CMI SPM controller interface

Note: this document describes the client-server connection to the hwserver, which is a key part of the DSP and all what needs to be known to operate it. For a more detailed description of the DSP and its internal API see the document “CMI Digital Signal Processor”.

Changes from v1 to v2.0

- separated instrument specific settings to an ini file
- added option for splitting the 20-bit dac and for running the z DAC via FPGA
- added possibility of selection of 20-bit DAC ranges
- added possibility of limiting the lock-in excitation data to positive voltages only
- speeded up the internal functions when doing a voltage scan
- changed the PID loop to use more standard algorithm
- all AD channels can now be requested
- added option for swapping the inputs direction for individual modes
- added speed set function from Lua interface

Changes from v2.0 to v2.1

- amplitude stabilisation added for Akiyama probe
- limits and phase filter added for Akiyama probe
- bug leading to piezo jumps while setting feedback on/off fixed
- workaround to prevent jumps when changing PID parameters implemented
- Akiyama probe operation tested

Changes from v2.1 to v3

- bug fixed in timing of SPI signals
- added and tested basic KPFM support
- cleaned API and PS-PL interconnections
- speeded Lua moveto command
- enabled output of internal signals via slow DAC
- slightly refined feedback loop PIDs
- added independent setup of high-resolution lock-in regimes
- added generator offset option to both channels, removed unused phase

Changes from v3 to v4

- changed API by renaming some of the communication commands
- modes in hwserver.ini are now defined independently
- added freq3 generator
- added support for DART

General introduction

The interface is realised via sockets passed over Ethernet. Server, in the following text called **gwyhwserver,** is running on some port of the RedPitaya device (or any other hardware controller, like a personal computer with data acquisition cards in case of some custom built systems at CMI).

Communication is done on a client-server basis, where the client (SPM control programme operated by user) sends some message to the server and gets some response.

All the messages are based on a simple protocol described below. It includes messages for setting different parameters, getting actual readouts, letting the system get into feedback, move on some profile, etc. Messages can be simple, e.g. to set the system into the feedback, or complex, e. g. to set all the feedback parameters. Every message is responded by the server, even if there is no request for passing any data back (at least “todo” is passed back). The response can be immediate (non-blocking), but also can be delayed when the requested work is done (blocking). No next message is treated unless the previous was responded.

Messages are passed as GwyFileObject structures serialised by Gwyfile library, with a variable number of items inside, depending on the message type and user’s wishes on the number of parameters that should be passed at the moment. For deserialization, the Gwyfile library can be used, or custom implementation based on the Gwyddion file format description can be done.

Hwserver parameters are in a separate file, either the hwserver.ini file read from default location (working directory), or file provided as a parameter of the hwserver. Ini file includes all the information that is hardware dependent and should be set while physically setting up the system.

A sample hwserver.ini file is also listed here:

```
; hwserver config file. Don't change this unless you know what you're are doing and match the settings to the hardware
[general]
debug = 0               ; report some extra information useful for debugging

[hardware]              ; System parameters
scan_mode = 1           ; 0: nothing, 1: voltage scan, 2: PI RS232 scan, 3: RP two axis controller scan
xrange = 100e-6         ; x scanner physical range in m
yrange = 100e-6         ; y scanner physical range in m
zrange = 10e-6          ; z scanner physical range in m
hrdac_regime = 0        ; 0: 20bit DACs controlled by FPGA, 1: 20bit DACs controlled by CPU, 2: mixed, 2xCPU + 1xFPGA,
hrdac1_range = 1        ; 20bit xy piezo range: 0: +-10 V, 1: 0-10 V, check jumpers
hrdac2_range = 1        ; 20bit xy piezo range: 0: +-10 V, 1: 0-10 V, check jumpers
hrdac3_range = 1        ; 20bit xy piezo range: 0: +-10 V, 1: 0-10 V, check jumpers
rp1_input_hv = 0        ; use this if RP fast ADC1 jumper is on HV range, which normally should not
rp2_input_hv = 0        ; use this if RP fast ADC1 jumper is on HV range, which normally should not
rp_bare_input = 0       ; RP fast inputs are used directly, without the divider board
rp_bare_output = 0      ; RP fast outputs are used directly without scaling to +-10
dds1_range = 0          ; dds1 output amplitude range: 0: +-1 V, 1: 0-1 V (scaled and shifted)
dds2_range = 0          ; dds2 output amplitude range: 0: +-1 V, 1: 0-1 V (scaled and shifted)
timestep = 500000       ; time step for most of the operations, in nanoseconds. was 3000000 for PI table, 1000000 for voltage scan, interrupted
fine up to 200000
oversampling = 6        ; slow ADC oversampling factor, 0 (none oversampling) to 6 (highest oversampling).

rpadc1_bare_lv_offset = 0.0       ; bare RP1 input, LV jumper, zero offset
rpadc1_bare_lv_slope = 8191       ; bare RP1 input, LV jumper, slope (integer corresponding to 1V)
rpadc1_bare_hv_offset = 0.0       ; bare RP1 input, HV jumper, zero offset
rpadc1_bare_hv_slope = 410        ; bare RP1 input, HV jumper, slope (integer corresponding to 1V)
rpadc2_bare_lv_offset = 0.0       ; bare RP2 input, LV jumper, zero offset
rpadc2_bare_lv_slope = 8191       ; bare RP2 input, LV jumper, slope (integer corresponding to 1V)
```

```
rpadc2_bare_hv_offset = 0.0       ; bare RP2 input, HV jumper, zero offset
rpadc2_bare_hv_slope = 410        ; bare RP2 input, HV jumper, slope (integer corresponding to 1V)

rpadc1_divhigh_lv_offset = 78     ; RP1 with input1_range=1 and LV jumper (silly choice), zero offset
rpadc1_divhigh_lv_slope = 720     ; RP1 with input1_range=1 and LV jumper (silly choice), slope (integer corresponding to 1V)
rpadc1_divhigh_hv_offset = 42     ; RP1 with input1_range=1 and HV jumper, zero offset
rpadc1_divhigh_hv_slope = 29      ; RP1 with input1_range=1 and HV jumper, slope (integer corresponding to 1V)
rpadc1_divlow_lv_offset = 120     ; RP1 with input1_range=0 and LV jumper, zero offset
rpadc1_divlow_lv_slope = 7200     ; RP1 with input1_range=0 and LV jumper, slope (integer corresponding to 1V)
rpadc1_divlow_hv_offset = 48      ; RP1 with input1_range=0 and HV jumper (silly choice), zero offset
rpadc1_divlow_hv_slope = 290      ; RP1 with input1_range=0 and HV jumper (silly choice), slope (integer corresponding to 1V)

rpadc2_divhigh_lv_offset = 78     ; RP2 with input1_range=1 and LV jumper (silly choice), zero offset
rpadc2_divhigh_lv_slope = 720     ; RP2 with input1_range=1 and LV jumper (silly choice), slope (integer corresponding to 1V)
rpadc2_divhigh_hv_offset = 48     ; RP2 with input1_range=1 and HV jumper, zero offset
rpadc2_divhigh_hv_slope = 29      ; RP2 with input1_range=1 and HV jumper, slope (integer corresponding to 1V)
rpadc2_divlow_lv_offset = 120     ; RP2 with input1_range=0 and LV jumper, zero offset
rpadc2_divlow_lv_slope = 7200     ; RP2 with input1_range=0 and LV jumper, slope (integer corresponding to 1V)
rpadc2_divlow_hv_offset = 48      ; RP2 with input1_range=0 and HV jumper (silly choice), zero offset
rpadc2_divlow_hv_slope = 290      ; RP2 with input1_range=0 and HV jumper (silly choice), slope (integer corresponding to 1V)

rpdac1_bare_offset = 0            ; bare RP1 output zero offset
rpdac1_bare_slope = 8191          ; bare RP1 output zero slope
rpdac2_bare_offset = 0            ; bare RP2 output zero offset
rpdac2_bare_slope = 8191          ; bare RP2 output zero slope
rpdac1_offset = 0                 ; RP1 output zero offset
rpdac1_slope = 819.1              ; RP1 output zero slope
rpdac2_offset = 0                 ; RP2 output zero offset
rpdac2_slope = 819.1              ; RP2 output zero slope

[mode0]
name = off                        ; name of the measurement mode
mux1 = 4                          ; ADC channel routed to RP1
mux2 = 5                          ; ADC channel routed to RP2
error_source = 2                  ; error source signal (0: off, 1: RP1, 2: RP2, 3: A1, 4: P1, 5: A2, 6: P2, 7: freq)
swap_in = 0                       ; swap error signal direction (0: lower signal retracts piezo, 1: higher signal retract piezo)
swap_out = 0                      ; swap zpiezo signal direction
error_bit_shift = 0               ; error signal bit shift (0-31)
pll = 0                           ; pll on/off, this also generates freq signal for feedback
pll_input = 0                     ; 0: phase1, 1: phase2
input1_range = 0                  ; RP1 input range (0: +-1 V 1: +-10 V)
input2_range = 0                  ; RP2 input range (0: +-1 V 1: +-10 V)
lockin1_hr = 1                    ; high resolution low signal option for lockin 1
lockin2_hr = 1                    ; high resolution low signal option for lockin 2
pidskip  = 1                      ; PID speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
pllskip = 0                       ; PLL speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
lockin1_filter_amplitude = 0 ; filter amplitude in lockin1 output
lockin1_filter_phase = 1          ; filter phase in lockin1 output
lockin2_filter_amplitude = 0 ; filter amplitude in lockin2 output
lockin2_filter_phase = 0          ; filter phase in lockin2 output
out1 = 0                          ; RP output routing, default 0: gen1 excitation (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6:
excs)
out2 = 4                          ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
outhr = 4                         ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: hrdac, 4: pid, 5: off, 6: excs)

[mode1]
name = proportional               ; name of the measurement mode
mux1 = 4                          ; ADC channel routed to RP1
mux2 = 5                          ; ADC channel routed to RP2
error_source = 1                  ; error source signal (0: off, 1: RP1, 2: RP2, 3: A1, 4: P1, 5: A2, 6: P2, 7: freq)
swap_in = 1                       ; swap error signal direction (0: lower signal retracts piezo, 1: higher signal retract piezo)
swap_out = 0                      ; swap zpiezo signal direction
error_bit_shift = 3               ; error signal bit shift (0-31), was 3
pll = 0                           ; pll on/off, this also generates freq signal for feedback
pll_input = 0                     ; 0: phase1, 1: phase2
input1_range = 1                  ; RP1 input range (0: +-1 V 1: +-10 V)
input2_range = 1                  ; RP2 input range (0: +-1 V 1: +-10 V)
lockin1_hr = 0                    ; high resolution low signal option for lockin 1
lockin2_hr = 0                    ; high resolution low signal option for lockin 2
pidskip  = 1                      ; PID speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
pllskip = 0                       ; PLL speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
lockin1_filter_amplitude = 0 ; filter amplitude in lockin1 output
lockin1_filter_phase = 1          ; filter phase in lockin1 output
lockin2_filter_amplitude = 0 ; filter amplitude in lockin2 output
lockin2_filter_phase = 0          ; filter phase in lockin2 output
out1 = 0                          ; RP output routing, default 0: gen1 excitation (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6:
excs)
out2 = 4                          ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
outhr = 4                         ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: hrdac, 4: pid, 5: off, 6: excs)

[mode2]
name = ncamplitude                ; name of the measurement mode
mux1 = 4                          ; ADC channel routed to RP1
mux2 = 5                          ; ADC channel routed to RP2
error_source = 3                  ; error source signal (0: off, 1: RP1, 2: RP2, 3: A1, 4: P1, 5: A2, 6: P2, 7: freq)
swap_in = 0                       ; swap error signal direction (0: lower signal retracts piezo, 1: higher signal retract piezo)
swap_out = 0                      ; swap zpiezo signal direction
error_bit_shift = 3               ; error signal bit shift (0-31)
pll = 0                           ; pll on/off, this also generates freq signal for feedback
pll_input = 0                     ; 0: phase1, 1: phase2
input1_range = 0                  ; RP1 input range (0: +-1 V 1: +-10 V)
input2_range = 0                  ; RP2 input range (0: +-1 V 1: +-10 V)
lockin1_hr = 0                    ; high resolution low signal option for lockin 1
lockin2_hr = 0                    ; high resolution low signal option for lockin 2
pidskip  = 2                      ; PID speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
pllskip = 0                       ; PLL speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
lockin1_filter_amplitude = 0 ; filter amplitude in lockin1 output
lockin1_filter_phase = 1          ; filter phase in lockin1 output
lockin2_filter_amplitude = 0 ; filter amplitude in lockin2 output
lockin2_filter_phase = 0          ; filter phase in lockin2 output
out1 = 0                          ; RP output routing, default 0: gen1 excitation (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6:
excs)
out2 = 4                          ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
```

```
outhr = 4                          ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: hrdac, 4: pid, 5: off, 6: excs)

[mode3]
name = ncphase                     ; name of the measurement mode
mux1 = 4                           ; ADC channel routed to RP1
mux2 = 4                           ; ADC channel routed to RP2
error_source = 4                   ; error source signal (0: off, 1: RP1, 2: RP2, 3: A1, 4: P1, 5: A2, 6: P2, 7: freq)
swap_in = 0                        ; swap error signal direction (0: lower signal retracts piezo, 1: higher signal retract piezo)
swap_out = 0                       ; swap zpiezo signal direction
error_bit_shift = 3                ; error signal bit shift (0-31)
pll = 0                            ; pll on/off, this also generates freq signal for feedback
pll_input = 0                      ; 0: phase1, 1: phase2
input1_range = 0                   ; RP1 input range (0: +-1 V 1: +-10 V)
input2_range = 0                   ; RP2 input range (0: +-1 V 1: +-10 V)
lockin1_hr = 0                     ; high resolution low signal option for lockin 1
lockin2_hr = 0                     ; high resolution low signal option for lockin 2
pidskip  = 2                       ; PID speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
pllskip = 0                        ; PLL speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
lockin1_filter_amplitude = 0 ; filter amplitude in lockin1 output
lockin1_filter_phase = 1           ; filter phase in lockin1 output
lockin2_filter_amplitude = 0 ; filter amplitude in lockin2 output
lockin2_filter_phase = 0           ; filter phase in lockin2 output
out1 = 0                           ; RP output routing, default 0: gen1 excitation (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6:
excs)
out2 = 4                           ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
outhr = 4                          ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: hrdac, 4: pid, 5: off, 6: excs)

[mode4]
name = akiyama                     ; name of the measurement mode
mux1 = 10                          ; ADC channel routed to RP1
mux2 = 11                          ; ADC channel routed to RP2
error_source = 7                   ; error source signal (0: off, 1: RP1, 2: RP2, 3: A1, 4: P1, 5: A2, 6: P2, 7: freq)
swap_in = 1                        ; swap error signal direction (0: lower signal retracts piezo, 1: higher signal retract piezo)
swap_out = 0                       ; swap zpiezo signal direction
error_bit_shift = 9                ; error signal bit shift (0-31)
pll = 1                            ; pll on/off, this also generates freq signal for feedback
pll_input = 0                      ; 0: phase1, 1: phase2
input1_range = 1                   ; RP1 input range (0: +-1 V 1: +-10 V)
input2_range = 1                   ; RP2 input range (0: +-1 V 1: +-10 V)
lockin1_hr = 0                     ; high resolution low signal option for lockin 1
lockin2_hr = 0                     ; high resolution low signal option for lockin 2
pidskip  = 0                       ; PID speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
pllskip = 0                        ; PLL speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
lockin1_filter_amplitude = 0 ; filter amplitude in lockin1 output
lockin1_filter_phase = 1           ; filter phase in lockin1 output
lockin2_filter_amplitude = 0 ; filter amplitude in lockin2 output
lockin2_filter_phase = 0           ; filter phase in lockin2 output
out1 = 0                           ; RP output routing, default 0: gen1 excitation (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6:
excs)
out2 = 4                           ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
outhr = 4                          ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: hrdac, 4: pid, 5: off, 6: excs)

[mode5]
name = nenoprobe                   ; name of the measurement mode
mux1 = 13                          ; ADC channel routed to RP1
mux2 = 13                          ; ADC channel routed to RP2
error_source = 3                   ; error source signal (0: off, 1: RP1, 2: RP2, 3: A1, 4: P1, 5: A2, 6: P2, 7: freq)
swap_in = 0                        ; swap error signal direction (0: lower signal retracts piezo, 1: higher signal retract piezo)
swap_out = 0                       ; swap zpiezo signal direction
error_bit_shift = 7                ; error signal bit shift (0-31)
pll = 0                            ; pll on/off, this also generates freq signal for feedback
pll_input = 1                      ; 0: phase1, 1: phase2
input1_range = 1                   ; RP1 input range (0: +-1 V 1: +-10 V)
input2_range = 1                   ; RP2 input range (0: +-1 V 1: +-10 V)
lockin1_hr = 0                     ; high resolution low signal option for lockin 1
lockin2_hr = 0                     ; high resolution low signal option for lockin 2
pidskip  = 2                       ; PID speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
pllskip = 0                        ; PLL speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
lockin1_filter_amplitude = 0 ; filter amplitude in lockin1 output
lockin1_filter_phase = 0           ; filter phase in lockin1 output
lockin2_filter_amplitude = 0 ; filter amplitude in lockin2 output
lockin2_filter_phase = 0           ; filter phase in lockin2 output
out1 = 0                           ; RP output routing, default 0: gen1 excitation (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6:
excs)
out2 = 4                           ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
outhr = 4                          ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: hrdac, 4: pid, 5: off, 6: excs)

[mode6]
name = stm                         ; name of the measurement mode
mux1 = 12                          ; ADC channel routed to RP1
mux2 = 12                          ; ADC channel routed to RP2
error_source = 1                   ; error source signal (0: off, 1: RP1, 2: RP2, 3: A1, 4: P1, 5: A2, 6: P2, 7: freq)
swap_in = 0                        ; swap error signal direction (0: lower signal retracts piezo, 1: higher signal retract piezo)
swap_out = 0                       ; swap zpiezo signal direction
error_bit_shift = 3                ; error signal bit shift (0-31)
pll = 0                            ; pll on/off, this also generates freq signal for feedback
pll_input = 0                      ; 0: phase1, 1: phase2
input1_range = 0                   ; RP1 input range (0: +-1 V 1: +-10 V)
input2_range = 0                   ; RP2 input range (0: +-1 V 1: +-10 V)
lockin1_hr = 0                     ; high resolution low signal option for lockin 1
lockin2_hr = 0                     ; high resolution low signal option for lockin 2
pidskip  = 2                       ; PID speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
pllskip = 0                        ; PLL speed factor. 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
lockin1_filter_amplitude = 0 ; filter amplitude in lockin1 output
lockin1_filter_phase = 0           ; filter phase in lockin1 output
lockin2_filter_amplitude = 0 ; filter amplitude in lockin2 output
lockin2_filter_phase = 0           ; filter phase in lockin2 output
out1 = 2                           ; RP output routing, default 0: gen1 excitation (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6:
excs)
out2 = 4                           ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
outhr = 4                          ; RP output routing, default 4: pid result (0: gen1, 1: gen2, 2: hrdac, 4: pid, 5: off, 6: excs)
```

API, hwserver version v4

Note: we do our best not to change API, however in the initial versions it can still happen.

Every GwyFileObject passed to gwyhwserver has to have a string item “todo”, which defines the type of request. GwyFileObjects without the “todo” item are ignored. Correct GwyFileObjects coming from the client will be called “message” in the following text and when these are returned back to the client they will be called “response”. The response always includes the “todo” string, which is used to acknowledge the reception of the message (and most often is accompanied by other data, depending on the request).

The messages below are organised by their function and listed by the “todo” strings. Apart from this parameter, they can have a variable number of other parameters. These are not obligatory and parameters that are not passed are simply not changed.

A. Messages to get and set the status or parameters of the microscope

state: sets and gets the general behaviour of the microscope, namely aspects that affect how the feedback loop will work in general. The mode parameter should match some of the modes defined in hwserver.ini file located on the server, which might be e.g. “proportional” for a general feedback loop operation proportional to the input, “ncamplitude”, which might be tapping mode based on amplitude feedback, etc. The real number of modes depends on the hardware and their names are provided by this command. Switching the mode to off has nothing to do with switching the feedback on and off (see the *set_feedback* message). The main goal of this command is to connect all the wires inside the digital feedback loop.

The various mode parameters can also be read. What will be returned will correspond to the actually selected mode parameters. Many of these parameters can’t be yet set as most of them depend on wiring in the electronics and values in hwserver.ini file should match them.

The following parameters can be get/set:
mode (string): use one from the list provided by this command.
mux1 (int): multiplexer settings for RP fast ADC 1
mux2 (int): multiplexer settings for RP fast ADC 2
bitshift (int): error signal bit shift inside the FPGA
input1_range (int): 0/1 for small/full RP1 ADC range
input2_range (int): 0/1 for small/full RP2 ADC range
lockin1_hr (int): use high resolution small signal option for lock-in1
lockin2_hr (int): use high resolution small signal option for lock-in2
lockin1_filter_amplitude (int): low pass filter amplitude result from lock-in1
lockin1_filter_phase (int): low pass filter phase result from lock-in1
lockin2_filter_amplitude (int): low pass filter amplitude result from lock-in2

lockin2_filter_phase (int): low pass filter phase result from lock-in2
lockin1_nwaves (int): number of waves to be evaluated by the lock-in, 0-6 means this sequence: 1 wave, 2 waves, 4, 8, 32, 128 and 512 waves
lockin2_nwaves (int): number of waves to be evaluated by the lock-in, 0-6 means this sequence: 1 wave, 2 waves, 4, 8, 32, 128 and 512 waves
pidskip (int): feedback loop speed: 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz
pllskip (int): feedback loop speed: 0: 125 MHz, 1: 1 MHz, 2: 120 kHz, 3: 15 kHz

On top of it, the following general hardware related parameters can be obtained, but not set (they should be set in hwserver configuration file, not by user, and should match together):

error_source (int): error source: (0: RP1, 1: RP2, 2: A1, 3: P1, 4: A2, 5: P2, 6: PLL freq)
swap_in (boolean): swap the error signal direction in the feedback loop
swap_out (boolean): swap the piezo direction
pll (boolean): use of PLL
pll_input (int): input for PLL: 1: lock-in1 phase, 2: lock-in2 phase
out1 (int): routing for RP fast DAC 1 (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
out2 (int): routing for RP fast DAC 2 (0: gen1, 1: gen2, 2: dac1, 3: dac2, 4: pid, 5: off, 6: excs)
outhr (int); routing for FPGA connected HR DAC
x_cal_factor (double): conversion factor of x position to voltage
y_cal_factor (double): conversion factor of y position to voltage
z_cal_factor (double): conversion factor of z position to voltage
x_shift_factor (double): conversion factor of x position to voltage
y_shift_factor (double): conversion factor of y position to voltage
z_shift_factor (double): conversion factor of z position to voltage
x_range (double): maximum scan range in the x direction in metres
y_range (double): maximum scan range in the y direction in metres
z_range (double): maximum range in the z direction in metres
dds1_range (int): full or half range of DDS generator 1
dds2_range (int): full or half range of DDS generator 2
rp1_input_hv (int): 1 if the jumper on RP board input 1 is on HV (only special cases)
rp2_input_hv (int): 1 if the jumper on RP board input 1 is on HV (only special cases)
hrdac_range (int): xy 20bit DAC range option: 0: +-10V, 1: 0-10 V
zpiezo_hrdac (boolean): use external 20bit DAC for z piezo motion
zpiezo_hrdac_range: z 20bit DAC range option: 0: +-10V, 1: 0-10 V
scan_mode (int): scan regime 0: nothing, 1: voltage scan, 2: PI RS232 scan, 3: RP 2D scan

Finally, the command sends a list of available modes as mode1 (string), mode2 (string), etc.

In the response, all the settings will be sent back. Sending a request with nothing to set is a way to get all the settings without altering anything. Depending on the hardware the lateral position can be controlled via voltage or via digital communication. When voltage position

control should be used, the voltage is output from 20 bit DAC as out = length*cal_factor + shift_factor, where length is in metres.

---

set: sets selected feedback parameters, like the PID loop settings, setpoint and hardware time that is used to report how long things take. This also includes an optional second feedback loop that is needed in some scanning regimes (e.g. KPFM) or second frequency generator and lock-in.

The following parameters can be set:
hwtime (double): hardware time, to be used to restart or alter the time elapsed by the server
pid_p (double): primary feedback loop P parameter
pid_i (double): primary feedback loop I parameter
pid_d (double): primary feedback loop D parameter
pid_setpoint (double): primary feedback loop setpoint, in basic units of particular quantity
freq1_f (double): primary frequency generator frequency, in Hz
freq1_a (double): primary frequency generator amplitude, in Volts, range 10 V
freq1_o (double): primary frequency generator offset, in Volts, range +-10 V
pid_pll_p (double): PLL feedback loop P parameter
pid_pll_i (double): PLL feedback loop I parameter
pid_pll_d (double): PLL feedback loop D parameter, used also for FM
pid_pll_setpoint (double): FM AFM PLL setpoint, in radians
pid_amplitude_p (double): FM mode amplitude feedback loop P parameter
pid_amplitude_i (double): FM mode amplitude feedback loop I parameter
pid_amplitude_d (double): FM mode amplitude feedback loop D parameter
pid_kpfm_p (double): KPFM feedback loop P parameter
pid_kpfm_i (double): KPFM feedback loop I parameter
pid_kpfm_d (double): KPFM feedback loop D parameter
pid_dart_p (double): DART feedback loop P parameter
pid_dart_i (double): DART feedback loop I parameter
pid_dart_d (double): DART feedback loop D parameter
freq2_f (double): secondary frequency generator frequency, in Hz
freq2_a (double): secondary frequency generator amplitude, in Volts, range 10 V
freq2_o (double): secondary frequency offset, in Volts, range +-10 V
freq3_f (double): auxiliary internal generator frequency, to be passed to lock-in output instead of , in Hz
filter1 (integer): adc filter 1, range 0-12, larger value means higher cut-on, 0 means off
filter2 (integer): adc filter 2, range 0-12, larger value means higher cut-on, 0 means off
pll_phase_limit_factor (integer): range is 0-7, about 1/32 of range to full range
pll_frequency_limit_factor (integer): range is 0-7, meaning approx 4 to 480 Hz
oversampling (integer): range is 0-6, meaning no averaging to 64 samples averaging
kpfm_mode: set the KPFM mode: 0: none, 1: am manual, 2: fm manual, 3: am, 4: fm
kpfm_feedback_source: lockin-data for feedback: 0: signal*sin, 1: signal*cos
kpfm_feedback_direction: 0: up, 1: down
kpfm_no_autoset: 0: start/stop kpfm automatically when doing profile scan, 1: don't do this
phaseshift1: phase shift of evaluated signal in lock-in, 0-3, meaning 0, 90, 180 and 270 deg
phaseshift2: phase shift of evaluated signal in lock-in, 0-3, meaning 0, 90, 180 and 270 deg

dart_mode: 0:off, 1:manual (no feedback), 2: on
dart_frequency: central excitation frequency, in Hz
dart_amplitude: excitation amplitude, in Volts
dart_freqspan: frequency to be subtracted and added from central frequency, in Hz

In the response, the altered settings are sent back.

---

get: gets only selected parameters, not setting anything (if there are values passed in the message, they are ignored). The same parameters as in *set* command can be returned. There are also some additional values that cannot be set but can be obtained:

version (string): hwserver version
moving (boolean): scanner is still moving
scanning_adaptive (boolean): adaptive scan in progress
scanning_line (boolean): line scan in progress
scanning_script (boolean): script scan in progress
ramp_running (boolean): ramp run in progress

---

reset_spi: resets the internal controller communication, to be used after cycling the power on external boards connected by SPI interface. Function has no parameters.

B. Messages to read live data

set_standby_storage: set which additional channels should be acquired internally in the free running regime. By default, all the available channels would be read, which is slow and mostly unnecessary. This function limits the storage to only what is needed and in this way speeds up everything. The function clears the so far measured data and previously set storage parameters. Some of the channels can be duplicated, depending on the feedback regime - e.g. the error signal will be equal to amplitude in the amplitude based tapping mode. The XYZ positions, timestamp, error signal and raw adc data are acquired and sent always.
The following parameters can be set to be acquired:
a1 (boolean): on/off, amplitude of primary lock-in signal
p1 (boolean): on/off, phase of primary lock-in signal
a2 (boolean): on/off, amplitude of secondary lock-in signal
p2 (boolean): on/off, phase of secondary lock-in signal
in1 (boolean): on/off, auxiliary ADC analog input 1
.
.
.
in16 (boolean): on/off, auxiliary ADC analog input 16
fmdrive (boolean): drive amplitude of the FM signal
kpfm (boolean): KPFM result (voltage)
l1x (boolean): primary lock-in x component
l1y (boolean): primary lock-in y component
l2x (boolean): secondary lock-in x component

l2y (boolean): secondary lock-in y component
dart (boolean): DART result (frequency)
In the response the function sends a full set of parameters.

---

read: get all the synchronously acquired data (default parameters plus all channels set by set_standby_storage commands).
Function has no parameters.

In the response, it sends a single set of all the available data (parameter names are the channel names). The values are doubles. All values are in basic SI units and are coded by following strings: x, y, z, e (error signal), adc1, adc2 a1, p1, a2, p2, ch1...ch16, ts (timestamp), fmd (fmdrive), kpfm. Values adc1 and adc2 are the raw data from FPGA high speed input.

C. Messages to set digital and analog outputs manually

Present version of electronics has three types of analog outputs. High speed (RF) Red Pitaya outputs are supposed to be externally connected to dither piezo and to excitation of the Kelvin probe signal. Slow 20-bit DAC outputs are supposed to be connected to scanners in all three axes, to control them, use the *move_to* command. Finally, there are 16 general purpose DACs that can be fed by anything, based on the user's wishes, so only these outputs are configurable using the *set_out* and *route_out* commands. The electronics also has four digital outputs that can be set by the *set_out* command.
Note that the electronics also has some inputs, even if these are not discussed in this section. There are 16 general purpose simultaneous inputs. These are not configurable (user can read them, set to store them or not, but not alter the routing, which depends on mode of operation). Similarly, the two fast RF inputs to the Red Pitaya FPGA are configured automatically based on the mode of operation. The details on how everything should be connected are given elsewhere.

---

set_out: sets static values of individual general purpose analog or digital outputs. These can be connected anywhere (e.g. to form tip or sample bias, or to control some electronics). They can be set at any moment, even during scan. Note that some measurement options can override the signals, e.g. when ramp is requested on any of the channels. After the ramp is finished, the values are recovered.

The following parameters can be set:

out1 (double): value of auxiliary analog output 1, in Volts
.
.
.
out16 (double): value of auxiliary analog output 16, in Volts

dout1 (boolean): status of digital output 1
dout2 (boolean): status of digital output 2
dout3 (boolean): status of digital output 3
dout4 (boolean): status of digital output 4

In the response, the altered values are sent back to confirm that the requested operation had happened.

---

set_routing: sets routing of general purpose analog outputs. By default, a constant value set by the set_out command is passed to them (which corresponds to the routing option “nothing”). If the routing option is e.g. “in1”, the signal from auxiliary analog input is fed to output. This is done on the software side, so some small delay might be expected. If the routing option is x, y, z, the actual values of voltage sent digitally to DACs for scanners/piezos are passed to the analog outputs. This can be useful for synchronisation to other instruments.

The following parameters can be set:
ch1...ch16 (string) routing: nothing, x, y, z,, in1,...,in16, error, zpiezo, amplitude1, phase1, amplitude2, phase2, fmresult, amresult, adc1, adc2 (string).

Command returns a complete routing table. Routing “nothing” does not affect the present settings, output value can be e.g. constant set by *set_out* command.
Some of the results are scaled to provide reasonable voltage values: x, y, z and zpiezo are multiplied by 10000, so +-100 um converts to +-1 V, fmresult (frequency shift) is divided by 100, so the shift +-100 Hz converts to +- 1 V.

D. Messages to approach or retract automatically, and to set feedback

There are not yet any messages for automated approach as the coarse z motion depends on instrumentation heavily. The approach still can be done manually, by using digital outputs to control the motor, using analog outputs to control piezo and monitoring the deflection signal on the client side. After that the feedback should be manually switched on.

---

set_feedback: switches the state of feedback from user control of z-piezo to feedback control or the PLL.

The following parameters can be set:
feedback (boolean)
feedback_pll (boolean)
feedback_amplitude (boolean)
feedback_kpfm (boolean)
zpiezo (double)

In the response it sends the feedback status. Feedback is related to the main feedback loop, feedback_pll to the PLL frequency feedback and feedback_ampitude to the PLL amplitude

feedback, in this case the setpoint value is the current value recorded at the time when the parameter is set. Zpiezo value is applied only when feedback is off and is applied immediately without any ramp. When the feedback is on, the zpiezo value still can be set, it does not impact the immediate value, however it will be used after the feedback is switched off (otherwise any other last known value of zpiezo would be used). The other additional feedback loops (e.g. KPFM, DART) are controlled internally.

---

move_motor: move some additional motor, either to approach or to control sample position
The following parameters can be set:
n (integer): motor identifier
distance (double): in metres

The realisation is hardware dependent, so hwserver and its settings should match the reality.

E. Messages to move somewhere

move_to: move the stage to some absolute position. Global speed settings are used.
The following parameters can be set
xreq (double), in metres
yreq (double), in metres
zreq (double), in metres

Zreq is applied only when feedback is off. In contrast to setting zpiezo from the set_feedback command where it is applied instantly, here it ramps to the value. Speed is controlled by the set_scan command and is independent for xy and for z.

---

stop: stop everything immediately.
This function stops all the motion, including scan or ramp if they are running.

F. Messages to scan something

Scanning starts by issuing a scan command. Different variants of scan commands are available.

- point by point scanning and storage based on xyz data (using Gwyscan library), using *run_scan_path* command
- continuous scanning on a line given by end points, storing automatically regularly sampled data using *run_scan_line* command
- custom scanning using Lua script that controls the whole scanning and data storage process on the server side using *run_scan_script* command.

The measured data are stored on hwserver and can be requested by user. Starting a scan clears the data scanned so far and eventually also re-allocates the amount of memory reserved for the storage. Scan then runs automatically - in case of *run_scan_path* or *run_scan_script* commands the whole image is collected and in case of *run_scan_line* a single line is collected. Client software can read the data scanned so far at any moment

using commands *get_scan_ndata* and *get_scan_data*. A scan can also be stopped at any moment by *stop_scan* command or paused and started again by *pause_scan* command. Prior to starting scan, client software can request that only a subset of available channels should be allocated and collected to speed up the microscope operation, by using the *set_storage* command. However, some channels are collected always for internal reasons, e.g. the positions.

---

set_scan_storage: set which additional channels should be stored internally during scan. By default, all the available channels are stored, which is slow and in most cases unnecessary. This function limits the storage to only what is needed and in this way speeds up everything. XYZ positions and error signal are always stored. The function clears the so far measured data and previously set storage parameters. Some of the channels can be duplicated, depending on the feedback regime - e.g. the error signal will be equal to amplitude in the amplitude based tapping mode.
The following parameters can be set to be acquired:
a1 (boolean): on/off, amplitude of primary lock-in signal
p1 (boolean): on/off, phase of primary lock-in signal
a2 (boolean): on/off, amplitude of secondary lock-in signal
p2 (boolean): on/off, phase of secondary lock-in signal
in1 (boolean): on/off, auxiliary ADC analog input 1
.
.
.
in16 (boolean): on/off, auxiliary ADC analog input 16
fmdrive (boolean): drive amplitude of the FM signal
kpfm (boolean): KPFM result
dart (boolean): DART result
l1x (boolean): primary lock-in x component
l1y (boolean): primary lock-in y component
l2x (boolean): secondary lock-in x component
l2y (boolean): secondary lock-in y component
set (boolean): sets of data to be used to distinguish different data when scanned from Lua script

---

set_scan: set global parameters of scanning
The following parameters can be set:
speed (double): in m/s, the value to be used for all lateral motion
zspeed (double): speed used in z move_to command, in m/s
delay (double): in seconds, to wait before each point acquisition
xslope (double): in radians, the sample tilt in x-direction to be compensated in PID loop
yslope (double): in radians, the sample tilt in y-direction to be compensated in PID loop
xsloperef (double): in meters, sample tilt zero point in x direction (usually scan x offset)
ysloperef (double): in meters, sample tilt zero point in y direction (usually scan y offset)
subtract_slope (boolean): subtract the sample tilt when reporting z piezo data

---

get_scan_ndata: get information about the scanned and stored data
The following parameters are returned:
n (integer): number of scanned data points from beginning of the scan command

---

get_scan_data: get some part of already scanned and stored data.
The following parameters can be requested:
from (integer): where the returned arrays should start (use 0 or -1 to set it to beginning)
to (integer): where the returned arrays should end (use -1 to set it to entire data)

In the response, it sends all the scanned data of stored channels in the range from-to as individual arrays of doubles, with the same naming as for the read command (i.e. the array name will be e.g. “in1”). It also returns an integer representing the number of values in the passed arrays. From and to parameter control range of data to be returned.

---

set_scan_path_data: provide a set of individual xy(z) points that will be used for run_scan_path command. Optionally, also send some trajectory in z (e.g. for lift mode operation).
The following parameters can be set:
n (integer), total number of positions (xy pairs) in the scan
from (integer), xy pair number to start from when filling the array at server side
to (integer), xy pair number to end at when filling the array at server side
xydata (array of interleaved doubles), the x and y positions organised as x1, y1, x2, y2...
z (array of doubles with n data points): z values to be used for scan line when feedback is off.

For short scans (e.g. 100x100 pixels) the data can be sent at once, however for larger scans it is more robust to send them piecewise. Parameters from and to are used to pass the data to the right location. Parameter n is used to allocate the arrays and find that the data are complete.

---

run_scan_path: run scan along path, given by a set of individual xy(z) points that is sent prior to the command. Optionally, also follow some trajectory in z (e.g. for lift mode operation).
The following parameters can be set:
n (integer), number of positions to scan

Function starts scanning pixel by pixel until all the n positions are not scanned. Data points should be passed before calling this function by single or multiple calls of set_scan_path_data. The measured data are stored internally and can be requested by the *get_scan_data* function.

---

run_scan_line: run scan along a line determined by present position used as start point and end point. Optionally, also follow some trajectory in z (e.g. for lift mode operation). The following parameters should be set:
xto (double), the end x position
yto (double), the end y position

regime (string): linear/smooth/sine
n (integer): number of data points to store
z (array of doubles with n data points): z values to be used for scan line when feedback is off. If this is missing, or feedback is on, no z motion is done.

Function checks and eventually reallocates the storage, calculates the x and y positions to stored data in regularly spaced data points (even if a sine pattern is used) and starts scanning and collecting data. Data points are stored internally and can be requested by the *get_scan_data* function. As the data points are corresponding to an equally sampled line profile, users could in future omit requesting the x and y values if a regular raster scan is supposed to be created, however these values are still stored internally and now sent for compatibility reasons.

---

run_scan_script: run Lua script to control scan and data points storage.
The following parameters can be set:
n (int): maximum number of data points to allocate for scanned data
script (string): script to perform the scan

More details can be found in the Lua scripting description.

---

set_script_param: add or change value of an entry to key-value table that can be read from Lua script.
The following parameters can be set:
key (string): the key to find the value in the table
value (double): the value

More details can be found in the Lua scripting description. At present up to 50 key/value pairs can be used.

---

clear_script_params: remove all the entries in the key-value table
Function has no parameters. More details can be found in the Lua scripting description.

---

stop_scan: stop scan immediately.
This function stops the scan and leaves the tip where it was at that moment. It does not alter the status of any of the control parameters including the feedback loop. Scanned data can be still retrieved, until the next scan command is entered.

---

pause_scan: pause or resume scan.
The following parameters can be set:
pause (boolean): pause (true) or run again (false)

This function pauses the scan and leaves the tip where it was at that moment. It does not alter the status of any of the control parameters including the feedback loop. Scanned data can be still retrieved, until the next scan command is entered. When the pause is cancelled, the scan continues. This function should not replace command *stop_scan* if we want to really stop, as pausing scan does not do any cleanup and scan status is still preserved as running.

G. Messages to ramp something

To collect a dependence of some quantity on another quantity in a single scan point, the ramp set of commands can be used. Ramp means that one of the quantities is going towards peak value and then back, potentially with some wait times between and behaviour of other quantities is monitored. Ramp can be realised via different quantities (z piezo, tip bias, etc.), providing different spectroscopy options (force-distance, tunnelling current, etc.). To set up the parameters of the ramp the *set_ramp* command is used. A single ramp is then run by *run_ramp* command. To get the last ramp values the *get_ramp_data* command is used, which returns all the ramp data obtained so far. Next ramp clears the data buffer. During ramp data acquisition all available channels are stored. The client can choose which should be passed back within the *get_ramp_data* command.

---

set_ramp_storage: set which additional channels should be stored internally during ramp acquisition. By default, all the available channels are stored, which is slow and mostly unnecessary. This function limits the storage to only what is needed and in this way speeds up everything. XYZ positions and error signal are always stored. The function clears the so far measured data and previously set storage parameters. Some of the channels can be duplicated, depending on the feedback regime - e.g. the error signal will be equal to amplitude in the amplitude based tapping mode.
The following parameters can be set to be acquired:
a1 (boolean): on/off, amplitude of primary lock-in signal
p1 (boolean): on/off, phase of primary lock-in signal
a2 (boolean): on/off, amplitude of secondary lock-in signal
p2 (boolean): on/off, phase of secondary lock-in signal
in1 (boolean): on/off, auxiliary ADC analog input 1
.
.
.
in16 (boolean): on/off, auxiliary ADC analog input 16
fmdrive (boolean): drive amplitude of the FM signal
kpfm (boolean): KPFM result
dart (boolean): DART result
l1x (boolean): primary lock-in x component
l1y (boolean): primary lock-in y component
l2x (boolean): secondary lock-in x component
l2y (boolean): secondary lock-in y component

---

run_ramp: run a single ramp
The following parameters can be set:
quantity: z/out1/out2/…/time , the quantity that will be ramped
from (double): the ramp quantity start value, in real units
to (double): the ramp quantity peak value, in real units

start_delay (double): in seconds, time before ramp start
peak_delay (double): in seconds, time at peak position
time_up (double): in seconds, how long the way up should last
time_down (double): in seconds, how long the way down should last
n (double): number of points in the ramp half-period (from start to peak), the number going up and down is equal

If *z* is the ramp quantity, it refers to the actual z position in the feedback, the feedback is stopped for the ramp and then restored again.

---

stop_ramp: stop ramp immediately.
This function stops the ramp and returns to the state before the ramp was issued. Measured data can still be retrieved, until the next scan command is entered.

---

get_ramp_ndata: get information about the scanned and stored data
The following parameters are returned:
n (integer): number of scanned data points from beginning of the scan command.

---

get_ramp_data:
Function returns ramp arrays in the specified order and number of data points in them (ndata). The behaviour is the same as for the get_scan_data function. Array `q` is the ramped quantity regardless it might or might not be also in some of the other fields.

H. Lua scripting interface

The scripting approach is based on using a Lua script that is passed by the client and runs on the server. The script controls the whole scanning process, going either point by point or by any more complex commands, that may now or in future include piecewise linear data acquisition without stopping, use of lift mode, starting/stopping feedback, conditions, etc.

All relevant internal data are passed to all functions as a single opaque structure p, that is globally available. It contains all the scanned points, their number and other data needed internally. The scanned points are independently passed to the client, which is not done by this script. The script only controls the scanning and data collection process.
To create your scan procedure, create the runit function as in the example script.

Collected data can be assigned to different sets. If sets (other than set 0) should be used, the set_scan_storage command should be used to ensure that the set arrays are allocated.
Note that there are no scan parameters that could be passed from the server via client-server interface. Everything that should be controlled by the script should be fixed in the script (range, resolution, number of points) and cannot be changed on the fly. It is assumed that every scan is initiated by sending the script using the **run_scan_script** command via the client-server interface**.** Everything that is not controlled by the script (e.g. feedback loop PID parameters) and is also related to the microscope operation when not

scanning, continues to be controlled independently via server and client and can be eventually changed on the fly.

Available functions:

gws_clear(p) -- clear all data points

gws_get_nvals(p) -- get number of data points stored so far, returns the number
gws_get_x(p) -- get actual x value
gws_get_y(p) -- get actual y value
gws_get_z(p) -- get actual z value
gws_get_e(p) -- get_actual error signal value
gws_get_in(p, channel) -- get actual additional value of particular channel
gws_get_t(p) -- get actual timestamp value

gws_get_z_at(p, x, y, from, to, set) -- get_stored z value closest to x,y,
from given data range and set, returns the value
gws_get_entry(p, index) -- get position and feedback data stored at given index,
returns x,y,z,e,ts,set
gws_get(p, string) – get actual value of a parameter defined in description of the
get command

gws_set_feedback(p, on) -- set feedback on or off. When off, the last z is used for piezo
gws_set_speed(p, speed) -- set speed, in m/s

gws_set_fm_feedback(p, on) – set fm feedback (PLL)
gws_set_fm_amplitude_feedback(p, on) – set fm amplitude feedback
gws_set_gen_amplitude(p, channel, amplitude) set generator amplitude
gws_set_kpfm_feedback(p, on) – set KPFM feedback
gws_move_to(p, x, y ,z) -- move to x, y, z (in metres), z runs only if feedback is off
gws_store_point(p, set) -- store actual readings with index of a specific set
returns the stored data index
gws_check_if_stopped(p) -- returns 1 if there was a stop command from client
gws_check_if_paused(p) -- returns the pause flag, if it is 1, scan should be paused
gws_get_scan_param(p, key) -- get value of scan parameter identified by string key

gws_scan_and_store(p, xto, yto, nvals, set) -- moves to xto, yto, while scanning n vals
with given set number,
vals locations are equidistant.
gws_scan_and_store_lift(p, xto, yto, nvals, set, zindex, lift, no_fb_action) – performs line scan in lift mode,
using globally available z data starting at zindex (and successive values) as the topography. Parameter no_fb_action can be used to prevent the feedback to be set automatically off before lift and on after that (user has to do it)

gws_set_result(p, key, value) set something that controller can read.

Example script:
Running a 5x5 points, 1x1 um range regular scan, followed by a lift scan.
Line by line scan, line by line lift scan.

```
-- file scan.lua ---------------------------------------------------------------------------------
local scan = {}

function scan.runit()
  gws_clear(p)                              -- clear all points
  gws_set_feedback(p, 1)                    -- feedback on, probably it already is
  for x = 0, 1e-6, 0.25e-6                  -- first pass
  do
         for y = 0, 1e-6, 0.25e-6
         do
            gws_move_to(p, x, y, 0)
            gws_store_point(p, x, y, 0)     -- we store the data as set 0
         end
  end
  n = gws_get_nvals(p)                      -- data index where first pass ends

  gws_move_to(p, 0, 0, 0)                   -- move back to 0,0
  z = gws_get_z(p)                          -- get the actual height
  gws_move_to(p, 0, 0, z + 100e-9)          -- set where to move when feedback off
  gws_set_feedback(p, 0)                    -- switch off feedback

  for x = 0, 1e-6, 0.25e-6                  --second pass
  do
         for y = 0, 1e-6, 0.25e-6
         do
            z = gws_get_z_at(p, x, y, 0, n, 0)
            gws_move_to(p, x, y, z + 100e-9)
            gws_store_point(p, x, y, 1) -- now we store the data as set 1
         end
  end

 gws_set_feedback(p, 1)                     -- switch feedback on again

 -- debugging: check last entry to see what happened
  n = gws_get_nvals(p) - 1;
  x, y, z, e, in1, in2, t, s = gws_get_entry(p, n)
  print(string.format("last entry: xyz %g %g %g  inputs %g %g %g  timestamp %g  set %g ",
x, y, z, e, in1, in2, t, s))
  print("Scan script successfully completed.")
end
```

```
return scan
-- end of file scan.lua ---------------------------------------------------------------------------

local scan = {}

function scan.runit()
	local z = gws_get_z(p)

	gws_move_to(p, 0, 0, z);

	for iy = 1,500 do
	local xfrom = 0
	local xto = 500*20e-9
	local y = iy * 20e-09

	gws_scan_and_store(p, xto, y, 500, 0);
	gws_move_to(p, xfrom, y, z);

	if gws_check_if_stopped(p)==1 then
	print("stopped on lua side")
	return scan
	end

	end
end
return scan
-- end of file scan.lua ---------------------------------------------------------------------------

local scan = {}

function scan.runit()
	local z = gws_get_z(p)

	gws_move_to(p, 0, 0, z);

	for iy = 1,500 do
	local xfrom = 0
	local xto = 500*20e-9
	local y = iy * 20e-09
	local zindex = gws_get_nvals(p);

	gws_scan_and_store(p, xto, y, 500, 0);
	gws_move_to(p, xfrom, y, z);

	gws_scan_and_store_lift(p, xto, y, 500, 1, zindex, 100e-9);
```

```
        gws_move_to(p, xfrom, y, z);

        if gws_check_if_stopped(p)==1 then
        print("stopped on lua side")
        return scan
        end

        end
end
return scan
```

Examples of how the microscope operation can be organised

Setting up the microscope operation mode

Choose the regime of operation using the *state* command. This does not affect much the interface between the client and server (described in this document), but it has a large impact on internal organisation of the feedback loop on the server side - using different data sources for feedback, running lock-ins, performing second feedback loop for the KPFM, etc. Using a state that does not match the real microscope wiring on the electronics side can lead to unpredictable behaviour.

Reading live signals

When the microscope doesn't do any scanning or ramping, either in feedback or not, it continuously reads all the channels requested by *set_standby_storage* command and offers them as live data using the *read* command.

Tapping mode probe setup

Use the general *set* command to gradually set the primary frequency generator frequency and *read* command to see the response to get the dependence of the phase or amplitude on frequency.

Akiyama probe setup

Use the general *set* command to gradually set the frequency and *read* command to see the response to get the dependence of the phase or amplitude on frequency. Something more will be added in the future.

Getting into contact and setting up feedback loop

Use the *set_out* digital or analog signal connected to something in your hardware to run your approach motor. Use *read* command to see if the setpoint value was already reached. To

create a more complex approach pattern, use *move_to* command to move the z-piezo while the feedback is off. After reaching the setpoint values suitable for feedback, use the *set_feedback* command to let the server know that from now on it should control the z-piezo values itself.

Input and output signals setup

To run an experiment where additional signals are needed, use the *set_out* command to set the static auxiliary analog output value. On the instrument side, wire this to the desired location. For example, a conductive AFM scan can be run by setting the auxiliary analog output 1 to our desired tip bias value and physically wiring the output 1 to the tip. To run photoconductive AFM with different illuminations, you could use the auxiliary analog output 2 to control the illumination. You can then switch the illumination on/off during the scan as needed.

If some more complex experiment has to be set up, probably involving third party hardware, it might be needed to route some other signals to the outputs. This can be done by *route_out* command.

Raster scan

To run the simplest raster scan, set up the scan parameters like speed by *set_scan* command. Then, optionally, set which channels should be stored internally by *set_scan_storage*. Run a single profile in the forward direction by the *run_scan_line* command, asking for N equally spaced data points to be measured. Check if all the data in the profile were already measured with the *get_scan_ndata* command. Read all the values by *get_scan_data* command and store them on the client side as a single line of the forward scan. (You might also read the values during the scan and display them live using *get_scan_data* repeatedly). Run a single profile moving back, and read data to store them as the reverse scan. In this manner go through all the M scan lines. Now you should have two images of M lines x N values, one representing forward and one reverse values. If you requested more channels to be passed, you will have multiple pairs of images.

Adaptive scan using Gwyscan library

Set up the scan parameters like speed by *set_scan* command. Then, optionally, set which channels should be stored internally by *set_scan_storage*. Create the scan path, ideally using the Gwyscan library. Pass the scan path to the server and collect the data using *get_scan_data* and *get_scan_ndata* commands until the scan is not finished. You can use Gwyscan preview functions to rasterize the data for the user preview.

Script based scan

Prepare the Lua script and estimate how much scan data points need to be allocated on the server side. If some parameters need to be changed on the fly, use the set_script_param

command to add them (use the command to modify them on the fly). Use the command *run_scan_script* to start the script. Collect the data using *get_scan_data* and *get_scan_ndata* commands. To check if the scan has finished use either the number of points known before scan or check the status of variable “scanning_script” using *get* command.

Single ramp acquisition

Start by obtaining a topography image (optional, but typical step) using either raster or adaptive mode. Use *move_to* command to get to some position on the surface. Optionally set a subset of channels to be stored by using *set_ramp_storage* command. Use *set_ramp* to set the basic parameters of ramp, like routing of the signals. Use *run_ramp* to perform the ramp and store internally the requested number of points. Use *get_ramp_data* to get the resulting data arrays.

Force volume data acquisition

Using the procedure shown in the previous paragraph, run a single ramp in every point of the scan.

Lift mode data acquisition

First run the regular profile or whole scan in feedback, either by raster mode or adaptive mode. Switch feedback off. Use the measured data to form the desired lift z path, using a constant value amended by lift height or following the topography amended by lift height. Use *run_scan_line* or *run_scan_path* commands containing the z data (of the same resolution as the requested new profile or path) to run the tip in lift mode. Collect data the same way as for the topography profile, but save them elsewhere, this will form your lift data channel.

AM KPFM data acquisition

Set the scan mode using the *state* command to AFM KPFM. Organise your scan in a similar procedure to the lift mode. In the second pass, set up the KPFM feedback loop parameters as pid2_p, pid2_i, pid2_d using the general *set* command. Store the auxiliary output value that was used for nulling the contact potential.

Lithography

After obtaining a topography scan, create a lithography path as a discrete set of points similarly to the xyz path used in adaptive mode, adding an array of delays in each point. Before starting the path, set the auxiliary output value that should produce the desired surface modification (setpoint, out1..out16) and revert it back after the lithography path was passed.

TODO before shipping

- set the physical ranges in hwserver.ini
- set all the pins on board and fit the ranges in hwserver.ini to them

- connect outputs to inputs, run test_spi or similar tool to check the fast ADC calibration coefficients, eventually change them in hwserver.ini
-

End of the document

To be done later

---

add_ramp_peak_rule: set additional threshold to stop ramp, e.g. when some quantity is above or below some value. You can add multiple rules. To remove them, use *remove_ramp_peak_rules* function.

Parameters:
quantity (string) x, y, z, err, a1, p1, a2, p2, in1...in16
logic (string), bigger, lower
value (double) value in base units

---

remove_ramp_peak_rules: remove all peak ramp rules. If you want to only change rules, run the set_ramp_peak rule again for all of them.