

met5gui
-
general architecture

Christopher N. Anderson, Ryan H. Miyakawa, Antoine Wojdyla
EUV Lithography - Center for X-Ray Optics - Lawrence Berkeley National Laboratory

March - November 2013

Summary

met5gui is a framework developed in MATLAB programming language in order to offer a scalable user interface for the .5NA Micro Exposure Tool (MET5) currently built at the Center for X-Ray Optics.

It is inspired by the current MET software and various data acquisition user interfaces developed in-house. We present here what are the requirements and considerations that were taken into account for designing it. Among those, compatibility with low-level controllers (*mostly*) written in JAVA, robustness and thorough debugging, coding good practice and inspiration on common stage and sensors components.

We provide a rather comprehensive documentation of the elements of met5gui, for further improvements and use in other situation, such as the building of a general purpose optical experiment. met5gui framework has already been use in numerous situations, what allowed a first-pass debugging and validated the choices we've made. Finally, we perform a critical assessment of the current framework.

Contents

1	Introduction	6
1.1	Goal of this document	6
1.2	met5gui framework	6
1.3	Inspirations	6
1.3.1	Coding guidelines	6
1.3.2	Common API components	7
1.3.3	Previous tools' software	10
1.4	User experience	13
1.4.1	Standardizing API for Matlab	13
1.4.2	Matlab class breakdown	13
1.5	History	13
1.5.1	The choice of Matlab	14
1.5.2	Controllers in JAVA	14
1.5.3	Initial Timeline	14
1.6	System integration	14
1.6.1	EPS and kill switches	14
1.6.2	Feedback loops	15
1.6.3	Compound measurements	15
1.6.4	Kinematics	15
1.6.5	Major shifts	15
1.6.6	Wrapping UIElements	16
1.6.7	A Clock to rule them all	16
1.6.8	Return of experiment	16
2	Implementation & Conventions	19
2.1	Software structure	19
2.2	Naming conventions	19
2.3	Matlab compliance	19
3	Documentation	20
3.1	Clock	20
3.2	Instrument control	20
3.2.1	Axis	20
3.2.2	Diode	21
3.2.3	DiodeAPI	21
3.2.4	HardwareIO	22
3.2.5	HardwareO	22
3.2.6	Shutter	22
3.2.7	Camera	22
3.2.8	Scan	22

3.2.9	Monitor	22
3.3	API & wrappers	22
3.3.1	nPoint	22
3.3.2	APIVHardwareIO	23
3.3.3	APIHardwareIOOnPoint	23
3.3.4	ConfigMotor	23
3.4	UI elements	23
3.4.1	UIEdit	23
3.4.2	UIButton	23
3.4.3	UIList	23
3.4.4	UIText	23
3.4.5	UICheckbox	23
3.4.6	UIToggle	23
3.4.7	UIPopup	23
3.4.8	UI2DNav	23
3.4.9	List	23
3.4.10	Panel	23
3.4.11	Utils	23
3.5	Controls	24
3.5.1	Mirror	24
3.5.2	ReticlePick	24
3.5.3	PupilFill	24
3.5.4	HeightSensor	25
3.6	Debugging and examples	25
4	Room for improvement	26
5	Annex	27
5.1	Newport ESP300 controller - 3-axes linear stage controller	27
5.2	SmarAct MCS 3-Axes Piezo motor controller	30
5.3	Stanford Research Systems SR830 DSP Lock-In Amplifier	32
5.4	GalilAxisVT50	32
5.5	Micos PMC100	34
5.6	Mirrorcle	35
5.7	Wago	35
5.8	PaulBox	36

1

Introduction

1.1 Goal of this document

The intent of this document is to give an overall perspective of the elements that drove the development of the met5gui software, the User Interface for controlling the .5NA Micro Exposure Tool (MET5) developed at the Center for X-Ray Optics at Lawrence Berkeley National Laboratory.

We also want to provide the a faithful picture of the the functions that are on the scope of the met5gui, and how to bind the basic elements of the framework together, for further uses.

1.2 met5gui framework

The met5gui framework is a set of Matlab classes that allow the development of robust interactive software, mainly for the user interface of MET5 but keeping it rather general so that any experiment involving motors and sensors can be easily easily built on top of this framework.

1.3 Inspirations

In order to build a robust framework, we have looked at available resources to provide some guidelines, listed here.

1.3.1 Coding guidelines

Computer programming does not have clear guidelines (“*Science is to computer science as hydrodynamics is to plumbing.*” – J. C. Clark.), so we managed to summarize best practices from various sources.

Coding best practice

1. Commenting & documentation (design and purpose, not mechanics)
 - (a) Consistent indentation (do not mix space and tabs)
 - (b) Avoid obvious comments
2. Consistent naming scheme (define it!)
 - (a) Minimize the use of abbreviations
 - (b) Consistent temporary names
 - (c) Avoid homonyms

- (d) Pair antonym procedures (open/close, etc.)
- 3. Keep your code simple (The “20 lines” rule)
 - (a) Avoid deep nesting (hard to read)
 - (b) Limit line length (hard to read & viewer dependancy)
 - (c) File and folder organization
 - (d) Don’t repeat yourself
 - (e) Code refactoring
 - (f) Code grouping
 - (g) Use globals sparingly
- 4. Write programs for people, not computers
 - (a) Do you have testers/hallway usability testing?
 - (b) Provide useful error messages
- 5. Have a high level design or a low level design and Function specification and proof of concepts
 - (a) Push interface up and implementation down
 - i. Level 1 (high): accept user input
 - ii. Level 2: taint check and normalize user input, and check for errors
 - iii. Level 3: process user input according to business logic
 - iv. Level 4 (low): store data
- 6. Separation of code and data
 - (a) Wrap built-in functions and third-party library functions with your own wrapper functions
 - (b) Don’t assume output formats
 - (c) Internal data should be in native format
 - (d) Operate on objects, not data structure
- 7. Test returned status for error conditions
 - (a) Recover or fail “gracefully” : Robust programs should report an error message
- 8. “Premature optimization is the root of evil” (Optimize software only after it works correctly)
- 9. Never rewrite code from scratch

1.3.2 Common API components

We have gathered together common API components, taken from NEWPORT ESP300 controller, SMARACT MCS 3-axes Piezo motor controller and a STANFORD RESEARCH SYSTEMS SR830 DSP Lock-In Amplifier and from controllers developed in house (see 5 for a comprehensive list). Here are the most common functions that are required to control a scientific tool.

Stage

motion

- **move**

The motor motion is obviously the most important function for controlling a motor. At the very least, an API should have the ability to read an *absolute* and *raw* position, for it can be converted downstream to a more user-friendly calibrated and relative movement.

It is a limited version of **move to target** function, since the motions stops after the target has been reached.

- **move to target**
Having the possibility to set a motion to a target (in a *non-blocking* fashion) is important in the case of iterative tracking, when the target is likely to change before the position has been reached. This is especially important for alignment and calibration.
- **pause**
`pause` is related with the *move to target* function : it temporarily stops the position update.
- **stop**
`stop` can be used while a `move` or a `move to target` is in progress.
The target position is set to the current position
- **abort**
`abort` stops the current motion and resets the target. It is more general than the `stop` in the sense that it breaks the feedback loop.
It can be viewed as an emergency stop.
- **jog**
Jogging is interesting for alignment, where the ideal position is not known in advance. It doesn't require the motor position to settle down after each visited position
- **move one step**
Increase the position by the smallest step possible.
This allows to have a common framework between stages with optical encoders and those with brush encoder.
- **move home**
Homing procedure is important for defining an absolute position, which can be relied on when the system is reinitialized
- **move to limit** Moving to the limit is sometimes important to calibrate the motion, and decouples the limits set by the encoder reading and the physical motion.

position

- **read position**
It is an essential component to check whether the stage is in the right position, and how far it is actually from the set position.
- **read target**
Lets you know, in conjunction with `read position` how far you are from the target.
- **wait for stop**
A blocking command that is used for enforcing position settlement.
- **pause/acquisition data/data done status**
Used to disconnect the feedback loop to have a steady state during acquisition.

settings

- **init/open/start**
Establishes the communication between the hardware and the computer.
- **motor on/off**
Allows the motor to be turned off, either for thermal constraints or to ensure stability.

- **axes properties, axis number, axis ID**
Each motor must have a unique ID to make sure we are actually communicating with the proper stage and check the status (on, unlocked, available for movement, not touching a limit switch)
- **general infos**
(controller version etc.)
- **Speed/Acceleration settings** (*e.g.* jog speed)
Useful for jogging and optimizing the speed/sensitivity ratio.
- **PID settings/refresh rate**
Useful for optimizing the speed/sensitivity ratio.
- **tolerance**
Sets the tolerance for which the motion can be considered as completed.
- **high/low limits**
Setting either physical limits (limited by the stage or the encoder) or safe-to-work positions.
- **isReady (for motion or acquisition)**
Essential to check if the stage is ready to proceed to the next step of the procedure (motion or data acquisition)
- **read internal data** (*e.g.* stage resolution)
- **status** (moving, hardware)
Checks whether the stage is connected and ready to work (can be coupled with `init.`)
- **offset/slope/units** (calibration)
Tells what are the internal units and calibration. This is especially useful when there is no documentation at hand, and to synchronize motion of different stages.
Ideally, they should all be expressed in the same set of units (ideally meters), provide a unit indicator or bear a name that is unambiguous (*e.g.* `slope_ct_um.`)
- **voltage, level, actual speed**
Useful to perform some troubleshooting (*e.g.* when there is a failure in the power supply etc.)
- **lock/unlock**
Allows to lock on stage when performing a synchronous motion, to avoid fatal interference with other commands.
- **min/max range**
Allows to know where the stage is relative to its extreme positions
(centering is usually a good habit, especially when coupling coarse and fine motion stages sharing the same axis)
- **polarity/direction**
Allows to change the polarity of the motor, to comply with a set of defined axes (following the hand rule is usually a good habit.)
- **limit switches status**
Allows to check whether a hard or a soft limit has been reached.

further comments In general, there are three categories of motion : the instruction to go to a position, the pursuit (continuous motion) and the scan (the motor settles between positions). We have not included specific parameters that can be found in certain stages (like the PID parameters for things other than motion); they must be defined case-by-case.

Sensor

reading

- **read**
Reads the sensor value. This is the purpose of any sensor

parameters

- **readout time**
The delay between signal readout (informs of the maximum speed at which the sensor can be read.)
- **time constant**
This is how much the signal is integrated before acquisition. It is different from the readout time, since the signal can be read at speed higher than the time constant (*e.g.* in the case of a feedback loop), or at lower speed (the SNR is then limited by the time constant). in the case of a camera, this is equivalent to exposure time.
- **sensitivity/gain**
When the sensor is read through an amplifier, or if the sensor is a camera, it is useful to be able to change the sensitivity/gain parameters to maximize the dynamic range.

1.3.3 Previous tools' software

We have learned a great deal by analyzing the software made by Ken Goldberg for other tools used at CXRO.

MET 3

The current MET3 lithography tool was designed by Ken Goldberg and coded in IDL programming language. It comprises many independent elements, all monitored by a core (PRONTO) that perform highly specialized functions (Fig. 1.1). It has been made thoroughly tested and debugged. It is worth mentioning that the soft consistently crashes when it has not been reinitialized after one day, for not obvious reasons.

SHARP

The latest software designed by Ken Goldberg benefited from all the previous tool software development experience. The main improvement was the creation of the concept of *Chaperon*, which is a global thread manager that allow the gracious synchronization of all the elements of the software, which are divided into many chunks to increase the scalability of the code (Fig 1.2).

In addition, the UX was carefully designed, especially for the keyboard navigation increasing ease-of-use for the users of the tool (Fig. 1.3).

Pupil Fill Monitor

The CXRO Pupil Fill Monitor (PFM) software was developed by Chris Anderson using MATLAB (Fig. 1.4.) It served as a proof-of-concept validation, and helped to lay the foundation of met3gui.

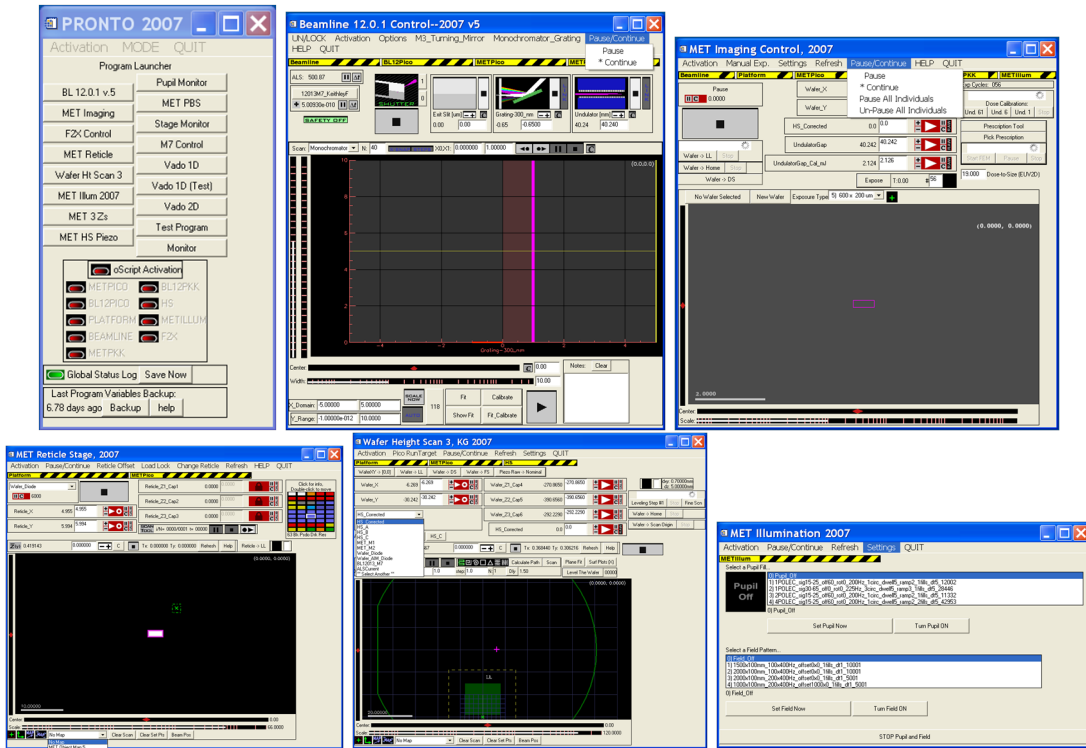


Figure 1.1: MET3 sotware GUI (courtesy of Ken Goldberg.)

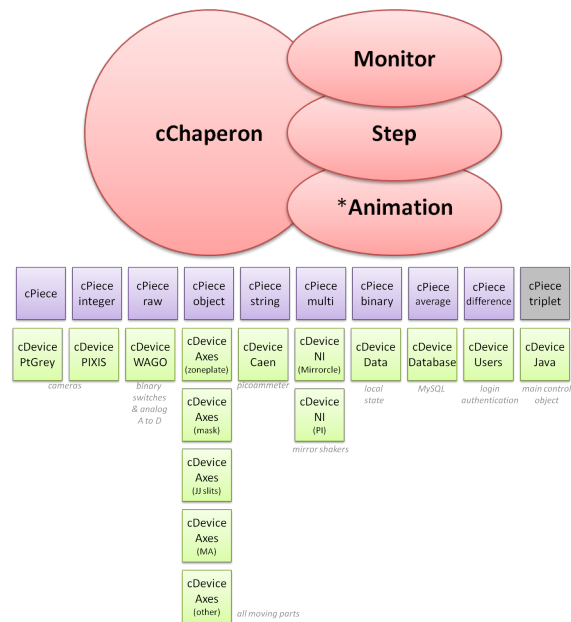


Figure 1.2: SHARP software structure breakdown (courtesy of Ken Goldberg.)

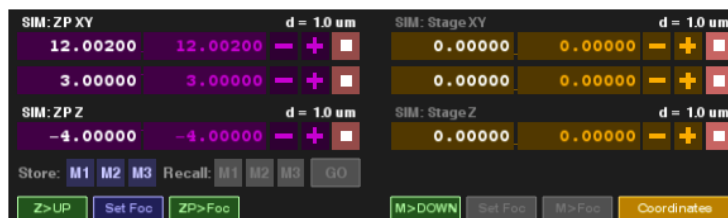


Figure 1.3: SHARP axis control

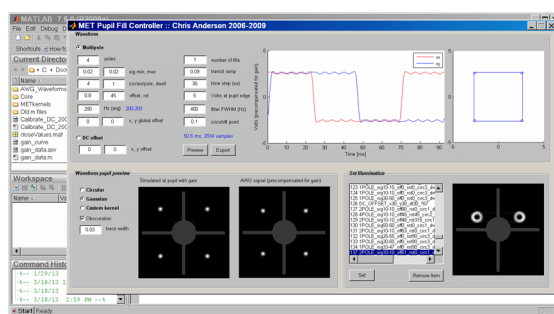


Figure 1.4: CXRO Pupil Fill Monitor GUI.

1.4 User experience

We wanted to leverage the user experience (UX) of users of the current MET3 tool, in order to produce the. We interviewed MET3 operators, who told us they were happy with the current software, especially its modular aspect (*a transcript of the interviews is available somewhere*), and they had no particular grief to tell us. This is why we tried to keep the met5gui structure as similar as possible to MET3 software.

1.4.1 Standardizing API for Matlab

Since there is no framework in MATLAB to perform measurements, we have defined a standard that all met5gui components should follow.

Vital components

In this framework, the most vital functions are :

- **Move** defined as absolute and in raw units,
- **Read position/sensor** defined as absolute and in raw units,
- **Completed Motion** \simeq **isThere** to check motion completion,
- **Abort** to stop/abort the current motion (change in target or emergency stop),

1.4.2 Matlab class breakdown

Most of met5gui classes must have these methods :

- **constructor** creates an instance of a class; it must not include vital elements such as motor initialization, which are refactored in the **init** method,
- **init** initializes the instruments, loads the libraries etc. It is decoupled from the constructor in order to allow for re-initialization,
- **build** builds the UI element corresponding to the class (controls and windows).
- **handleClock** defines what are the actions to perform when the class is called by the common clock,
- **delete** frees the memory gracefully.
Even if there is a garbage collector in MATLAB, things can go awry if the class is not isolated from the rest of the elements before it is deleted (the object must be removed from the clock task list, UI elements must be killed, all members that are instances of of controller class must be disconnected and deleted);

1.5 History

It is certainly interesting to recall the decisions and why we have made them.

1.5.1 The choice of Matlab

We have favored MATLAB over JAVA and C++ for versatility and stability. The use of an interpreted language is

JAVA would have been interesting for it is ubiquitous and could be ported to any kind of device (computer, tablet or even smartphone), and could have been good for connectivity purposes (e.g. communicating data through a website). However, some stability issues and the lack of powerful tools for scientific computing to lead us dismiss this choice.

PYTHON was overlooked, for it is not the first choice for a robust GUI. Available Framework (such as SCIPY or PYQT) could have been a good help for scientific computing. C++ is a compiled language, but the fact that the bytecode imposes an extreme dependence on the choice of the machine the code runs on would impose problems on deployment of the software solution.

MET3's software and SHARP demonstrated that a high level language would work well for the purpose of controlling a lithography tool. That is, MATLAB being agile with both C++ DLL and JAVA classes and somehow platform-independent, added to the fact that newer version of MATLAB allows Object-Oriented Programming easy to debug, to build robust Graphical User Interface and that it can be compiled and easily deployed without the actual need for a license, it turned out to be the best choice for MET5 software development.

1.5.2 Controllers in JAVA

At the very beginning, we had to choose whether the low-level components would be coded in C++ or JAVA. A series of test were lead to compare the speed of both solutions, and JAVA proved to perform well. It also has the advantage of being somehow more compliant with CORBA or ICE. Many in-house low-level controllers have already been coded using JAVA, hence empowering us to leverage on already available controllers. The 32 bits vs. 64 bits problem also seems to be less relevant in the case of JAVA, so we eventually decided to use this programming language for hardware control.

A solution using MATLAB MEX files was dismissed, for they do not allow the keep the hand on the hardware at all time (*it this really true ?*).

1.5.3 Initial Timeline

The initial schedule for met5gui is presented Fig.1.5. This schedule was defined at a time when the beamline components were believed to be delivered by August 2013, and the MET5 end-station ready to use by mid-October 2013. As of November 2013, the project is not completed. Most of the components are ready, and deploying is believed to be relatively easy.

The met5gui infrastructure has not been tested on the actual system for obvious reasons. Yet, it has been thoroughly debugged, tested and validated on several experiments presented in section 1.6.8, and we believe that adapting to the lithography tool and fine-tuning should be straightforward, given that all the lower-level elements are operative (motors, cables, controllers, DLL & JAVA classes).

1.6 System integration

met5gui is a on top of many elements, and it is legitimate to questions whether some desirable system features, such as kinematics calculations or closed-loop action, can be delegated to lower layers. The actual communication route via VMI or CORBA must also be discussed.

1.6.1 EPS and kill switches

All the EPS and kill switches should be low level, to prevent any malfunction in the software. However, some 'security' features such as camera can be implemented at higher level, for they require an action from the users.

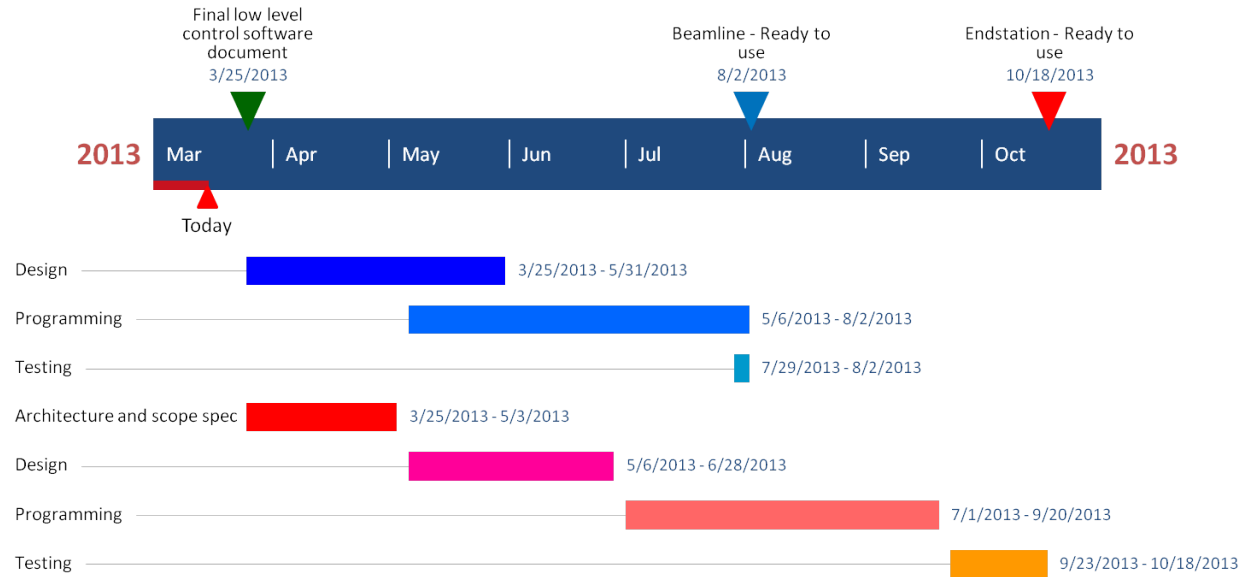


Figure 1.5: Initial project timeline, as defined March 15th, 2013

1.6.2 Feedback loops

Feedback loops, essential to keep the wafer at level by compensating for drifts, must be defined at low level, since they need to operate at high speed (1 KHz). Moreover, having feedback loops at different levels of the system might be detrimental, for that might cause interferences between loops acting at different levels.

1.6.3 Compound measurements

In the case of compound measurements, *i.e.* measurements that are aggregated into a single value, like the height of the height sensor, it would be preferable to have the calculations done at lower level, since these value are meant to feed feedback loop, which might be disturbed by too much overhead in the software.

However, since these measurements rely on calibration (E.G. the actual spot size on each segmented diode, or a best-fit), having the compound measurements at higher level can be beneficial. Though, the main software could easily interact with lower-level controllers to perform calibration.

1.6.4 Kinematics

The kinematics could be handled by the software, though some stage kinematics algorithm might be proprietary and provided by the vendors, like in the case of an Hexapod. For simpler systems (E.G. wafer stage, Z-Tip-Tilt), we might envision to use the software to perform these calculations.

Again, the need to operate a feedback loop at high-speed might drive the need to have these algorithms implemented at low-level, though it might be helpful to have control on the the kinematics algorithm to correct errors in calibration, E.G. imperfect parallelism between stages.

1.6.5 Major shifts

The major shifts we've made during the process of developing met5gui are the following :

Hungarian notation

MATLAB performing loose enforcement on data types, we decided to use the Hungarian notation to prevent obfuscation in the code.

1.6.6 Wrapping UIElements

To comply with the requirement for separation between data and interface, we decided to create UI elements that perform basic data validation and can be easily abstracted from the data.

Save & load

To save the state of the tool, we decided to implement a recursive procedure, so that the parameters can be (re-)loaded by a simple function call.

1.6.7 A Clock to rule them all

The discussion we had with Ken Goldberg about SHARP software, and the difficulties we had to synchronize all the timers included in many different classes, lead us to the creation of a class, `Clock`, inspired by SHARP's *Chaperon* that is shared by most of the other classes, defining a clear sequence for the actions to be performed. We had issues with speed at first, that were corrected by tuning memory allocation.

Separation of Axis and HardwareIO

Using the met5gui framework on other experiments, it became clear that there is a fundamental difference between the motion of a stage (performed using `HardwareIO`) and the motion of an object in space (*e.g.* along a direction, performed using `Axis`), which can be a compound motion.

1.6.8 Return of experiment

During the course of programming for met5gui, many experiments were led in the optical prototyping room. Since most wrappers were non-existent or wildly out-of-date, it was required to write all the MATLAB from scratch, leveraging on-the-spot training, knowledge and available experience. That is, it was a good occasion to put at use the met5gui framework, debugging its features, extending its possibilities and questioning its functions. Here is what we've learned from building acquisition tools :

AIS

Building the optical prototype for *AIS* required to write from many stage control and sensor classes, including a flexure stage and diode controlled by a *PaulBox*, a set of two *Micronix* VT50 linear stages using a GALIL controller, and a Z-Tip-Tilt tripod developed in-house.

Z-tip-tilt tripod

The Z-tip-tilt tripod developed at CXRO is meant to provide increased stiffness in the Z-direction compared to other tripod structures. It comprises three *Micronix* Pp30-8MM stick-slip translation stage, whose motions are transferred to a main platform (Fig. 1.6) The kinematics relations were not defined yet; we managed to figure them out, and allow a user to independently change the height, the tip and the tilt, by abstracting individual motors position (*a detailed report is available upon request.*) This lead us to separate the notion of `Axis` and `HardwareIO` in the met5gui framework (Fig. 1.7, left).

The AIS Experiment also gave us some feedback to carve out a coherent stage and sensor framework : the flexure stage is actuated through a C++ DLL, the VT50 linear stage directly through a JAVA class, while the tripod uses a JAVA class to communicate with the stage, allowing to send queries to the individual motor.

At the end of it, *AIS* experimental results were found satisfactory.

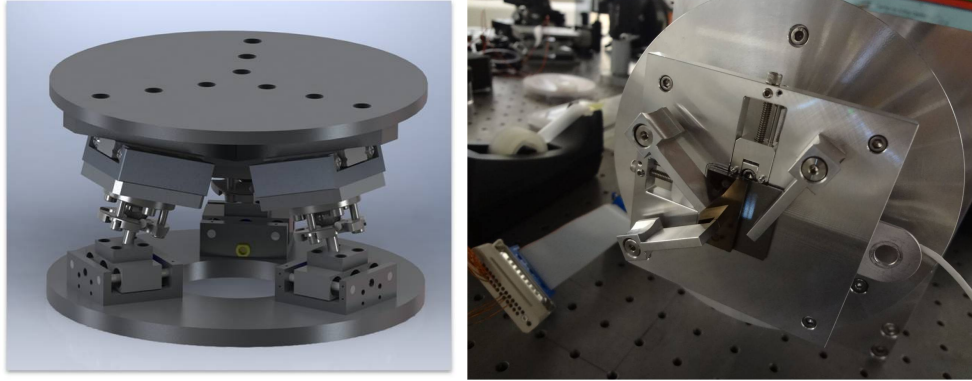


Figure 1.6: CXRO's Z-Tip-Tilt tripod. left : 3D CAD model – right : mounted tripod in the AIS experiment.

Coded Aperture

The *coded aperture* experiment taught us how to deal with other kinds of motors and sensors, slightly different : a XZ two-axis stage coupled to an optical encoder through a GALIL controller and a photo-diode whose output was read through a WAGO ADC. It was a good occasion to interact with Carl Cork and get a mutual better understanding of what were our needs and our issues, and become aware of certain subtleties such as the partial incompatibility of JAVA versions 1.6 and 1.7 (only one can be loaded with MATLAB), but also Operating Systems incompatibilities with some drivers (some drivers do not work on *Windows 7*.)

The alignment of the setup required the use of a camera, what was the starting point for programming the `Camera` class. With all the fine-tuning allowed by the use of a coherent framework (not really *met5gui* yet), *coded aperture* experimental results were deemed suitable for publication.

Picomotors and cap sensors vacuum test

Deploying some of the tools created for *AIS* and *coded aperture* experiments on an other setup proved the robustness of a limited subset of the framework (Fig. 1.7, right), while attracting the attention on the fact that MATLAB versions are not fully compatible. However, newest iterations of the language seems to be pretty solid.

Coherent Multiplexed Ptychography

The current study of *Coherent Multiplexed Ptychography* allowed us to try stage motion, coupled with camera acquisition and asynchronous scanning procedure (the first real-life test for the `Scan` class.)

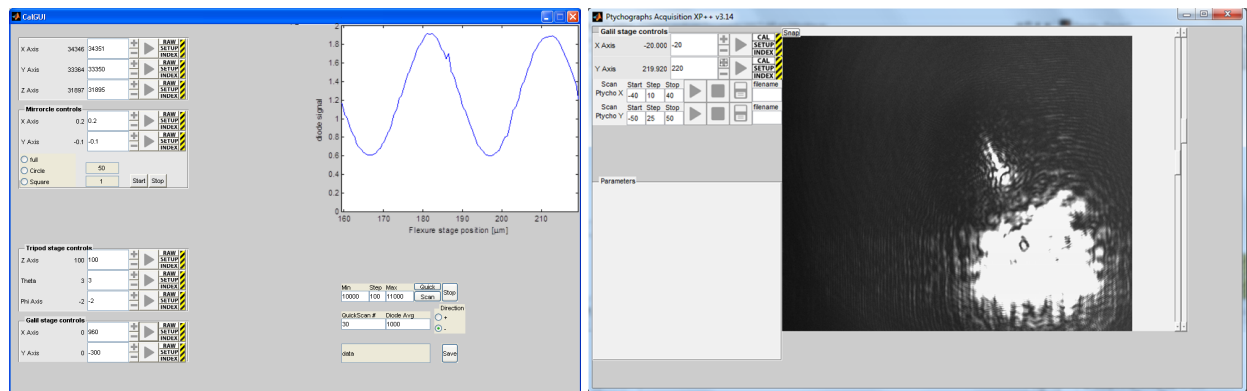


Figure 1.7: Examples of two UIs based on met5gui. left : CalGUI (*AIS*) – right : Ptychography Acquisition.

2

Implementation & Conventions

2.1 Software structure

(needs further developments)

2.2 Naming conventions

met5gui follows both *CamelCase* notation (*e.g.* `nameOfMyVariable`) and *Hungarian* notation, *i.e.* the name of a variable is prefixed by the data type (*e.g.* a string is prefixed with a `c`, like `cMyString`). Properties of a class which are objects have a two letter prefix, *e.g.* an `AxisStetup` property within an `Axis` class will be named `asSetup`.

Underscores must be avoided (they conflict with the *CamelCase* notation), except in the case a property that has a dimension, which (ideally) should be suffixed (*e.g.* `currentWavelength_nm`).

2.3 Matlab compliance

To allow for a unrestricted class capabilities, all of our classes inherit from the MATLAB `handle` class. Further, all the classes inherit from a custom `HandlePlus` class which has a few saving and load methods added to the `handle` class.

3

Documentation

We provide here a description of the main components of met5gui.

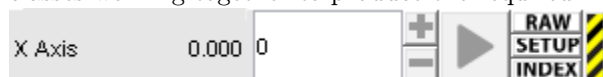
3.1 Clock

`Clock` is the class that coordinates all the actions of the other classes. It is shared by most of the components to allow for an asynchronous operation, *i.e.* to avoid blocking procedures. It registers all the actions that are automated, such as reading updates or scanning procedures. It also allows the user to pause the program, making it easier to troubleshoot specific components.

3.2 Instrument control

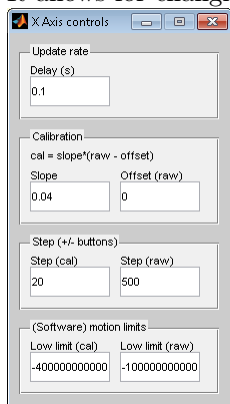
3.2.1 Axis

`Axis!` is the class that It has the capability to operate in a raw or calibrated mode (set through `AxisSetup`), and provides simulation mode through a virtual axis `AxisVirtual`. It is different from `HardwareIO` in the sense that it doesn't command the position of an actual motor, rather the position of an object. Typically, the position input will be fed to a low-level kinematics algorithm, or to a bunch of (undisplayed) `HardwareIO` classes working together to produce the required movement.



AxisSetup

It allows for changing the settings of the `Axis` class display.



AxisWithSave

(what are the specifics of this class?)

AxisAPI

It is a template class to show the capability of `Axis` to perform of compound movement (*i.e.* it contains the kinematic transforms.)

AxisAPITest

Provides the binding between `Axis` and the stage, when the stage has the generic methods listed in section 1.4.1.

(deprecated, used in AIS, will be turned into `APIGeneric`)

AxisVirtual

Simulates the behavior of a motor for testing purposes.

3.2.2 Diode

Diode! is the class that communicate with a sensor, pretty much like `Axis` communicates with a motor. The reading can be a compound of several readings, contrary to `Hardware0`.



DiodeSetup

The equivalent of `AxisSetup`, for a diode.

3.2.3 DiodeAPI

(There is no such class yet, but there should be one, for general coherence.)

DiodeVirtual

The equivalent of `AxisVirtual`, for a diode

3.2.4 HardwareIO

HardwareIO is the class that has the direct control over a stage. It is structurally similar to **Axis** class, except that it doesn't allow for compound movements.

SetupHardwareIO


The equivalent of **AxisSetup**, for a stage motion. (Why is it called **SetupHardwareIO** ?)

3.2.5 HardwareO

HardwareO completes the same role as **HardwareIO** for **Axis**, but for the **Diode** class.

3.2.6 Shutter

Shutter allows to trigger a shutter, *i.e.* it is a countdown timer.

test	10	trigger	
------	----	---------	-----------------------------------------------------------------------------------

ShutterVirtual

It is a virtual shutter, used for software testing and troubleshooting.


3.2.7 Camera

The **Camera** class establishes the communication with a camera, allowing continuous measurements and snapshots, with the ability to change exposure and gain settings.

(It is not fully implemented yet. The *Ptychography* setup currently running serves as an iterator.)

3.2.8 Scan

The **Scan** class allows to scan any number (2D, 3D, 4D, etc.) of any kind of dimension (space (stage), time (shutter), angle, averaging, etc.) in an asynchronous, non-blocking fashion. Its versatility has been tested through the **ScanTest** class (which comprises many examples of scanning procedure).

Scan	Start	Step	Stop				filename
Ptycho X	-40	10	40				

3.2.9 Monitor

The **Monitor** class is pretty much like the scan, but performs continuous measurement in only one dimension (time), pretty much like an oscilloscope. It is very helpful for debugging and troubleshooting procedures, or simply for quick tests performed on an instrument.

3.3 API & wrappers

3.3.1 nPoint

(to be edited)

3.3.2 APIVHardwareIO

(to be edited)

3.3.3 APIHardwareIOOnPoint

(to be edited)

3.3.4 ConfigMotor

(to be edited; what is that, actually?)

3.4 UI elements

All the previous classes uses custom UI elements classes, rather than the MATLAB `uicontrol` class, to enforce coherence within the code and performs some extra functions.

3.4.1 UEdit

It is an editbox that performs data validation and imposes min/max constraints on the entered data.

3.4.2 UIButton

3.4.3 UIList

3.4.4 UIText

3.4.5 UICheckbox

3.4.6 UIToggle

It is similar to a button, but the displayed text/image changes after it is clicked.

3.4.7 UIPopup

3.4.8 UI2DNav

UI2DNav allows to pan and zoom an object.

3.4.9 List

(what is that?)

3.4.10 Panel

(what is that ?)

3.4.11 Utils

It is a set of common purpose functions, *e.g.* correcting the user-unfriendly UI elements of MATLAB or for allowing scrolling iteration in an editbox.

3.5 Controls

3.5.1 Mirror

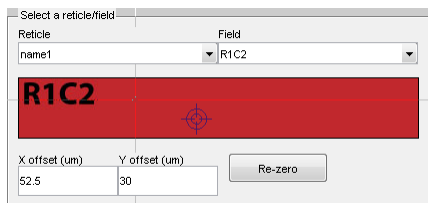
This panel is meant to control the beamline mirrors positions.

(it is not fully implemented yet, for there still are uncertainties on the low-level controls)

3.5.2 ReticlePick

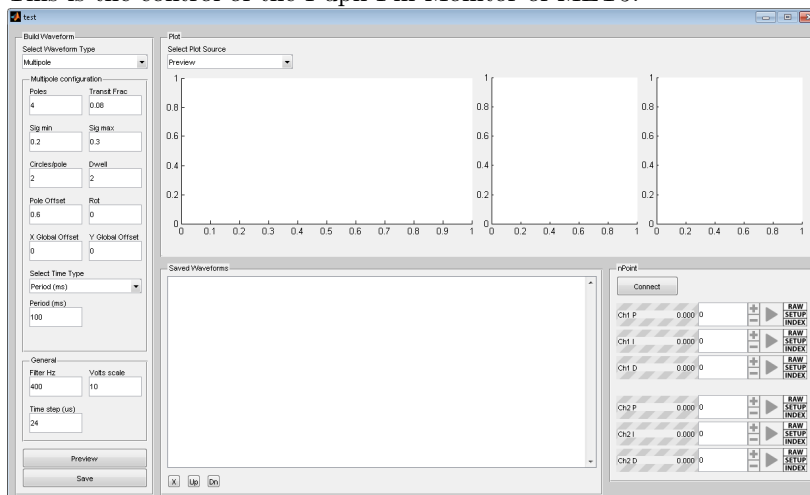
This is the Reticle Pick of the MET5 software, that allow users to chose among different configuration sets.

ReticlePickPanel



3.5.3 PupilFill

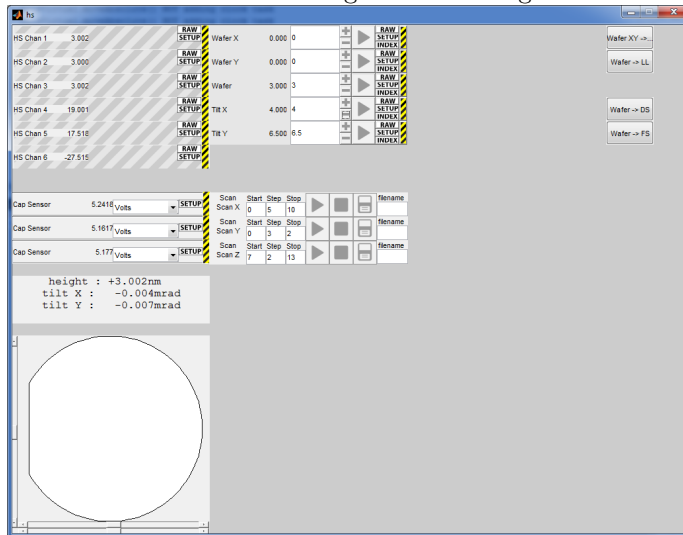
This is the control of the Pupil Fill Monitor of MET5.



PupilFillCore

3.5.4 HeightSensor

This is the control of the Height Sensor integrated to MET5.



HeightSensorCore

3.6 Debugging and examples

There are a few classes that were coded to allow for quick debugging and provide examples of how some the elements are meant to be used :

- Window
- DemoPanel
- ClockTest
- ScanTest

4

Room for improvement

We have successfully overcome many of the issues we've faced. There are still some (non-critical) issues that haven't been tackled yet :

- When using the clock, the debugging procedure is quite difficult. Adding some verbosity to the process would be a great ,
- The clock doesn't know how to handle deleted class (that generally happens when a class is deleted before the clock),
- Clock should have an interface that allows to see what is happening in the current cycle, with maybe the ability to slow down the processes and to log and verbose all automated actions,
- in `UIEdit`, setting the min and max values to $\pm\infty$ is still a problem : Matlab sometimes handles the min/max allowed value symbolically or as number (like `eps`),
- When a `UIEdit` is edited in a modal window, the user has to validate the choice (by hitting *Enter* or clicking in another box). This is cumbersome, since when the user closes the window, he might have the impression that the edit box has been validated, which might be not,
- It would be nice to have a fully coherent layout manager (a component would be placed at the bottom of another),
- Keyboard control and general user-friendliness should be improved, and some automation should be implemented (like "go to previous position").
- the issue of 32 vs. 64 bits controllers has not been carefully examined. It might be a problem if the computers get updated.
- the issue of JAVA 1.6 vs. 1.7 has not been solved. Newer versions of MATLAB use JAVA 1.7, but their might be some broken legacy compatibility.

5

Annex

5.1 Newport ESP300 controller - 3-axes linear stage controller

1. AB abort motion
2. AC set acceleration.
3. AE set e-stop deceleration
4. AF set acceleration feed-forward gain.
5. AG set deceleration.
6. AP abort program
7. AU set maximum acceleration and deceleration.
8. BA set backlash compensation
9. BG assign DIO bits to execute stored programs
10. BK assign DIO bits to inhibit motion
11. BL enable DIO bits to inhibit motion .
12. BM assign DIO bits to notify motion status.
13. BN enable DIO bits to notify motion status.
14. BO set DIO port A, B direction
15. BP assign DIO bits for jog mode
16. BQ enable DIO bits for jog mode
17. CL set closed loop update interval.
18. CO set linear compensation
19. DB set position deadband.
20. DC setup data acquisition
21. DD get data acquisition done status
22. DE enable/disable data acquisition
23. DF get data acquisition sample count
24. DG get acquisition data .
25. DH define home.
26. DL define label
27. DO set dac offset
28. DP read desired position.

29. DV read desired velocity.
30. EO automatic execution on power on.
31. EP enter program mode
32. ES define event action command string.
33. EX execute a program
34. FE set maximum following error threshold
35. FP set position display resolution.
36. FR set encoder full-step resolution
37. GR set master-slave reduction ratio
38. HA set group acceleration
39. HB read list of groups assigned
40. HC move group along an arc
41. xii PrefaceHD set group deceleration
42. HE set group e-stop deceleration
43. HF group off.
44. HJ set group jerk.
45. HL move group along a line
46. HN create new group.
47. HO group on
48. HP read group position
49. HQ wait for group command buffer level
50. HS stop group motion
51. HV set group velocity
52. HW wait for group motion stop.
53. HX delete group
54. HZ read group size
55. ID read stage model and serial number.
56. JH set jog high speed
57. JK set jerk rate
58. JL jump to label .
59. JW set jog low speed
60. KD set derivative gain.
61. KI set integral gain
62. KP set proportional gain
63. KS set saturation level of integral factor.
64. LP list program
65. MD read motion done status
66. MF motor off.
67. MO motor on
68. MT move to hardware travel limit.
69. MV move indefinitely
70. MZ move to nearest index

71. OH set home search high speed.
72. OL set home search low speed
73. OM set home search mode
74. OR search for home
75. PA move to absolute position .
76. PH get hardware status.
77. PR move to relative position
78. QD update motor driversettings
79. QG set gear constant
80. QI set maximum motor current
81. QM set motor type.
82. QP quit program mode
83. QR reduce motor torque.
84. QS set microstep factor
85. QT set tachometer gain
86. QV set average motor voltage .
87. Preface xiii RS reset the controller
88. SB set / get DIO port A, B bit status.
89. SI set master-slave jog velocity update interval
90. SK set master-slave jog velocity scaling coefficients
91. SL set left travel limit.
92. SM save settings to non-volatile memory
93. SN set axis displacement units
94. SR set right travel limit
95. SS define master-slave relationship
96. ST stop motion.
97. SU set encoder resolution
98. TB read error message .
99. TE read error code .
100. TJ set trajectory mode.
101. TP read actual position
102. TS read controller status
103. TV read actual velocity
104. TX read controller activity.
105. UF update servo filter.
106. UH wait for DIO bit high.
107. UL wait for DIO bit low
108. VA set velocity
109. VB set base velocity for step motors
110. VE read controller firmware version
111. VF set velocity feed-forward gain .
112. VU set maximum velocity

- 113. WP wait for position
- 114. WS wait for motion stop
- 115. WT wait
- 116. XM read available memory.
- 117. XX erase program
- 118. ZA set amplifier I/O configuration.
- 119. ZB set feedback configuration.
- 120. ZE set e-stop configuration.
- 121. ZF set following error configuration
- 122. ZH set hardware limit configuration
- 123. ZS set software limit configuration
- 124. ZU get ESP system configuration
- 125. ZZ set system configuration

5.2 SmarAct MCS 3-Axes Piezo motor controller

- GetDLLVersion.
- GetAvailableSystems
- AddSystemToInitSystemsList
- ClearInitSystemsList.
- InitSystems
- ReleaseSystems
- GetNumberOfSystems.
- GetSystemID
- GetNumberOfChannels
- GetChannelType
- SetHCMEnabled.
- Functions for Synchronous Communication.
 1. SetClosedLoopMaxFrequency
 2. SetClosedLoopMoveSpeed
 3. GetClosedLoopMoveSpeed
 4. SetPosition.
 5. SetZeroPosition
 6. GetPhysicalPositionKnown
 7. SetPositionLimit.
 8. GetPositionLimit.
 9. SetAngleLimit.
 10. GetAngleLimit.
 11. SetStepWhileScan.
 12. SetSensorEnabled.
 13. GetSensorEnabled.
 14. SetSensorType.
 15. GetSensorType.

16. SetAccumulateRelativePositions
 17. SetEndEffectorType.
 18. GetEndEffectorType.
 19. SetZeroForce
 20. StepMove.
 21. ScanMoveAbsolute.
 22. ScanMoveRelative.
 23. GotoPositionAbsolute
 24. GotoPositionRelative
 25. GotoAngleAbsolute
 26. GotoAngleRelative
 27. Stop
 28. CalibrateSensor
 29. FindReferenceMark
 30. GotoGripperOpeningAbsolute.
 31. GotoGripperOpeningRelative.
 32. GotoGripperForceAbsolute
 33. GetVoltageLevel
 34. GetPosition.
 35. GetAngle.
 36. GetStatus
 37. GetGripperOpening
 38. GetForce
- Functions for Asynchronous Communication
 1. GetClosedLoopMoveSpeed
 2. GetPhysicalPositionKnown
 3. GetPositionLimit.
 4. GetAngleLimit.
 5. SetReportOnComplete
 6. GetSensorEnabled.
 7. GetSensorType.
 8. GetEndEffectorType.
 9. GetVoltageLevel
 10. GetPosition.
 11. GetAngle.
 12. GetStatus
 13. GetGripperOpening
 14. GetForce.
 15. SetReceiveNotification
 16. ReceiveNextPacket
 17. ReceiveNextPacketIfChannel.
 18. LookAtNextPacket.
 19. DiscardPacket.

5.3 Stanford Research Systems SR830 DSP Lock-In Amplifier

1. OUX i The SR830 OUX i command sets the output interface to RS232 (i=0) or GPIB (i=1).
2. FMODE i The SR530 is always in external reference mode.
3. AX AY AR The AX, AY and AR commands auto offset the X, Y and R outputs.
4. AP The AP command performs the Auto Phase function.
5. B n The SR830 has no bandpass filter.
6. C n Changes the Reference display.
7. D n Change the dynamic reserve. Unlike the SR530, all reserves are allowed at all sensitivities.
8. E m ,n Change the Channel m expand. n=2 selects expand by 100.
9. F x The F command Reads the frequency.
10. G n Change the sensitivity from 10 nV (n=1) to 500 mV (n=24).
11. I n Change the remote/local status. The SR830 Override Remote mode can override the I2 command.
12. L m ,n Change the line notch filter status.
13. M n Change the reference mode to 2f.
14. N m Change the noise bandwidth.
15. OX n ,v OY n ,v OR n ,v Change the X, Y or R offsets.
16. P v Change the reference phase shift. The value of v is limited to -360.0=v=729.99.
17. Q1 Q2 QX QY Read the output values in Volts or degrees.
18. R n Change the reference input mode.
19. S n Change the Output displays.
20. T m ,n Change the time constant.
21. V n Change the value of the SRQ mask.
22. X n ,v Set or query the auxiliary analog ports.
23. Y n Not implemented. Do not use.
24. Z Reset the SR830. The instrument is reset to the SR830 default setup listed in the Operation section.

5.4 GalilAxisVT50

(from CXRO java controller)

1. abortMove()
2. destroy()
3. disable()
4. enable()
5. getAcceleration()
6. getAuxEncoderOffset()
7. getAuxEncoderPosition()
8. getAuxEncoderScale()
9. getAxisUnits()
10. getLowerLimitHard()
11. getLowerLimitSoft()
12. getName()
13. getOffset()

14. `getPosition()`
15. `getPositionRaw()`
16. `getScale()`
17. `getSpeed()`
18. `getSwitches()`
19. `getTarget()`
20. `getTargetRaw()`
21. `getUpperLimitHard()`
22. `getUpperLimitSoft()`
23. `hasAuxEncoder()`
24. `initialize()`
25. `isEnabled()`
26. `isInitialized()`
27. `isLocked()`
28. `isReady()`
29. `isStopped()`
30. `loadConfigs()`
31. `moveAbsolute(double dest)`
32. `moveAbsoluteRaw(double destRaw)`
33. `moveRelative(double dist)`
34. `moveRelativeRaw(double distRaw)`
35. `saveConfigs()`
36. `setAcceleration(double accel)`
37. `setAuxEncoderOffset(double auxEncoderOffset)`
38. `setAuxEncoderPosition(double auxEncoderPosition)`
39. `setAuxEncoderScale(double auxEncoderScale)`
40. `setAxisUnits(java.lang.String axisUnits)`
41. `setLowerLimitSoft(double lowerLimit)`
42. `setOffset(double offset)`
43. `setPosition(double pos)`
44. `setScale(double inScale)`
45. `setSpeed(double speed)`
46. `setTarget(double dest)`
47. `setTargetRaw(double rawDest)`
48. `setUpperLimitSoft(double upperLimit)`
49. `stopMove()`
50. `unlock()`

5.5 Micos PMC100

(from .h, wrapped with CXRO java class; uses queries)

- GetNumDevices(lpNumDevices);- This function is used to get total number of all types of Performax USB modules connected to the PC.
 - GetProductString(dwNumDevices, lpDeviceString, dwOptions); This function is used to get the Performax product string. This function is used to find out Performax USB module product string and its associated index number. Index number starts from 0.
 - ComOpen(dwDeviceNum, pHandle); This function is used to open communication with the Performax USB module and to get communication handle. Index number starts from 0.
 - ComClose(pHandle); This function is used to close communication with the Performax USB module.
 - ComSetTimeouts(dwReadTimeout, DWORD dwWriteTimeout); This function is used to set the communication read and write timeout. Values are in milliseconds.
 - ComSendRecv(pHandle, wBuffer, dwNumBytesToWrite, dwNumBytesToRead, rBuffer); This function is used to send command and get reply. Number of bytes to read and write must be 64 characters.
1. ABORT Immediately stops the motor if in motion. For decelerate stop, use STOP command. This command is used for clearing the StepNLoop error status.
 2. ACC Returns current acceleration value in milliseconds.
 3. ACC=[Value] Sets acceleration value in milliseconds.
 4. DN Returns 4 character device ID.
 5. DN=[Value] Sets 4 character device ID.
 6. DO Returns driver general purpose output status 1 – motor power enabled 0 – motor power disabled 0 1
 7. DO=[0 or 1] Enables (value 1) or disables (value 0) general purpose digital output 0 1
 8. EO Returns driver power enable 1 – motor power enabled 0 – motor power disabled 0 1
 9. EO=[0 or 1] Enables (value 1) or disables (value 0) motor power 0 1
 10. EX Returns current encoder counter value Position value in 32 bit
 11. EX=[value] Sets the current encoder counter value
 12. HSPD Returns High Speed Setting Value in pps 1 6M
 13. HSPD=[Value] Sets High Speed. 1 6M
 14. H+ Homes the motor in positive direction
 15. H- Homes the motor in negative direction
 16. ID Return product ID. PerformaxUSB-EX
 17. J+ Jogs the motor in positive direction
 18. J- Jogs the motor in negative direction
 19. LSPD Returns Low Speed Setting Value in pps 1 6M
 20. LSPD=[Value] Sets Low Speed 1 6M
 21. LT=[0 or 1] Enable or disable position latch feature
 22. LTE Returns latched encoder position Encoder position
 23. LTP Returns latched pulse position Pulse position
 24. LTS Returns latch status. 0 – Latch off 1 – Latch on and waiting for latch trigger 2 – Latch triggered.
 25. MST Returns motor status Bit 0 – constant speed Bit 1 – accelerating Bit 2 – decelerating ...
 26. POL Returns current polarity Bit 0 - Pulse Bit 1 - Dir ..
 27. POL=[value] Sets polarity.
 28. PS Returns current pulse speed

29. PX Returns current position value Position value in 32 bit
30. PX=[value] Sets the current position value
31. SCV Returns the s-curve control 0 or 1 0 1
32. SCV=[0 or 1] Enable or disable s-curve. If disabled, trapezoidal acceleration/deceleration will be used.
33. SL Returns StepNLoop control status 0 – StepNLoop Off 1 – StepNLoop On
34. SL=[0 or 1] Enable or disable StepNLoop control
35. SLA Returns maximum number of StepNLoop control attempt
36. SLA=[value] Sets maximum number of StepNLoop control attempt 1 16
37. SLE Returns StepNLoop correction value.
38. SLE=[value] Sets StepNLoop correction value.
39. SLR Returns StepNLoop ratio value
40. SLR=[factor] Sets StepNLoop ratio value.
41. SLS Returns current status of StepNLoop control -1 – Not applicable 0 – Idle 1 – Moving ...
42. SLT Returns StepNLoop tolerance value
43. SLT=[value] Sets StepNLoop tolerance value.
44. SSPD[Value] Performax on-the-fly speed change.
45. STOP Decelerates to stop the motor if in motion. For immediate stop, use ABORT command.
46. VER Returns current firmware software version number Vxxx
47. X[value] Moves the motor to absolute position value using the HSPD, LSPD, and ACC values. Maximum allowed incremental move amount is 262143.
48. ZH+ Homes the motor in positive direction using the home switch and then Z index encoder channel.
49. ZH- Homes the motor in negative direction using the home switch and then Z index encoder channel.

5.6 Mirrorcle

(from CXRO's DLL header file)

1. Uninit(unsigned long phandle);
2. Init(unsigned long phandle, uint id, float Vmaxin);
3. SendDataStream(unsigned long handle, float* x, float* y, float* dout, int len, int sps, uint loop);
4. GoToXY(unsigned long handle, float x, float y);
5. EnableAmplifier(unsigned long handle, uint state);
6. StartDataStream(unsigned long handle);
7. StopDataStream(unsigned long handle);
8. AbortDataStream(unsigned long handle);
9. ClearDataStream(unsigned long handle, int sps);
10. Home(unsigned long handle);
11. IsInitialized(unsigned long handle);

5.7 Wago

(from CXRO's java wrapper)

1. connect
2. disconnect
3. gettAllInputs
4. getSingleInput

5.8 PaulBox

(from CXRO's DLL header file)

1. init(void);
2. beepTest(long iCount);
3. attach(int, char *argv[]);
4. deattach(void);
5. serverStatus(void);
6. startServer(void);
7. killServer(void);
8. motorMoveto(long box, long mot, long goal);
9. motorLock(long box, long mot, long goal);
10. motorUnlock(long box, long mot);
11. motorRun(long box, long mot, long dir, long sp);
12. motorStop(long box);
13. stopAll();
14. motorAbort(long box);
15. abortAll();
16. motorGetPos(long box, long mot);
17. motorReadPos(long box, long mot);
18. sensorGet(long box, long sens);
19. sensorRead(long box, long sens);
20. getState(long box);
21. getPort(long box);
22. getTol(long box, long mot, long delay);
23. getLim(long box, long mot, long high);
24. getTun(long box, long mot, long wx);
25. getPol(long box, long mot);
26. getFil(long wx);
27. getUpdate();
28. getAvg(long box, long mot);
29. getSensorAvg(long box, long sens);
30. getContUpd();
31. getSmrFlag();
32. getActiveMotor(long box, long mot);
33. getLockedMotor(long box, long mot);
34. getActiveVoltage(long box, long sensor);
35. setUpdate(long update);
36. setPort(long box, unsigned short int port);
37. setTol(long box, long mot, long tol, long delay);
38. setLim(long box, long mot, long low, long high);
39. setTun(long box, long mot, long w1, long w2, ...);
40. setPol(long box, long mot, long pol);

41. setFil(long f1, long f2, long f3, long f4);
42. setAvg(long box, long mot, long avg);
43. setSensorAvg(long box, long sens, long avg);
44. setContUpd(long f1);
45. setSmrFlag(long f1);
46. setActiveMotor(long box, long mot, bool value);
47. setActiveVoltage(long box, long sensor, bool value);
48. autoTun(long box, long mot, long method);
49. autoPol(long box, long mot);