get_pos_axes(channel_cnf))`.

Return value:

- `int`: Suggested default number of steps along this axis in a scan process.

See also:

- `get_pos_steps_min(channel_cnf, axis_cnf)`
- `get_pos_steps_max(channel_cnf, axis_cnf)`

4.1.22 get_pos_steps_min(channel_cnf, axis_cnf)

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.
- `int axis_cnf`: Index identifying the axis. Must be larger or equal 1 and smaller or equal the number of available axis as returned by `len(get_pos_axes(channel_cnf))`.

Return value:

- `int`: Minimum number of steps along this axis in a scan process.

See also:

- `get_pos_steps_default(channel_cnf, axis_cnf)`
- `get_pos_steps_max(channel_cnf, axis_cnf)`

4.1.23 get_pos_steps_max(channel_cnf, axis_cnf)

Parameters:

- `int channel_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_pos_ch_count()`.
- `int axis_cnf`: Index identifying the axis. Must be larger or equal 1 and smaller or equal the number of available axis as returned by `len(get_pos_axes(channel_cnf))`.

Return value:

- `int`: Maximum number of steps along this axis in a scan process.

See also:

- `get_pos_steps_default(channel_cnf, axis_cnf)`
- `get_pos_steps_min(channel_cnf, axis_cnf)`

4.1.24 close_positioner()

This class function is to be called when unloading the software. It shuts down the connection to the hardware.

4.2 ati_counter_interface

The `ati_counter_interface` needs to be implemented by all hardware module classes used for controlling count rate detector devices like SPCMs.

These are the required class methods comprising the interface:

4.2.1 get_name()

Returns a string with the name of the photon counter device. This method is expected to return a meaningful name already after the class has been instanced.

Return value:

- `str`: name of device.

See also:

- `set_name(new_name)`
- `get_pcnt_channel_names(source_cnf)`

4.2.2 set_name(new_name)

This method can be used to change the name of the device.

Parameter:

- `str new_name`: New name of the device.

See also:

- `get_name()`

4.2.3 get_pcnt_sources_count()

A single instance of a class implementing this `ati_counter_interface` can not only be connected to a single (photon) counting device (SPCM), but (potentially) also to a controller that actually processes the input from several sources simultaneously. A "source" may be an actual single hardware device, which in itself may provide one or more output streams, or alternatively a "source" is just a logical grouping of two or more separate devices.

Each source is identified by an individual "source" index, numbered from 1 to n (with n being the number of devices). The method `get_pcnt_sources_count()` returns the number of sources:

Return value:

- `int`: number of sources, i.e.: number of counting devices.

4.2.4 get_pcnt_channel_count(source_cnf)

Every "source" may provide one or more data streams, called "channels". This could be, for example, the count-rates of two separate SPCM's which just have been grouped together in a "source", or it also may be the naked count-rate of just a single SPCM, combined with the somehow processed count-rate (e.g. counting only somehow correlated counts).

The class method `get_pcnt_channel_count(source_cnf)` returns the number of such channels provided by a specific source:

Parameter:

- `int source_cnf`: Index identifying the source. Must be larger or equal 1 and smaller or equal the number of available spurces as returned by `get_pcnt_sources_count()`.

Return value:

- `int`: number of channels for this source, i.e.: number of data streams from this source.

See also:

- `get_pcnt_channel_names(source_cnf)`

4.2.5 get_pcnt_channel_names(source_cnf)

Every "source" may provide one or more data streams, called "channels". This could be, for example, the count-rates of two separate SPCM's which just have been grouped together in a "source", or it also may be the naked count-rate of just a single SPCM, combined with the somehow processed count-rate (e.g. counting only somehow correlated counts).

The class method `get_pcnt_channel_names(source_cnf)` returns the names of such channels provided by a specific source:

Parameter:

- `int source_cnf`: Index identifying the source. Must be larger or equal 1 and smaller or equal the number of available spurces as returned by `get_pcnt_sources_count()`.

Return value:

- `str list[n]`: A list with n entries (where n is the number of channels for this source). Each list entry consists of a text string representing the channel's name.

See also:

- `get_pcnt_channel_count(source_cnf)`

4.2.6 get_pcnt_rate(source_cnf)

This class method triggers a new measurement for the given source, waits for the measurement to be completed, and then returns a list containing the count-rates of all channels of the specified source.
Parameter:

- `int source_cnf`: Index identifying the source. Must be larger or equal 1 and smaller or equal the number of available spurces as returned by `get_pcnt_sources_count()`.

Return value:

- `float list[n]`: A list with n entries (where n is the number of channels for this source). Each list entry contains the count-rate for the corresponding channel.

See also:

- `update_cur_count_rate(output_source_cnf, channel_idx)`
- `get_last_cnt_rates()`

4.2.7 update_cur_count_rate(output_source_cnf, channel_idx)

This class method first checks if a measurement that has already been started before (by a call of this very function) is currently still in progress. If so, it just immediately returns control to the calling instance.

If no, then it assumes that a new count-rate is available, and the result is retrieved and can be afterwards accessed by means of the `get_last_cnt_rates()` method. Then, a new measurement is started, but the function does not wait for the measurement to finish, but rather immediately returns control to the calling instance.

Therefore, in contrast to `get_pcnt_rate(source_cnf)`, which blocks until a measurement is finished, this class method can be used (together with `get_last_cnt_rates()`) for regular polling of count-rates without blocking the software (even if a single measurement takes a significant amount of time).

Parameter:

- `int source_cfg`: Index identifying the source. Must be larger or equal 1 and smaller or equal the number of available sources as returned by `get_pcnt_sources_count()`.
- `int channel_idx`: Index identifying a source's channel. Must be larger or equal 0 and smaller than the number of available channels for this source as returned by `get_pcnt_channel_count(source_cnf)`.

See also:

- `get_pcnt_rate(source_cnf)`
- `get_last_cnt_rates()`

4.2.8 get_last_cnt_rates()

This class method returns a list containing the latest available count-rates of all channels of the specified source. It does *not* start a new measurement and is intended to be called after `update_cur_count_rate(output_source_cnf, channel_idx)`. However, it can also be called after `get_pcnt_rate(source_cnf)` to retrieve the latest count-rates again.

Return value:

- `float list[n]`: A list with n entries (where n is the number of channels for this source). Each list entry contains the count-rate for the corresponding channel.

See also:

- `update_cur_count_rate(output_source_cnf, channel_idx)`
- `get_pcnt_rate(source_cnf)`

4.2.9 close_counter()

This class function is to be called when unloading the software. It shuts down the connection to the hardware.

4.3 ati_camera_interface

The `ati_camera_interface` needs to be implemented by all hardware module classes used for controlling a camera. Currently the interface is designed to control a single camera per class instance. It is partially based on the code in `camera_interface.py` in the standard Qudi distribution.

These are the required class methods comprising the interface:

4.3.1 get_name()

Returns a string with the name of the camera.

Return value:

- `str`: Name of camera.

4.3.2 is_connected()

Return value:

- `bool`: Returns `True`, if the class instance could connect successfully to a camera, otherwise `False`.

4.3.3 get_size()

This class method returns a tuple representing the (currently configured) size of the image.

Return values (in a tuple):

- `int`: width (in pixels).
- `int`: height (in pixels).

See also:

- `get_resolutions()`
- `set_resolution(res_index)`

4.3.4 get_resolutions()

This class method returns a list with all allowed resolutions.

Return value:

- `list[n]`: A list with n entries, where n stands for the number of possible resolutions. Each list entry represents an allowed resolution and contains a list with two `int` values representing the x- and y resolution in pixels.

  For example, a camera allowing the resolutions 512×512 and 256×256 pixels, would return the list `[[512,512], [256,256]]`

See also:

- `set_resolution(res_index)`
- `get_size()`

4.3.5 set_resolution(res_index)

This class method allows to set the resolution to one of the values returned by `get_resolutions()`.

Parameter:

- `int res_index`: Index representing one of the allowed resolutions as returned by the function `get_resolutions()`. Must be larger or equal 0 and smaller than `len(get_resolutions())`.

See also:

- `get_resolutions()`
- `get_size()`

4.3.6 start_single_acquisition(wait_for_acquisition)

This class method starts the acquisition of a single camera image (either synchronously or asynchronously). The image can then be retrieved with `get_acquired_data()`.

Parameter:

- `int wait_for_acquisition`: If `True`, the method waits until the image has been taken. This is not recommended for long exposure times, as it blocks the software. If `False`, the image acquisition is started and then control is immediately given back to the calling instance. By calling the class method `is_acquiring()`, it can be checked whether the acquisition is still in progress.

Return value:

- `bool`: `True` if successful, otherwise `False`.

See also:

- `is_acquiring()`
- `stop_acquisition()`
- `get_acquired_data()`
- `get_ready_state()`
- `set_exposure(exposure)`
- `set_shutter(index)`

4.3.7 is_acquiring()

Return value:

- `bool`: Returns `True` if an image requisition started previously with with `start_single_acquisition(False)` is still in progress, otherwise `False`.

See also:

- `start_single_acquisition(wait_for_acquisition)`

4.3.8 stop_acquisition()

This method prematurely aborts an image requisition started previously with with `start_single_acquisition(False)`.

See also:

- `start_single_acquisition(wait_for_acquisition)`

4.3.9 get_acquired_data()

This class method returns the image previously acquired by calling the function `start_single_acquisition(wait_for_acquisition)`.

Return value:

- `int numpy.array`: Numpy array of shape (`hres`, `vres`), with `hres` being the horizontal resolution, and `vres` being the vertical resolution, containing the gray-scale values of the acquired image as integer values.

See also:

- `start_single_acquisition(wait_for_acquisition)`

4.3.10 get_ready_state()

Return value:

- `bool`: Returns `True` if the camera is ready to acquire a new image, otherwise `False`.

See also:

- `start_single_acquisition(wait_for_acquisition)`

4.3.11 set_cooler(state)

Parameter:

- `bool state`: `True` to turn the camera cooler on; `False` to turn it off.

Return value:

- `bool`: Returns `True` if successful, otherwise `False`.

See also:

- `is_cooler_on()`
- `set_temperature(temp)`

4.3.12 is_cooler_on()

Return value:

- `bool`: Returns `True` if the camera cooler is turned on, or `False` if it is turned off. It also returns `False`, if it is in an unknown state.

See also:

- `set_cooler(state)`

4.3.13 set_temperature(temp)

This class method sets the camera's target temperature in °C. This only takes effect if the cooler has been turned on with `set_cooler(True)`.

Parameter:

- `int temp`: Target temperature in °C

Return value:

- `bool`: `True` if successful, otherwise `False`.

See also:

- `set_cooler(state)`
- `is_cooler_on()`

4.3.14 provides_temperature_information()

Return value:

- `bool`: `True` if the camera provides temperature information, otherwise `False`.

See also:

- `get_temperature()`
- `get_temperature_precision()`

4.3.15 get_temperature()

Return values (tuple):

- `float`: Current temperature of the camera in °C.
- `int`:

  0 . . . no temperature range information available

  1 . . . the temperature is out of range, or within range but not yet stabilized

  2 . . . the temperature is within range and stabilized

See also:

- `set_temperature(temp)`
- `get_temperature_precision()`

4.3.16 get_temperature_precision()

Return value:

- `int`: Number of digits after the decimal point up to which the current temperature returned by `get_temperature()` is significant.

See also:

- `get_temperature()`

4.3.17 set_exposure(exposure)

Parameter:

- `float exposure`: exposure time in seconds

Return value:

- `bool`: `True` if successful, otherwise `False`.

See also:

- `get_exposure(get_actual_val_from_hardware = True)`
- `get_max_exposure()`

4.3.18 get_exposure(get_actual_val_from_hardware = True)

This class method returns the currently configured exposure time.

Parameter:

- `optional bool get_actual_val_from_hardware`: It `True` or not specified, then the actual exposure time is returned. This time may slightly deviate from the exposure time set with `set_exposure(exposure)`). If `False`, the exposure time as set with `set_exposure(exposure)`) is returned.

Return value:

- `float`: exposure time in seconds.

See also:

- `set_exposure(exposure)`

4.3.19 get_max_exposure()

Return value:

- `float`: maximum exposure time in seconds.

See also:

- `set_exposure(exposure)`

4.3.20 has_shutter()

Return value:

- `bool`: `True` if the camera has a mechanical shutter, `False` otherwise.

See also:

- `get_shutter_settings()`
- `set_shutter(index)`

4.3.21 get_shutter_settings()

This class method returns a list of all possible shutter settings.

Return value:

- `list[n]`: List with n entries, where n is the number of possible shutter settings. Each list entry represents a possible setting and consists of a sub-list with two entries: The first entry is an unique integer index, and the second entry a text string describing the setting. This is an example of a list for two allowed shutter settings: `[[0, 'Fully Auto'], [1, 'Permanently Open']]`

See also:

- `get_default_shutter_setting()`
- `set_shutter(index)`

4.3.22 get_default_shutter_setting()

Return value:

- `int`: Index of the default shutter setting out of the indexes returned by `get_shutter_settings()`.

See also:

- `get_shutter_settings()`
- `set_shutter(index)`

4.3.23 set_shutter(index)

Parameter:

- `int index`: Index of the shutter setting to be set. Must be one of the indexes returned by `get_shutter_settings()`.

See also:

- `get_shutter_settings()`
- `get_default_shutter_setting()`

4.3.24 get_additional_settings()

This class method returns a dictionary containing detailed information about all additional camera-specific settings not covered by dedicated class methods of this interface.

The main structure of this dictionary looks like this:

```
{'group_1' : {'label' : <label>,
              'setting_1' : <sdef>,
              'setting_2' : <sdef>, ... }
 'group_2' : {'label' : <label>,
              'setting_1' : <sdef>,
              'setting_2' : <sdef>, ... }
 ...
}
```

As shown above, all settings are assigned to groups. These groups have keys `'group_1'`, `'group_2'`, etc. in successive order. The placeholder `<label>` stands for a human-readable name of the corresponding group that could be used in an user interface, e.g. as the title of a frame.

The settings in each group have keys `'setting_1'`, `'setting_2'`, etc. in successive order. Each placeholder `<sdef>` represents a dictionary describing the properties of the setting. This is the structure of a `<sdef>` entry:

```
{'ID': <ID>, 'label': <label>, 'type': <type>,
'dependent_on': <depend_ID>, 'dependent_values': <depend_value_list>
'default_value': <default_value>, 'values': <value_list}
```

This is what above placeholders stand for:

- `string <ID>` ... Unique ID identifying this setting
- `string <label>` ... Human-readable designation that could be used in an user-interface, e.g. to be displayed before the corresponding input field.
- `string <type>` ... Indicates the type of the setting. Currently the only implemented type is `'list'` which represents a setting where the user has to choose from a list of possible options.
- `string <depend_ID>` ... Optional identifier to indicate that this setting is dependent on another setting For example: If the setting with ID `'B'` is only to be displayed if the setting with ID `'A'` has the values `1` or `2`, then the `<depend_ID>` of setting `'B'` would be `'A'`.
- `<depend_value_list>` ... Optional. If there is a `<depend_ID>`, then the `<depend_value_list>` placeholder stands for a list of values. For example: If

the setting with ID `'B'` is only to be displayed if the setting with ID `'A'` has the values `1` or `2`, then the `<depend_value_list>` should be `[1, 2]`

- `<default_value>` ... The default value of this setting. In case of a setting of `<type>:'list'` this should be the list index of the default value.
- `value_list` ... If the setting is of `<type>:'list'`, then the placeholder `value_list` stands for a list with n entries, where n is the number of possible list values. Each entry is yet another list with two entries, the first being a unique integer index, the second being a unique text string. For example, this is how the `value_list` would look like for two possible options: `[[0, 'Electron Multiplying'], [1, 'Conventional']]`

Here is an example of a complete additional settings dictionary:

```
{'group_1':
  {'label' :'Vertical Pixel Shift',
   'setting_1':
     {'ID': 'vs_speed',
      'label': 'Shift Speed',
      'type': 'list',
      'default_value': 4,
      'values': [[0, '0.3μs'], [1, '0.5μs'], [2, '0.9μs']]}},
 'group_2':
  {'label' : 'Horizontal Pixel Shift',
   'setting_1':
     {'ID': 'output_amp',
      'label': 'Output Amplifier',
      'type': 'list',
      'default_value': 0,
      'values': [[0, 'Electron Multiplying'], [1, 'Conventional']]},
   'setting_2':
       {'ID': 'hs_speed0',
        'dependent_on': 'output_amp',
        'dependent_values': [0],
        'label': 'Readout Rate',
        'type': 'list',
        'default_value': 0,
        'values': [[0, '17.0μs'], [1, '10.0μs'], [2, '5.0μs']]}
    'setting_3':
      {'ID': 'hs_speed1',
       'dependent_on': 'output_amp',
       'dependent_values': [1],
       'label': 'Readout Rate',
       'type': 'list',
```

```
        'default_value': 0,
        'values': [[0, '3.0μs'], [1, '1.0μs'], [2, '0.08μs']]}
  }
}
```

Return value:

- `dictionary`: Dictionary with additional settings as explained above.

See also:

- `set_additional_settings(ID, value)`

4.3.25 set_additional_settings(ID, value)

This class method allows to set one of the "additional setting" as defined in the dictionary returned by `get_additional_settings()`.

Parameters:

- `string ID`: The `<ID>` of the "additional setting" as defined in the dictionary returned by `get_additional_settings()`.
- `int value`: In case of settings of `<type>:'list'`, this is the index of the chosen list-entry.

See also:

- `get_additional_settings()`

4.4 ati_actuator_interface

The `ati_actuator_interface` needs to be implemented by all hardware module classes used for controlling actuators like, for example, flip-mounts.

These are the required class methods comprising the interface:

4.4.1 get_name()

Returns a string with the name of the actuator. This method is expected to return a meaningful name already after the class has been instanced.

Return value:

- `str`: name of actuator.

See also:

- `set_name(new_name)`

4.4.2 set_name(new_name)

This method can be used to change the name of the actuator device.

Parameter:

- `str new_name`: new name of the actuator device.

See also:

- `get_name()`

4.4.3 get_act_ch_count()

A single instance of a class implementing this `ati_ActuatorInterface` can not only represent a single actuator device, but (potentially) also multiple actuator devices simultaneously. An example for that would be an analog IO card controlling several separate flip-mounts.

Each device is accessed through a separate "channel", numbered from 1 to n (with n being the number of devices). The method `get_act_ch_count()` returns the number of channels (hence, the number of controllable actuator devices):

Return value:

- `int`: number of channels, i.e.: number of controllable actuator devices.

See also:

- `get_act_ch_name(ch_cnf)`

4.4.4 get_act_ch_name(ch_cnf)

In case an instance of a class implementing this interface is representing multiple actuator devices simultaneously, the `get_name()` method is not sufficient, as it returns only one name.

In such case, `get_act_ch_name(ch_cfg)` can be used to get the specific name of the individual actuator identified by channel index `ch_cfg`.

Parameter:

- `int ch_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_act_ch_count()`.

Return value:

- `str`: name of actuator.

See also:

- `get_act_ch_count()`
- `get_act_ch_output(ch_cnf)`

4.4.5 get_act_ch_output(ch_cnf)

In case an instance of a class implementing this interface is representing multiple actuator devices simultaneously, the method returns a string representation of the actual hardware channel used to access the actuator identified by `ch_cfg`.

For example, if the class instance represents an National Instruments IO card, this function returns `'line2'` for digital output number 2, or `'AO1'` for analog output number 1. Or, if the class instance represents multiple Kinesis flip-mounts, then this function returns the serial number of the flip-mount.

Parameter:

- `int ch_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_act_ch_count()`.

Return value:

- `str`: Text string representation of the actual hardware channel.

See also:

- `get_act_ch_name(ch_cnf)`

4.4.6 get_act_ch_is_digital(ch_cnf)

Parameter:

- `int ch_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_act_ch_count()`.

Return value:

- `bool`: Returns `True` if the actuator identified by channel index `ch_cfg` has only two states and therefore `set_act_output(ch_cnf, value)` expects `value` to be boolean (with `0` or `False` representing one actuator position, and `1` or `True` representing the other position). Returns `False` if the actuator has more than two positions, or is controlled with an analog (float) value.

See also:

- `set_act_output(ch_cnf, value)`
- `get_act_ch_output(ch_cnf)`

4.4.7 set_act_output(ch_cnf, value)

Parameters:

- `int ch_cnf`: Channel index identifying the device. Must be larger or equal 1 and smaller or equal the number of available channels as returned by `get_act_ch_count()`.
- `float/int/bool value`: This value indicates which state the actuator should take on. If `get_act_ch_is_digital(ch_cnf)` returned `True`, this `value` must be `0` or `False` for the actuator to take on the first of two states, and `1` or `True` for the actuator to take on the second of two states. If `get_act_ch_is_digital(ch_cnf)` returned `False`, this `value` can be `integer` or `float`, depending on the hardware.

See also:

- `get_act_ch_is_digital(ch_cnf)`

5 Logic Layer Class Interface

In this chapter we describe the the `ati_confocal_logic_interface` by which the confocal GUI layer can access the logic layer.

5.1 Remarks on Absolute and Relative Coordinates

Many of the following class methods processing coordinates expect a boolean parameter named `absolute`. This is the purpose of that parameter:

If `absolute` is set to `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device in question. The current position of any other positioning device or pre-positioning device is not considered.

If, however, `absolute` is set to `True`, then - depending on the current configuration - the positions of other positioning devices and pre-positioning devices may also be taken into account.

For example, if `absolute` is set to `False`, the coordinate $x = 50\,\mu\text{m}$ may identify the x-center position of a stage with a total range of $100\,\mu\text{m}$ in x-direction. However, if at the same time a pre-positioning stepper motor has moved the whole stage to $x = 3\,\text{mm}$ and `absolute = True`, then the very same x-center position of the stage has to be referenced as $x = 3.05\,\text{mm}$.

5.2 General Scan Methods

5.2.1 get_scan_type_count()

This method returns the number of available *scan types*, where each *scan type* has a unique identifying name, and typically represents a configuration for creating an image by scanning along two of two or three available axes with a positioning device like a stage or a laser mirror. Alternatively, a specific *scan type* can represent a configuration where the image is taken by a camera.

Many of the class methods below refer to a specific *scan type*, which needs to be identified by an `scan_type` index larger or equal zero and smaller than `get_scan_type_count()`.

Return value:

- `int`: Number of available *scan types*.

See also:

- `get_scan_type_max_index()`
- `get_scan_type_name(scan_type)`

5.2.2 get_scan_type_max_index()

This method returns the maximum `scan_type` index, which is exactly the value of `get_scan_type_count()` minus one.

Return value:

- `int`: Maximum `scan_type` index value.

See also:

- `get_scan_type_count()`
- `get_scan_type_name(scan_type)`

5.2.3 get_scan_type_name(scan_type)

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `str`: name of `scan_type`

5.2.4 set_cur_scan_type_index(scan_type)

In some of the class methods defined below, the `scan_type` parameter is optional. If in such a method the `scan_type` parameter is omitted, then the method assumes that it applies to the "current" scan type, which can be defined by `set_cur_scan_type_index`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

See also:

- `get_cur_scan_type_index()`

5.2.5 get_cur_scan_type_index()

Return value:

- `int`: Current *scan type* index, as set with `set_cur_scan_type_index(scan_type)`

See also:

- `set_cur_scan_type_index(scan_type)`

5.2.6 set_scan_center_position(scan_type, x=None, y=None, z=None, a=None, absolute = False)

This method defines a so-called "center position" for the positioning device linked to the `scan_type`. When a scan is finished (or interrupted), the axes involved in the scan return to this "center position".

For example, if a certain *scan type* is controlling a 3-axis piezo-stage, and if the "center position" of this xyz stage is set to 10 µm / 20 µm / 30 µm, then after a xy-scan the x-axis is re-set to 10 µm, and the y-axis to 20 µm, whereas the z-axis is left unchanged. However, after a yz-scan (a so-called depth-scan) the y-axis is re-set to 20 µm, the z-axis to 30 µm, and the x-axis is left unchanged.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `optional float x`: Center position of the first axis (in meters).
- `optional float y`: Center position of the second axis (in meters).
- `optional float z`: Center position of the third axis, if available (in meters).
- `optional float a`: Center position of a potential fourth axes, if available (in meters).
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred *scan type*, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Please note that the actual names of the axes may differ from "x", "y" and "z".

See also:

- `absolute_pos_coords_available(scan_type)`

5.2.7 set_scan_X_range(scan_type, new_range, absolute = False)

This method defines the scan range for the positioning device linked to the `scan_type` in the direction of the first axis. Please note that the actual name of the first axes may differ from "x". The next scan involving that axis will be executed covering this range.

This method does not apply to a `scan_type` representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `list new_range`: A list with two float values, representing the lower and upper bound of the next scan's scan range along the first axis (in meters).
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred *scan type*, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

See also:

- `set_scan_Y_range(scan_type, new_range, absolute = False)`
- `set_scan_Z_range(scan_type, new_range, absolute = False)`
- `get_scan_range(scan_type, xyz, absolute = False)`
- `get_pos_range(scan_type, xyz, absolute = False)`
- `absolute_pos_coords_available(scan_type)`

5.2.8 set_scan_Y_range(scan_type, new_range, absolute = False)

This method defines the scan range for the positioning device linked to the `scan_type` in the direction of the second axis. Please note that the actual name of the second axes may differ from "y". The next scan involving that axis will be executed covering this range.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `list new_range`: A list with two float values, representing the lower and upper bound of the next scan's scan range along the second axis (in meters).
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred *scan type*, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

See also:

- `set_scan_X_range(scan_type, new_range, absolute = False)`
- `set_scan_Z_range(scan_type, new_range, absolute = False)`
- `get_scan_range(scan_type, xyz, absolute = False)`
- `get_pos_range(scan_type, xyz, absolute = False)`
- `absolute_pos_coords_available(scan_type)`

5.2.9 set_scan_Z_range(scan_type, new_range, absolute = False)

This method defines the scan range for the positioning device linked to the `scan_type` in the direction of the third axis. Please note that the actual name of the third axes may differ from `"z"`. The next scan involving that axis will be executed covering this range.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `list new_range`: A list with two float values, representing the lower and upper bound of the next scan's scan range along the third axis (in meters).
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred *scan type*, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

See also:

- `set_scan_X_range(scan_type, new_range, absolute = False)`
- `set_scan_Y_range(scan_type, new_range, absolute = False)`
- `get_scan_range(scan_type, xyz, absolute = False)`
- `get_pos_range(scan_type, xyz, absolute = False)`
- `absolute_pos_coords_available(scan_type)`

5.2.10 get_scan_range(scan_type, xyz, absolute = False)

This method returns the (user-defined) scan range for the positioning device linked to the `scan_type` in the direction of the axis specified with the `xyz` parameter. Please note that these are not the scan range limits set by the hardware, but rather the scan range as defined by the user using the class methods `set_scan_X_range`, `set_scan_Y_range`, and `set_scan_Z_range`.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates for which axis the scan range should be returned. `xyz = 0` will return the scan range of the first axis, `xyz = 1` will return the scan range of the second axis, and `xyz = 2` will return the scan range of the third axis (if available).
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return value:

- `list[2]`: A list with two float values, indicating the lower and upper bound of the scan range (in meters).

See also:

- `set_scan_X_range(scan_type, new_range, absolute = False)`
- `set_scan_Y_range(scan_type, new_range, absolute = False)`
- `set_scan_Z_range(scan_type, new_range, absolute = False)`
- `get_pos_range(scan_type, xyz, absolute = False)`
- `absolute_pos_coords_available(scan_type)`

5.2.11 matches_target_position(scan_type, xyz, pos, absolute = False)

This class method can be used to determine whether a given position `pos` matches the target position of a specific axis of the positioning device linked to the `scan_type` (plus/minus minor rounding deviations).

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates the position of which axis should be compared with `pos`. Setting `xyz = 0` will compare the position of the first axis with `pos`, setting `xyz = 1` will compare the position of the second axis with `pos`, and setting `xyz = 2` will compare the position of the third axis with `pos` (if available).
- `float pos`: Position in meters to which the target position should be compared to.
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return value:

- `bool`: `True`, if the given position `pos` matches the target position of the axis referenced by `xyz_index` (plus/minus minor rounding deviations). `False` otherwise, or if `pos` is `None`, or if no target position has yet been set.

This method does not apply to a *scan type* representing a camera.

See also:

- `set_position(tag, scan_type=-1, x=None, y=None, z=None, a=None, absolute = False)`
- `absolute_pos_coords_available(scan_type)`

5.2.12 set_scan_resolution(scan_type, xyz, new_res)

This method defines the scan resolution (i.e. number of scan steps) for the given `scan_type` in the direction of the axis indicated by `XYZ_index`. The next scan involving that axis will be executed with the given number of scan steps along this axis.

The logic module can be configured so that the specific axis of the main positioning device of a *scan type* must always have the same resolution. If so, setting the resolution of one axis with this method may therefore also affect the resolution of another axes.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates along which axis the scan resolution should be set. `xyz = 0` sets the resolution for the first axis, `xyz = 1` sets the resolution for the second axis, and `xyz = 2` sets the resolution for the third axis (if available).
- `int new_res`: Scan resolution (i.e. number of scan steps). Must be larger or equal `get_min_resolution(scan_type, xyz)` and smaller or equal `get_max_resolution(scan_type, xyz)`.

See also:

- `same_scan_resolution(scan_type, xyz_1, xyz_2)`
- `get_scan_resolution(scan_type, xyz)`
- `get_min_resolution(scan_type, xyz)`
- `get_max_resolution(scan_type, xyz)`

5.2.13 same_scan_resolution(scan_type, xyz_1, xyz_2)

The logic module can be configured so that a scan along two specific axes of a *scan type* must always have the same scan resolution for both axes. This method returns whether or not two axes are correlated in that way.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz_1`: This indicates the first axis. Must be $\in \{0, 1\}$ for a main positioning device with two axes, or $\in \{0, 1, 2\}$ for a main positioning device with three axes.
- `int xyz_2`: This indicates the second axis. Must be $\in \{0, 1\}$ for a main positioning device with two axes, or $\in \{0, 1, 2\}$ for a main positioning device with three axes.

Return value:

- `bool`: `True`, if a scan along the axes indicated by `xyz_index_1` and `xyz_index_2` must always have the same resolution for both axes. `False` otherwise.

See also:

- `set_scan_resolution(scan_type, xyz, new_res)`
- `get_scan_resolution(scan_type, xyz)`

5.2.14 get_scan_resolution(scan_type, xyz)

This method returns the scan resolution (i.e. number of scan steps) for the given `scan_type` in the direction of the axis indicated by `XYZ_index` as previously set with `set_scan_resolution(scan_type, xyz, new_res)`.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates of which axis the scan resolution should be returned. `xyz = 0` returns the resolution for the first axis, `xyz = 1` returns the resolution for the second axis, and `xyz = 2` returns the resolution for the third axis (if available).

Return value:

- `int`: resolution (i.e. number of scan steps) along the specified axis.

See also:

- `set_scan_resolution(scan_type, xyz, new_res)`

5.2.15 get_min_resolution(scan_type, xyz)

This method returns the smallest scan resolution (i.e. minimum number of scan steps) for the given `scan_type` in the direction of the axis indicated by the `xyz`-index.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates of which axis the minimum scan resolution should be returned. `xyz = 0` returns the minimum resolution for the first axis, `xyz = 1` returns the minimum resolution for the second axis, and `xyz = 2` returns the minimum resolution for the third axis (if available).

Return value:

- `int`: minimum resolution (i.e. smallest number of scan steps) along the specified axis.

See also:

- `get_max_resolution(scan_type, xyz)`
- `set_scan_resolution(scan_type, xyz, new_res)`

5.2.16 get_max_resolution(scan_type, xyz)

This method returns the largest scan resolution (i.e. maximum number of scan steps) for the given `scan_type` in the direction of the axis indicated by `XYZ_index`.

This method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates of which axis the maximum scan resolution should be returned. `xyz = 0` returns the maximum resolution for the first axis, `xyz = 1` returns the maximum resolution for the second axis, and `xyz = 2` returns the maximum resolution for the third axis (if available).

Return value:

- `int`: maximum resolution (i.e. largest number of scan steps) along the specified axis.

See also:

- `get_min_resolution(scan_type, xyz)`
- `set_scan_resolution(scan_type, xyz, new_res)`

5.2.17 start_scanning(scan_type, depth_scan = False, tag = 'logic')

This method triggers the start of a main scan or depth scan of the *scan type* indicated by parameter `scan_type`. Before the actual scan procedure starts, the method may (depending on the configuration of the logic module) activate various actuators as to bring the confocal microscope into the required configuration.

Then, control is immediately given back to the calling instance, and the scanning starts. Every time a new line of the scan is completed, a `signal_xy_image_updated(scan_type)` or `signal_depth_image_updated(scan_type)` signal is emitted, so that the user interface module gets a chance to update the image using the `get_main_scan_image(scan_type)` or `get_depth_scan_image(scan_type)` method.

The index of the freshly added line (column) can be retrieved using the `get_scan_counter()`-method.

Qudi modules provide a method `module_state()` by default. This method returns a string and can be used to determine whether the scan has already stopped. As long as `module_state()` returns `'locked'`, the scan has not stopped yet.

The `start_scanning`-method does not apply to a *scan type* representing a camera.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `optional bool depth_scan`: If `False` (default value), a scan along the main axes is started. If `True`, a depth scan is started.
- `optional str tag`: for future use.

Return value:

- `bool`: `True` if the scan was successfully started, otherwise `False` .

See also:

- `stop_scanning()`
- `continue_scanning(depth_scan, tag = 'logic')`
- `cur_scan_is_depth_scan()`
- `get_main_scan_image(scan_type)`
- `get_depth_scan_image(scan_type)`

5.2.18 stop_scanning()

This method triggers an early stop of an ongoing scan. Control is given back to the calling instance immediately, while the scan logic still aims to finish the current line of the scan. If the current scan-line cannot be finished within 5 seconds, then the scan is aborted nevertheless with the current scan line unfinished.

Qudi modules provide a method `module_state()` by default. This method returns a string and can be used to determine whether the scan has already stopped. As long as `module_state()` returns `'locked'`, the scan has not stopped yet.

The `stop_scanning`-method does not apply to a *scan type* representing a camera.

See also:

- `start_scanning(scan_type, depth_scan = False, tag = 'logic')`
- `continue_scanning(depth_scan, tag = 'logic')`

5.2.19 continue_scanning(depth_scan, tag = 'logic')

This method allows to continue a scan that has been suspended using the `stop_scanning()`-method. Before calling this method to continue a main scan, the return value of `get_cur_main_scan_continuable()` should be checked. Before calling this method to continue a depth scan, the return value of `get_cur_depthscan_continuable()` should be checked.

Control is immediately given back to the calling instance, and the scanning proceeds. The re-started scan can be handled as explained for the `start_scanning` method.

The `continue_scanning`-method does not apply to a *scan type* representing a camera.

Parameters:

- `bool depth_scan`: If `False`, a scan along the main axes ("main scan") is continued. If `True`, a depth scan is continued.
- `optional str tag`: for future use.

See also:

- `start_scanning(scan_type, depth_scan = False, tag = 'logic')`
- `stop_scanning()`

5.2.20 cur_scan_is_depth_scan()

The `cur_scan_is_depth_scan`-method does not apply to a *scan type* representing a camera.

Return value:

- `bool`: Returns `True` if the *current* scan type (as set using the method `set_cur_scan_type_index(scan_type)`) is a depth-scan. Otherwise the method returns `False`.

See also:

- `start_scanning(scan_type, depth_scan = False, tag = 'logic')`

5.2.21 get_cur_main_scan_continuable()

The `get_cur_main_scan_continuable`-method does not apply to a *scan type* representing a camera.

Return value:

- `bool`: Returns `True` if the *current* scan type (as set using the method `set_cur_scan_type_index(scan_type)`) is a depth-scan that has been suspended with `stop_scanning()`. Otherwise the method returns `False`.

See also:

- `get_cur_depthscan_continuable()`
- `continue_scanning(depth_scan, tag = 'logic')`
- `cur_scan_is_depth_scan()`

5.2.22 get_cur_depthscan_continuable()

The `get_cur_depthscan_continuable`-method does not apply to a *scan type* representing a camera.

Return value:

- `bool`: Returns `True` if the *current* scan type (as set using the method `set_cur_scan_type_index(scan_type)`) is a depth-scan that has been suspended with `stop_scanning()`. Otherwise the method returns `False`.

See also:

- `get_cur_main_scan_continuable()`
- `continue_scanning(depth_scan, tag = 'logic')`
- `cur_scan_is_depth_scan()`

5.2.23 get_main_scan_image(scan_type)

This method returns the scan data of the previously finished or currently ongoing main scan along the two main axes (which depend on the configuration and need not necessarily be x and y).

The returned `numpy.array` can be read as indicated below, with `row` and `col` indexing the respective image pixel. `row` and `col` must be larger or equal zero and smaller than the resolution along the respective axes as set with `set_scan_resolution(scan_type, xyz, new_res)`

- `[row, col, 0]` ...`float`: first axis coordinate value
- `[row, col, 1]` ...`float`: second axis coordinate value
- `[row, col, 2]` ...`float`: third axis coordinate value
- `[row, col, 3]` ...`float`: measured value from first data channel
- `[row, col, 4]` ...`float`: measured value from second data channel
- ...
- `[row, col, n]` ...`float`: measured value from $(n-2)$-th data channel

The `get_XY_image`-method can be invoked by the user interface module to update the scan image every time a `signal_xy_image_updated(scan_type)` signal is emitted. The index of the latest `row` added to the image can be determined by calling `get_scan_counter()` method.

The `get_XY_image`-method does not apply to a `scan_type` representing a camera.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `numpy.array`: Array with scan data as explained above.

See also:

- `get_depth_scan_image(scan_type)`
- `set_scan_resolution(scan_type, xyz, new_res)`
- `get_scan_counter()`
- `signal_xy_image_updated(scan_type)`

5.2.24 get_depth_scan_image(scan_type)

This method returns the scan data of the previously finished or currently ongoing depth scan along the two defined depth-scan axes (which depend on the configuration and need not necessarily be y and z).

The returned `numpy.array` can be read as indicated below, with `row` and `col` being integer values larger or equal zero and indexing the respective image pixel.

- `[row, col, 0]` ...`float`: first axis coordinate value
- `[row, col, 1]` ...`float`: second axis coordinate value
- `[row, col, 2]` ...`float`: third axis coordinate value
- `[row, col, 3]` ...`float`: measured value from first data channel
- `[row, col, 4]` ...`float`: measured value from second data channel
- ...
- `[row, col, n]` ...`float`: measured value from $(n-2)$-th data channel

The `get_depth_image`-method can be invoked by the user interface module to update the scan image every time a `signal_depth_image_updated(scan_type)` signal is emitted. The index of the latest `row` added to the image can be determined by calling `get_scan_counter()` method.

The `get_depth_image`-method does not apply to a `scan_type` representing a camera.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `numpy.array`: Array with scan data as explained above.

See also:

- `get_main_scan_image(scan_type)`
- `set_scan_resolution(scan_type, xyz, new_res)`
- `get_scan_counter()`
- `signal_depth_image_updated(scan_type)`

5.2.25 get_scan_counter()

The `get_scan_counter`-method returns the `col` index of the line just added to the scan image in the numpy-array returned by `get_main_scan_image(scan_type)` and `get_depth_scan_image(scan_type)`.

Hence, it can be used by the user interface module to update just this line of the scan image every time a `signal_xy_image_updated(scan_type)` signal or `signal_depth_image_updated(scan_type)` signal is emitted.

The `get_scan_counter`-method does not apply to a `scan_type` representing a camera.

See also:

- `get_main_scan_image(scan_type)`
- `get_depth_scan_image(scan_type)`
- `signal_xy_image_updated(scan_type)`
- `signal_depth_image_updated(scan_type)`

5.2.26 get_ax_index_from_xyz_index(d, scan_type, xyz)

It is one of the responsibilities of the scan logic to determine for a given *scan type*, which of the available axes are the *main scan* axes, and - if three axes are available - which of them are the *depth scan* axes.

Let us assume that, for a given *scan type*, three axes are available. They can have any name, but to keep things simple, let us further assume that they are named 'X', 'Y' and 'Z'. Then, the scan logic may determine that the main scan axes are 'X' (horizontal) and 'Y' (vertical), and that the depth scan axes are 'X' and 'Z' (which, again, is just an example. The scan logic could be configured to any other axes combinations).

Now, the `get_ax_index_from_xyz_index`-method allows to determine for a given axis (here: 'X', 'Y', or 'Z') if it is used as horizontal or vertical *main-scan* or *depth-scan* axis.

Parameters:

- `int d`: Indicates if we need information for the *main-scan*-axes (`d=0`), or the *depth-scan*-axes (`d=1`).
- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates for which axis the "horizontal-or-vertical-main-or-depth-scan"-information is required. `xyz = 0` returns information for the first axis, `xyz = 1` returns the information for the second axis, and `xyz = 2` returns the information for the third axis (if available). Please note, that the first, second, and third axes may have different names than 'X', 'Y' and 'Y'.

Return value:

- `int`: If `0`, then the given axis is the horizontal axis. If `1`, then the given axis is the vertical axis. If `2`, then the given axis is not part of the chosen (main- or depth-)scan.

In the example above, to get the information for the third ('Z') axis with regards to the *depth-scan*, one would call the function as `get_ax_index_from_xyz_index(1, scan_type, 2)` and would get the return value `1`, as this is the vertical *depth-scan*-axis in our example.

5.2.27 get_third_scan_axis_xyz_index(scan_type, d)

This method allows to determine which of the available axes is *not* used for the main-scan or depth-scan of a given *scan-type*.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int d`: Indicates whether we need information for the *main-scan* (`d=0`), or the *depth-scan* (`d=1`).

Return value:

- `int`: If `0` then the first axis, if `1` then the second axis, or if `2` then the third axis is not used in the referenced *main-scan* or *depth-scan*.

5.2.28 allows_depth_scan(scan_type)

This method returns `True` if the referenced *scan type* allows to run a *depth-scan*, otherwise it returns `False`.

Parameters:

- `int scan_type`: Index identifying the *scan-type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `int bool`: `True`, if the referenced *scan-type* allows to run a depth-scan, otherwise `False`.

5.2.29 save_file(scan_type, file_name, main_image)

This method saves the main scan data or (if available) the depth scan data of the *scan type* referenced by parameter `scan_type` into a comma-separated csv text file. This method can also be used for saving the camera image of a camera controlling *scan type*.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `str file_name`: File name (including file path and file extension).
- `bool main_image`: If `True`, the main scan image or (if *scan type* is controlling a camera) the camera image is saved. If `False` the depth scan image is saved.

Every line of the generated csv file represents a pixel of the scan image and consists of the following data:

- `int hpix` ... horizontal (column) index of the pixel, counting from left to right, starting with `0` for the leftmost pixel of the line
- `int vpix` ... vertical (row) index of the pixel, counting from bottom to top, starting with `0` for the bottom-most pixel
- `float ax1_coord` ... coordinate of the first axis in meters (scientific number format)
- `float ax2_coord` ... coordinate of the second axis in meters (scientific number format)
- `float scientific ax3_coord` ... coordinate of the third axis in meters, if available (scientific number format)
- `float ch_data_1` ... data of the first counter channel (scientific number format)
- `float ch_data_2` ... data of the second counter channel (scientific number format)
- ...
- `float ch_data_n` ... data of the last counter channel (scientific number format)

The first line of the file is a header file with textual column descriptions. Separator character is a comma (',').

5.2.30 load_file(scan_type, file_name)

This method loads scan-images or camera-images from a comma-separated csv text-file saved previously by the `save_file(scan_type, file_name, main_image)` method into the *scan type* referenced by parameter `scan_type` .

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `str file_name`: Name of the file from which the image should be loaded (including file path and file extension).

Return values (tuple):

- `string`: Error message in case the file could not be loaded, or empty string (if no error occurred).
- `bool`: Returns `True` if the loaded image is a depth-scan image, `True` otherwise.

Besides basic syntax checks, the method also determines whether or not the file to be loaded matches the referenced *scan-type*. It is checked whether the axes names and data source names in the file header match the referenced *scan-type*.

In case this *scan-type* provides both *main-scan* and *depth-scan*, the method additionally determines the file type with respect to this automatically by checking which axis coordinates are constant.

5.3 Main Positioning Device Control Methods

Each *scan type* can be identified by a unique index, and has a specific main positioning device assigned (e.g. a *scan type* named "stage scan" may have a three axes piezo stage as main positioning device , or a *scan type* named "mirror scan" may have a laser mirror with two axes as main positioning device).

In this chapter we document all class methods related to controlling these main positioning devices.

5.3.1 set_position(tag, scan_type=-1, x=None, y=None, z=None, a=None, absolute = False)

This method moves the main positioning device of the *scan type* indicated by `scan_index` along one or more axes. The target positions of the axes to be moved must be passed on to the method via the four optional parameters `x`, `y`, `z`, and `a`; where `x` stands for the devices' first axes, `y` stands for the devices' second axes, `z` stands for the devices' third axes (if available), and `a` stands for any potentially available fourth axes. Please note that the actual naming of the axes as returned by `get_pos_axes(scan_type = -1)` may differ from "`x`", "`y`", and "`z`".

Parameters:

- `str tag`: Every time this method is called, a `signal_change_position(tag)` signal is emitted with the `tag` information attached.
- `optional int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index is omitted, then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.
- `optional float x` : Target position of the first axis (in meters). Must be within the range returned by `get_pos_range(scan_type, xyz, absolute = False)[0]`.
- `optional float y` : Target position of the second axis (in meters). Must be within the range returned by `get_pos_range(scan_type, xyz, absolute = False)[1]`.
- `optional float z` : Target position of the third axis, if available (in meters). Must be within the range returned by `get_pos_range(scan_type, xyz, absolute = False)[2]`.
- `optional float a` : Target position of a potential fourth axes, if available (in meters). Must be within the range returned by `get_pos_range(scan_type, xyz, absolute = False)[3]`.
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning

device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

See also:

- `get_position(scan_type = -1, absolute = False)`
- `get_target_position(scan_type, absolute = False)`
- `get_pos_range(scan_type, xyz, absolute = False)`
- `set_cur_scan_type_index(scan_type)`
- `absolute_pos_coords_available(scan_type)`
- `signal_change_position(tag)`

5.3.2 get_position(scan_type = -1, absolute = False)

Provided the main positioning device of the referenced `scan_type` has sensors to determine the current *actual* position (which can be queried by calling `pos_provides_cur_pos_info(scan_type)`), this class method returns:

- information about the current position of all axes,
- information whether or not the current position of each axes currently matches the target position (withing tolerances), and
- for each axis information about how long the position already matches the target position within tolerances (or, zero, if not matching).

If the positioning device does not have senors to determine the current actual position, this class method will just return the target position and assume that we are already on target.

Parameters:

- `optional int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index is omitted, then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return values (tuple):

- `float list[n]`: A list with n entries, where n stand for the number of axes. The list contains the current positions of each axis in meters. If the device has no sensors to determine the actual positions, the list will just contain the target positions. For each axis, the list contains the boolean
- `bool list[n]`: A list with n entries, where n stand for the number of axes. For each axis, the list contains `True`, if the current axis' position is on target (within tolerance), otherwise `False`. If the device has no sensors to determine the actual positions, the list will just contain `True` values.
- `float list[n]`: A list with n entries, where n stand for the number of axes. For each axis, the list contains the duration in seconds for which the actual position is already matching the target position (within tolerance). If, for an axis, the current position is *not* matching the target position, a value of 0 is returned. If the device has no sensors to determine the actual positions, the list will just contain the duration in seconds since calling of `set_position(...)`.

See also:

- `set_position(...)`
- `get_target_position(scan_type, absolute = False)`
- `pos_provides_cur_pos_info(scan_type)`
- `set_cur_scan_type_index(scan_type)`
- `absolute_pos_coords_available(scan_type)`

5.3.3 get_target_position(scan_type, absolute = False)

This class method returns the target positions of the main positioning device of the referenced `scan_type`.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index has the value `-1` then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return value:

- `float list[n]`: A list with n entries, where n stand for the number of axes. The list contains the target positions of each axis in meters. If no target position has been defined for an axis, the list will contain an `None` entry for this axis.

See also:

- `set_position(...)`
- `get_position(scan_type = -1, absolute = False)`
- `set_cur_scan_type_index(scan_type)`
- `absolute_pos_coords_available(scan_type)`

5.3.4 pos_provides_cur_pos_info(scan_type)

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index has the value `-1` then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.

Return value:

- `bool`: `True` if the positioning device can provide information about the actual current position along all axes, `False` otherwise.

See also:

- `get_position(scan_type = -1, absolute = False)`
- `set_cur_scan_type_index(scan_type)`

5.3.5 get_pos_range(scan_type, xyz, absolute = False)

This method returns the physical range of all axes of the main positioning device of the referenced `scan_type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates for which axis the physical range is requested. `xyz = 0` returns the range for the first axis, `xyz = 1` returns the range for the second axis, and `xyz = 2` returns the range for the third axis (if available). Please note, that the first, second, and third axes may have different names than 'X', 'Y' and 'Y'.
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return value:

- `list[n][2]`: A list with n entries (where n is the number of axes of the positioning device). Each entry consists of another list with two float values, representing the lower and upper bound of the axis' physical range in meters.

See also:

- `set_position(...)`
- `absolute_pos_coords_available(scan_type)`

5.3.6 set_pos_ini_position(scan_type, main_axes, absolute_center)

Depending on the boolean value of `main_axes`, this method re-sets either the main-scan-axes or the depth-scan-axes of the main positioning device of the referenced `scan_type`. The boolean parameter `AbsoluteCenter` defines whether the axes should to be re-set to the initial position as defined by the hardware layer class, or alternatively to the center position as defined with the method `set_scan_center_position`.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `bool main_axes`: If `True` then the two *main-scan*-axes are being re-set. Otherwise, the two *depth-scan*-axes are being re-set.
- `bool absolute_center`: If `True` then the two axes are re-set to the initial position as defined by the hardware layer class. If `False` then the two axes are re-set to the center position as defined with method `set_scan_center_position`.

See also:

- `get_pos_ini_position(scan_type, xyz, absolute, absolute_center)`

5.3.7 get_pos_ini_position(scan_type, xyz, absolute, absolute_center)

This class method can be used to determine the initial position of the main positioning device of the referenced `scan_type`. Depending on the boolean parameter `absolute_center` it returns the initial position as defined by the hardware layer class, or alternatively the center position as defined with the method `set_scan_center_position`.

When called with a `scan_type` controlling a camera, then the center position of the camera image is returned.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: Index identifying the axis. `0` represents the first axis, `1` represents the second axis, and `2` represents the third axis (if available).
- `bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

- `bool absolute_center`: If `True` then the initial position as defined by the hardware layer class is returned. If `False` then the center position as defined with method `set_scan_center_position` is returned.

Return value:

- `float`: Initial position for this axis in meters.

See also:

- `set_pos_ini_position(scan_type, main_axes, absolute_center)`
- `absolute_pos_coords_available(scan_type)`

5.3.8 get_pos_is_centered(scan_type, main_axes)

Depending on the boolean value of `main_axes`, this method checks whether the main-scan-axes or the depth-scan-axes of the main positioning device of the referenced `scan_type` are in the initial position as defined by the hardware layer class.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `bool main_axes`: If `True` then it is checked whether the two *main-scan*-axes are centered. Otherwise, it is checked whether the two *depth-scan*-axes are centered.

Return value:

- `bool`: `True` if the relevant axes are in their initial position, `False` otherwise.

See also:

- `set_pos_ini_position(scan_type, main_axes, absolute_center)`

5.3.9 absolute_pos_coords_available(scan_type)

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `bool`: If `True` then absolute coordinates as explained in chapter 5.1 are available for all main positioner functions with an `absolute` parameter.

5.3.10 pos_abs_to_rel(scan_type, xyz, pos_abs)

This function converts an absolute coordinate (where the positions of other main positioning devices and pre-positioning devices are taken into account as explained in chapter 5.1) into a relative coordinate (within the coordinate system of the single main positioning device of this `scan_type`).

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates for which axis the coordinate conversion is requested. `xyz = 0` converts the coordinate for the first axis, `xyz = 1` converts the coordinate for the second axis, and `xyz = 2` converts the coordinate for the third axis (if available). Please note, that the first, second, and third axes may have different names than 'X', 'Y' and 'Y'.
- `float pos_abs`: absolute coordinate value in meters.

Return value:

- `float`: Coordinate value `pos_abs` converted into a relative coordinate value within the coordinate system of the single main positioning device of this `scan_type`.

See also:

- `pos_rel_to_abs(scan_type, xyz, pos_rel)`
- `absolute_pos_coords_available(scan_type)`

5.3.11 pos_rel_to_abs(scan_type, xyz, pos_rel)

This function converts a relative coordinate (within the coordinate system of the single main positioning device of this `scan_type`) into an absolute coordinate (where the positions of other positioning devices and pre-positioning devices are taken into account as explained in chapter 5.1).

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates for which axis the coordinate conversion is requested. `xyz = 0` converts the coordinate for the first axis, `xyz = 1` converts the coordinate for the second axis, and `xyz = 2` converts the coordinate for the third axis (if available). Please note, that the first, second, and third axes may have different names than 'X', 'Y' and 'Y'.
- `float pos_rel`: relative coordinate value in meters.

Return value:

- `float`: Coordinate value `pos_rel` converted into an absolute coordinate value (where the positions of other positioning devices and pre-positioning devices are taken into account as explained in chapter 5.1).

See also:

- `pos_abs_to_rel(scan_type, xyz, pos_abs)`
- `absolute_pos_coords_available(scan_type)`

5.3.12 get_pos_length_unit(scan_type)

As all coordinates are always expressed in meters, it is helpful for the user interface to get a hint from the scan logic about the typical magnitude of the coordinates of the main positioning device of a `scan_type`, i.e. to get information in which unit the coordinates should be displayed.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `string`: ('m'... millimeters, 'u'... micrometers, 'n'... nanometers).

See also:

- `get_pos_precision(scan_type)`
- `get_position(scan_type = -1, absolute = False)`

5.3.13 get_pos_precision(scan_type)

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `int`: Number of decimal points to be display in an user interface when using the length unit as returned by `get_pos_length_unit(scan_type)`.

See also:

- `get_pos_length_unit(scan_type)`
- `get_position(scan_type = -1, absolute = False)`

5.3.14 get_pos_axes(scan_type = -1)

This function returns a list containing the names of all axes of the positioning device of this `scan_type`.

Parameter:

- `optional int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index is omitted, then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.

Return value:

- `list[n]`: A list with n entries (where n is the number of axes of the main positioning device of this `scan_type`. Each entry consists of a string value with the name of the corresponding axis (e.g. `['X','Y','Z']`).

See also:

- `get_pos_ch_device_name(scan_type, first_char_upper)`
- `set_cur_scan_type_index(scan_type)`

5.3.15 get_pos_ch_device_name(scan_type, first_char_upper)

This method can be used to get the name of the positioning device of a `scan_type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `str`: name of positioning device.

See also:

- `get_pos_axes(scan_type = -1)`

5.4 Pre-Positioning Device Control Methods

Each *scan type* optionally may also have a so-called pre-positioning device assigned (e.g. stepper motors along each axes).

In this chapter we document all class methods related to controlling these pre-positioning devices.

5.4.1 set_pre_position(scan_type, x=None, y=None, z=None, a=None, absolute = False)

This method moves the pre-positioning device of the `scan_type` along one or more axes. The target positions of the axes to be moved must be passed on to the method via the four optional parameters `x`, `y`, `z`, and `a`; where `x` stands for the devices' first axes, `y` stands for the devices' second axes, `z` stands for the devices' third axes (if available), and `a` stands for any potentially available fourth axes. Please note that the actual naming of the axes as returned by `get_pre_pos_axes(scan_type = -1)` may differ from `"x"`, `"y"`, and `"z"`. By default, the pre-positioning-device should start to move immediately after this method has been called. However, it may also happen that the device is a slow-moving positioning device (like, stepper motors) and is currently re-setting (e.g. homing). Then, the movement is delayed and can be triggered at a later point in time by calling the method `pre_pos_trigger_queue(scan_type)`.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `optional float x` : Target position of the first axis (in meters). Must be within the range returned by `get_pre_pos_range(...)[0]`.
- `optional float y` : Target position of the second axis (in meters). Must be within the range returned by `get_pre_pos_range(...)[1]`.
- `optional float z` : Target position of the third axis, if available (in meters). Must be within the range returned by `get_pre_pos_range(...)[2]`.
- `optional float a` : Target position of a potential fourth axes, if available (in meters). Must be within the range returned by `get_pre_pos_range(...)[3]`.
- `bool absolute`: If `False`, then all coordinates are interpreted as coordinates just within the coordinate system of the single pre-positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return value:

- `int`: `-1` if the `scan_type` has no pre-positioning device, `0` otherwise.

See also:

- `get_pre_pos_position(scan_type, absolute = False)`
- `pre_pos_trigger_queue(scan_type)`
- `absolute_pre_pos_coords_available(scan_type)`

5.4.2 get_pre_pos_position(scan_type, absolute = False)

Provided the pre-positioning device of the referenced `scan_type` has sensors to determine the current *actual* position, this class method returns:

- information about the current position of all axes,
- information whether or not the current position of each axes currently matches the target position (withing tolerances), and
- for each axes information about how long the position already matches the target position within tolerances (or, zero, if not matching).

If the pre-positioning device does not have senors to determine the current actual position, this class method will just return the target position and assume that we are already on target.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index is set to `-1` then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single pre-positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return values (tuple):

- `float list[n]`: A list with n entries, where n stand for the number of axes. The list contains the current positions of each axis in meters. If the pre-positioning-device has no sensors to determine the actual positions, the list will just contain the target positions. For each axis, the list contains the boolean
- `bool list[n]`: A list with n entries, where n stand for the number of axes. For each axis, the list contains `True`, if the current axis' position is on target (within tolerance), otherwise `False`. If the pre-positioning-device has no sensors to determine the actual positions, the list will just contain `True` values.
- `float list[n]`: A list with n entries, where n stand for the number of axes. For each axis, the list contains the duration in seconds for which the actual position is already matching the target position (within tolerance). If, for an axis, the current position is *not* matching the target position, a value of 0 is returned. If the pre-positioning-device has no sensors to determine the actual positions, the list will just contain the duration in seconds since calling of `set_pre_position(...)`.

See also:

- `set_pre_position(scan_type, ...)`
- `set_cur_scan_type_index(scan_type)`
- `absolute_pre_pos_coords_available(scan_type)`

5.4.3 pre_pos_allows_homing(scan_type)

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `bool`: `True` if the pre-positioning device has a callable homing procedure, which moves all axis to a "home" position, or `False` otherwise.

See also:

- `pre_pos_home(scan_type)`

5.4.4 pre_pos_home(scan_type)

This class method triggers a "homing" procedure for the pre-positioning device of this `scan_type`, which moves all axis to a "home" position, provided the pre-positioning-device controller has such a procedure.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

See also:

- `pre_pos_allows_homing(scan_type)`

5.4.5 get_pre_pos_ini_position(scan_type, xyz, absolute = False)

This class method can be used to determine the initial position of the pre-positioning device of the referenced `scan_type`. The initial position is defined by the hardware layer class.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: Index identifying the axis. `0` represents the first axis, `1` represents the second axis, and `2` represents the third axis (if available).
- `bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the single pre-positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return value:

- `float`: Initial position for this axis in meters.

See also:

- `set_pre_position(scan_type, ...)`
- `absolute_pre_pos_coords_available(scan_type)`

5.4.6 absolute_pre_pos_coords_available(scan_type)

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `bool`: If `True` then absolute coordinates as explained in chapter 5.1 are available for all pre-positioning functions with an `absolute` parameter.

5.4.7 pre_pos_abs_to_rel(scan_type, xyz, pos_abs)

This function converts an absolute coordinate (where the positions of other main positioning devices and pre-positioning devices are taken into account as explained in chapter 5.1) into a relative coordinate (within the coordinate system of the pre-positioning device of this `scan_type`).

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates for which axis the coordinate conversion is requested. `xyz = 0` converts the coordinate for the first axis, `xyz = 1` converts the coordinate for the second axis, and `xyz = 2` converts the coordinate for the third axis (if available). Please note, that the first, second, and third axes may have different names than 'X', 'Y' and 'Y'.
- `float pos_abs`: absolute coordinate value in meters.

Return value:

- `float`: Coordinate value `pos_abs` converted into a relative coordinate value within the coordinate system of the pre-positioning device of this `scan_type`.

See also:

- `absolute_pre_pos_coords_available(scan_type)`

5.4.8 pre_pos_rel_to_abs(scan_type, xyz, pos_rel)

This function converts a relative coordinate (within the coordinate system of the pre-positioning device of this `scan_type`) into an absolute coordinate (where the positions of other positioning devices and pre-positioning devices are taken into account as explained in chapter 5.1).

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates for which axis the coordinate conversion is requested. `xyz = 0` converts the coordinate for the first axis, `xyz = 1` converts the coordinate for the second axis, and `xyz = 2` converts the coordinate for the third axis (if available). Please note, that the first, second, and third axes may have different names than 'X', 'Y' and 'Y'.
- `float pos_rel`: relative coordinate value in meters.

Return value:

- `float`: Coordinate value `pos_rel` converted into an absolute coordinate value (where the positions of other positioning devices and pre-positioning devices are taken into account as explained in chapter 5.1).

See also:

- `absolute_pre_pos_coords_available(scan_type)`

5.4.9 pre_pos_trigger_queue(scan_type)

By default, the pre-positioning device should start to move immediately after the class method `set_pre_position(scan_type, ...)` has been called. However, it may also happen that the pre-positioning device is slow-moving (like, stepper motors), and is currently re-setting (e.g. homing). Then, the movement is delayed and can be triggered at a later point in time by calling the method `pre_pos_trigger_queue(scan_type)`.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

See also:

- `set_pre_position(scan_type, ...)`

5.4.10 get_pre_pos_length_unit(scan_type)

As all coordinates are always expressed in meters, it is helpful for the user interface to get a hint from the scan logic about the typical magnitude of the coordinates of the pre-positioning device of a `scan_type`, i.e. to get information in which unit the coordinates should be displayed.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `string`: ('m'… millimeters, 'u'… micrometers, 'n'… nanometers).

See also:

- `get_pre_pos_precision(scan_type)`

5.4.11 get_pre_pos_precision(scan_type)

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `int`: Number of decimal points to be display in an user interface when using the length unit as returned by `get_pre_pos_length_unit(scan_type)`.

See also:

- `get_pre_pos_length_unit(scan_type)`

5.4.12 matches_pre_pos_target_position(scan_type, xyz, pos, absolute = False)

This class method can be used to determine whether a given position `pos` matches the target position of a specific axis of the pre-positioning device linked to the `scan_type` (plus/minus minor rounding deviations).

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates the position of which axis should be compared with parameter `pos`. `xyz = 0` will compare the position of the first axis with `pos`, `xyz = 1` will compare the position of the second axis with `pos`, and `xyz = 2` will compare the position of the third axis with `pos` (if available).
- `float pos`: Position in meters to which the target position should be compared to.
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the pre-positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return value:

- `bool`: `True`, if the given position `pos` matches the target position of the axis referenced by `xyz_index` (plus/minus minor rounding deviations). `False` otherwise, or if `pos` is `None`, or if no target position has yet been set.

See also:

- `set_pre_position(scan_type, ...)`
- `absolute_pre_pos_coords_available(scan_type)`

5.4.13 get_pre_pos_range(scan_type, xyz, absolute = False)

This method returns the physical range of all axes of the pre-positioning device of the referenced `scan_type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int xyz`: This indicates for which axis the physical range is requested. `xyz = 0` returns the range for the first axis, `xyz = 1` returns the range for the second axis, and `xyz = 2` returns the range for the third axis (if available). Please note, that the first, second, and third axes may have different names than 'X', 'Y' and 'Y'.
- `optional bool absolute`: If `False` (default value), then all coordinates are interpreted as coordinates just within the coordinate system of the pre-positioning device of the referred scan type, not considering the position of any other positioning devices or pre-positioning devices. If `True` then - depending on configuration - the positions of other positioning devices and pre-positioning devices are also taken into account (see chapter 5.1).

Return value:

- `list[n][2]`: A list with n entries (where n is the number of axes of the positioning device). Each entry consists of another list with two float values, representing the lower and upper bound of the axis' physical range in meters.

See also:

- `set_position(...)`
- `absolute_pos_coords_available(scan_type)`

See also:

- `set_pre_position(scan_type, ...)`
- `absolute_pre_pos_coords_available(scan_type)`

5.4.14 get_pre_pos_axes(scan_type = -1)

This function returns a list containing the names of all axes of the pre-positioning device of this `scan_type`.

Parameter:

- `optional int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index is omitted, then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.

Return value:

- `list[n]`: A list with n entries (where n is the number of axes of the pre-positioning device of this `scan_type`. Each entry consists of a string value with the name of the corresponding axis (e.g. `['X','Y','Z']`).

See also:

- `get_pre_pos_ch_device_name(scan_type, first_char_upper)`
- `set_cur_scan_type_index(scan_type)`

5.4.15 get_pre_pos_ch_device_name(scan_type, first_char_upper)

This method can be used to get the name of the pre-positioning device of a `scan_type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `str`: name of pre-positioning device.

See also:

- `get_pre_pos_axes(scan_type = -1)`

5.5 Camera Control Methods

5.5.1 is_camera(scan_type)

Whereas different *scan types* by default represent different image generating scanning methods (like, confocal scanning with a laser mirror or confocal scanning with a piezo stage), a specific *scan type* may also as well represent a camera.

As the user interface will have to vary between "default" *scan types* and a camera *scan type*, we need a method to distinguish between this two different types. Therefore, the `is_camera` method allows to find out whether a specific `scan type` represents a camera.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index is set to `-1` then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.

Return value:

- `bool`: Returns `True`, if the `scan_type` represents a camera, `False` otherwise.

See also:

- `cam_is_connected(scan_type)`
- `set_cur_scan_type_index(scan_type)`

5.5.2 cam_is_connected(scan_type)

Even if `is_camera(scan_type)` returns `True`, the camera still may be disconnected or offline. This method allows to check if that is the case.
Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `bool`: Returns `True`, if the camera of this `scan_type` is online and ready. If the camera is offline (e.g. disconnected or turned off), or if this is no "camera" `scan_type`, then `False` is returned.

See also:

- `is_camera(scan_type)`

5.5.3 get_cam_resolutions(scan_type)

This method returns a list of available resolutions for the camera of a `scan type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `list[n]`: A list with n entries, where n stands for the number of possible resolutions. Each list entry represents an allowed resolution and contains a list with two `int` values representing the x- and y resolution in pixels.

  For example, a camera allowing the resolutions 512×512 and 256×256 pixels, would return the list `[[512,512], [256,256]]`

See also:

- `set_cam_resolution(scan_type, res_index)`

5.5.4 set_cam_resolution(scan_type, res_index)

This method allows to set the resolution to one of the values returned by `get_cam_resolutions(scan_type)`.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int res_index`: Index representing one of the allowed resolutions as returned by the function `get_cam_resolutions(scan_type)`. Must be larger or equal 0 and smaller than `len(get_cam_resolutions(scan_type))`.

See also:

- `get_cam_resolutions(scan_type)`

5.5.5 cam_has_shutter(scan_type)

This method determines whether the camera of a `scan_type` has a mechanical shutter.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `bool`: Returns `True` if the camera has a mechanical shutter. If the camera has no mechanical shutter, or if this `scan_type` does not control a camera, `False` is returned.

See also:

- `get_cam_shutter_settings(scan_type)`
- `get_cam_default_shutter_setting(scan_type)`
- `set_cam_shutter(scan_type, mode)`

5.5.6 get_cam_shutter_settings(scan_type)

This method returns a list of all possible shutter settings for the camera of a `scan_type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `list[n]`: List with n entries, where n is the number of possible shutter settings. Each list entry represents a possible setting and consists of a sub-list with two entries: The first entry is an unique integer index, and the second entry a text string describing the setting. This is an example of a list for two allowed shutter settings: `[[0, 'Fully Auto'], [1, 'Permanently Open']]`

See also:

- `cam_has_shutter(scan_type)`
- `get_cam_default_shutter_setting(scan_type)`
- `set_cam_shutter(scan_type, mode)`

5.5.7 get_cam_default_shutter_setting(scan_type)

This method returns the index of the default shutter setting for the camera of a `scan_type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `int`: Index of the default shutter setting out of the indexes returned by `get_cam_shutter_settings(scan_type)`.

See also:

- `cam_has_shutter(scan_type)`
- `get_cam_shutter_settings(scan_type)`
- `set_cam_shutter(scan_type, mode)`

5.5.8 set_cam_shutter(scan_type, mode)

This method sets the shutter of the camera of a `scan_type` to one of the possible settings.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `int index`: Index of the shutter setting to be set. Must be one of the indexes returned by `get_cam_shutter_settings(scan_type)`.

See also:

- `cam_has_shutter(scan_type)`
- `get_cam_shutter_settings(scan_type)`
- `get_cam_default_shutter_setting(scan_type)`

5.5.9 set_cam_exposure(time, scan_type=-1)

This method sets the exposure time of the camera of a `scan_type`.

Parameters:

- `float time`: exposure time in seconds
- `optional int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`. If this index is omitted, then the current scan type as set with `set_cur_scan_type_index(scan_type)` is assumed.

Return value:

- `bool`: `True` if successful, otherwise `False`.

See also:

- `get_cam_exposure(scan_type)`
- `get_cam_max_exposure(scan_type)`
- `set_cur_scan_type_index(scan_type)`

5.5.10 get_cam_exposure(scan_type)

This method returns the currently configured exposure time of the camera of a `scan_type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `float`: exposure time in seconds.

See also:

- `get_cam_max_exposure(scan_type)`
- `set_cam_exposure(time, scan_type=-1)`

5.5.11 get_cam_max_exposure(scan_type)

This method returns the maximum exposure time of the camera of a `scan_type`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `float`: maximum exposure time in seconds.

See also:

- `get_cam_exposure(scan_type)`
- `set_cam_exposure(time, scan_type=-1)`

5.5.12 cam_provides_temperature_information(scan_type)

This method returns a boolean value indicating whether the camera of a `scan_type` provides current temperature information.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `bool`: Returns `True` if the camera provides temperature information. If the camera does not provide current temperature information, or if this `scan_type` does not control a camera, `False` is returned.

See also:

- `cam_get_temperature_precision(scan_type)`
- `cam_get_temperature(scan_type)`

5.5.13 cam_get_temperature_precision(scan_type)

This method returns the number of digits after the decimal point up to which the current temperature information of the camera of a certain `scan_type` is significant.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `int`: Number of digits after the decimal point up to which the current temperature returned by `cam_get_temperature(scan_type)` is significant.

See also:

- `cam_provides_temperature_information(scan_type)`
- `cam_get_temperature(scan_type)`

5.5.14 cam_get_temperature(scan_type)

Returns the current temperature of the camera of a certain `scan_type` and an indicator about the temperature stability.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return values (tuple):

- `float`: Current temperature of the camera in °C.
- `int`:

  0 . . . no temperature range information available

  1 . . . the temperature is out of range, or within range but not yet stabilized

  2 . . . the temperature is within range and stabilized

See also:

- `cam_provides_temperature_information(scan_type)`
- `cam_get_temperature_precision(scan_type)`

5.5.15 cam_start_acquistion(scan_type, first_call)

This method starts the acquisition of a single camera image with the camera of thescan_type. When called the first time in this scan_type, the first_call parameter should be set to True. This ensures that, before the image acquisition starts, the method can (depending on the configuration of the logic module) activate various actuators as to bring the confocal microscope into the required configuration (e.g. wide field configuration). At subsequent calls with the same scan_type, the first_call parameter can be set to False.

If the current exposure time is less than a second, the method waits for image acquisition to finish before returning control. For longer exposure times, the acquisition is started and control is given back to the calling instance immediately (before end of acquisition). In the latter case, the method cam_wait_for_acquisition_end(scan_type) can be used to determine whether the image acquisition is still in progress. The final image can be retrieved with cam_fetch_image(scan_type) and get_last_cam_image().

Parameters:

- int scan_type: Index identifying the *scan type*. Must be larger or equal 0 and smaller or equal get_scan_type_max_index().
- bool first_call: True, if this is the first call of the class method for this scan_type. In such case actuators may be activated to bring the confocal microscope in the corresponding configuration. False otherwise.

Return value:

- bool: True if successful, otherwise False.

See also:

- cam_wait_for_acquisition_end(scan_type)
- cam_is_acquiring(scan_type)
- cam_fetch_image(scan_type)
- get_last_cam_image()
- cam_stop_acquistion(scan_type)

5.5.16 cam_wait_for_acquisition_end(scan_type)

This method returns two boolean values. The first one indicates whether we are currently waiting for the camera of the given `scan_type` to end an image acquisition. That is the case when `cam_start_acquistion(scan_type, first_call)` has been called before with an exposure time of 1 second or longer.

The second boolean return value indicates (if the first value is `True`) whether the camera is still acquiring or has finished acquisition.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return values (tuple):

- `bool`:
  - `True` if we are currently waiting for the camera of the given `scan_type` to end an image acquisition.
  - `False` if we are not waiting for an image acquisition to end, or if the given `scan_type` is not controlling a camera at all.
- `bool`:
  - `True` if the first return value is `True` and the camera is still acquiring.
  - `False` if the first return value is `True` and the acquisition has finished, and always `False` if the first return value is `False`.

See also:

- `cam_start_acquistion(scan_type, first_call)`
- `cam_is_acquiring(scan_type)`
- `cam_fetch_image(scan_type)`
- `get_last_cam_image()`
- `cam_stop_acquistion(scan_type)`
- `signal_cam_acquisition_finished()`

5.5.17 cam_is_acquiring(scan_type)

This method returns a value indicating whether the camera of the given `scan_type` is currently acquiring an image and has not yet finished acquisition. That can be the case when `cam_start_acquistion(scan_type, first_call)` has been called before with an exposure time of 1 second or longer.

The return value of this method is equivalent the second return value of the method `cam_wait_for_acquisition_end(scan_type)`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `bool`:
  - `True` if the camera is still acquiring.
  - `False` if the camera is not acquiring, or if the current `scan_type` is not controlling a camera at all.

See also:

- `cam_start_acquistion(scan_type, first_call)`
- `cam_wait_for_acquisition_end(scan_type)`
- `cam_fetch_image(scan_type)`
- `get_last_cam_image()`
- `cam_stop_acquistion(scan_type)`

5.5.18 cam_fetch_image(scan_type)

This method fetches the image of the camera of the given `scan_type` after a successful image acquisition, triggered by `cam_start_acquistion(scan_type, first_call)`.

After fetching the image, it emits a `signal_cam_update_display(scan_type)` signal. This can be used to call a function in the user interface module which uses the `get_last_cam_image()` method to update the screen display.

Then, after emitting the `signal_cam_update_display(scan_type)` signal the class method resets the "cam-is-waiting-for-acquisition"-flag, so that the method `cam_wait_for_acquisition_end(scan_type)` returns `False` in its first return value again.

Eventually it issues a `signal_cam_acquisition_finished()` signal before returning control to the calling instance. This signal can be used by the user interface to indicate to the user that the image acquisition has finished.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

See also:

- `cam_start_acquistion(scan_type, first_call)`
- `cam_wait_for_acquisition_end(scan_type)`
- `cam_is_acquiring(scan_type)`
- `get_last_cam_image()`
- `cam_stop_acquistion(scan_type)`
- `signal_cam_update_display(scan_type)`
- `signal_cam_acquisition_finished()`

5.5.19 get_last_cam_image()

This class method returns the image previously fetched by calling the function `cam_fetch_image(scan_type)`.

Return value:

- `int numpy.array`: Numpy array of shape (`hres`, `vres`), with `hres` being the horizontal resolution, and `vres` being the vertical resolution, containing the gray-scale values of the acquired image as integer values.

See also:

- `cam_start_acquistion(scan_type, first_call)`
- `cam_wait_for_acquisition_end(scan_type)`
- `cam_is_acquiring(scan_type)`
- `cam_fetch_image(scan_type)`
- `cam_stop_acquistion(scan_type)`

5.5.20 cam_stop_acquistion(scan_type)

This method prematurely stops an image acquisition previously started with `cam_start_acquistion(scan_type, first_call)` (This method is only relevant, if `cam_start_acquistion(scan_type, first_call)` was called with an exposure time of 1 second or longer).

The method also resets the "cam-is-waiting-for-acquisition"-flag, so that the method `cam_wait_for_acquisition_end(scan_type)` returns `False` in its first return value again.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

See also:

- `cam_start_acquistion(scan_type, first_call)`
- `cam_wait_for_acquisition_end(scan_type)`
- `cam_is_acquiring(scan_type)`
- `cam_fetch_image(scan_type)`
- `get_last_cam_image()`

5.5.21 get_additional_cam_settings(scan_type)

This class method returns a dictionary containing detailed information about all additional camera-specific settings of the camera of the given scan_type which are not covered by dedicated class methods of this interface.

The main structure of this dictionary looks like this:

```
{’group_1’ : {’label’ : <label>,
              ’setting_1’ : <sdef>,
              ’setting_2’ : <sdef>, ... }
 ’group_2’ : {’label’ : <label>,
              ’setting_1’ : <sdef>,
              ’setting_2’ : <sdef>, ... }
 ...
}
```

As shown above, all settings are assigned to groups. These groups have keys ’group_1’, ’group_2’, etc. in successive order. The placeholder <label> stands for a human-readable name of the corresponding group that could be used in an user interface, e.g. as the title of a frame.

The settings in each group have keys ’setting_1’, ’setting_2’, etc. in successive order. Each placeholder <sdef> represents a dictionary describing the properties of the setting. This is the structure of a <sdef> entry:

```
{’ID’: <ID>, ’label’: <label>, ’type’: <type>,
’dependent_on’: <depend_ID>, ’dependent_values’: <depend_value_list>
’default_value’: <default_value>, ’values’: <value_list}
```

This is what above placeholders stand for:

- string <ID> ... Unique ID identifying this setting
- string <label> ... Human-readable designation that could be used in an user-interface, e.g. to be displayed before the corresponding input field.
- string <type> ... Indicates the type of the setting. Currently the only implemented type is ’list’ which represents a setting where the user has to choose from a list of possible options.
- string <depend_ID> ... Optional identifier to indicate that this setting is dependent on another setting For example: If the setting with ID ’B’ is only to be displayed if the setting with ID ’A’ has the values 1 or 2, then the <depend_ID> of setting ’B’ would be ’A’.

- <depend_value_list> ... Optional. If there is a <depend_ID>, then the <depend_value_list> placeholder stands for a list of values. For example: If the setting with ID 'B' is only to be displayed if the setting with ID 'A' has the values 1 or 2, then the <depend_value_list> should be [1, 2]

- <default_value> ... The default value of this setting. In case of a setting of <type>:'list' this should be the list index of the default value.

- value_list ... If the setting is of <type>:'list', then the placeholder value_list stands for a list with n entries, where n is the number of possible list values. Each entry is yet another list with two entries, the first being a unique integer index, the second being a unique text string. For example, this is how the value_list would look like for two possible options: [[0, 'Electron Multiplying'], [1, 'Conventional']]

Here is an example of a complete additional settings dictionary:

```
{'group_1':
  {'label' :'Vertical Pixel Shift',
   'setting_1':
     {'ID': 'vs_speed',
      'label': 'Shift Speed',
      'type': 'list',
      'default_value': 4,
      'values': [[0, '0.3µs'], [1, '0.5µs'], [2, '0.9µs']]}},
 'group_2':
  {'label' : 'Horizontal Pixel Shift',
   'setting_1':
     {'ID': 'output_amp',
      'label': 'Output Amplifier',
      'type': 'list',
      'default_value': 0,
      'values': [[0, 'Electron Multiplying'], [1, 'Conventional']]},
   'setting_2':
       {'ID': 'hs_speed0',
        'dependent_on': 'output_amp',
        'dependent_values': [0],
        'label': 'Readout Rate',
        'type': 'list',
        'default_value': 0,
        'values': [[0, '17.0µs'], [1, '10.0µs'], [2, '5.0µs']]}
    'setting_3':
      {'ID': 'hs_speed1',
       'dependent_on': 'output_amp',
       'dependent_values': [1],
```

```
        'label': 'Readout Rate',
        'type': 'list',
        'default_value': 0,
        'values': [[0, '3.0µs'], [1, '1.0µs'], [2, '0.08µs']]}
  }
}
```

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `dictionary`: Dictionary with additional settings as explained above.

See also:

- `set_additional_cam_setting(scan_type, id, value)`

5.5.22 set_additional_cam_setting(scan_type, id, value)

This class method allows to set one of the "additional setting" as defined in the dictionary returned by `get_additional_cam_settings(scan_type)`.

Parameters:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.
- `string id`: The `<ID>` of the "additional setting" as defined in the dictionary returned by `get_additional_cam_settings(scan_type)`.
- `int value`: In case of settings of `<type>:'list'`, this is the index of the chosen list-entry.

See also:

- `get_additional_cam_settings(scan_type)`

5.6 Photon Counter Control Methods

5.6.1 get_pcounter_channel_names(scan_type, default="")

Every `scan_type` that is not controlling a camera has a data source of a (photon) counting device assigned. Every such data source may provide one or more data streams, called "channels". This could be, for example, the count-rates of two separate SPCM's which just have been grouped together in a "source", or it also may be the naked count-rate of just a single SPCM, combined with the somehow processed count-rate (e.g. counting only somehow correlated counts).

The class method `get_chnames` returns a list with the names of these data channels. If there are no channels (typically because the references `scan_type` is controlling a camera) then the list is either empty, or contains a single entry with the text string passed on to the method in the `default` parameter.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `str list[n]`: A list with *n* entries (where *n* is the number of channels for this source). Each list entry consists of a text string representing the channel's name.

See also:

- `get_cur_pcounter_rate(scan_type, ch_idx)`
- `get_cur_pcounter_channel_names()`

5.6.2 get_cur_pcounter_channel_names()

This method returns the channel name for the current *scan type* as set with `set_cur_scan_type_index(scan_type)`.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `str list[n]`: A list with *n* entries (where *n* is the number of channels for this source). Each list entry consists of a text string representing the channel's name.

See also:

- `get_pcounter_channel_names(scan_type, default="")`

5.6.3 get_cur_pcounter_rate(scan_type, ch_idx)

This class method triggers a new measurement of the photon counter source assigned to the `scan_type` and then returns a list containing the newly measured count-rates of all channels of the specified source.

Parameter:

- `int scan_type`: Index identifying the *scan type*. Must be larger or equal `0` and smaller or equal `get_scan_type_max_index()`.

Return value:

- `float list[n]`: A list with n entries (where n is the number of channels for this source). Each list entry contains the count-rate for the corresponding channel.

See also:

- `get_pcounter_channel_names(scan_type, default="")`

5.7 Signals

5.7.1 signal_xy_image_updated(scan_type)

This signal is emitted during a main scan every time a whole line is completed. It can be used to trigger a function in the user interface module, so that - by use of the class method `get_main_scan_image(scan_type)` - the screen image of the ongoing scan is continuously updated.

5.7.2 signal_depth_image_updated(scan_type)

This signal is emitted during a depth scan every time a whole line is completed. It can be used to trigger a function in the user interface module, so that - by use of the class method `get_depth_scan_image(scan_type)` - the screen image of the ongoing scan is continuously updated.

5.7.3 signal_cam_update_display(scan_type)

This signal is emitted by the `cam_fetch_image(scan_type)` method when an image is fetched from the camera after an image acquisition. It can be used to trigger a function in the user interface module, so that - by use of the class method `get_last_cam_image()` - the screen image can be updated.

5.7.4 signal_cam_acquisition_finished()

This signal is emitted by the `cam_fetch_image(scan_type)` method after the `signal_cam_update_display(scan_type)` signal. When this signal has been emitted, the `cam_wait_for_acquisition_end(scan_type)` method no longer returns `True` in the first return parameter.

5.7.5 signal_change_position(tag)

This signal is emitted every time the `set_position(tag, scan_index, ...)` method is called.

6 Software Installation and Configuration

6.1 Installation of all Components and Drivers

This chapter describes how to install the COMICS package, including all prerequisites, on Windows 10, 64-bit.

6.1.1 Install Git for Windows

- Get the Git-for-Windows installation file from `https://gitforwindows.org/`.
- Alternatively, use the `Git-2.29.1-64-bit.exe` installation file in the installation repository.
- Double-Click the installation file.
- Click through the installation wizard using all suggested default settings.
- Optionally, you may decide to also install TortoiseGit, a Windows Shell interface to Git (which is not required to run COMICS). If you want to do so, these are the required steps:
  - Download the TortoiseGit installer from `https://tortoisegit.org/`
  - Alternatively, use the `TortoiseGit-2.11.0.0-64bit.msi` installation file in the installation repository.
  - Run the installation wizard by double-clicking the installation file. Keep all default settings.

6.1.2 Install Anaconda

- Get the Anaconda Python 3.x "Individual Edition" distribution from `https://www.anaconda.com/products/individual`.
- Alternatively, use the `Anaconda3-2020.07-Windows-x86_64.exe` installation file in the installation repository.
- Double-Click the installation file.
- During the installation, select "Install for all users".
- When asked during the installation, check both checkboxes "Add Acaconda to the system PATH environment variable" and the "Register Anaconda3 as the system

Python 3.8", even if it says that that the first option is "not recommended" (see figure 7).

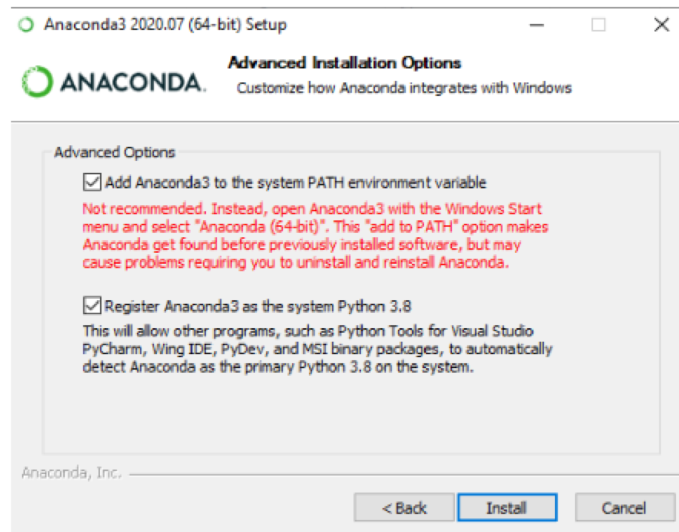


Figure 7: Check both check-boxes at this step.

6.1.3 Install PyCharm

- Get the PyCharm Community Edition installation file from the following page: `https://www.jetbrains.com/pycharm/download`
- Alternatively, use the `pycharm-community-2020.2.3.exe` file in the installation repository.
- Run the installation wizard by double-clicking the installation file. Keep all default settings.

6.1.4 Copy COMICS Source Code and Install Python Modules

- Copy the whole `COMICS` folder from the installation repository into a suitable folder onto your hard-disk.
- With the Windows file-explorer, navigate into the `COMICS\tools` sub-directory.
- Double-click the batch-file `install-python-modules-windows.bat` and wait until the message "`Installation procedure for conda environment 'qudi' finished.`" appears.
- Press any key to make the batch-file window disappear.

6.1.5 First Start and Configuration of PyCharm

- Start "PyCharm Community Edition" from the Start menu
- When asked, confirm the Licence Agreement
- When asked, choose whether or not you want PyCharm to send statistical data to the JetBrain company.
- When asked, select an UI theme.
- Then Click "Skip Remaining and Set Defaults"
- In the next Window, click "open"

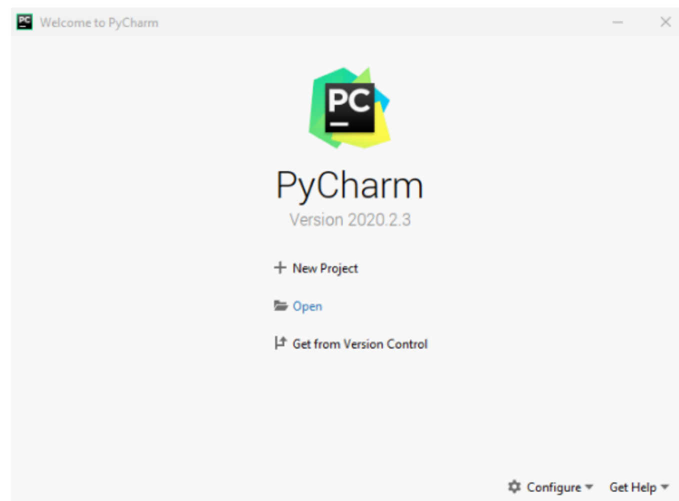


Figure 8: At this window, click on "Open".

- Select the COMICS source code folder you have copied onto your hard-drive and click "OK".

- After the source code has loaded, press Ctrl+Alt+S or go to File/Settings. In the upcoming window, select "Project: COMICS" on the left side, and then click on "Python interpreter"

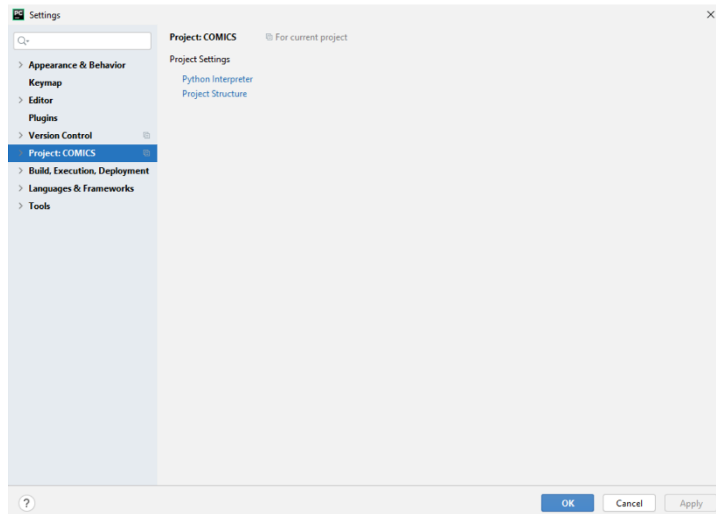
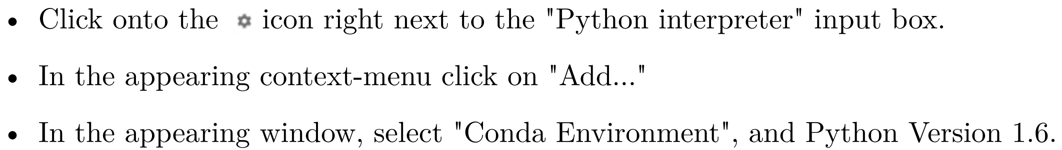


Figure 9: Select "Project: COMICS" and then click on "Python interpreter".



- Click onto the ⚙ icon right next to the "Python interpreter" input box.
- In the appearing context-menu click on "Add..."
- In the appearing window, select "Conda Environment", and Python Version 1.6.

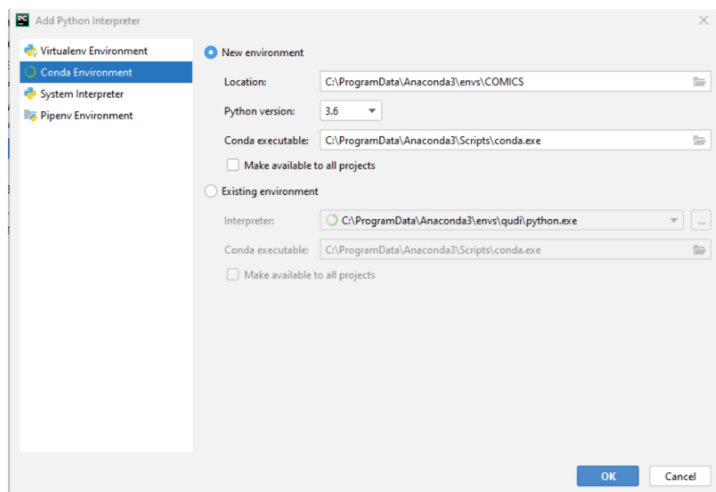


Figure 10: Select "Select "Conda Environment", and Python Version 1.6

- In the following window click "OK" once more.

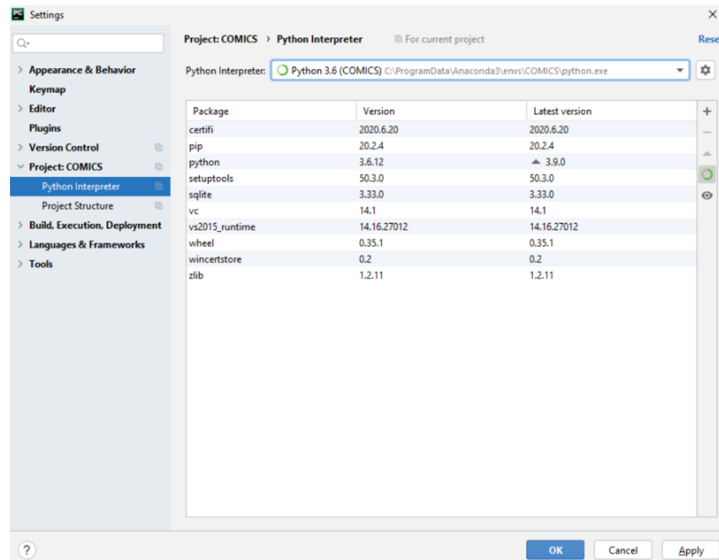


Figure 11: Click "OK" once more

- Now close PyCharm.

6.1.6 Install Python Modules in Python 3.6 Environment

Start "Anaconda Prompt" from the Windows start-menu, and enter the following lines (each concluded by pressing <ENTER>)

```
activate COMICS
pip install QtPy
pip install pyqt5-tools
pip install numpy
pip install ruamel.yaml
pip install fysom
pip install rpyc
pip install pyqtgraph
pip install gitpython
pip install qtconsole
pip install lmfit
pip install cycler
pip install matplotlib
pip install PyDAQmx
```

Now close the "Anaconda Prompt" window.

6.1.7 Install NI PyDAQ Driver Software

The NI PyDAQ Driver Software is required to use the `ati_ni_card` hardware module. To install the PyDAQ Driver software, which allows to access National Instruments IO cards, follow these instructions:

- Double click the `ni-daqmx_19.5_online_repack.exe` installation file in the `Drivers\DAQmx` directory of the installation repository.
- When asked, confirm all license agreements, and keep all default settings.
- After the installation, restart the computer.

6.1.8 Install PI Driver and Configuration Software

The PI Driver and SDK software (specifically, the PI USB driver, the PI GCS Library, and the PI Programming Files) are required to use the `ati_pi_positioner` hardware module. To install this driver software, which allows to access the E-727 and other digital multi-channel piezo microcontrollers from Physik Instrumente GmbH, follow these instructions:

- **PI USB driver**
  - Double click the `PI_USB_Driver_Setup_64Bit.exe` installation file in the `Drivers\PI` directory of the installation repository to run the installation wizard.
  - When asked, confirm the license agreement.
  - When asked, confirm that you trust software from "Physik Instrumente (PI) GmbH".
  - Finish the installation wizard.
- **PI GCS Library**
  - Double click the `PI_GCS_Library_PI_GCS2_DLL_Setup.exe` installation file in the `Drivers\PI` directory of the installation repository to run the installation wizard.
  - When asked, confirm the license agreement.
  - Finish the installation wizard.
- **PI Programming Files**
  - Double click the `PI_Programming_Files_PI_GCS2_DLL_Setup` installation file in the `Drivers\PI` directory of the installation repository to run the installation wizard.
  - When asked, confirm the license agreement.
  - Finish the installation wizard.

In addition to the abovementioned software components required to use the `ati_ni_card` hardware module, it makes sense to also install the PI MikroMove software. It allows to optimize the control parameters of the piezo servo controller, especially to compensate for undesired resonances in the mechanical system.

- **PI MikroMove Software**
  - Double click the `PIMikroMove_Setup.exe` installation file in the `Drivers\PI` directory of the installation repository to run the installation wizard.
  - When asked, confirm the license agreement.
  - Finish the installation wizard.

6.1.9 Install Thorlabs Kinesis Driver Software

The Thorlabs Kinesis driver software (specifically, the Kinesis software package, and the APT software package) is required to use the `ati_kinesis_positioner` and the `ati_kinesis_actuator` hardware modules. To install this software, which allows to control multiple types of Thorlabs devices, follow these instructions:

(1) Kinesis Software Package

- Double click the `kinesis_17290_setup_x64.exe` installation file in the `Drivers\Thorlabs` directory of the installation repository.
- When asked, confirm the license agreement, and keep all default settings.
- Choose "Complete Installation", when asked.
- Finish the installation wizard with all other settings at default.

(2) APT Software Package

- Double click the `APT_setup.exe` installation file in the `Drivers\Thorlabs` directory of the installation repository.
- When asked, confirm the license agreement, and keep all default settings.
- Choose "Complete Installation", when asked.
- Finish the installation wizard with all other settings at default.

6.1.10 Install Swabian Time Tagger Driver Software

The Swabian Time Tagger driver software is required to use the `ati_swabian_tt` hardware module. To install this software, which allows to control Time-Tagger devices from Swabian instruments, follow these instructions:

- Double click the `TimeTagger-2.4.4-64bit.msi` installation file in the `Drivers\TimeTagger` directory of the installation repository.
- When asked, confirm the license agreement, and keep all default settings.
- Finish the installation wizard.

6.1.11 Install Andor SDK Software

The Andor SDK software is required to use the `ati_andor` hardware module. To install this software, which allows to control Andor cameras from Oxford Instruments, follow these instructions:

- Double click the `AndorSDKSetup-2.103.30022.0.exe` installation file in the `Andor` directory of the installation repository.
- When asked, confirm the license agreement, and keep all default settings.
- When asked to choose between a 32-bit and a 64-bit driver, choose the 64-bit driver.

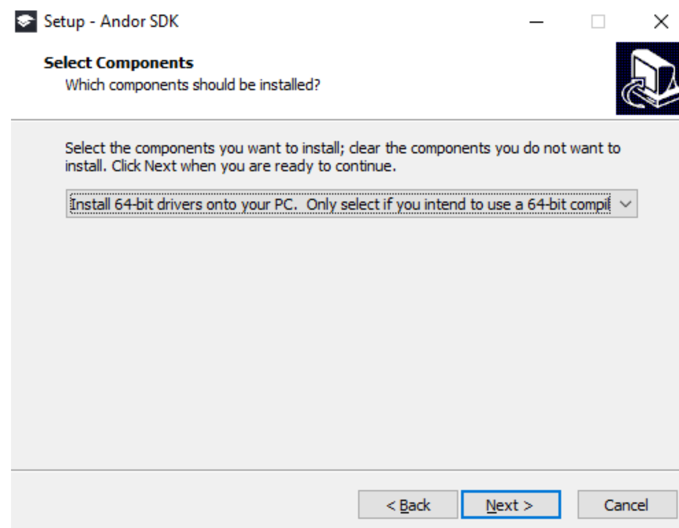


Figure 12: Choose the 64-bit Andor driver

- At the next screen, select the camera type matching your model.

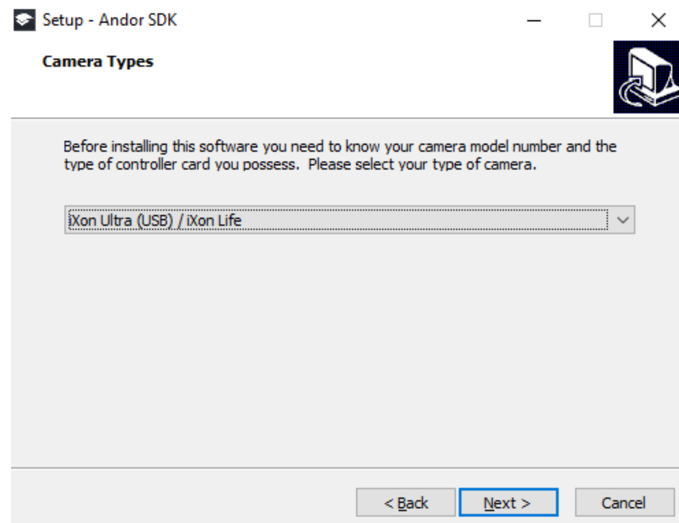


Figure 13: Select your Andor camera model

- Finish the installation wizard by clicking "Install" on the next screen.

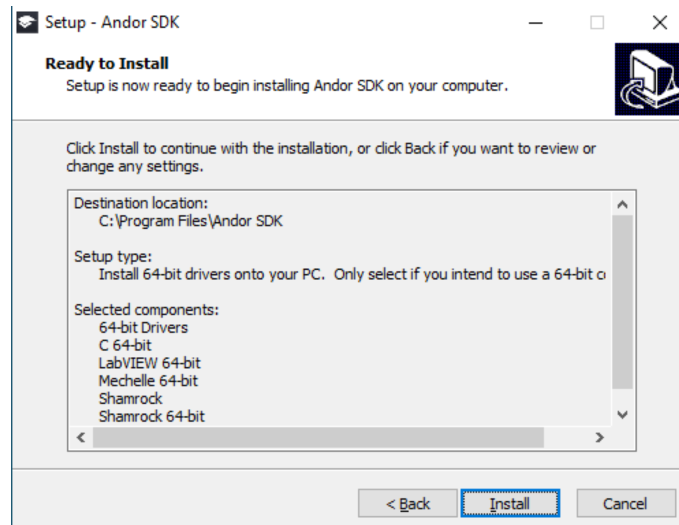


Figure 14: Click "Install"

- When asked, confirm that you trust software from "Andor Technology PLC".

6.1.12 Create Desktop Shortcut Icon

To create a Desktop shortcut follow these instuctions:

- On your Desktop, right click and then select New/Shortcut from the context menu.
- Enter the following into the "location" text field (without line-breaks):

```
<windir>\System32\cmd.exe "/K" <path-to-activation-script>
"<path-to-qudi-environment>" && <drivename> &&
cd "<path-to-qudi-directory>" && python "start.py"
```

- In the above string you need to replace the following placeholders:
  - `<windir>` : the windows directory, usually `C:\Windows`
  - `<path-to-activation-script>` : the path to the Anaconda activate.bat file, for example `C:\ProgramData\Anaconda3\Scripts\activate.bat`
  - `<path-to-qudi-environment>` : this can be found using command `conda info -envs` in a terminal, usually `C:\ProgramData\Anaconda3\envs\qudi`
  - `<drivename>` : Name of hard drive hosting the qudi directory followed by a colon, e.g. `C:` or `D:`
  - `<path-to-qudi-directory>` : the path where Qudi's `start.py` can be found.
- Click Next
- Enter the name for the shortcut : `COMICS`
- Click Finish
- Right click on the newly created shortcut and go to Properties
- Click "Change Icon"
- Browse and select the `COMICS` icon from the `artwork` folder of the `quidi` source code directory.

6.2 Software Configuration

This chapter describes how to configure the currently implemented COMICS modules in the Qudi configuration file.

6.2.1 Kinesis Positioner

This chapter describes the necessary entries in the `hardware` section of the configuration file to make stepper motors controlled by the `ati_kinesis_positioner` class available to the program logic. This is how an exemplary configuration for controlling three axes with Kinesis stepper motors looks like:

```
hardware:
    kinesis_positioner:
        module.Class: 'positioner.ati_kinesis_positioner.ati_kinesis_positioner'
        wrapper_dll: 'wrapper\\x64\\Release\\KCwrapper.dll'
        pos_ch1:
            name: 'motor'
            axis1:
                name: 'X'
                serial: 26002900
                step_range: [0, 54920433] # steps
                phys_range_hardware: [0, 25.1406] # mm
                center_pos: 3.000 #mm
                max_on_target_delta: 0.0001 #mm

            axis2:
                name: 'Y'
                serial: 26002928
                step_range: [0, 54920433] # steps
                phys_range_hardware: [0, 25.1406] #mm
                flip: True
                max_on_target_delta: 0.0001 #mm

            axis3:
                name: 'Z'
                serial: 26002855
                step_range: [0, 54920433] # steps
                phys_range_hardware: [0, 25.1406] # mm
                max_on_target_delta: 0.0001 #mm
                velocity: 75482536
```

The settings shown above have the following meaning:

- `kinesis_positioner:` This is the header of the Kinesis positioning hardware configuration section. The name can be freely chosen and it is an identifier by which the Kinesis motors can be referenced in the `logic` section of the configuration file. This header and the subsequent configuration options must be placed in the `hardware` section of the configuration file.
- `module.Class:` This string value references path, file name and class name of the `ati_kinesis_positioner` Qudi class.

- `wrapper_dll`: The `ati_kinesis_positioner` Qudi class needs the wrapper dll referenced by this string setting to access the Kinesis API.
- `pos_ch1`: The available axes (and hence the corresponding stepper motors) must be grouped into "positioning channels", identified by headers `pos_ch1`, `pos_ch2`, etc. User controls for stepper motors grouped together in one "positioning channel" are made available together in the user interface. Our exemplary configuration file uses just one positioning channel, and hence only has a single `pos_ch1` section.
- `name`: The entry `name` in each section `pos_ch1`, `pos_ch2`, etc. is a text string with a descriptive name of the motor group controlled by this "positioning channel". It appears in the user interface above the corresponding user controls.
- `axis1`, `axis2`, `axis3`: Each of these headers identify a configuration section relevant for one specific stepper motor.
- `name`: The entry `name` in sections `axis1`, `axis2` and `axis3` assigns an identifying name to each axis (here: `'X'`, `'Y'`, and `'Z'`).
- `serial`: This integer value represents the serial number of the respective stepper motor.
- `step_range`: Kinesis stepper motors are positioned via the Kinesis API by integer step values. This configuration parameter consists of two integer values representing the total positioning range measured in these step values.
- `phys_range_hardware`: This configuration parameter consists of two float values representing the total positioning range measured in millimeters.
- `max_on_target_delta`: This value represents the maximum allowed deviation in millimeters between target position and current actual position. If the current actual position is smaller or equal this value, then the current position is assumed to be "on target".
- `center_pos`: This optional float parameter (default value: `0`) must be within the `phys_range_hardware` range. It is interpreted as a millimeter value, and defines the zero position. For example, `center_pos: 3` means that the motor position `3 mm` (on the controller's display) will be interpreted as position `0 mm`. Consequently, a motor position `2 mm` (on the controller's display) would be interpreted as position `-1 mm`.
- `flip`: This optional boolean parameter (default value: False) defines, whether the coordinates are flipped. In the example above, the setting `flip: True` means that the physical range $0\,\text{mm} \ldots 25.1406\,\text{mm}$ (on the controller's display) is translated into a logical range $-25.1406\,\text{mm} \ldots 0\,\text{mm}$ If this setting is used together with `center_pos`, then the scale is mirrored around the defined center position.
- `velocity`: This optional integer parameter defines the stepper motor's velocity. The value range is dependent on the stepper motor model.

6.2.2 PI Positioner

This chapter describes the necessary entries in `hardware` section of the configuration file to make piezo stages and other piezo positioning devices controlled by the `ati_pi_positioner` class available to the program logic. This is how an exemplary configuration for controlling the three axes of a PI piezo stage looks like:

```
hardware:
    pi_positioner:
        module.Class: 'positioner.ati_pi_positioner.ati_pi_positioner'
        controller: 'PI E-727'
        serial: '0120024771'
        name: 'stage'

        axis1:
            name: 'X'
            output: '2'
            max_on_target_delta: 0.02 #um
            center: True
            flip: True

        axis2:
            name: 'Y'
            output: '1'
            max_on_target_delta: 0.02 #um
            steps_default: 50
            steps_min: 2
            steps_max: 1000
            center: True
            flip: True

        axis3:
            name: 'Z'
            output: '3'
            max_on_target_delta: 0.02 #um
            center: True
            flip: False
```

The meaning of these settings is explained below. Please note that, unlike with the Kinesis controller configuration described in the previous chapter, it is not necessary to define the physical range of each axis, as this information is provided by the PI controller.

- `pi_positioner:` This is the header of the PI positioning hardware configuration section. The name can be freely chosen and it is an identifier by which the PI controller can be referenced in the `logic` section of the configuration file. This header and the subsequent configuration options must be placed in the `hardware` section of the configuration file.
- `module.Class:` This string value references path, file name and class name of the `ati_PI_positioner` Qudi class.
- `controller:` This string value assigns a name to the controller.
- `serial:` This string value represents the controller's serial number. If there is only one PI controller attached, then this setting is optional. If there is more than one controller attached, this setting is mandatory to identify the controller.
- `name:` The string value `name` in the main section is a text string with a descriptive name of the controlled positioning device (e.g. `'piezo stage'`). It appears in the user interface above the corresponding user controls.
- `axis1`, `axis2`, `axis3`: Each of these headers identifies a specific axis.
- `name`: The string value `name` in sections `axis1`, `axis2` and `axis3` assigns an identifying name to each axis (here: `'X'`, `'Y'`, and `'Z'`).
- `output`: Internally, the PI controller identifies the axis by numbers. This setting links a logical axis (e.g. `'X'`) to the corresponding PI controller axis number (e.g. `'2'`).
- `max_on_target_delta`: This value represents the maximum allowed deviation in micrometers between target position and current actual position. If the current actual position is smaller or equal this value, then the current position is assumed to be "on target".
- `center`: This optional boolean parameter (default value: `False`) defines whether the physical axis range is translated into a logical range with the zero position in the center. For example, let us assume that the physical range is 0 µm . . . 50 µm. It that case, setting `center: True` means that this physical range is translated into a logical range −25 µm . . . 25 µm, with the −25 µm logical position corresponding to the 0 µm physical position, and the 25 µm logical position corresponding to the 50 µm physical position.
- `flip`: This optional boolean parameter (default value: False) defines whether the coordinates are flipped. For example, let us assume that the physical range is

0 µm . . . 50 µm. It that case, setting `flip: True` means that this physical range is translated into a logical range −50 µm . . . 0 µm, with the −50 µm logical position corresponding to the 0 µm physical position, and the 0 µm logical position corresponding to the 50 µm physical position. If this setting is used together with `center: True`, then the same range would be translated into −25 µm . . . 25 µm, with the −25 µm logical position corresponding to the 50 µm physical position, and the 25 µm logical position corresponding to the 0 µm physical position.

- `steps_default`: This optional integer setting defines the default number of steps along this axis in a scan process (default value 100 for main axes, and 50 for the depth axis)
- `steps_min`: This optional integer setting defines the minimum number of steps along this axis in a scan process (default value: `steps_default / 10`)
- `steps_max`: This optional integer setting defines the maximum number of steps along this axis in a scan process (default value: `steps_default * 10`)

When using a piezo microcontrollers from Physik Instrumente GmbH for the first time in a new physical setting, it makes sense to optimize the PID parameters of the piezo servo controller, especially to compensate for undesired resonances in the mechanical system.

This can be done by using the PI MIkroMove software (see chapter 6.1.8). Please follow the instructions in chapter 3.6 of the PIMIkroMove Software Manual (File `PIMikroMove_UserManual_EN_SM148E.pdf` in directory `Drivers\PI` of the software repository). The password to permanently save the settings into the controller's non-volatile memory is "advanced".

6.2.3 NI Card

This chapter describes the required entries in `hardware` section of the configuration file to make positioning devices controlled by the `ati_NI_card` class by means of analog voltage output available to the program logic.

This is how an exemplary configuration for controlling the two axes of an laser mirror looks like:

```
hardware:
    ati_ni_card:
        module.Class: 'ati_ni_card.ati_ni_card'

        positioner:
            channel1:
                name: 'mirror'
                axis1:
                    name: 'X'
                    output: '/Dev1/AO1' # analog output
                    input: '/Dev1/AI15' # analog input
                    max_on_target_delta: 0.05 #um
                    ctrl_method: 2
                    ctrl_steps: 5
                    ctrl_step_goal: 0.95
                    ctrl_step_time: 0.05
                    volt_range: [-8.8, 8.8]
                    phys_range: [-110e-6, 110e-6]
                    steps_default: 25
                    steps_min: 10
                    steps_max: 500

                axis2:
                    name: 'Y'
                    output: '/Dev1/AO0' # analog output
                    input: '/Dev1/AI14' # analog input
                    max_on_target_delta: 0.05 #um
                    ctrl_method: 2
                    ctrl_steps: 5
                    ctrl_step_goal: 0.95
                    ctrl_step_time: 0.05
                    volt_range: [-8.8, 8.8]
                    phys_range: [-110e-6, 110e-6]
                    steps_default: 25
                    steps_min: 10
                    steps_max: 500
```

The settings shown above have the following meaning:

- `ati_ni_card:` This is the header of the NI card hardware configuration section. The name can be freely chosen and it is an identifier by which the NI card can be referenced in the `logic` section of the configuration file. This header and the subsequent configuration options must be placed in the `hardware` section of the configuration file.
- `module.Class:` This string value references path, file name and class name of the `ati_NI_card` Qudi class.
- `positioner:` Besides the fully implemented positioning functionality (which needs to be configured under the `positioner` sub-header as shown above), the

ati_ni_card class also includes yet unfinished implementations for actuators and counters. Each of these functionalities will get their own sub-headers in the configuration file.

- channel1: All axes of one device must be grouped into "channels", identified by headers channel1, channel2, etc. User controls for devices grouped together in one "positioning channel" are made available together in the user interface. Our exemplary configuration file uses just one positioning channel, and hence only has a single channel1 section.
- name: The string value name in sections channel1, channel2, etc. assigns an identifying name to each channel (here: 'mirror').
- axis1, axis2: Each of these headers identifies a specific axis.
- name: The string value name in sections axis1 and axis2 assigns an identifying name to each axis (here: 'X' and 'Y').
- output: This string value defines the analog NI card output by which the given axis is controlled.
- input: This optional string value defines the analog NI card input by which the current position is indicated.
- max_on_target_delta: This value represents the maximum allowed deviation in micrometers between target position and current actual position. If the current actual position is smaller or equal this value, then the current position is assumed to be "on target".
- volt_range: This configuration parameter consists of two float values representing the total volt range.
- phys_range: This configuration parameter consists of two float values representing the total positioning range measured in meters.
- steps_default: This optional integer setting defines the default number of steps along this axis in a scan process (default value 100 for main axes, and 50 for the depth axis)
- steps_min: This optional integer setting defines the minimum number of steps along this axis in a scan process (default value: steps_default / 10)
- steps_max: This optional integer setting defines the maximum number of steps along this axis in a scan process (default value: steps_default * 10)

- `ctrl_method`: This optional integer setting (default value `1`) defines how the device is positioned along the given axes.
  - `ctrl_method: 1`
    - If no `input` channel is provided, then the voltage corresponding to the target position is calculated based on the `volt_range` setting and the `phys_range` setting and simply set as output value to the defined `output`. This is fast, but may not be accurate enough.
    - If an `input` channel is provided, then a calibration procedure is performed on startup. Each axis is moved to the extreme position given by the `volt_range` setting five times. The input voltage for each extreme position is recorded and averaged over these five attempts. Then, a linear calibration curve is calculated, matching output voltages to input voltages. The output voltage henceforth is set to a value according to this calibration curve, so that the expected input voltage corresponds to the desired target position.
    - For either of these two cases the following setting is relevant:
      - `ctrl_step_time`: This optional float value (default `0.05`) defines the time in seconds to wait after all output voltages have been set. In case multiple axes are moved simultaneously, the longest `ctrl_step_time` is taken.
  - `ctrl_method: 2`
    - This control method requires the `input` channel to be defined. The method works iteratively, with the number of steps defined by the `ctrl_steps` setting (default: `4`). In each step, the difference between the current input voltage and the target voltage is calculated. Then, this difference is multiplied with the `ctrl_step_goal` factor, which is a float value slightly smaller than `1` (default `0.95`). This now slightly discounted difference is added to the current output voltage. After having set all axes, the algorithm waits for the time period defined by `ctrl_step_time`, before repeating this procedure in the next iterative step.
    - These are the relevant settings for this control method:
      - `ctrl_steps`: This optional integer value (default `4`) defines the number of iterative steps.
      - `ctrl_step_goal`: This optional float value (default `0.95`) should be a number larger or equal `0.5` and smaller than `1`. In each step, the difference between the current target voltage and the current input voltage is multiplied with this factor, and then the result is added to the current output voltage.
      - `ctrl_step_time`: This optional float value (default `0.05`) defines the time in seconds to wait between the iterative steps. In case multiple axes are moved simultaneously, the longest `ctrl_step_time` is taken.

6.2.4 Swabian Time Tagger

This chapter describes the necessary entries in the `hardware` section of the configuration file to make the Swabian Time Tagger device available to the program logic. Below we show an exemplary configuration:

```
hardware:
    swabian_tt:
        module.Class: 'ati_swabian_time_tagger.ati_swabian_tt'
        acquisition_time: 1e11 # pico seconds
        serial: '1729000IBI'

        ch0:
            input_delay: 50

        ch1:
            test_signal: True

        ch2:
            test_signal: True
            divider: 2

        virt_ch1:
            type: 'delay'
            input: 0
            delay: 50

        virt_ch2:
            type: 'coincidence'
            input1: v1
            input2: 3
            window: 1e9 #ps

        virt_ch3:
            type: 'gated'
            input: 0
            gate_start: 3
            gate_stop: 4

        virt_ch4:
            type: 'combine'
            inputs: [1, 2, v3]

        output_source1:
            acquisition_time: 1e11 # ps optional

            output_channel1:
                name: 'SPCM1'
                input: 0

            output_channel2:
                name: 'coinc'
                input: v2
```

The settings shown above have the following meaning:

- swabian_tt: This is the header of the Swabian Time Tagger hardware configuration section. The name can be freely chosen and it is an identifier by which the Time Tagger can be referenced in the logic section of the configuration file. This header and the subsequent configuration options must be placed in the hardware section of the configuration file.
- module.Class: This string value references path, file name and class name of the ati_swabian_tt Qudi class.
- acquisition_time: This optional setting in the swabian_tt section defines the acquisition time in picoseconds, provided a more specific acquisition time has not been defined in the relevant output_source section. Default value: 10e11 ps.
- serial: This string value represents the Time Tagger's serial number. If there is only one Swabian Time Tagger device attached, then this setting is optional. If there is more than one controller attached, this setting is mandatory to identify the intended Time Tagger.
- **Channel configuration**: Input channels can be optionally configured for simple signal manipulations. Such configuration needs to be introduced by a header ch followed by the channel number. For example, the configuration of channel 1 must be introduced by a header ch1. Under each header, the following optional configuration settings (which can also be combined) are available:
  - input_delay: This option expects an integer delay-time in picoseconds. The signals from this channel are then consequently registered with the configured delay time.
  - test_signal: If this boolean setting is set to True, the actual input signal is replaced by a test signal.
  - divder: This integer setting defines that the number of counts is to be divided by the given number. For example, divder: 2 will lead to a halfed count rate at this channel.
- **Virtual channel configuration**: Besides the actual input channels, so-called virtual channels can be defined. These channels need to be configured under consecutive headers named virt_ch1, virt_ch2 etc. Under each header the following configuration options are available:
  - type: This string value defines the type of virtual channel:
    * type: 'delay': This channel type takes an (actual or virtual) channel as input and creates an output where the input signals are delayed for a defined period of time. Hence, for actual channels this is equivalent to the channel configuration's input_delay option. These are the required configuration settings:

- `input`: This defines the input channel. Integer numbers `0`, `1`, `2`, etc. represent actual input channels, whereas `v1`, `v2`, `v3`, etc. represent virtual channels.
- `delay`: integer delay-time in picoseconds.

* `type: 'coincidence'`: This channel type takes two (actual or virtual) channels as input, and creates an output signal every time an input signal is within a defined time window. These are the required configuration settings:
  - `input1`: This defines the first input channel. Integer numbers `0`, `1`, `2`, etc. represent actual input channels, whereas `v1`, `v2`, `v3`, etc. represent virtual channels.
  - `input2`: This defines the second input channel. As before, integer numbers `0`, `1`, `2`, etc. represent actual input channels, whereas `v1`, `v2`, `v3`, etc. represent virtual channels.
  - `window`: coincidence time-window in picoseconds.
* `type: 'gated'`: This channel type takes an (actual or virtual) input channel, an (actual or virtual) gate-start channel, and an (actual or virtual) gate-stop channel as input. The output then includes exactly these signals from the input channel which lie between a gate-start and a gate-stop signal. These are the required configuration settings:
  - `input`: This defines the input channel. Integer numbers `0`, `1`, `2`, etc. represent actual input channels, whereas `v1`, `v2`, `v3`, etc. represent virtual channels.
  - `gate_start`: This defines the gate-start channel. Again, integer numbers `0`, `1`, `2`, etc. represent actual input channels, whereas `v1`, `v2`, `v3`, etc. represent virtual channels.
  - `gate_stop`: This defines the gate-stop channel. Once more, integer numbers `0`, `1`, `2`, etc. represent actual input channels, whereas `v1`, `v2`, `v3`, etc. represent virtual channels.
* `type: 'combine'`: This channel type takes two or more (actual or virtual) input channels and combines them into one signal. These are the required configuration settings:
  - `inputs`: Here, a list of one or more (actual or virtual) input channels is expected. The above sample configuration merges the actual input of channels 1 and 2 and the virtual channel v3.

- **Output-sources**: A single Swabian Time Tagger device may process the input from several data sources simultaneously. A "data source" in this context may be an actual single hardware device, which in itself may provide one or more output streams. Alternatively, a data source is just a logical grouping of two or more separate devices, like multiple SPCMs. All data sources belonging together logically should be grouped into a single "output-source". The settings for all output-sources can be configured under consecutive headers named `output_source1`, `output_source2`, etc. Under each header the following configuration options are available:
  - `acquisition_time`: This optional setting in each `output_source` section defines the specific acquisition time in picoseconds for the given `output_source` and overrules the general setting in the `sqabian_tt` section.
  - **Output-channel configuration**: Every output-source may provide one or more data streams, called "output-channels". This could be, for example, the count-rates of two separate SPCM's which just have been grouped together in a output-source, or it also may be the naked count-rate of just a single SPCM, combined with the somehow processed count-rate from a virtual channel. These output-channels need to be configured under consecutive headers named `output_channel1`, `output_channel2`, etc. Under each header the following configuration options are available:
    - `name`: This string setting designates a name to the related data stream.
    - `input`: This defines the input channel from which the data stream is drawn. Integer numbers `0`, `1`, `2`, etc. represent actual input channels, whereas `v1`, `v2`, `v3`, etc. represent virtual channels.

6.2.5 Andor Camera

This chapter describes the necessary entries in the `hardware` section of the configuration file to make an Andor camera available to the program logic. Below we show an exemplary configuration:

```
hardware:
    camera:
        module.Class: 'camera.andor.ati_andor.ati_andor'
        dll_location: 'C:\\Program Files\\Andor SDK\\atmcd64d.dll' # path to library file
        serial: 12139
        exposure: 0.01
        read_mode: 'IMAGE'
        acquisition_mode: 'SINGLE_SCAN'
        trigger_mode: 'INTERNAL'
        temperature: -85
        cooler_on: True
        provides_temperature_information: True
```

The settings shown above have the following meaning:

- camera: This is the header of the Andor camera configuration section. The name can be freely chosen and it is an identifier by which the Andor camera can be referenced in the logic section of the configuration file. This header and the subsequent configuration options must be placed in the hardware section of the configuration file.
- module.Class: This string value references path, file name and class name of the ati_andor Qudi class.
- dll_location: This string value references path and file name of the Andor driver library dll.
- serial: This integer value represents the camera's serial number. If there is only one Andor camera attached, then this setting is optional. If there is more than one Andor camera attached, this setting is mandatory to identify the intended camera.
- exposure: This float value indicates the initial exposure time in seconds.
- read_mode: This optional string value represents the read-mode. Currently only the 'IMAGE' mode has been tested. Other (not yet tested) settings are: 'FVB', 'MULTI_TRACK', 'RANDOM_TRACK' and 'SINGLE_TRACK'. The default value is 'IMAGE'. For details please see the Andor documentation.
- acquisition_mode: This optional string value represents the acquisition mode. Currently only the 'SINGLE_SCAN' mode has been tested. Other (not yet tested) settings are: 'ACCUMULATE', 'KINETICS', 'FAST_KINETICS' and 'RUN_TILL_ABORT'. The default value is 'SINGLE_SCAN'. For details please see the Andor documentation.
- trigger_mode: This optional string value represents the trigger mode. Currently only the 'INTERNAL' mode has been tested. Other (not yet tested) settings are: 'EXTERNAL', 'EXTERNAL_START', 'EXTERNAL_EXPOSURE', SOFTWARE_TRIGGER and 'EXTERNAL_CHARGE_SHIFTING'. The default value is 'INTERNAL'. For details, please see the Andor documentation.
- temperature: This optional integer value represents the target temperature in degrees Celsius. It takes only effect if the cooler is activated. The default value is -70 degrees Celsius.
- cooler_on: This optional boolean setting turns the camera cooling system on if it is set to True, otherwise the cooler is turned off. The default value is True.
- provides_temperature_information: This optional boolean setting indicates whether the camera has a temperature sensor and returns information about the current actual temperature. The default value is True.

6.2.6 Kinesis Actuators

This chapter describes the necessary entries in `hardware` section of the configuration file to Kinesis flip mounts and other Kinesis actuators controlled by the `ati_kinesis_actuator` class available to the program logic. This is how an exemplary configuration for controlling two Kinesis flip mounts looks like:

```
hardware:
    kinesis_actuator:
        module.Class: 'actuator.ati_kinesis_actuator.ati_kinesis_actuator'
        wrapper_dll: 'flipper_wrapper\\x64\\Release\\FlipperWrapper.dll'

        channel1:
            name: 'WF-1'
            serial: 37002468

        channel2:
            name: 'WF-2'
            serial: 37002400
```

The settings shown above have the following meaning:

- `kinesis_actuator:` This is the header of the Kinesis actuator hardware configuration section. The name can be freely chosen and it is an identifier by which the Kinesis actuators can be referenced in the `logic` section of the configuration file. This header and the subsequent configuration options must be placed in the `hardware` section of the configuration file.
- `module.Class:` This string value references path, file name and class name of the `ati_kinesis_actuator` Qudi class.
- `wrapper_dll:` The `ati_kinesis_actuator` Qudi class needs the wrapper dll referenced by this string setting to access the Kinesis API.
- `pos_channel1`, `pos_channel2`, etc.: Each available actuator must be assigned to a channel, identified by headers with subsequent numbering `pos_channel1`, `pos_channel2`, etc.
- `name:` This string setting assigns a name to the channel (and, hence, to the actuator).
- `serial:` This integer value represents the actuator's serial number. This setting is mandatory to identify the intended actuator.

6.2.7 COMICS Logic Module

This chapter describes the necessary entries in `logic` section of the configuration file for configuring the COMICS logic module. This configuration determines the type and number of so-called *scan types*, where each *scan type* has a unique identifying name, and typically represents a configuration for creating an image by scanning along two of two or three available axes with a positioning device like a stage or a laser mirror. Alternatively, a specific *scan type* can represent a configuration where the image is taken by a camera.

The User Interface module then queries the program logic about the configured *scan types*, and builds up accordingly (see chapter 6.2.8).

On the following page we show a typical configuration for a confocal microscope which provides a wide-filed camera, laser mirrors, a piezo-stage, stepping motors for pre-positioning, and Kinesis flip-mounts to switch configuration between wide field and scanning.

```
logic:
    ati_scanner4:
        module.Class: 'ati_confocal_logic4.ati_ConfocalLogic4'

        connect:
            positioner1: 'kinesis_positioner' # motors
            positioner2: 'pi_positioner' # piezo stage
            positioner3: 'ati_ni_card' # controls the mirrors
            camera1: 'camera'
            counter1: 'swabian_tt'
            actuator1: 'kinesis_actuator'

        scan_type_1:
            name: 'Wide Field'
            is_camera: True
            camera: 1
            pre_positioner: 1
            pre_positioner_ch: 1
            coord_offset_positioner_channels: [[1,1]]
            phys_range:
                - [-50e-6, 50e-6]
                - [-50e-6, 50e-6]
            act_ini: 'AC1:CH2:1:W100'

        scan_type_2:
            name: 'Mirror Scan'
            positioner: 3 # use ati_ni_card for controlling the mirrors
            positioner_ch: 1 # use configuration channel 1 of ati_ni_card
            coord_offset_positioner_channels: [[1,1], [2,1]]
            main_scan_axes: [1, 2]
            min_stable_time: 0.1 #s
            readjust_time: 1
            pre_positioner: 1 # use motors for pre-positioning
            pre_positioner_ch: 1
            counter: 1
            counter_source: 1
            act_ini: 'AC1:CH2:0:W100'

        scan_type_3:
            name: "Piezo Stage Scan"
            positioner: 2 # piezo stage
            coord_offset_positioner_channels: [[1,1], [3,1]]
            main_scan_axes: [1, 2]
            depth_scan_axes: [2, 3]
            min_stable_time: 0.1 #s
            pre_positioner: 1 # use motors for pre-positioning
            pre_positioner_ch: 1
            counter: 1
            counter_source: 1
            act_ini: 'AC1:CH2:0<:W100'
```

The settings shown on the previous page have the following meaning:

- `ati_scanner4:` This is the header of the COMICS logic module section. The name can be freely chosen and it is an identifier by which the logic module can be referenced in the `gui` section of the configuration file. This header and the subsequent configuration options must be placed in the `logic` section of the configuration file.
- `module.Class:` This string value references path, file name and class name of the `ati_ConfocalLogic4` Qudi class.
- `connect:` This header introduces the configuration section which links the `logic` layer to the `hardware` layer.
  - `positioner1`, `positioner2`, etc.: These consecutive string settings refer to headers of positioning device configurations in the hardware section.
  - `camera1`, `camera2`, etc.: These consecutive string settings refer to headers of camera configurations in the hardware section. The sample configuration refers to only one camera configuration.
  - `counter1`, `counter2`, etc.: These consecutive string settings refer to headers of counter configurations in the hardware section. The sample configuration refers to only one counter configuration.
  - `actuator1`, `actuator2`, etc.: These consecutive string settings refer to headers of actuator configurations in the hardware section. The sample configuration refers to only one actuator configuration.
- `scan_type1`, `scan_type2`, etc.: Each of these consecutive headers introduces a configuration section for a specific *scan type*. A *scan type* typically represents a configuration for creating an image by scanning along two of two or three available axes with a positioning device like a stage or a laser mirror. Alternatively, a specific *scan type* can represent a configuration where the image is taken by a camera.
  - **Scan-type configuration settings for cameras** (see `scan_type1` in the sample configuration):
    - `name:` This string setting designates a name to the *scan type* (e.g. "Camera").
    - `is_camera:` This optional boolean setting must be set to `True` for *scan types* representing a camera.
    - `camera:` This integer setting selects one of the cameras from the `connect` section.
    - `pre_positioner:` This integer setting selects one of the positioners from the `connect` section as "pre-positioning" device.
    - `pre_positioner_ch:` This optional integer setting selects a channel of the chosen pre-positioner (only relevant for positioners allowing multiple

channels like the Kinesis Positioner or the NI Card positioner). Default value: `1`.

* `coord_offset_positioner_channels`: This optional setting expects a list of lists in the format $[[p_1, c_1], [p_2, c_2], \ldots]$, with each p_i representing a positioner number according to the `connect` section, and c_i representing a positioner channel (or channel 1 for positioners without channels). The default value is an empty list. If this setting is omitted, then the pixels coordinates of the camera image are converted into physical coordinates according to the `phys_range` setting. The current position of the pre-positioning device is not considered. If, however, a list of one or more (pre-)positioning devices (and channels) is provided, then the positions of these positioning devices and pre-positioning devices is also be taken into account. For example, if the setting is omitted, then the coordinate $x = 0\,\mu\text{m}$ will (according to our sample configuration) identify the x-center position of the image. However, if the setting references the pre-positioning stepper motors (like in our example), and if the pre-positioning stepper-motor has moved the whole stage to $x = 3\,\text{mm}$, then the very same x-center position of the stage is now referenced as $x = 3\,\text{mm}$.
* `phys_range`: This setting expects a list of two lists with two float values each. These two lists then represent the physical range of the horizontal and vertical axis in meters.
* `act_ini`: This optional string setting encodes commands send to the actuator(s) before the first image is taken in this *scan mode*. All commands are to be separated by colon (':'). These are the allowed commands:
  · `AC`: The command `AC` followed by an integer number selects an set of actuators according to the `connect` configuration section. For example, `'AC1'` selects set number one. In our sample configuration file this would be the Kinesis acutators. By default (if this command is omitted), set `1` is selected.
  · `CH`: The command `CH` followed by an integer number selects an channel for the current set of actuators according to the corresponding configuration in the `hardware` section. For example, `'AC1:CH2'` selects the second channel of set number one.
  · `number`: A integer or float number without preceding letters is interpreted as an output value. If the current actuator and channel expects a boolean value, then `0` is interpreted as `False`, and `1` is interpreted as `True`. For example, `'AC1:CH2:1'` brings the corresponding actuator device into position `1`.
  · `W`: The command `W` followed by an integer number is interpreted as wait time in milliseconds. For example, `'AC1:CH2:1:W100'` brings the

corresponding actuator device into position `1` and then the software waits for 100 ms.

- **Scan-type configuration for controlling positioning devices** (see `scan_type2` and `scan_type3` in the sample configuration):
  * `name`: This string setting designates a name to the *scan type* (e.g. "Stage Scan" or "Mirror Scan")
  * `is_camera`: This optional boolean setting must be omitted or set to `False` for regular *scan types* representing a positioning device.
  * `same_resolution`: This optional setting expects a list of two integer values, both either `1`, `2`, or `3`, but different from one another. These values define which two axes should be forced to have the same "scan resolution" (number of scan steps).
  * `positioner`: This integer setting selects one of the positioners from the `connect` section as positioning device.
  * `positioner_ch`: This optional integer setting selects a channel of the chosen positioner (only applicable for positioners allowing multiple channels like the Kinesis Positioner or the NI Card positioner). Default value: `1`.
  * `coord_offset_positioner_channels`: This optional setting expects a list of lists in the format $[[p_1, c_1], [p_2, c_2], \ldots]$, with each p_i representing a positioner number according to the `connect` section, and c_i representing a positioner channel (or channel 1 for positioners without channels). The default value is an empty list. If this setting is omitted, then the scan pixels coordinates are just converted into physical coordinates according to coordinate system of the scanning device (e.g. the piezo stage). The current positions of other positioning devices or the pre-positioning device is not considered. If, however, a list of one or more (pre-)positioning devices (and channels) is provided, then the positions of these positioning devices and of the pre-positioning devices is also taken into account. For example, if the setting is omitted, then the coordinate $x = 50\,\mu\text{m}$ may identify the x-center position of a stage with a total range of 100 µm in x-direction. However, if the setting references the pre-positioning stepper motors, and the x-axis stepper-motor has moved the whole stage to $x = 3\,\text{mm}$, then the very same x-center position of the stage is referenced as $x = 3.05\,\text{mm}$.
  * `main_scan_axes`: This optional setting expects a list of two different integer values. The first integer value (`1`, `2`, or `3`) defines which axis is the horizontal axis and the second integer value (also `1`, `2`, or `3`) defines which axis is the vertical axis in a `main scan`. The default value is `[1, 2]`.

* `depth_scan_axes`: This optional setting expects a list of two different integer values. The first integer value (`1`, `2`, or `3`) defines which axis is the horizontal axis and the second integer value (also `1`, `2`, or `3`) defines which axis is the vertical axis in a `depth scan`. The default value is `[1, 3]`. This setting is only applicable if the positioning device has more than two axes.
* `min_stable_time`: This optional setting (default value: `0`) defines the minimal time in seconds both scan coordinates must remain within tolerance before they are accepted as valid.
* `readjust_time`: This optional setting defines the time in seconds after which the scan logic makes a second (singulary) attempt to position the main positioning device if the current coordinates are still not on target plus/minus tolerance. This strategy has proven to allow for a smaller positioning tolerance especially with laser mirrors. If the setting is omitted, then no re-positioning attempt is made.
* `pre_positioner`: This integer setting selects one of the positioners from the `connect` section as pre-positioning device.
* `pre_positioner_ch`: This optional integer setting selects a channel of the chosen pre-positioner (only applicable for positioners allowing multiple channels like the Kinesis Positioner or the NI Card positioner). Default value: `1`.
* `act_ini`: This optional string setting encodes commands send to the actuator(s) before the first image is taken in this *scan mode*. All commands are to be separated by colon (':'). These are the allowed commands:
  - `AC`: The command `AC` followed by an integer number selects a set of actuators according to the `connect` configuration section. For example, `'AC1'` selects set number one. In our sample configuration file this would be the Kinesis acutators. By default (if this command is omitted), the set `1` is selected.
  - `CH`: The command `CH` followed by an integer number selects an channel for the current set of actuators according to the corresponding configuration in the `hardware` section. For example, `'AC1:CH2'` selects the second channel of set number one.
  - `number`: A integer or float number without preceding letters is interpreted as an output value. If the current actuator and channel expects a boolean value, then `0` is interpreted as `False`, and `1` is interpreted as `True`. For example, `'AC1:CH2:0'` brings the corresponding actuator device into position `0`.
  - `W`: The command `W` followed by an integer number is interpreted as wait time in milliseconds. For example, `'AC1:CH2:0:W100'` brings the

corresponding actuator device into position 0 and then the software waits for 100 ms.

6.2.8 COMICS User Interface

This chapter describes the necessary entries in gui section of the configuration file for configuring the COMICS user interface module.

Below we show a typical configuration for the COMICS user interface.

```
gui:
    ati_confocalgui4:
        module.Class: 'confocal.ati_confocalgui4.ConfocalGui4'
        connect:
            confocallogic4: 'ati_scanner4'
```

The settings shown on the previous page have the following meaning:

- ati_confocalgui4: This is the header of the COMICS user interface module section. The name can be freely chosen and it is an identifier by which the user interface module can be referenced by other gui modules. This header and the subsequent configuration options must be placed in the gui section of the configuration file.
- module.Class: This string value references path, file name and class name of the ati_ConfocalGui4 Qudi class.
- connect: This header introduces the configuration section which links the gui layer to the logic layer.
    - confocallogic4: This string setting refers to the header of the confocal logic module configuration in the logic section.

7 User's Guide

This chapter explains how to use the COMICS user interface based on the exemplary configuration from chapter 6.2.

7.1 Start Qudi and COMICS

After starting the Qudi software, the Qudi Manager shows up (see fig. 15).

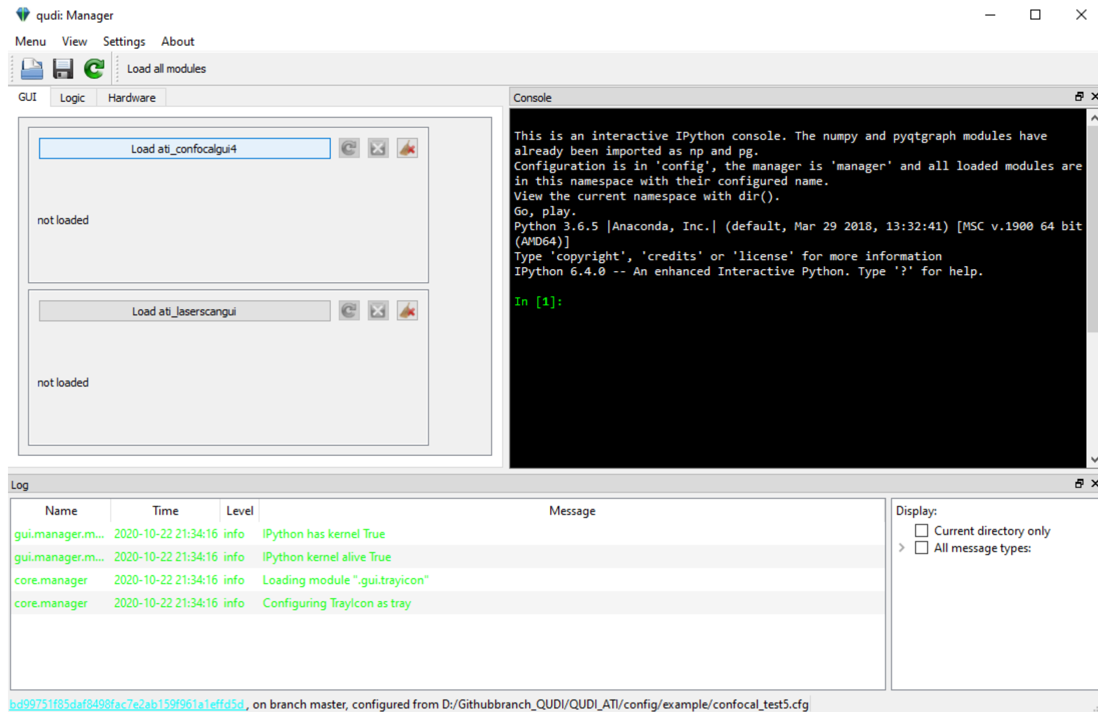


Figure 15: Qudi program manager

On start-up, the Qudi Manager loads the previously used configuration file and provides a start-button for each module represented in this configuration file. Hence, to load the COMICS user interface, a corresponding configuration file with all required COMICS modules (like the one from chapter 6.2) must either already have been used last time, or must explicitly be opened using the "File Open" button in the button-bar.

To start the COMICS user interface, the button `ati_confocalgui4` on the `GUI` tab must be clicked. The COMICS user interface will then be loaded and executed. Right after start-up, the user interface loads the corresponding program logic as configured in the `ati_confocalgui4.connect` section of the configuration file (see chapter 6.2.8), which then consequently loads all required hardware modules (again, based on the corresponding `connect` section in the `logic` section of the configuration file).

When all connected modules are loaded, the User Interface queries the program logic about the total number of *scan modes*, their individual names and all other configuration details.

Therefore, almost everything in the user interface's appearance is either determined by the configuration of the program logic (see chapter 6.2.7), or by the configuration details of the related hardware modules (see chapters 6.2.1 - 6.2.6).

For example, a configuration similar to the one presented in chapter 6.2 will create an user interface with three tabs (one for each `scan mode`), and will look like the screenshot in Figure 16.

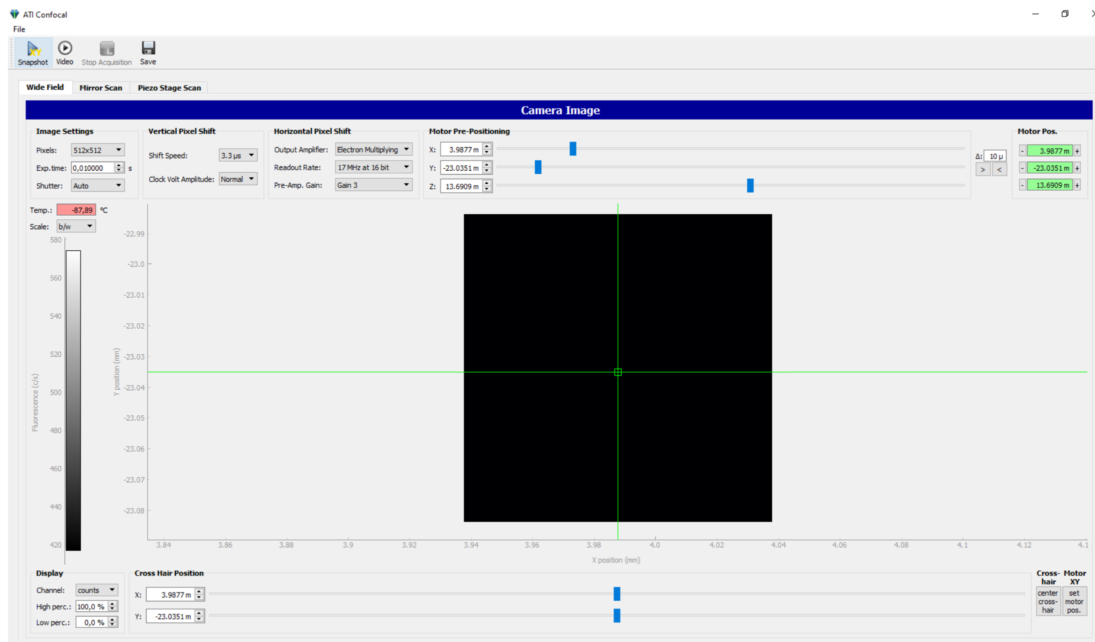


Figure 16: The COMICS start screen.

The three *scan modes* in the `logic` part of the configuration file are represented in the user interface by three tabs, showing the scan mode names (here: "Wide Field", "Mirror Scan", and "Piezo Stage Scan")

7.2 Camera Scan Mode

7.2.1 Image Settings

The "Image Settings" frame on the top left of the window hosts the following settings:

- **Pixels:** This dropdown-box allows to choose the image resolution. For Andor cameras, resolutions below the maximum resolution are realized by pixel binning.
- **Exposure Time:** This input box allows you to set an exposure time within the valid range of the connected camera.
- **Shutter:** This dropdown-box is only visible if the connected camera has an mechanical shutter. The selectable values depend on the connected camera. The currently implemented `ati_andor` module offers the settings "Auto", "Open", and "Closed".

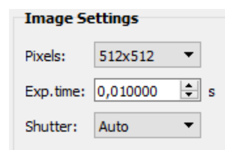


Figure 17: Camera image settings.

7.2.2 Extended Camera Settings

If the connected camera provides additional settings, these are displayed right next to the "Image Settings" frame. For example, the Andor camera provides the settings "Shift speed" and "Clock Voltage Amplitude" in the "Vertical Pixel Shift" frame, and "Output Amplifier", "Readout Rate" and "Pre-Amp. Gain" in the "Horizontal Pixel Shift" frame.

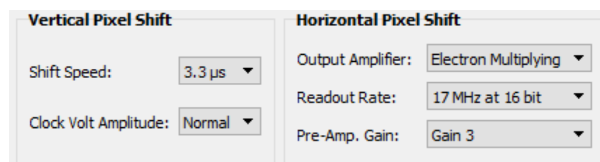


Figure 18: Extended settings for the Andor camera.

7.2.3 Pre-Positioning

Provided that a `pre_positioner` has been defined in the according `scan_type` section of the configuration file (see chapter 6.2.7), the user interface will show pre-positioning controls as shown in Figure 19.

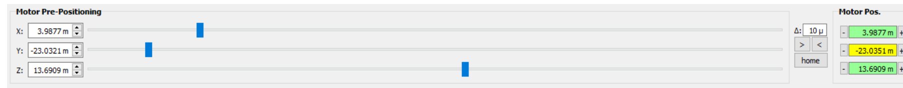


Figure 19: Pre-positioning controls for the camera.

To move the pre-positioning axes (here: the stepper-motors) to specific target-coordinates, one can either enter these target-coordinates into the leftmost input fields, or alternatively use the sliders. Clicking the "`>`"-button triggers the motors to move to these target coordinates.

The coordinate ranges, positions of the zero-coordinate and the coordinate orientations for each axis are determined by the pre-positioning hardware module connected by the program logic. In our case, that would be the settings `phys_range_hardware`, `center_pos`, and `flip` in the `kinesis_positionier` configuration section (see chapter 6.2.1).

The rightmost fields show the current motor coordinates. The back-color indicates the status:

- **green**: This pre-positioning axis is steady and currently at the target position.
- **yellow**: This pre-positioning axis is steady, but currently not at the target position.
- **red**: This pre-positioning axis is either currently moving and has not yet reached the target position, or is supposed to already be at the target position, but did not reach the target position within tolerance.

The little "`+`" and "`-`" buttons to the left and right side of the rightmost fields allow to directly "jog" the pre-positioning motors in small delta-steps. The step-size is determined by the "Δ" input field.

To copy the current motor position to the target coordinate input fields (and sliders), the "`<`"-button must be clicked.

If at least one of the pre-positioning stepper-motors provides a "homing" function (where the stepper-motor is moved to a "home" position for calibration), a "home"-button is additionally displayed below the ">" and "<" buttons. Clicking this button starts the "homing" procedure for all stepper-motors.

7.2.4 Temperature

On the middle-left side of the window there is a field displaying the current temperature of the camera's image sensor, provided that the camera provides that information. The back-color encodes the temperature status:

- **green**: The target temperature (as configured in the camera's configuration section) has been reached and is stable.
- **red**: The target temperature has not (yet) been reached, or is unstable.
- **gray**: There is no status information available.

7.2.5 Color Scale Setting

Also on the left side, below the temperature information, there is a drop-down box allowing to choose the color scale to be used for an acquired image. For *scan types* representing a camera, the default setting is "b/w" (for "black-and-white"). Alternatively, a color-coded scale can be chosen.

7.2.6 Display Settings

- **Channel:** For `scan-types` representing a camera this setting is fixed to "counts".
- **High. perc.**, **Low. perc.:** These values can be used to increase the contrast of an acquired image.

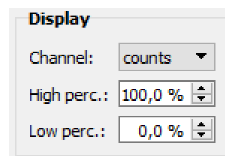


Figure 20: Camera image display settings.

7.2.7 Crosshair Position Controls

The crosshair of the main image can be moved directly by dragging it with the mouse. Alternatively, the cross-hair position controls in the lower middle and right part of the window can be used (see Fig. 21).

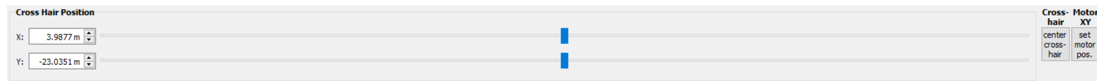


Figure 21: Camera cross-hair position controls.

Using these controls, the crosshair position can be changed by entering the desired coordinates into the leftmost input fields, or by dragging the sliders.

If the `coord_offset_positioner_channels` setting is not used in the configuration of the logic module (see chapter 6.2.7), then the crosshair coordinates are just coordinates within the coordinate system of the camera (as configured in the `ati_andor` section, see chapter 6.2.5). The current position of the pre-positioning device is not considered.

If, however, the `coord_offset_positioner_channels` setting refers to the pre-positioner (here: the stepper-motors), then the current position of the stepper-motors is taken into account.

For example, the coordinate $x = 50\,\mu\text{m}$ may identify an x-position right from the image center in the camera's coordinate system. However, if at the same time a pre-positioning stepper motor has moved the whole stage to $x = 3\,\text{mm}$, then the same x-position in the image will be referenced as $x = 3.05\,\text{mm}$.

The crosshair position control panel also includes the following two buttons:

- The "`center cross hair`" button to move the crosshair to the center position,
- and the "`set motor pos.`" button, which is only available if a `pre_positioner` has been defined in the according `scan_type` section of the configuration file (see chapter 6.2.7). Clicking this button moves the pre-positioning device (here: the stepper-motors) to the current cross hair position.

7.2.8 Taking a Snapshot

To take a snapshot with the current settings, the "Snapshot" button on the button bar must be clicked. During the image acquisition the tab title flashes in red. In case of a long-time exposure, the image acquisition can be canceled by clicking the "Stop Acquisition" button. Figure 22 shows the situation after an image acquisition.

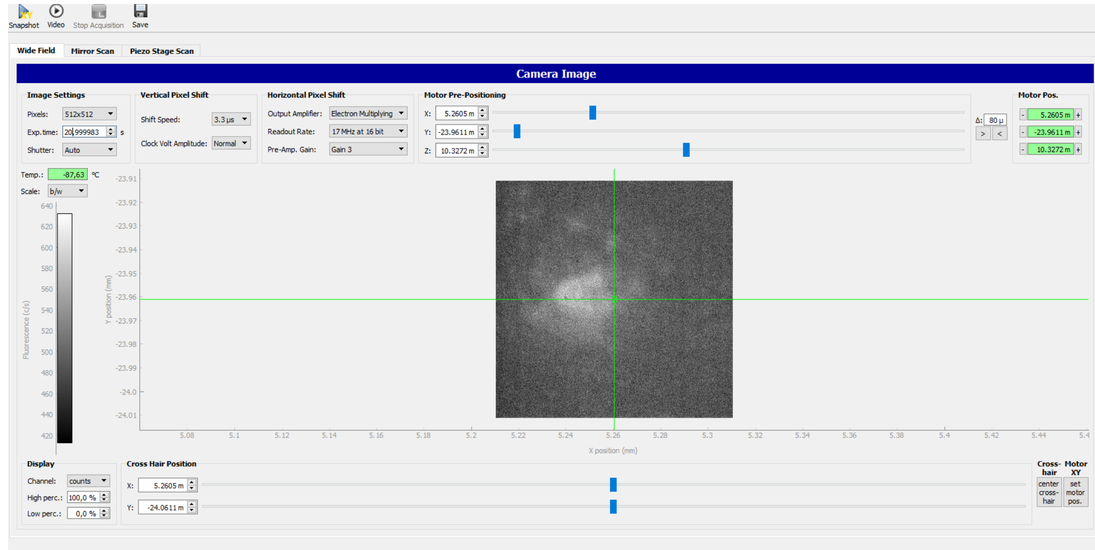


Figure 22: An image acquired with the camera.

7.2.9 Live Video Acquisition

To get a live video stream from the camera, the "Video" button on the button bar must be clicked. The camera settings may be changed during the acquisition to see the immediate effect on the image. The video can be stopped by clicking the "Stop Acquisition" button. The last taken frame of the live video remains visible like a snapshot image.

7.3 Confocal Scan With Two-Axes Positioning-Device (e.g. Mirror Scan)

This chapter describes the user interface for a confocal scan configuration with a positioning device that can be moved along two independent axes. An example for such a device is a stage with two degrees of freedom, or - as in our example - a laser mirror. Figure 23 shows the user interface for this case.

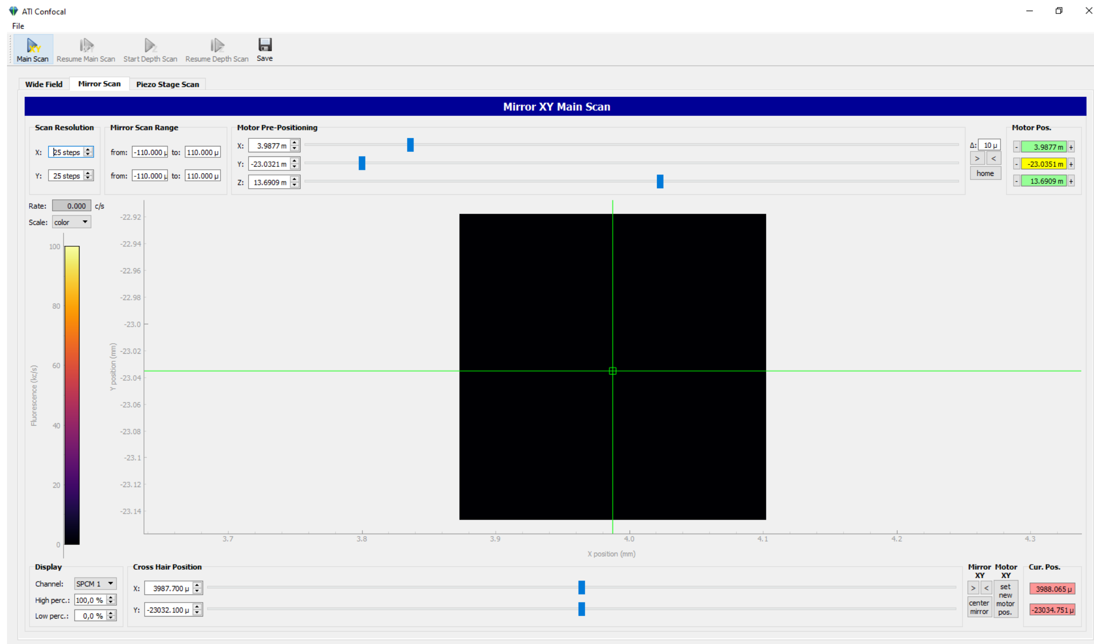


Figure 23: Mirror-scan window.

7.3.1 Scan Resolution Settings

The "Scan Resolution" frame on the top left side of the window hosts the input-fields determining the number of scan steps for either axis. The axis names are taken from the configuration of the corresponding hardware module (here: `ati_NI_card`, see chapter 6.2.3).

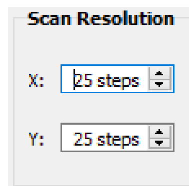


Figure 24: Scan resolution settings.

7.3.2 Scan Range Settings

Right next to the "Scan Resolution" frame there is the "Scan Range" Frame, hosting the input fields allowing to determine the physical scan range for either axis.

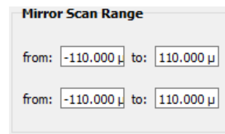


Figure 25: Scan range settings.

This scan range is interpreted as the range around the *current mirror position*, which can be set using the "`>`"-button on the "Cross Hair Position" control (see chapter 7.3.7).

7.3.3 Pre-Positioning

Provided that a `pre_positioner` has been defined in the according `scan_type` section of the configuration file (see chapter 6.2.7), the user interface will show Pre-Positioning controls. They work exactly the same way as already explained in chapter 7.2.3.

7.3.4 Count Rate

On the left side of the window there is a "Rate" label, showing the current count-rate of the selected data channel (see chapter 7.3.6).

7.3.5 Color Scale Setting

Also on the left side, below the count-rate label, there is a drop-down box allowing to choose the color scale to be used for an acquired image. For *scan types* representing a confocal scan, the default setting is "color". Alternatively, a black-and-white scale can be chosen.

7.3.6 Display Settings

- **Channel:** For every *scan-type*, the program logic connects to a specific counter module and a channel of this counter-module (see chapter 6.2.7). This channel then has one or more *output-channels*, representing one ore more data streams. The "Channel" dropdown-box lists all of these *output-channels*. For the Swabian Time Tagger implementation `ati_swabian_tt`, the *output-channels* (and their respective names) are configured in the `output_channel` configuration settings, as described in chapter 6.2.4.

- **High. perc.**, **Low. perc.:** These values can be used to increase the contrast of an acquired image.

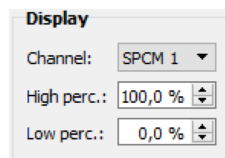


Figure 26: Confocal scan display settings.

7.3.7 Crosshair Position Controls

The crosshair of the main image can be moved directly by dragging it with the mouse. Alternatively, the cross-hair position controls in the lower part of the window can be used (see Fig. 27).

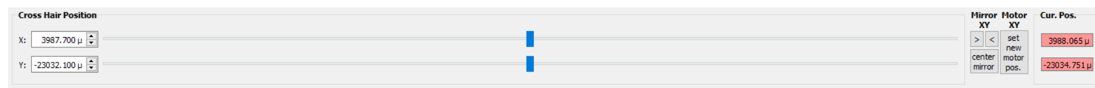


Figure 27: Cross-hair position controls for confocal scan.

Using these controls, the crosshair position can be changed by entering the desired coordinates into the leftmost input fields, or by dragging the sliders.

If the `coord_offset_positioner_channels` setting is not used in the configuration of the logic module (see chapter 6.2.7), then the crosshair coordinates are just coordinates within the coordinate system of the mirror (as, in this example, configured in the `ati_NI_card` section, see chapter 6.2.3). in this case, the current position of the prepositioning device is not taken into account.

If, however, the `coord_offset_positioner_channels` setting refers to positioning devices and pre-positioning-devices (here: the piezo stage and the stepper-motors), then the current position of both the piezo-stage and the stepper-motors is taken into account.

For example, the coordinate $x = 0\,\mu m$ may identify the x-center-position of the mirror in the mirror's coordinate system. However, if at the same time a pre-positioning stepper motor has moved the whole stage to $x = 3\,mm$, then the same x-center-position will be referenced as $x = 3\,mm$. If then, additionally, the piezo-stage is also moved $10\,\mu m$ in x-direction, then the x-center-position of the mirror is referenced as $x = 3.01\,mm$.

The crosshair position control panel also includes the following buttons:

- The "`>`"-button moves the positioning device (here: the mirror) to the cross-hair position. The current position is displayed on the rightmost side in the "Cur- pos." frame. The back-color of the current-position-fields encodes the current status:
  - **green**: This axis of the positioning device (here: the mirror) is steady and currently at the target position.
  - **yellow**: This axis of the positioning device (here: the mirror) is steady, but currently not at the target position.
  - **red**: This axis of the positioning device (here: the mirror) is either currently moving and has not yet reached the target position, or is supposed to already be at the target position, but did not reach the target position within tolerance.
- The "`center mirror`"-button (replace "mirror" with any other device name for different configurations) moves both the on-screen cross-hair, as also the the positioning device (here: the mirror) physically to the center position.
- The "`set new motor pos.`" button is only available if a `pre_positioner` has been defined in the according `scan_type` section of the configuration file (see chapter 6.2.7). Clicking this button moves the pre-positioning device (here: the stepper-motors) to the current cross hair position.

7.3.8 Start a Scan

To start a scan with the current settings, the "Main Scan" button on the button bar must be clicked. During the scan process the tab title flashes in red. After the scan has finished, the positioning device (here: the mirror) is moved to the central position along the scan-axes.

7.3.9 Stop and Resume a Scan

A currently running scan can be canceled by clicking the "Stop Scan" button. When clicked, the software will attempt to finish the current line. However, if the line cannot be finished within 5 seconds, the scan is stopped in any way. After the scan has been stopped, the positioning device (here: the mirror) is moved to the central position along the scan-axes.

A stopped scan can be resumed by clicking the "Resume Main Scan" button, provided the user has not yet changed the settings or moved any positioning-device or pre-positioning device.

7.4 Confocal Scan With Three-Axes Positioning-Device (e.g. Stage Scan)

This chapter describes the user interface for a confocal scan configuration with a positioning device that can be moved along three independent axes. An example for such a device is a piezo stage with three degrees of freedom. Figure 28 shows the user interface for this case.

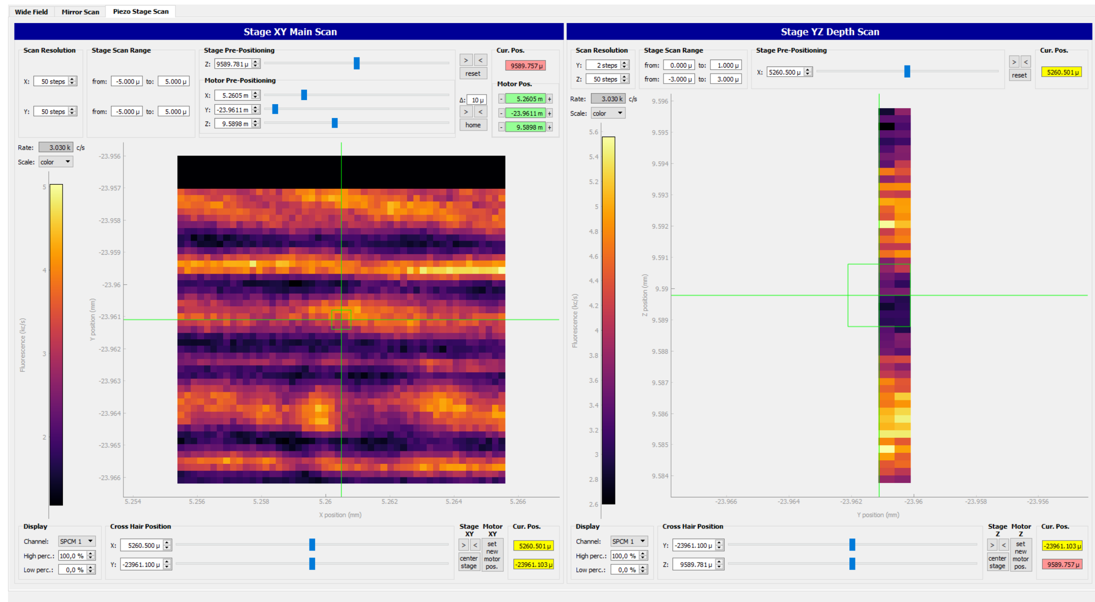


Figure 28: Three-axes stage-scan window.

As can be seen in the screen-shot in Fig. 28, the main difference compared to the two-axes scan is that with the three-axes scan the screen is now split into two areas: The "Main Scan" area on the left side, and the "Depth-Scan" area on the right side.

Which axes define the main-scan axes, and which axes the depth-scan axes is determined by the `main_scan_axes`-setting and the `depth_scan_axes`-setting in the corresponding

`scan_type`-section of the program logic (see chapter 6.2.7).

Both the main-scan side and the depth-scan side have the same controls as already explained in the previous chapter. However, there are some minor abbreviations which are explained below.

7.4.1 Main Scan Pre-Positioning

In the given sample configuration, the main-scan is performed along the xy-axes, and the depth-scan along the yz-axes. This means that the z-axis can be seen as an pre-postioning-axis for the main-scan, and hence it shows up in the pre-positioning area of the main scan:

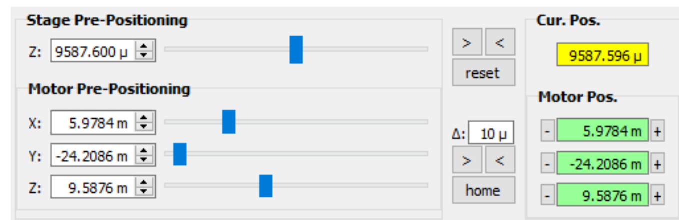


Figure 29: Main-scan pre-positioning controls for a three-axes piezo stage scan.

The "Motor Pre-Positioning" controls are also displayed as usual if the stepper-motors have been defined as `pre_positioner` in the according `scan_type` section of the configuration file (see chapter 6.2.7). They work exactly the same way as explained in the previous chapters.

The "reset" button next to the "Stage Pre-Positioning" frame simply centers the stage's z-axis.

7.4.2 Depth Scan Pre-Positioning

In the given sample configuration the main-scan is performed along the xy-axes, and the depth-scan along the yz-axes. Therefore, the x-axis can be seen as an pre-postioning-axis for the depth-scan, and hence it shows up in the pre-positioning area of the depth scan:

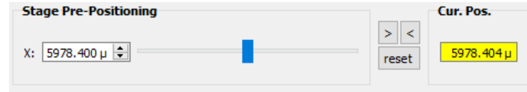


Figure 30: Depth-scan pre-positioning controls for a three-axes piezo stage scan.

7.4.3 Cross Hair Position Controls

The cross hair position controls in the main-scan area work exactly as described in the previous chapters with regards to the two main-scan axes (here: x and y).

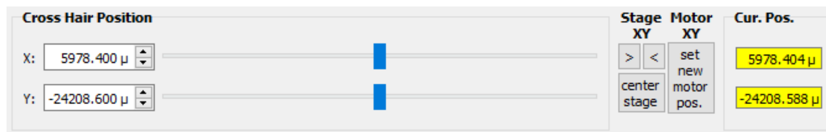


Figure 31: Cross-hair position controls for the main-scan area.

The cross hair position controls in the depth-scan area differ insofar as the buttons ">", "<", and "set new motor pos." do *not* affect both axes, but instead only affect the vertical depth-scan axis (here: the z-axis).

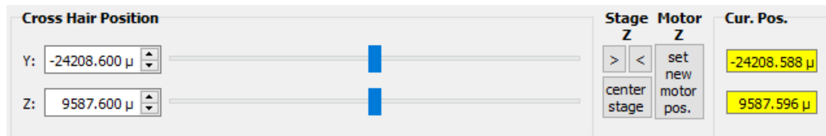


Figure 32: Cross-hair position controls for the depth-scan area.

7.4.4 Starting and Stopping Main-Scan and Depth-Scan

A main-scan can be started by clicking the "Main Scan"-button in the button bar, and a depth-scan can be started by clicking the "Start Depth Scan"-button in the button bar.

When a scan is completed (or has been prematurely stopped), the positioning device (here: the stage) is centered along the respective axes. This means that, in our sample configuration, when a main-scan is finished or stopped, the x- and y-axis is centered, and when a depth-scan is finished or stopped, the y- and z-axis is centered.

Both main-scan and depth-scan can be resumed when they have been prematurely stopped, as long as the user has not modified the scan parameters or moved any positioning devices or pre-positioning devices.

7.5 Saving and Loading Images

7.6 Saving Images

In all of the scan-types (camera, two-axes scan and three-axes scan) the resulting image-data can be saved as files in csv-format. This can be done by clicking the "Save"-button in the button bar (or, the "Save Depth"-button for saving depth-scan data). The file format is documented in chapter 5.2.29.

7.7 Loading Images

By clicking the "Load"-button in the button bar, scan-images or camera-images saved previously into a csv-file can be loaded. The load-function determines whether or not the file to be loaded matches the currently active *scan-type*. In case the referenced *scan type* provides both main-scan and depth-scan, the method additionally determines the file type with respect to this automatically.

List of Figures

1 Stepper motors, piezo stage and sample holder 10
2 K-Cube Stepper Motor Controllers 11
3 Beam paths in the confocal microscope. The confocal beam path is displayed in green, the wide-field beam-path in red. The two movable lenses and the movable mirror displayed in red color are depicted in the wide-field imaging position, and must be flipped out of the beam-path for confocal imaging. 12
4 orientation of axes according to current configuration (Jan. 2021) 13
5 Qudi functional and structural design. From: [Binder et al, 2017, p. 87] . 14
6 COMICS software architecture 15
7 Check both check-boxes at this step. 106
8 At this window, click on "Open". 107

9 Select "Project: COMICS" and then click on "Python interpreter". 108
10 Select "Select "Conda Environment", and Python Version 1.6 108
11 Click "OK" once more 109
12 Choose the 64-bit Andor driver 113
13 Select your Andor camera model 114
14 Click "Install" 114
15 Qudi program manager 138
16 The COMICS start screen. 139
17 Camera image settings. 140
18 Extended settings for the Andor camera. 140
19 Pre-positioning controls for the camera. 141
20 Camera image display settings. 142
21 Camera cross-hair position controls. 143
22 An image acquired with the camera. 144
23 Mirror-scan window. 145
24 Scan resolution settings. 145
25 Scan range settings. 146
26 Confocal scan display settings. 147
27 Cross-hair position controls for confocal scan. 147
28 Three-axes stage-scan window. 149
29 Main-scan pre-positioning controls for a three-axes piezo stage scan. . . . 150
30 Depth-scan pre-positioning controls for a three-axes piezo stage scan. . . . 151
31 Cross-hair position controls for the main-scan area. 151
32 Cross-hair position controls for the depth-scan area. 151

References

[Binder et al, 2017] J. M. Binder, A. Stark, N. Tomek, J. Scheuer, F. Frank, K. D. Jahnke, C. Müller, S. Schmitt, M. H. Metsch, T. Unden, T. Gehring, A. Huck, U. L. Andersen, L. J. Rogers, and F. Jelezko: "Qudi: A modular python suite for experiment control and data processing", *Software X* **Volume 6**, (2017), pages 85-90. https://doi.org/10.1016/j.softx.2017.02.001