
Inspector

Release 0.01.00

AI and MPIbpC

Nov 09, 2019

Contents

1	Introduction	3
1.1	Features	3
1.2	Current state of Documentation	8
1.3	Citing Inspector	8
2	Quickstart	9
2.1	Installation	9
2.2	Adding devices	10
2.3	Adjust the hardware settings	10
2.4	Configuring a measurement	10
2.5	Measuring with just one NiDAQmx card	11
2.6	Analyzing data	11
3	Short Tutorial	13
3.1	Getting Started	13
3.2	Configuration Dir	13
3.3	Loading Data	13
3.4	Displaying Data	13
3.4.1	Magnifying Glass	13
3.4.2	Shortcuts	14
3.5	Graphs	14
3.6	Drag, Drop, Cut and Paste	14
3.7	The Change Stack Size Dialog	14
3.8	Export Data	14
4	Installing Inspector	17
4.1	Setting up the MPD panel	17
4.1.1	Install the drivers	18
4.1.2	Run MpdCtrl.exe	19
4.1.3	Configure the panel	20
4.1.4	Remarks	21
5	Graphical User Interface	23
5.1	Color maps	23
5.1.1	Changing the Colormap	23
5.1.2	Copy Colormaps	23
5.1.3	Locking the Minimum/Maximum Gray Values of the Colormap	25

5.1.4	Linear and Logarithmic Representation of the Colormap	25
5.1.5	Adjusting the Colormap Values	25
5.1.6	Custom Colormaps	26
5.2	Toolbars	26
5.2.1	Display toolbar	26
5.2.2	Files toolbar	27
5.2.3	Measurement toolbar	27
5.2.4	Zoom toolbar	28
6	Handling Data with Inspector	29
6.1	Importing Data into Inspector	29
6.2	Data Inspection	29
6.3	Data Analysis Functions	29
6.3.1	The Arithmetics Plugin	29
6.3.1.1	The Function Parser	30
6.3.1.2	Summary	33
6.3.2	Examples for Arithmetics Plugin	33
6.3.2.1	Thresholding an Image (one sided thresholding)	33
6.3.2.2	Thresholding an Image (two sided thresholding)	33
6.3.2.3	Creation of an Image with Poisson noise (no structure)	33
6.3.2.4	Creation of an Image with Poisson noise on an existing structure	33
6.3.2.5	Scale stack content about a factor a without changing the stack size	33
6.3.2.6	Rotate a 2D stack by angle	34
6.3.2.7	Rotate a 2D-stack and produce a rotational symmetric 3D-stack out of it	34
6.3.2.8	Calculate a 2D Gaussian peak at the center	34
6.3.3	Interpolation	34
6.3.4	Smoothing	35
6.4	Image Deconvolution	35
6.4.1	Creating an idealized PSF	36
6.4.2	Convolution	38
6.4.3	Point Deconvolution	39
6.4.4	Wiener Filtering	39
6.4.5	Richardson-Lucy	39
7	Driving Hardware with Inspector	43
7.1	The Measurement Process	43
7.2	Hardware Configuration	43
7.3	Measurement Templates	43
7.3.1	Use of Existing Measurements as Templates	44
7.4	Live Dialogs	44
7.4.1	Experimental Control Live Dialog	44
7.4.2	Hardware Acquisition Timing Live Dialog	46
7.4.3	Lasers and Channels Live Dialog	46
7.4.4	Microscope Control Live Dialogs	48
7.4.5	Time Lapse Live Dialog	49
7.5	DAC Scanning Channels	50
7.6	TTL Channels	52
7.6.1	Wait times	53
7.6.2	Current settings	53
7.6.3	Channel action settings	53
7.6.4	Synced axis behaviour	54
7.6.5	Live Dialog	54
7.7	Generic Drivers	54
7.7.1	Com Driver	54

7.7.1.1	What does wait/line mean?	55
7.7.1.2	What are Consts?	55
7.7.2	SyncDriver	55
7.7.2.1	Dummy Axes	55
7.7.2.2	Passive Syncing	55
7.7.2.3	Active Syncing	55
7.7.3	Timer Driver	55
7.8	Camera Drivers	55
7.8.1	SpecCam Based Camera Drivers	55
7.8.1.1	Common Settings and Functions	55
7.8.1.2	Specifics of some Frequently Used Drivers	56
7.8.2	Deprecated non-standard Camera Drivers	56
7.8.2.1	Apogee Drive	56
7.9	NiDAQmx Cards	57
7.10	TCSPC hardware	58
7.10.1	Introduction	58
7.10.2	Supported Devices	58
7.10.3	Operating Principle	58
7.10.4	Settings to be adjusted	59
7.11	Becker&Hickl SPCM Cards (without AI FPGA scanner)	61
7.11.1	Configuring the Card	61
7.11.2	Configuring Measurements	62
7.11.2.1	Multi-channel measurements	63
7.11.2.2	Global Timegates	63
7.11.2.3	Channel-specific Timegates	63
7.11.3	FIFO Measurements	64
7.11.4	Live Dialogs	64
7.11.5	Rate monitor	64
8	Inspector Python Interface	67
8.1	Setup	67
8.2	Start	67
8.3	Interface	68
8.3.1	SpecPy constants	68
8.3.2	Inspector	68
8.3.3	Measurement	70
8.3.4	Configuration	71
8.3.5	File	72
8.3.6	Stack	73
8.4	Examples	75
8.4.1	Python Interface Examples	75
8.4.1.1	Hello Inspector Example	75
8.4.1.2	Data Analysis Example	76
8.4.1.3	Measurement Example	77
9	Inspector Matlab Interface	81
10	Utility applications	83
10.1	Direct Port Control	83
10.2	MPD Panel Control	83
11	HOWTOs	85
12	Shortcuts	87

13 Contributions to the Documentation	89
13.1 Correct Mistakes	89
13.2 Committing	89
14 FAQ	91
14.1 Errors During Startup	91
14.1.1 Inspector does not start because the ‘application configuration’ is incorrect.	91
14.1.2 Registering Hardware Ports takes too long.	91
14.2 Hardware Initialization	92
14.2.1 COM ports beyond COM9 cannot be opened.	92
14.3 Measurement Configuration	93
14.3.1 Is it possible to time two (or more) configurations in Inspector so that one automatically starts directly after the other?	93
15 File Formats	95
15.1 The Inspector MSR File Format	95
15.2 The OBF File Format	95
15.3 The DBL File Format (<i>deprecated</i>)	102
15.4 The OmasIo API, Bindings	103
15.5 Meta information data model	103
16 Developing Code for Inspector	105
16.1 A little KB	105
16.1.1 SpecDll versions, revisions and module revisions	105
16.1.2 Exception Handling and Crash Reporting	107

Contents:

Inspector is a robust software system for experimental control and quantitative data analysis in microscopy and spectroscopy. Integration of data processing and acquisition allows real-time analysis and visualization of experimental results.

1.1 Features

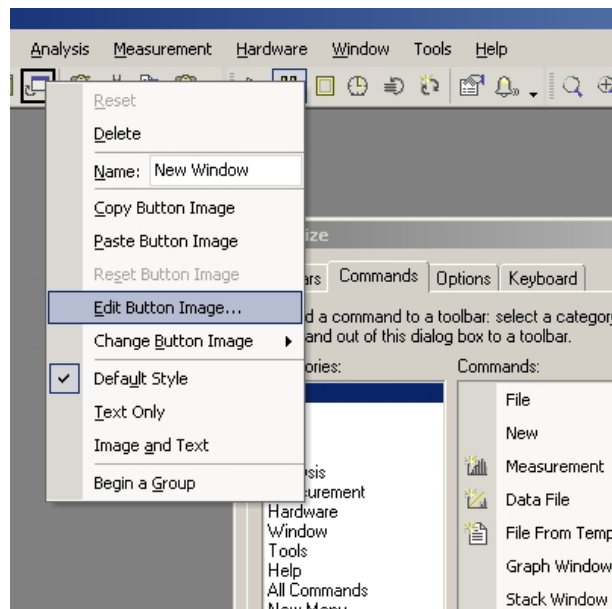


Fig. 1: Customization of toolbars, menus and shortcuts.

Inspector offers a variety of functions for speed-optimized visualization of up to 4-dimensional data as graphs and pictures, an intuitive user interface and access to analysis methods. Data can be graphically cropped, moved, copied

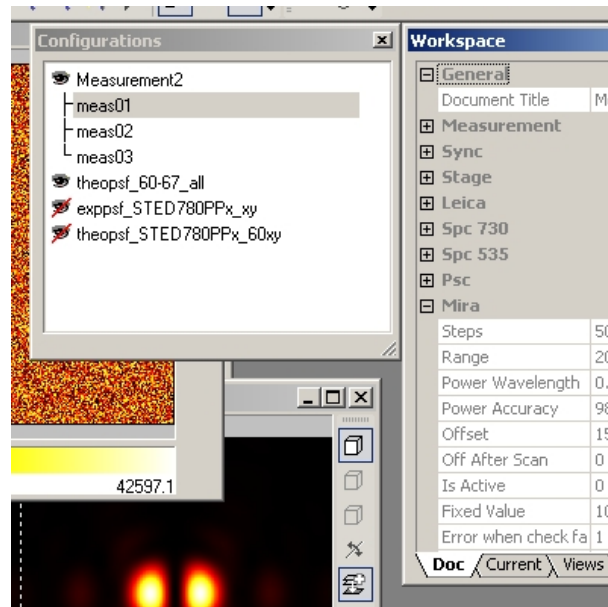


Fig. 2: The workspace allows you to keep track of all your measurements.

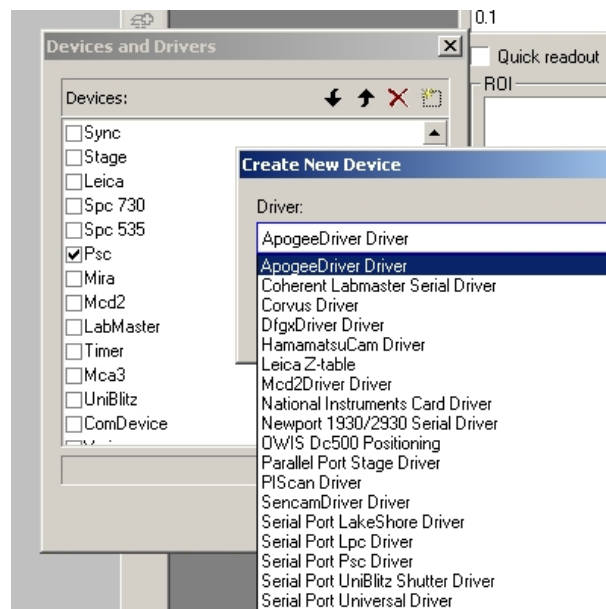


Fig. 3: Add new devices derived from standard or custom drivers.

and combined in overlays. In addition to ordinary cuts, zooming, multi-channel display and user-defined color tables, many tools are provided that are especially useful in quantitative microscopy. There are tools for the calculation of point-spread functions and simple linear de-convolution, frequency filtering etc. A tool for off-line nonlinear de-convolution is also included.

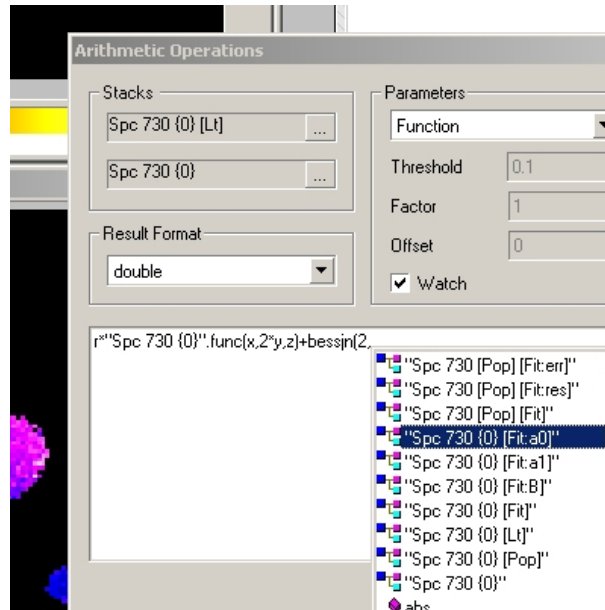


Fig. 4: Use the built-in parser to analyse and process your data.

A built-in function parser allows for user-defined filters, transformations and other numerical operations on the data and as part of the 'FitPlugin' nonlinear parameter fitting with user-defined functions and a choice of several optimized algorithms.

Inspector imports and exports many commonly used data formats and communicates with other applications as e.g. Origin, Photoshop, Freehand and Excel through cut- and paste operations.

Hardware drivers exist for analog output devices, multi-channel scales, time correlated single photon counting (TC-SPC), various CCD cameras (e.g. Apogee, Hamamatsu, PCO), laser power controllers, positioning stages (PIFOCs etc.).

Adding the ability to control new hardware components is straight-forward and achieved through a Plugin structure. Such hardware drivers can provide dialogs for hard- and software-specific settings and parameter adjustment during measurements. Data readout can be synchronous or asynchronous, the program handles the measurement flow, synchronization of different devices and the coordination of data readout, analysis and visualization during the measurement.

The program administers the settings defined by the Hardware drivers and allows the creation of template measurements including embedded analysis and visualization. Measurements can therefore be repeated at any time with identical settings by pressing a single button. Experimental data is always saved together with all relevant settings for later reference.

All data dependencies are remembered by the program so if data changes during a measurement or manual processing all necessary steps to update dependent data are repeated automatically. Inspector will even remember dependencies on data saved on disk and can 'watch' these files and re-load them if necessary. It can therefore serve as a graphical front-end for your command-line numerical analysis tools. All such dependencies, as well as settings and window positions are conserved when saving and re-loading documents.

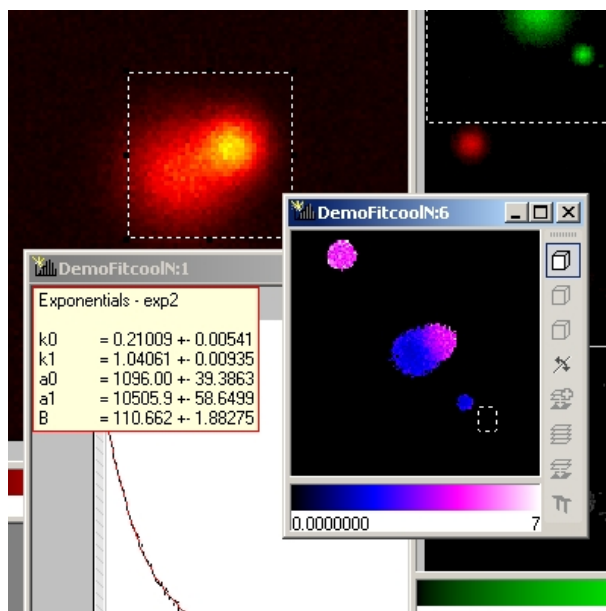


Fig. 5: Nonlinear fitting of single data curves or e.g. along the time axis of a TCSPC stack in each spatial point.

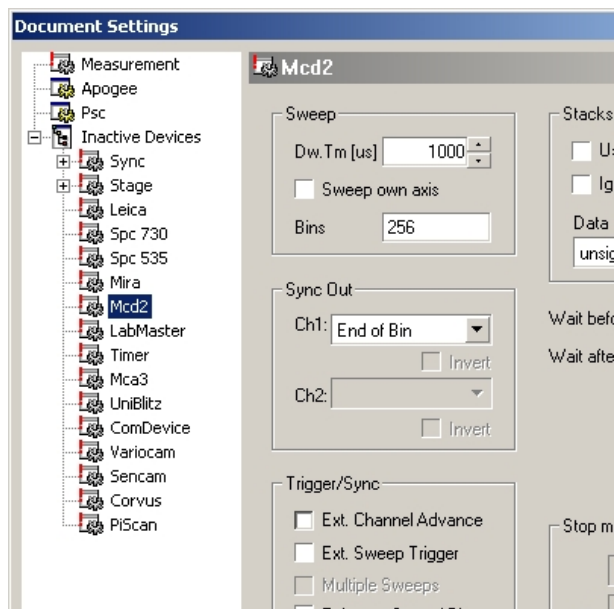


Fig. 6: An intuitive GUI allows you to adjust your measurement parameters.

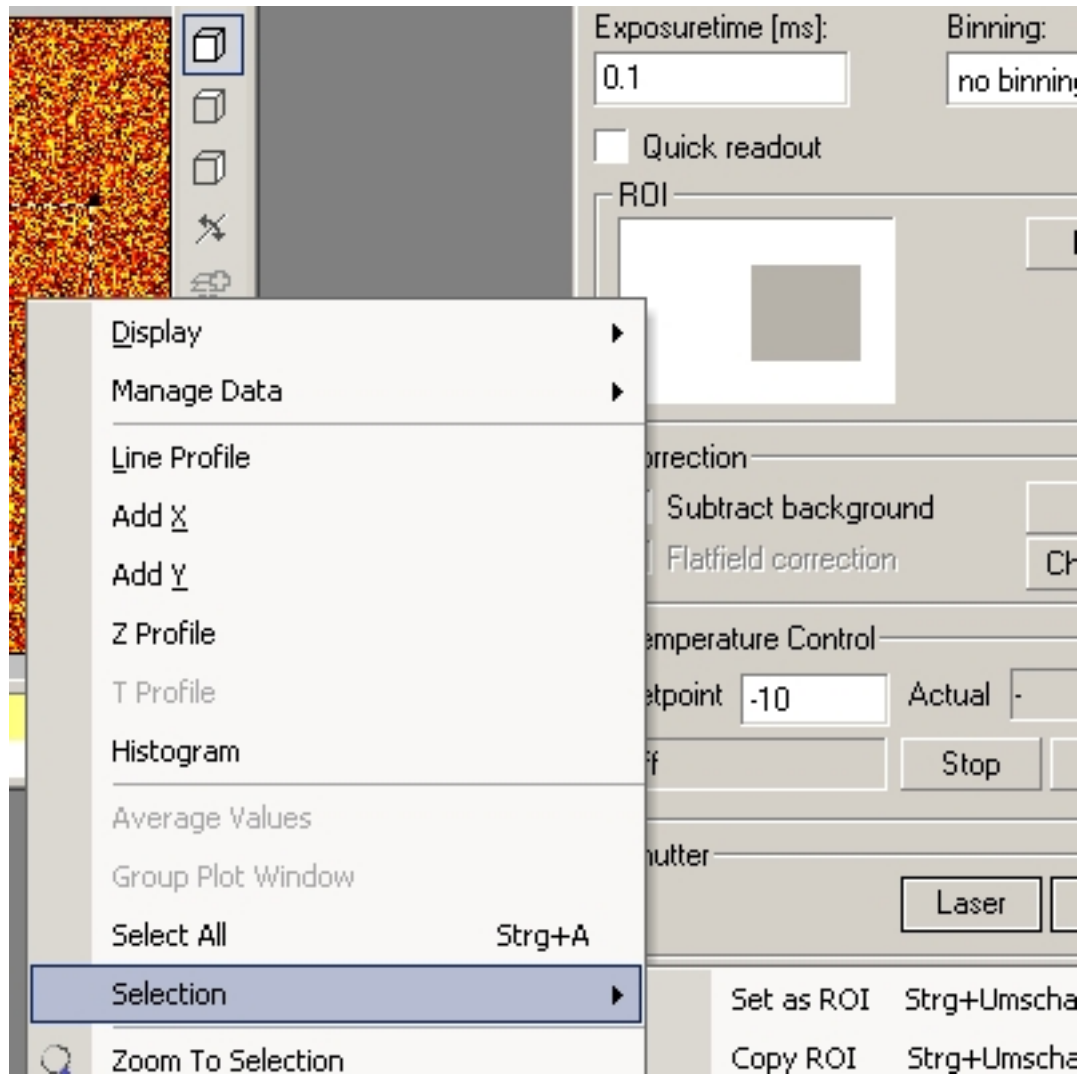


Fig. 7: All measurement parameters are remembered. So you can derive measurements from previous ones, image regions of interest etc.

1.2 Current state of Documentation

Many features of Inspector were inspired by its users in the department of NanoBiophotonics at the Max Planck Institute for biophysical Chemistry.

Not all of them found their way into this documentation as of now. Also, Inspector is under constant development, so features might be added or functionality might be replaced by superior implementations. Thus some of the information presented here might be outdated. Anybody is encouraged to explore the context-menus - a lot of the functionality is intuitive enough to figure it out by trial and error.

Inspector has been originally developed in the department of NanoBiophotonics at the Max Planck Institute for biophysical Chemistry in Göttingen, Germany.

Inspector was always and is currently lacking appropriate documentation. All users of Inspector are welcome to contribute. The source format of this manual is *restructured text* and we are using *Sphinx* to create the manual from it. You may send manual pages or sections in any format and they will be incorporated.

1.3 Citing Inspector

If you are using Inspector for your data acquisition or analysis and would like to cite the program or its documentation please use the following reference:

Schönle A., 2006. *Inspector Image Acquisition & Analysis Software*, v0.1
<http://www.inspector.de>

Inspector contains a few data analysis plugins and drivers for

- VidCap (for most webcams)
- SimCam (a simulated camera for testing)
- SyncDriver (provides dummy axes, allows synchronization via serial port)
- ComDriver (a generic driver for com/gpib devices with a simple protocol)
- Timer (for time-lapse type measurements)
- Becker&Hickl SPCM cards
- National instruments DAQ cards (through the NiDAQmx drivers) which control scanners, shutters and read out detectors
- Scanners from PI
- Various CCD cameras

Warning: Drivers are simply dll's that Interface Inspector to the drivers you install with your hardware. They have to be compatible with both, the version of Inspector used and the version of the hardware drivers and support dll's installed on your system. Always download all necessary drivers together with a new version of Inspector. And make sure you have updated your hardware drivers.

2.1 Installation

For simply running Inspector (e.g. for data analysis) all you have to do is to extract the zip Archive into a directory on your computer. Inspector will start and ask you for configuration directory. If this is the first time you use Inspector, create one and select it. Otherwise point Inspector to your existing directory.

Note: Depending on your OS configuration you may have to install some additional libraries. Please see the [program start chapter in the FAQ](#) if you encounter errors during startup.

The configuration directory can be chosen independently for each user on each computer Inspector is run. The directory itself can contain computer-specific configurations and also contains custom color maps, fit functions, formulas created by the user. Therefore it is reasonable to use one directory per user which is accessible from (or synchronized between) computers. You can change its location at any time by copying/moving it and directing Inspector to the new location through *Edit* → *Preferences* → *Configuration Directory*.

There are other useful options in this menu, too. To avoid an error message upon startup, explicitly disable logging for now.

Warning: You should regularly back up your configuration directory as settings, custom colormaps etc. can take some effort to re-create if you lose them. Also a regular backup allows you to reconstruct a working configuration if, for some reason your hardware does not behave as expected any more and you are unsure what you changed.

2.2 Adding devices

Go to *Hardware* → *Add/Remove Devices* to add devices and then click on the add button of the list box. You get a dialog with a combo list. Choose the device type. you will be asked a name. Choose one that is recognized by you and other potential users and keep in mind you may want to add more than one 'Camera' or 'Scanner'.

Note: Inspector can be extended by custom drivers compiled against the SDK. This requires knowledge of C++, the MFC and an installation of the Visual Studio 2008 (currently)

2.3 Adjust the hardware settings

Hardware settings are set in *Hardware* → *Configure*. Usually you will have to enter an identifier for the device. After adjusting the settings press initialize to see whether the device was found and can be configured using your chosen settings.

Note: The device id for the NI card is a string, e.g. "Dev1" and can be found in the measurement & automation explorer. For NIDAQ cards you should restart Inspector right after initializing it for the first time and saving the hardware settings because the the number of available channels (and thus the configuration of the settings dialogs) depends on the device model. The μm → Volts mapping is done in the Hardware settings of the NiCard after the restart (extra page with a channel selector and the possibility to set min and max Volts and the corresponding area your scanner will do).

2.4 Configuring a measurement

Open a document and go to *Measurement* → *Edit Settings*. This lets you configure parameters that define your measurement. You can have many measurements (i.e. documents) open at the same time and start them in turn to switch between different settings.

Most property pages are specific to the devices you configured in the hardware settings, only the first page configures the measurement itself, i.e. selects which axes will be scanned and whether this is controlled automatically by trigger signals or through Inspector.

‘Sync first axis’ in the first page means that the first axis is controlled by hardware. The computer assumes that a pixel sync is shared by the devices (but has no way of checking it). The devices have to be configured such that exactly one will be responsible for creating the sync pulses (and will tell the framework that it does) [e.g. when you enable ‘Create sync pulses’ in the NiDAQ card]

Exactly one device is responsible for ‘waiting’ until the axis sync has finished during each measurement stack. Currently this is always the same device that also creates the sync pulses.

All other devices have to acknowledge that they can deal with the synced axis and must be configured to listen to the sync pulses (and tell the framework about it).

2.5 Measuring with just one NiDAQmx card

Choose an appropriate ‘sync out’ for the NI card and ‘disabled’ for sync in and set the dwell time in the ‘DACs’ configuration page. Also set the ‘Create Sync pulses’ option in the DACs configuration

Select all other settings to your liking. (e.g. whether you want to measure histograms or not in the SPCM, AI channels in the NI card, have one or two counter inputs etc. Please roam the config dialogs and tell me what does and what doesn’t make sense to you).

Select at least one analog input or a CNTR input if only using the NiDAQ card. When starting the measurement, a stack should then pop up.

If you need to configure the TTL outputs of the NiDAQ card, please check back with us.

2.6 Analyzing data

All analysis functions are accessible through context menus. Right click on a stack, graph, axis, color map for those. For most of the frequently used functions there are toolbar buttons.

Please write an email with as much detail of what you intended to do and what you already tried if there is trouble.

This is a short tutorial that takes you on a tour of some of the more important features of Inspector. If this is the first time you use the program, it is a good idea to follow it step by step using the test data that should have been provided to you with the program.

3.1 Getting Started

3.2 Configuration Dir

Todo: Empty

3.3 Loading Data

Todo: Empty

3.4 Displaying Data

3.4.1 Magnifying Glass

If you hold down the `shift` key and drag with the left mouse button in a window a magnifying glass is displayed. `shift + alt` keys display an even bigger magnifying glass.

When you are pressing the `shift` key AFTER starting to drag it will force a vertical or horizontal line when selecting a line profile.

3.4.2 Shortcuts

- `F9 / F10` Fit maximum/minimum. In a stack view this adjusts the colormap max/min to the maximum/minimum of the current selection, in a graph view the y-axis scale to the maximum/minimum value between the slider bars.
- `Page Up / Down` Go up/down one slice along the third (z-) axis. Also pressing the `shift` key moves 10, pressing `shift + alt` moves 100 slices along the z-axis.
- `ctrl + Page Up / Down` Go up/down one layer along the fourth (hidden) axis. Also pressing the `shift` key moves 10, pressing `shift + alt` moves 100 layers along the hidden axis.

3.5 Graphs

Todo: Empty

3.6 Drag, Drop, Cut and Paste

Graphs, image stacks and color maps can be dragged and dropped between windows. This is done by pressing the `ctrl` key while dragging with the left mouse button. Color maps can be dragged onto stacks which will then be displayed using the dropped colormap. For data the following rules apply

- `ctrl` Copy all slices of the current selection to the new window. If no rectangle is selected the whole stack is copied.
- `ctrl + shift` Copy only the current slice of the current selection to the new window
- `ctrl + alt` Do not copy any data. Open a new view of the data in the new window

In the graph window the selection is determined by the vertical bars you can drag in from the border (they turn red when they are in use), NOT the rectangle selection. For 4d stacks the following additional rule applies:

In add-up and maximum intensity projection mode all layers (along the hidden axis) are copied. In parse-through mode only the current layer is copied.

3.7 The Change Stack Size Dialog

This dialog allows you to change the physical size, offset and pixel dimensions of the stack as well as its data type. It can be accessed using the button at the side of the image or via the shortcut `ctrl + t`.

3.8 Export Data

Data from Inspector measurements can be exported into several file formats:

- Colormap Tiff files/stacks (.tif/.tiff)
- RGB Tiff files/stacks (.tif)
- binary double files (.dbl)
- Avi files/movies (.avi)

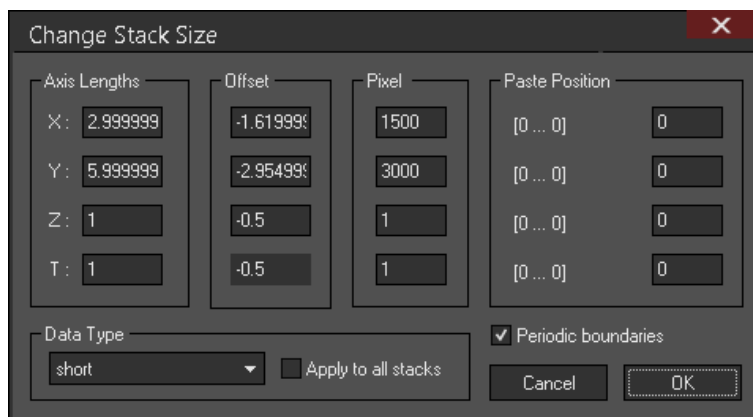


Fig. 1: Change Stack Size dialog.

- Visualization toolkit files (.vtk)
- MRC files (.st, .map, .ccp4, .mrc)
- ASCII data files (.dat, .asc)
- Becker&Hickl data files (.sdt)

To export data select the data Stack and select *File* → *Export* or use the shortcut `ctrl + e` to open the Export Data dialog.

Installing Inspector

If this is your first time setting up Inspector please refer to the section *Quickstart* for installation instructions. The basics of setting up the configuration directory are covered in the *Configuration Dir* chapter of the *Short Tutorial*.

This chapter contains some more advanced aspects of setting up your measurement environment in Inspector and how to set up and test your measurement-independent hardware.

4.1 Setting up the MPD panel

The MPD panel is a custom made USB device that allows you to control the origin of scan axes. Support for controlling other parameters is pending.



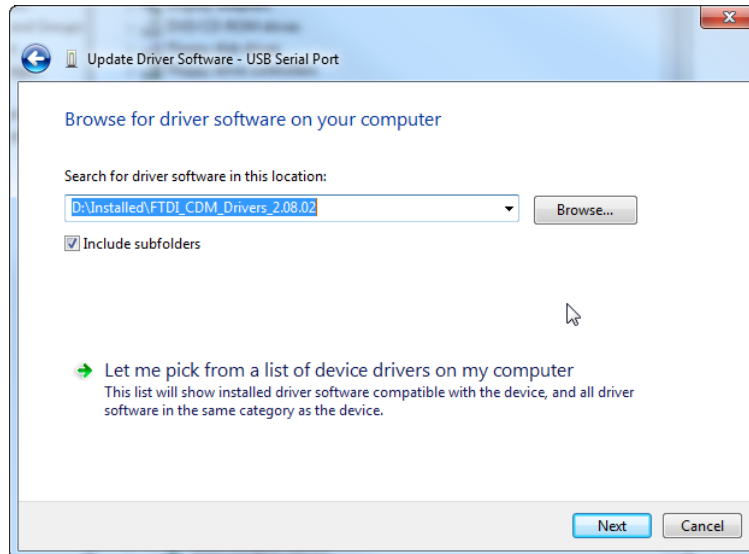
Currently the MPD panel is not available to parties outside the MPI. If you are interested in using it, please contact the support team and let us know. If there is enough interest we may try to find a solution to this problem.

4.1.1 Install the drivers

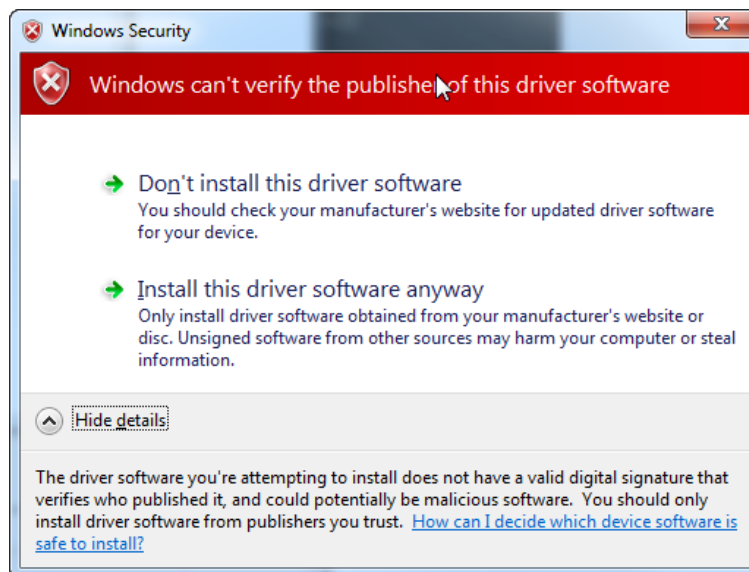
Download the [modified FTDI drivers](#) from the Inspector website.

Note: As a side remark these are just the standard drivers found at [FTDI here](#) with modified INF files to fit the IDs of our MPD panel. So in case the above version is incompatible with another FTDI device you can create a package based on another version of the FTDI drivers by modifying the INF files analogously.

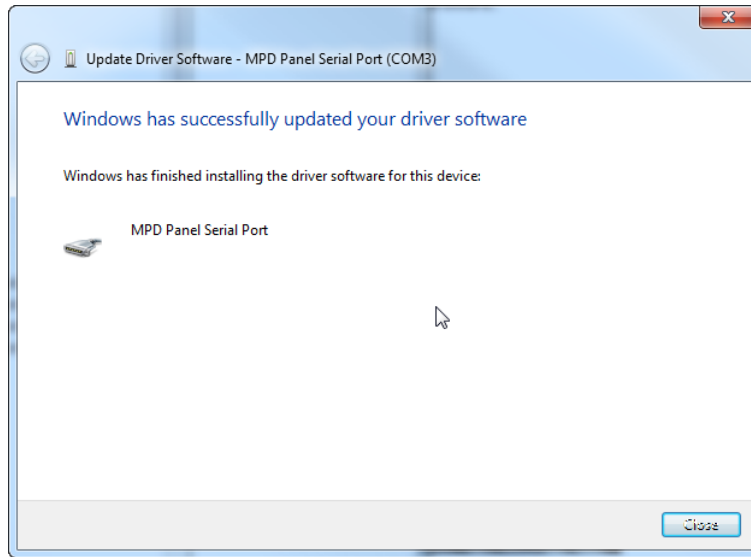
Unpack the drivers to your local hard drive, connect the panel. When asked you should choose to manually point windows to the location of the drivers. Depending on the windows version this will look something like this:



where the path is the location of the unpacked driver files. You may have to do this twice, once for a 'USB to Serial converter' and once for a 'USB Serial Port'. You may be warned that the driver is a security risk:

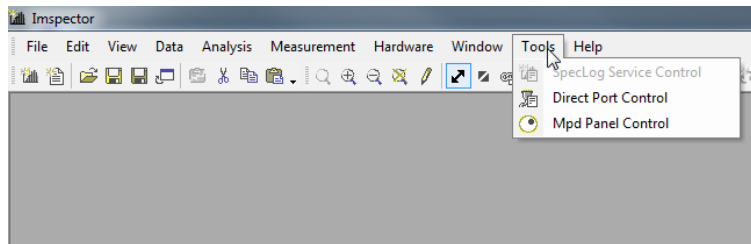


In case installation just fails without asking you to provide drivers, go to the device manager, find the new device not running properly, right click and select 'Update Driver'. Then continue as above. Again you may have to do this twice. Eventually you should be successful:



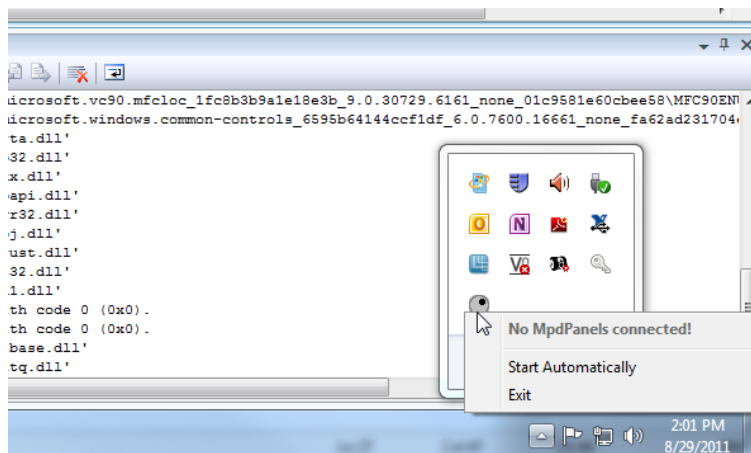
4.1.2 Run MpdCtrl.exe

Currently the MPD panel is controlled by a separate application, MpdCtrl.exe which manages access to the panel(s) for different programs (we internally have some other applications that can share the panel with Inspector). You can start MpdCtrl.exe directly from the installation directory of Inspector or through the tools menu:

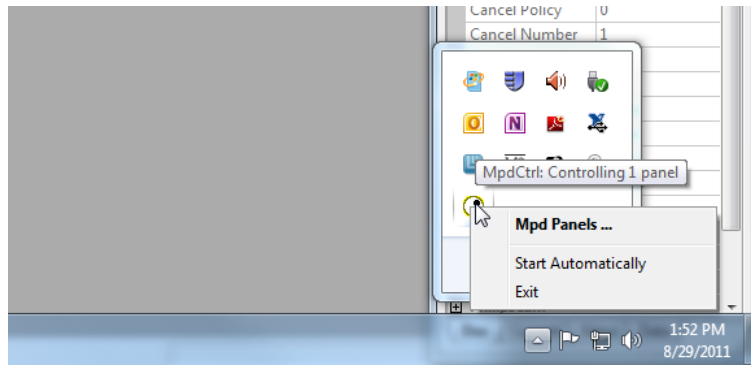


If the entry is greyed out this is either a glitch in Inspector (try to start it directly) or it is missing from your installation. In this case make sure you did not accidentally delete it and then contact support.

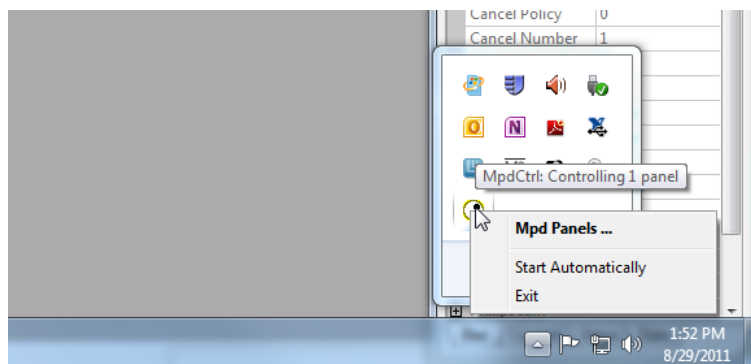
When MpcCtrl starts it creates a tray icon which looks either like this:



if either the panel is not connected or the driver is not correctly installed or like this:



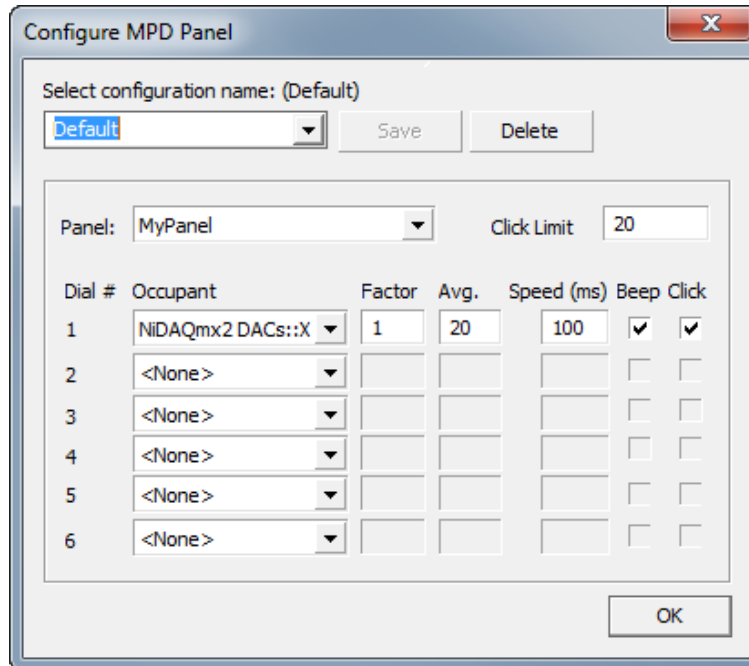
if everything is just fine. You can start the app which does nothing except allowing you to rename a panel (if you use more than one this is an important feature) and to show you which dials are in use by which app. Through the tray icon you can also tell MpdCtrl.exe to start up automatically when you log on.



Warning: It is important that the MPDCtrl.exe running is the same version as the Inspector executable accessing it. This is due to bad software design but will not be changes soon. If there is any trouble with the MPD panel, exit MPDCtrl.exe through its tray icon and restart it from the correct directory.

4.1.3 Configure the panel

You configure the panel from the Inspector main menu. Go to *Edit -> Preferences -> Configure MPD Panel*. You will see the following dialog provided that MpcCtrl.exe is running and has recognized a panel.



You can load previous configurations by choosing them in the 'Select configuration name' combo box. You can save the current configuration by typing a (new) name in the combo box (or choosing an old one to overwrite) and pressing 'Save'. Delete configurations by selecting them and pressing 'Delete'. To modify configurations you edit the dial entries for all connected panels. The values to configure are:

Click Limit The panel generates a clicking sound when the value is changed (on every step). If the time between value changes goes below the entered value in ms the clicking is suppressed. (i.e. the higher the value, the earlier this happens when rotating a dial fast). This is to avoid a high-pitched sound when moving values fast which will quickly annoy your colleagues in the lab.

Occupant The value to assign to a dial

Factor The dial's response is nonlinear, i.e. when advancing N 'clicks' the value will be advanced by $N \cdot (1 + \text{Factor} \cdot \text{Speed})$ where speed is the speed at which the dial is rotated and Factor is the one you set here.

Average The speed is determined as the minimum speed registered during the last 'Average' clicks. Average is thus mis-labeled but that has no practical consequences. It just turned out to work better this way.

Speed While you can vary values using the panel at very high speeds it is not always advisable to update a scanning parameter at very high rates. The speed in ms given here tells Inspector how often it should test the (modified) value and adjust the measurement process or move the stage depending on the internal configuration.

Click Whether or not to issue the clicking sound for each movement.

Beep Whether or not to beep (system beep from the computer, will not be audible if the computer is muted) when hitting the minimum or maximum values.

4.1.4 Remarks

Note: While all parameters directly related to the MPD panel are set here, the size of a single step (i.e. the movement during a single click) is configured through the hardware parameters of the axis in question. Go to [Hardware](#) → [Configure](#) and select the appropriate page (usually the 'DACs' subpage of a scanning device). For the NIDAQ driver the resolution is determined by the 'Resolution' parameter in logical units. A negative value will use a reasonable default

which may, however, be too coarse for you. Other devices may be configured differently, refer to their documentation for details.

Warning: The FTDI chip is used by many devices. All of them will eventually share one dll and driver located in the windows system directory. If installing the panel stops other hardware from working or vice versa, this is most probably due to incompatibilities of some older software with the newly installed drivers. Get someone with a thorough knowledge of such matters. MpcCtrl.exe should work with most version of the drivers - so all you have to do is to make sure that the dlls ('ftd2xx.dll' for 32bit Imspector and 'ftd2xx64.dll' for 64 bit Imspector) Imspector finds in the path and loads (dynamically) match the installed drivers.

Graphical User Interface

The graphical user interface of Inspector contains some basic elements, that will be explained in the following (*General layout of the Inspector graphical user interface.*). Each measurement may contain one or several measurement windows shown here in the image view (left) and the graph view (middle). The imaging settings are defined and shown in the live dialogs shown on the right (*General layout of the Inspector graphical user interface.*). Furthermore several drop down menus are available. The live dialogs and measurement windows can be arranged freely in the program workspace. The arrangement can be saved together with the hardware settings or as a workspace. The menu bar can be modified and saved.

5.1 Color maps

In Inspector the colormap 'Fire' is typically selected (*'Fire' colormap.*). Furthermore a variety of built-in colormaps can be chosen (*Built-in colormaps.*) The active colormap is displayed at the bottom of the measurement window. On the left and right bottom of the colormap, the minimum and maximum gray value is given. In a stack view the colormap is adjusted to the maximum/minimum of the current selection.

Properties of the colormap can be changed using right-click context menu. To access this menu the active colormap has to be right-clicked.

5.1.1 Changing the Colormap

Enter colormap right-click context menu. To change the colormap choose the first entry of the menu 'Colormap' and simply select the colormap of choice.

5.1.2 Copy Colormaps

Colormaps can be copied between measurements using standard Windows shortcuts.

1. The colormap has to be selected using left-click and is copied using `ctrl + c`. Then insert the copied colormap into the new image using `ctrl + v`

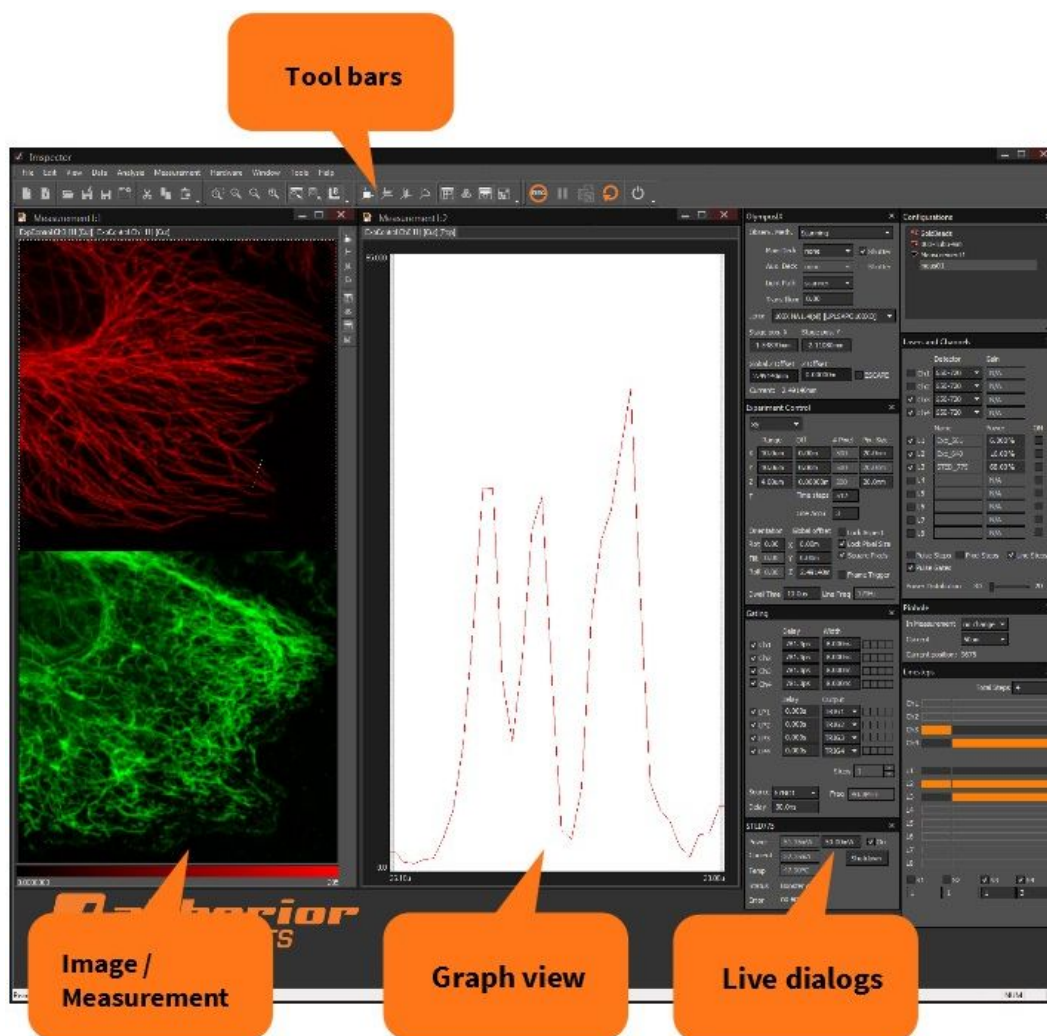


Fig. 1: General layout of the Inspector graphical user interface.



Fig. 2: 'Fire' colormap.

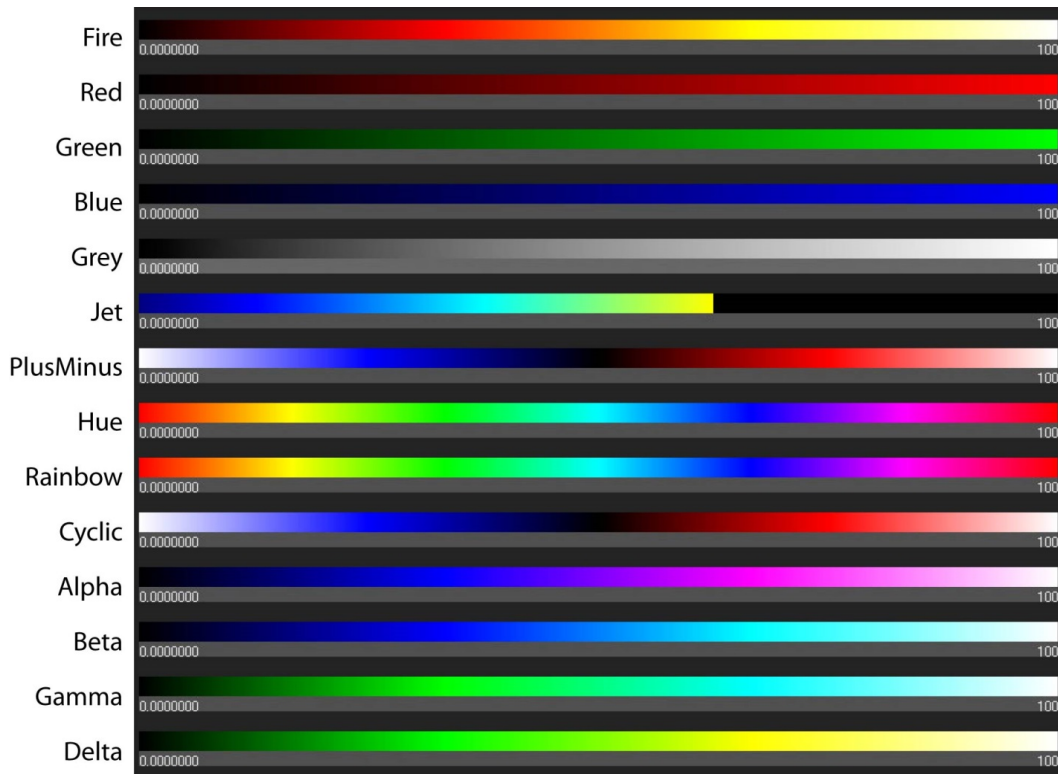


Fig. 3: Built-in colormaps.

- Alternatively, the colormap can be copied using drag and drop. To this end, the active colormap has to be selected using `ctrl + left-click`, drag to the new image and drop it.

5.1.3 Locking the Minimum/Maximum Gray Values of the Colormap

Enter colormap right-click context menu. To fix the gray values of the colormap, choose 'Lock' and select if you like to fix the maximum or the minimum value.

Note: This option may be particularly helpful during image acquisition.

5.1.4 Linear and Logarithmic Representation of the Colormap

In Inspector gray values can be represented in a linear or logarithmic mode. Typically the colormap is set to a linear mode. To change the colormap mode to logarithmic scale, please enter colormap right-click context menu and click 'Logarithmic'. If the logarithmic mode is active, a tick is shown in front of this entry.

5.1.5 Adjusting the Colormap Values

The displayed minimum and maximum gray values can be adjusted by different means

- The respective value can be accessed and changed after double-clicking the value at the bottom of the colormap.

2. The values can be adjusted to the minimum/maximum gray value of the image or a selected region in the colormap right-click context menu.
3. The values can be adjusted to the minimum/maximum gray value of the image or a selected region by pressing F9 / F10. (F9: maximum value; F10: minimum value).

Note: The colormap must be visible at the bottom to do this.

5.1.6 Custom Colormaps

In addition, new colormaps can be created using a colormap editor (*'Colormap editor' dialog.*). To create a custom colormap, the colormap editor has to be opened. To this end, the colormap context has to be opened by right-clicking the active colormap. Then select 'New'.

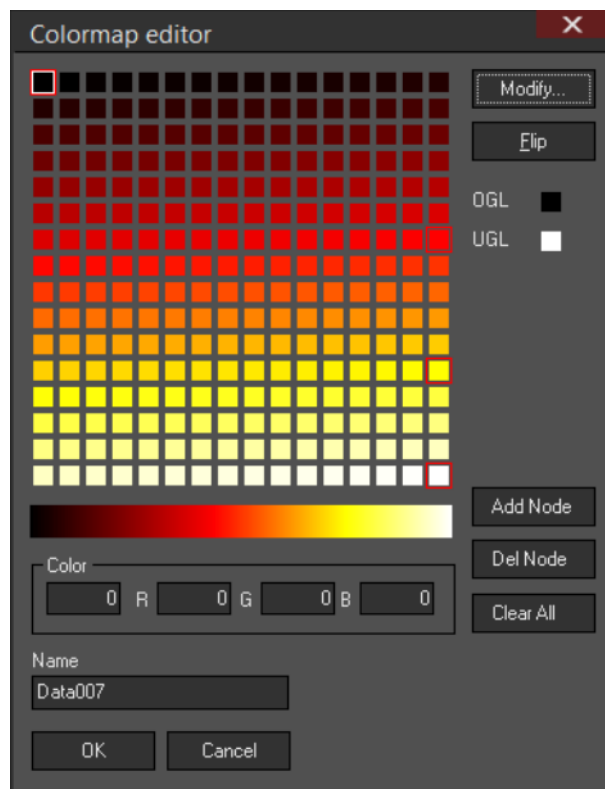


Fig. 4: 'Colormap editor' dialog.

Note: Colormaps can be saved and loaded again.

5.2 Toolbars

5.2.1 Display toolbar

The 'Display' toolbar (*The 'Display' toolbar.*) allows to change the way the data is displayed in a very fast way.

- The first three buttons allow to rotate a 3D data stack from XY to XZ to YZ.
- The fourth button allows to exchange the displayed X and Y axis and thereby to tilt the data axes.
- The fifth button changes the data representation from overlaid to side by side.
- The sixth button is to show the different data channels in RGB colors. The first channel will be shown in red, the second in green, and the third in blue. All other channels will not be changed.
- The seventh button shows/hides the information and comment pane on top of the measurement view.
- The eighth button shows information on the data stack dimensions.



Fig. 5: The 'Display' toolbar.

5.2.2 Files toolbar

- The first button of the 'File' toolbar (*The 'Files' toolbar.*) creates a **new measurement** with default imaging parameters (which are given during the installation of the microscope).
- The second button of the file toolbar creates a **new measurement from a template** (containing the imaging parameters).
- The third button of the file toolbar **opens** a measurement from a file. Here as well files from other compatible imaging formats can be opened.
- The fourth button of the file toolbar **saves** a measurement with a given file name (**save as**).
- The fifth button of the file toolbar saves a measurement under the same file name the current measurement is entitled with.
- If the 'overwrite warning' is activated Inspector will ask if the current measurement should be overwritten. Otherwise it saves it with the same file name. **An already existing file may be overwritten!**
- The sixth button of the file toolbar creates a **new window** within the current measurement.
- The seventh button of the file toolbar **cuts** out the selected data window. The data is purged to the clipboard.
- The eighth button of the file toolbar **copies** the selected data to the clipboard.
- The ninth button of the file toolbar **pastes** the selected data window to the measurement or graph window.



Fig. 6: The 'Files' toolbar.

5.2.3 Measurement toolbar

- The first button of the 'Measurement' toolbar (*The 'Measurement' toolbar.*) is the 'REC' button. It starts a measurement with the given parameters.

- The second button of the ‘Measurement’ toolbar pauses a measurement at the point when it is pressed.
- The third button of the ‘Measurement’ toolbar clones a measurement including the imaging parameters.
- The fourth button of the ‘Measurement’ toolbar is labeled ‘Auto repetition’. When pressed, the measurement is continued until being stopped.

Note: If the ‘overwrite warning’ is activated Inspector will ask if the current measurement should be overwritten. **An already existing file may be overwritten!**



Fig. 7: The ‘Measurement’ toolbar.

5.2.4 Zoom toolbar

- The first three buttons of the ‘Zoom’ toolbar (*The ‘Zoom’ toolbar*.) are similar to the ones known from Windows applications: **zoom to selection, zoom in, zoom out**.
- The fourth button of the ‘Zoom’ toolbar (‘Reset Zoom’) zooms the data to a scale where one pixel on the screen equals one pixel in the measurement.
- The fifth button of the ‘Zoom’ toolbar enlarges the measurement window to the size of the shown data (containing the zoom factor).
- The sixth button of the ‘Zoom’ toolbar shrinks the measurement window to the size of the shown data (containing the zoom factor).
- The seventh button of the ‘Zoom’ toolbar locks the dimensions of shown data.

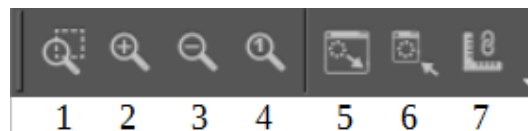


Fig. 8: The ‘Zoom’ toolbar.

Handling Data with Inspector

6.1 Importing Data into Inspector

Todo: Empty

6.2 Data Inspection

Todo: Empty

6.3 Data Analysis Functions

6.3.1 The Arithmetics Plugin

Inspector offers image arithmetic functionality. The Analysis/Arithmetics plugin is used to create new data stacks from already existing data stacks and the relation between the old and the new data is specified by a function. To access this, please open the ‘Arithmetic Operations’ dialog (*Analysis → Arithmetic Operations, [Arithmetic operations dialog](#)*).

One or two Stacks can be selected (upper left panel of the Arithmetic Operations dialog). The format of the data values of the output stack is selected below.

On the right upper panel, you either select a pre-determined function (Scale, Add, Subtract, Multiply, Divide, Offset, Invert, Normalize) with obvious meanings of the parameters and requiring one (scale, offset, invert, normalize) or two (add, subtract, multiply, divide) input stacks.

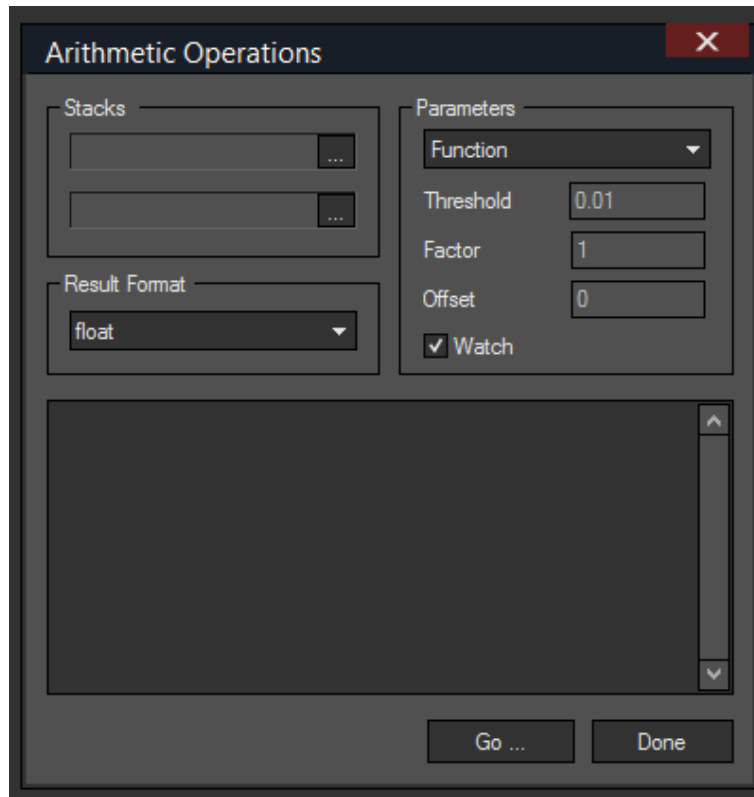


Fig. 1: Arithmetic operations dialog.

The last option (Function) allows to input an arbitrary term (edit box in the lower part) which is then evaluated for every position and of the input stack(s) and determines the values of the output stack.

The function parser expression executes point-wise through the input stack and therefore has some restrictions compared to a general treatment of input data with a script containing loops and conditions (although simple conditions can be included).

The output stack can re-evaluate itself, should the data of the input stack change (Watch checkbox). And a convenient way to modify the function later on is Right-Click/Manage and Manipulate Data/Edit Stack (ctrl + shift + e).

The Function parser which is used to evaluate the custom expressions is explained below.

6.3.1.1 The Function Parser

The dimensions of the output stack will equal the dimensions of the input stack. The given parser expression is evaluated at every position and the resulting value is stored in the output stack.

A very simple expression is r which just replicates the input stack (because r denotes the actual value of the input stack).

Substitutions are possible, new variables can be assigned and are used in all remaining terms, the last term must then yield a value which determines the value of the output stack.

Example: $a=5, r*a$

Creates a variable a with the value 5 and then computes $r*a$ which means the output stack will be equal to values of the input stack multiplied by five.

Numerical values can be also inserted in scientific notation, for example: $3.5e-6$.

In the following the different element types are listed.

Built-in variable	Description
r	Value of each pixel in the input stack (selected stack).
R	Value of each pixel in the second input stack (if given).
s, u, v, w	Pixel position (starting at zero) of the actual position in the input stack(s) in 1st, 2nd, 3rd, 4th dimension.
x, y, z, t	Physical position of the actual position in the input stack in 1st, 2nd, 3rd, 4th dimension.

Built-in operators	Description
$+, -, *, /$	Basic algebraic operations.
$<, >, ==, >=, <=$	Logical comparisons, results in 0 (false) or 1 (true).
$^$	Power
$\%$	Modulo. Also for real valued input (example: $r\%(2*\pi)$ wraps to $[0, 2\pi)$)
$//$	Integer division (example: $b*(a//b) + a\%b == a$ should return 1)

Built-in constants	Description
π	Pi
S, U, V, W	Total number of pixels in the input stack(s) in 1st, 2nd, 3rd, 4th dimension.
X, Y, Z, T	Total physical lengths of input stack(s) in 1st, 2nd, 3rd, 4th dimension.
$x0, y0, z0, t0$	Physical offsets of input stack(s) in 1st, 2nd, 3rd, 4th dimension.

Some common built-in functions are listed here. A complete list can be obtained by pressing F2 in the parser window. Built-in functions are used with parenthesis and comma separated arguments.

Built-in functions	Description
sqrt	Square root
$\text{sin}, \text{cos}, \text{tan}$	Sine, cosine, tangent
$\text{asin}, \text{acos}, \text{atan}$	Inverse sine, cosine, tangent
asin2	Inverse tangent given y, x separately.
min, max	Minimum/maximum of two values
abs	Absolute value (for integer and floating point numbers)
$\text{floor}, \text{ceil}$	Closest integer values rounded towards -Infinity/Infinity
exp, ln	Exponential, natural logarithm

Example of simple parser expressions

$\text{sqrt}(r)$

Takes the square root of the count values in the individual pixels

r^2

Squares the count values in the individual pixels.

$r-R$

Subtracts the intensity value of the second selected image/stack from those in the first selected image/stack.

Example of an advanced parser expression

```
sigma=5e-7,exp(-(x-(x0+X/2))^2+(y-(y0+Y/2))^2)/(2*sigma^2)
```

Creates a new stack with the same dimensions and pixel sizes of the input stack and the values of this new stack will correspond to a 2D centered Gaussian with a certain standard deviation `sigma`. Current values in the input stack are not regarded.

Access of input stack values

When applying a parser expression the arithmetics plugin:

- loops over all pixel of the input stack,
- evaluates the parser expression at each pixel position and
- assign the generated value as value of the output stack at this position.

Access to the current value of the input stack in the parser expression is via the variable `r` (and `R` for the second input stack if a second stack has been selected).

However, it is also possible to access values at different positions in the actual data stack or values in a different data stack with variables `s, u, v, w` or `x, y, z, t`. In this case, the stack name (can access any open data stack in Inspector) has to be given followed by a dot and `val(s, u, v, w)` or `func(x, y, z, t)`. Stack names are typically printed in quotes since they can contain spaces.

Example: `"ExpControl #2 {6}".val(s, u, v, w)`

This expression would produce equal results to the much simpler `r` if the selected stack has the name `"ExpControl #2 {6}"`. (Press F2 to get a list of all known objects.) The arguments `(s, u, v, w)` do not have to be in this order and can be complex expressions themselves (see example Rotate a 2D stack).

Example: `"ExpControl #2 {6}".val(u, s, v, w)`

For a square (equal number of pixels in 1st and 2nd dimension) 2D stack this exchanges the 1st and 2nd dimension, effectively mirroring the stack along the `y=x` line.

Notes:

- `func(x, y, z, t)` will interpolate if pixel positions are not hit directly
- `val(...)` and `func(...)` will return 0 if the given arguments are outside of the current data stack position ranges

Built-in advanced expressions

Conditionals: `Condition ? expression 1 : expression 2`

Condition is a logical expression (zero is regarded as false and everything not zero is regarded as true). Depending on the outcome either expression 1 (true) or expression 2 (false) is evaluated.

Example: `a > b ? a : b` is equivalent to `max(a, b)`.

Random number generation

Random numbers can be generated at each pixel position and the parameter for the random number generation can depend on the value of the input stack or an expression containing the value of the input stack.

<code>rand(max, min)</code>	Equally distributed random numbers in [min, max)
<code>gaussdev(sigma)</code>	Normally distributed random numbers with a certain standard deviation.
<code>poidev(avrg)</code>	Poisson distributed random numbers with a certain mean value.

6.3.1.2 Summary

The Arithmetic Operations dialog can be used to create derived data stacks calculating functions depending on values of input stacks. The functions are calculated point-wise which restricts the flexibility compared to for example running a custom script on the data with Python. Nevertheless, advanced features like conditionals or generation of random numbers make it a versatile and relatively easy to use tool. Inspector can keep track of updates in underlying input stacks and update the values of the derived stacks automatically by re-evaluating the stored parser expressions.

6.3.2 Examples for Arithmetics Plugin

6.3.2.1 Thresholding an Image (one sided thresholding)

First select the image to threshold at 'Stacks' using the pipette. Then set parameters to function. Then insert the formula: `r>15?r:0`. Finally Press 'Go ...' The formula contains following function: If a gray value is larger than 15 counts, leave it like it is. Otherwise set it to 0.

6.3.2.2 Thresholding an Image (two sided thresholding)

First select the image to threshold at 'Stacks' using the pipette. Then set parameters to function. Then insert the formula: `r>=15&r<20?r:0`. Finally Press 'Go ...' The formula contains following function: If a gray value is larger than or equal to 15 counts and smaller than 20 counts, leave it like it is. Otherwise set it to 0.

6.3.2.3 Creation of an Image with Poisson noise (no structure)

Insert the formula: `poidev(n)`, where you substitute `n` with the desired average intensity value within the resulting image stack.

6.3.2.4 Creation of an Image with Poisson noise on an existing structure

First select the image to threshold at 'Stacks' using the pipette. Then set parameters to function. Insert the formula: `poidev(r)` which uses `r`, the current intensity value of each pixel of the selected image stack.

6.3.2.5 Scale stack content about a factor a without changing the stack size

Solution using the Interpolation-function

- Obtain the size of the stack in pixel (using `ctrl + t`)
- Using 'Analysis'-'Interpolation' create a new stack with size `aN_i` (possible rounding errors!)
- Change the stack size of the interpolated stack (using `kbd:ctrl + t`) to the old size and chose thereby the middle of the interval in past position (whatever the signs are)

Solution using Analysis/Arithmetics

If you did not have set an offset on the stack but want to scale about the center position:

Parser expression: `zoom=a, "stack name".func((x-X/2)/zoom+X/2, (y-Y/2)/zoom+Y/2, z, t)`

Scaling is here done in the first two dimensions. Extension to 1D or 3D is straightforward. Insert the correct stack name instead of "stack name".

6.3.2.6 Rotate a 2D stack by angle

Assuming the rotation should be anti-clockwise in the XY plane (first two dimensions) and the rotation angle alpha is given in deg.

Parser expression: `arad=alpha/180*pi, stack.func((x-X/2)*cos(arad)-(y-Y/2)*sin(arad)+X/2, (x-X/2)*sin(arad)+(y-Y/2)*cos(arad)+Y/2, z, t)`

where stack is the name of the input data stack and alpha is the rotation angle in deg. Interpolation is applied.

6.3.2.7 Rotate a 2D-stack and produce a rotational symmetric 3D-stack out of it

This recipe is for a vertical rotation axis at $x = a$ (a is number of pixel):

- Copy 2D stack in new window, and set third dimension (ctrl + t) to the desired pixel size (in most case comparable to first dimension)
- Rename the new stack to eg. stack3d
- Use the following formula on the new stack: `a=??, stack3d.val(sqrt((s-a)^2+(v-a)^2)+a, u, 0, 0)`

6.3.2.8 Calculate a 2D Gaussian peak at the center

Parser expression: `sigma=5e-7, exp(-(x-(x0+X/2))^2+(y-(y0+Y/2))^2)/(2*sigma^2))`

Can be used to calculate a 2D Gaussian peak at the center of the current data stack.

A shortcut is: `gaussian2D(x-(x0+X/2), y-(y0+Y/2), 5e-7)`

where the width is given as full width half maximum (FWHM) and you can use 1D/2D/3D versions and as well for a Lorentzian function (use `lorentzian1/2/3D` in this case).

Note: Often made mistake is applying this function to a data stack with an integer data type and not changing the output data type to a floating point type before evaluating the parser expression.

6.3.3 Interpolation

Inspector offers to interpolate images. The interpolation can be done in up to four directions. In the 'Interpolation' dialog (*'Interpolation' dialog*.) first the resolution of the resulting image is given. Here typically integers are used as scaling factors. Inspector offers to use two algorithms that can be used to interpolate the data: 'Resample' and 'Lagrange Interpolation'.

The results of the interpolation directly show the differences of the algorithms. The 'Lagrange Interpolation' leads to rather smooth images where the 'Resample' algorithm preserves the data.

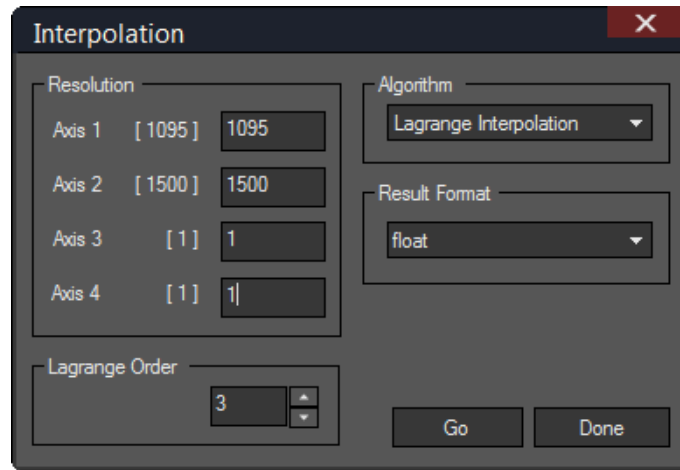


Fig. 2: 'Interpolation' dialog.

6.3.4 Smoothing

In Inspector images can be smoothed using the 'Smooth stack' dialog (*'Smooth Stack' dialog*). To smooth an image, the stack that should be smoothed is selected on the left side of the dialog. Then the type of smoothing has to be selected on the right side. Inspector offers different methods to smooth images:

- Typically smoothing is performed using Gaussian kernels with different sizes. To smooth an image using a Gaussian select 'Smooth' on the right side as 'Smoothing Type'.
- 'Median' allows to apply a median filter. This is commonly done to reduce 'salt & pepper noise'.
- Lowpass gauss
- Lowpass cutoff

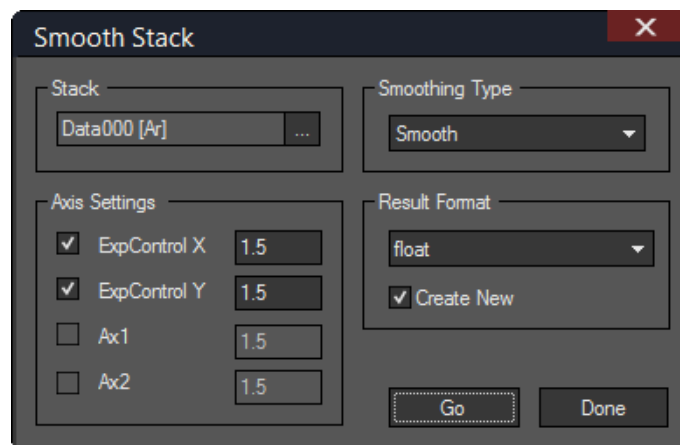


Fig. 3: 'Smooth Stack' dialog.

6.4 Image Deconvolution

In this chapter practical tips for performing a deconvolution of microscopic data with Inspector is given. First, some general remarks:

- Deconvolution is often used in image processing to remove the influence of the properties of your imaging device (represented by the Point Spread Function (PSF)) on the acquired image. The image is thereby assumed to be a convolution of the original object (dye molecules) distribution and the imaging device's PSF (point spread function aka blurring function), all of which is additionally distorted by ever present noise of mostly Gaussian or Poisson statistics.
- While the resolution of your image cannot be improved by deconvolution in a strict sense (i.e. information cannot be added if it is not already there), the deconvolved image is often less affected by imaging noise as a side effect. The obtained representation is often interpreted as the best estimate for the unknown object which was to be measured. The goal of deconvolution is to find this best estimate.
- However, especially for very dark images with a low number of collected counts (photons, ...) the usual deconvolution algorithms have to be applied with great care - the removal of noise also removes part of the information and we can end up with an useless or artifact-containing estimate. Also be aware that the deconvolved image is only an estimate of the object and can have systematic differences depending on the parameter of the deconvolution algorithm. Even small distances between peaks are normally preserved, however widths of objects in the order of the resolution can hardly be quantified (and indeed should not) and are somewhat arbitrary.

The deconvolution method allows to incorporate certain a-priori knowledge, like the positivity of the object or boundaries on the variation of objects. These are implemented for example by regularization parameter in Inspector.

And as a last thing: deconvolution can also be done only partly, e.g. when removing the side lobes of the PSF of a 4Pi microscope.

6.4.1 Creating an idealized PSF

For a deconvolution process the knowledge about the PSF of the imaging system is mandatory. The measurement of the PSF is often compromised by noise itself, most of the times only parameters like its width are known with high significance. Sometimes the shape of the PSF can be approximated well by a Gaussian or a Lorentzian peak. In this case there is an easy way to compute such a PSF in Inspector to have it ready, when it is needed for deconvolution.

We demonstrate the creation of a data stack containing a single gaussian or lorentzian peak in the center of a data stack in 2D or 3D here. The more general task to compute an arbitrary function with arguments being a data stack is discussed elsewhere but very closely related.

Here is the recipe:

1. Select an image stack that has the same size (physical and logical) as the PSF that you want to calculate. This will be mostly a measured data stack. Check the data stack size by **Right Click on Image/Manage and Manipulate Data/Stack Size & Data Type** and make sure, that the offset column is set to minus half the stack size¹ and the physical size column has the right units²
2. While the stack is selected (stack window heading is highlighted) select the from the menu **Analysis/Arithmetics** and select function as parameters in the dialog box that has appeared.
3. In the large edit box in the lower half of the dialog input you can input the formula to calculate your PSF. It will be fit into a new stack exactly the size as your image. For example enter

$$a=0.05, 2^{-(x^2+y^2)/(a/2)^2}$$

for a Gaussian with width 0.05 in 2D,

$$a=100, 2^{-(x^2+y^2+z^2)/(a/2)^2}$$

for a Gaussian with width 100 (for units see footnote) in 3D,

¹ So the origin of the internal coordinate system is at the center of the stack.

² Will be microns or nm in most cases. Given is the edge length of the field of view. A unit is not given, however all parameters later on have to have the same units, whatever they are.

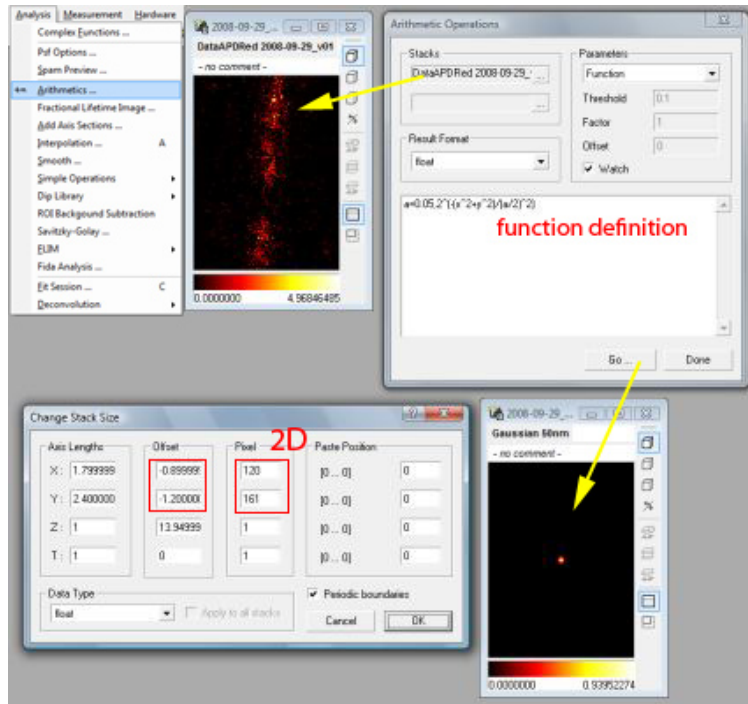


Fig. 4: The image stack (upper middle) is a 2D stack with the offset set at minus half the stack length, as can be seen in **Stack Size & Data Type (Ctrl.+T)**. Selecting this stack in the **Analysis/Arithmetics** menu, applying a function definition as explained in the text and clicking on **Go** creates a new stack with identical dimensions and the image of a PSF.

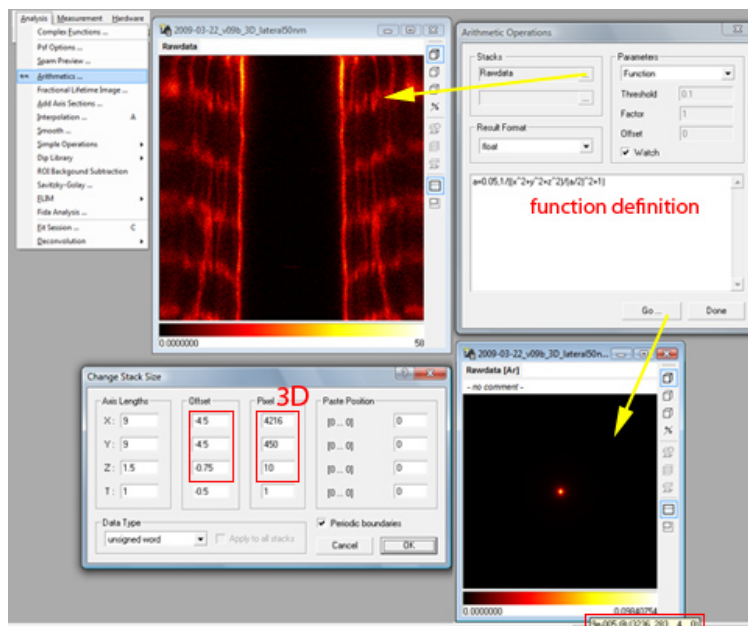


Fig. 5: The image stack (upper middle) is a 3D stack with the offset set as needed. Selecting this stack in the **Analysis/Arithmetics** menu, applying a function definition as explained in the text and clicking on **Go** creates a new stack with identical dimensions and the image of a PSF. The new stack is displayed at frame zero for default, scroll to the middle frame of the stack (Page up / down) to see the PSF's shape.}

$$a=0.05, 1/((x^2+y^2)/(a/2)^2+1)$$

for a Lorentzian with width 0.05 in 2D, or

$$a=100, 1/((x^2+y^2+z^2)/(a/2)^2+1)$$

for a Lorentzian with width 100 in 3D³. Adjust the formulas to your needs.

When clicking on **Go** a new stack should appear with a single centered peak (in 3D stacks one can see it only after scrolling to the central frame) which can be used in the following for deconvolving images. Inspector screenshots of the processes described above are shown in figures ref{fig:deconv_psf1} and ref{fig:deconv_psf1}.

6.4.2 Convolution

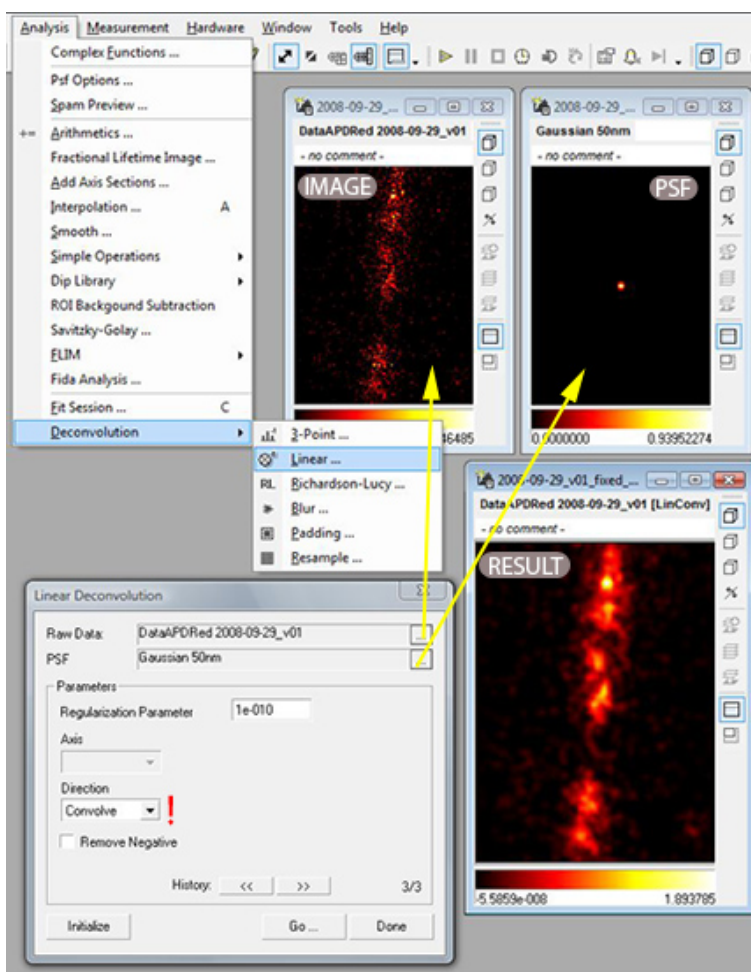


Fig. 6: Demonstration of the convolution of two data stacks. The direction in the dialog should be set to convolve. First select two data stacks in the fields Raw Data and PSF. Both stacks must have same data type and stack size. Then click on Initialize and Go. The convolved image will be computed. Leave the dialog with click on Done.

Smoothing is probably the easiest way to improve an image and is recommended especially for images with only a few counts where noise is the largest problem. The blurring effect of the PSF is here not removed but even more enhanced.

³ The normalization in this case is so that the maximum of the stack 1 (in the center). Although sometimes deconvolution algorithms expect a integral over the PSF of one (to resemble a probability distribution) this does not matter here in Inspector and is always (not sure) done automatically if necessary.

However, the noise is greatly reduced. The smoothing kernel will be in most cases a gaussian function. That means we have to provide a stack with equal physical and logical dimensions as the image stack (up to 4D possible) containing a centered gaussian function of certain width. Convolution of these two stacks (the order of the stacks can be exchanged thereby) is then performed via the menu command: **Analysis/Deconvolution/Linear** as shown in figure [ref{fig:deconv_conv}](#).

6.4.3 Point Deconvolution

Todo: Empty.

6.4.4 Wiener Filtering

Wiener Filtering or linear deconvolution is the optimal procedure when the image is compromised with gaussian noise. Its algorithm is based in fourier space where the convolution of PSF and object is represented by a simple multiplication. The reverse operation, the division is therefore simple to implement and will fail only where the fourier transform of the PSF (the optical transfer function, OTF) is zero or has a small amplitude. These is unfortunately true for many high spatial frequencies in all practical cases, therefore a regularization factor has to be added that dampens frequencies that were not transmitted very well and are dominated by noise and cannot be restored therefore. The way to do it in the program is via the menu command: **Analysis/Deconvolution/Linear** as shown in figure [ref{fig:deconv_lin}](#).

The regularization parameter has to adjusted so that the outcome is regularized properly. The scale for adjusting is mostly logarithmic, we advice to try 1E-1, 1E-2, ... 1E-10 and values between. A lower regularization parameter will result in largely overshooting positive and negative signal with many artifact. A larger than optimal regularization parameter will result in a smoothed version of the image.⁴

Because of the necessary regularization the resulting estimate is smoothed but sometimes does not get significantly smaller as expected when removing the PSF influence (noise prevents hard deconvolution in this case).

6.4.5 Richardson-Lucy

When we additionally to Wiener Filtering want to impose the restriction of a purely positive object (e.g. dye concentration) on the deconvolution process we end up with the Richardson-Lucy algorithm [cite{??}](#). This algorithm now is iterative, that means that next to a regularization parameter (as in the previous section to dampen the influence of high spatial frequencies which are dominated by noise) we have the number of iterations to be made as an additional parameter. The Inspector way of invoking this non-linear deconvolution method is via the **Analysis/Deconvolution/Richardson-Lucy** menu command as illustrated in figure [ref{fig:deconv_rl}](#).

Although in principle the optimal regularization parameter can be estimated from statistical theory, this is almost never done in applications. If the optimal regularization parameter would be found, the algorithm could run forever, every number of iterations which is high enough would be sufficient. Another, more practical approach is to save the resulting image after a fixed number of iterations each and choose from the images. In the beginning they will show too much blur, in the end, even the noise in the image will be translated to a crumbling structure, clearly representing artifacts.⁵

⁴ As a rule of thumb, we advice to adjust the parameter so that the smallest negative value present in the result is not more than 10% in absolute value of the highest positive value.

⁵ For most real world application we found an regularization parameter of 1E-10 and up to 100 iterations with stopping every 10 iterations sufficient.

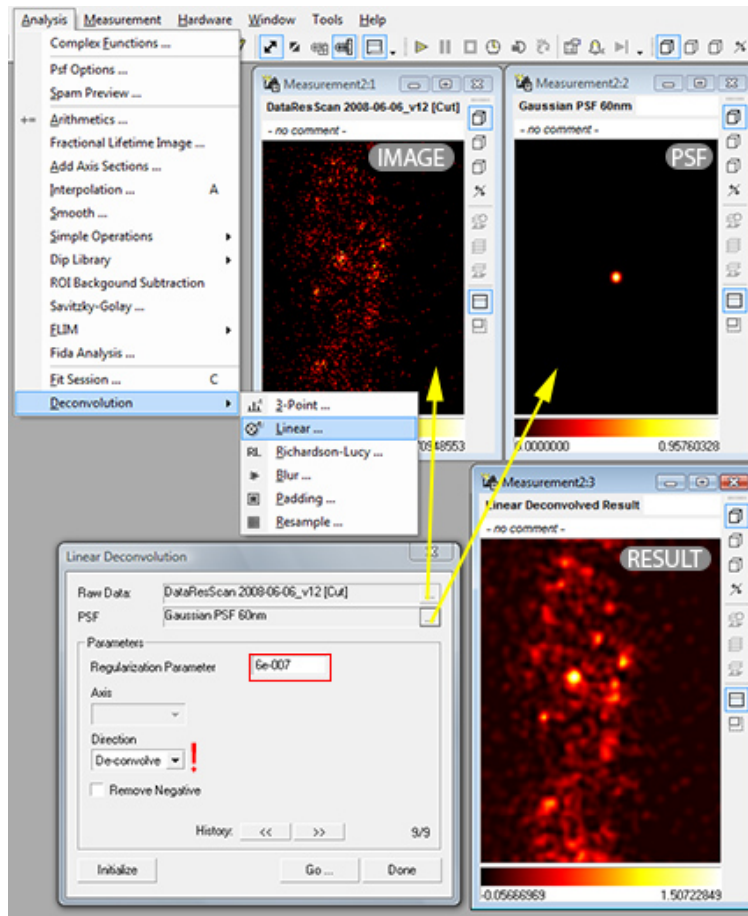


Fig. 7: Demonstration of linear deconvolution of two data stacks of equal size. The direction in the dialog should be set to de-convolve and the regularization parameter should be set to the smallest possible value where the artifacts (due to ringing, negative values in result) is still tolerable, which is normally achieved by values between $1e-4$ to $1e-8$. After selecting the image and the PSF (which are stacks of the same dimensions and the same data type) click on Initialize then on Go. A new stack with the linearly deconvolved image will appear.

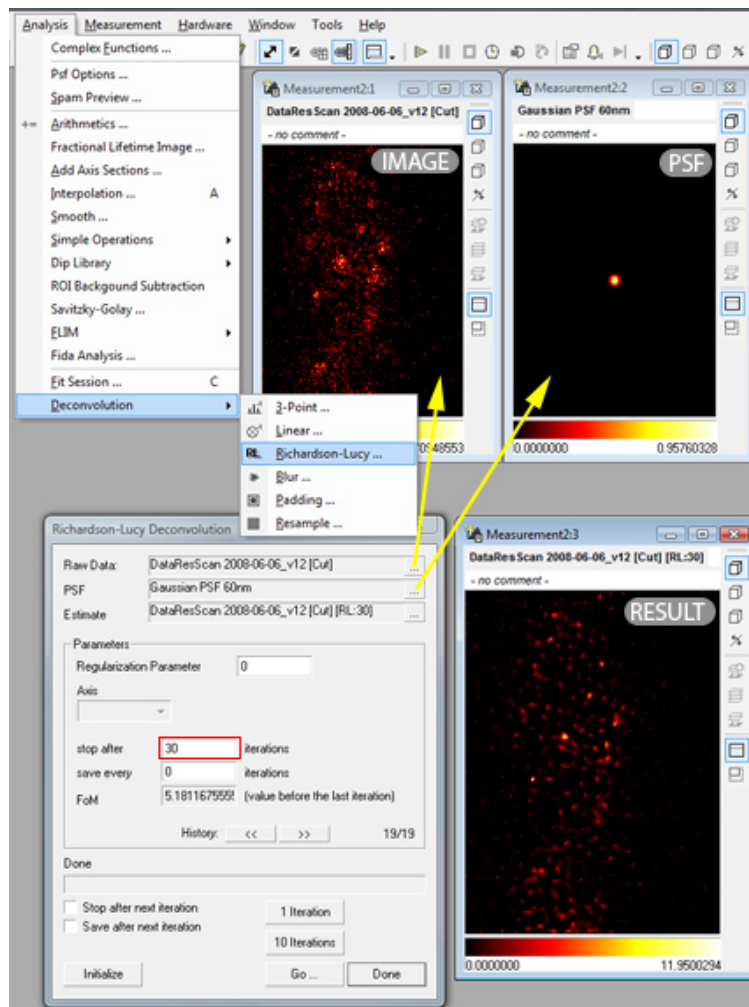


Fig. 8: Demonstration of Richardson-Lucy deconvolution of two data stacks of equal size. First select the image and the PSF in the two upper lines of the dialog. This type of deconvolution is iterative, so an estimate (as starting point) can be given (but is never necessary). The number of iterations is normally not above 100. First click on initialize then on Go. A new image appears. The estimate line is automatically replaced by the current result. Setting iterations to 30 and clicking two times on Go is equivalent to setting iterations to 60 and performing the algorithm only once. Intermediate results can be saved, a regularization parameter can additionally be set (1-0.001 are good values) - resulting in smoothed images.

Driving Hardware with Inspector

7.1 The Measurement Process

Todo: Empty

7.2 Hardware Configuration

Todo: Empty

7.3 Measurement Templates

In Inspector a template-driven workflow is implemented.

Measurement templates are ready-to-use parameter sets that enable a quick start into the use of Inspector and the microscope without the need for the user to be familiar with each and every detail of the microscope and the software. They contain the acquisition parameters (as field of view, pixel size, scan speed, scan direction, dimensionality, activated lasers, activated detectors...) that are required for a type of measurement. As in an Inspector measurement, multiple windows may be included.

To open a Template select *File* → *New* → *File from Template...*

During installation of the system a set of standard measurements schemes is pre-defined (*'Load Template' dialog*).

Based on these templates users can start with several basic measurement. Later, the given templates can be either adapted for the users measurement of interest or new templates can be created by the user. In contrast to easy-Commander driven work-flows, measurement templates are not restricted to simplified measurement settings, but may contain most parameters that are available in Inspector at a given the setup.

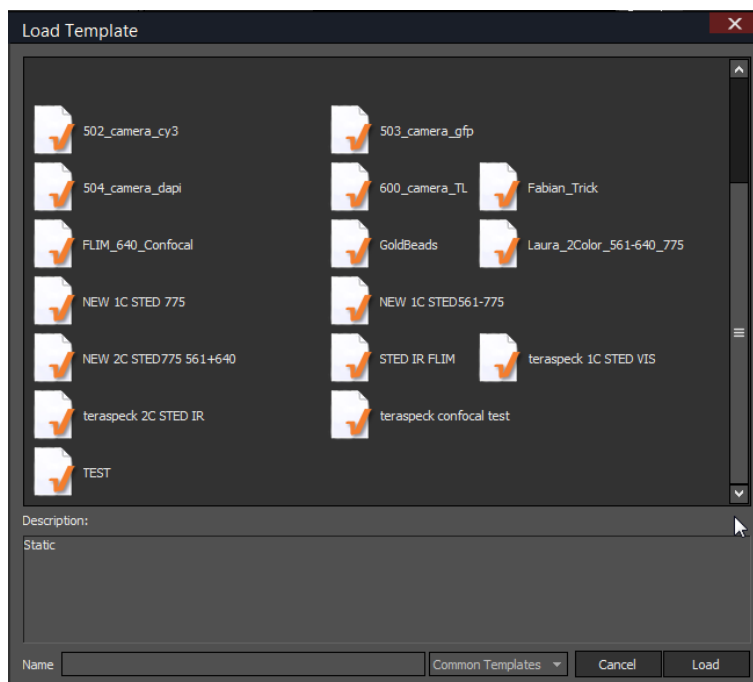


Fig. 1: 'Load Template' dialog.

7.3.1 Use of Existing Measurements as Templates

In Inspector existing measurements can be used as templates, because they contain required instrument parameters/meta-data for starting a new measurement. After opening a saved measurement, all experiments can be done in the exact same way by restarting the saved measurement with 'REC'.

Note: When several measurements are imported into Inspector, the scan settings of the active measurement window will be applied, when the 'REC' button is pressed. This allows a rapid change between different measurement modes.

7.4 Live Dialogs

In Inspector most imaging settings/parameters may be adjusted in live dialogs including activity of devices as lasers and detectors, settings such as laser power or pinhole size and guiding activity along the measurement process such as time lapse measurements.

Live dialogs can be closed individually and re-opened by opening the list in *Windows* → *Live Dialogs* from the main menu and choosing the corresponding dialogs from the list (*'Live Dialogs' window*). You can toggle the visibility of all live dialogs (with the exception of the 'Stack display' live dialog) by pressing the **TAB** button on the keyboard. Different layouts of live dialogs can be saved as workspaces and quickly restored by selecting *Windows* → *Save/Load Workspace* from the main menu.

7.4.1 Experimental Control Live Dialog

In the 'Experiment Control' live dialog (*'Experiment Control' live dialog*.) the basic scan settings are done:

- Type of scan (xy, yx, yz, xz, xt, xyt, xyz ...)

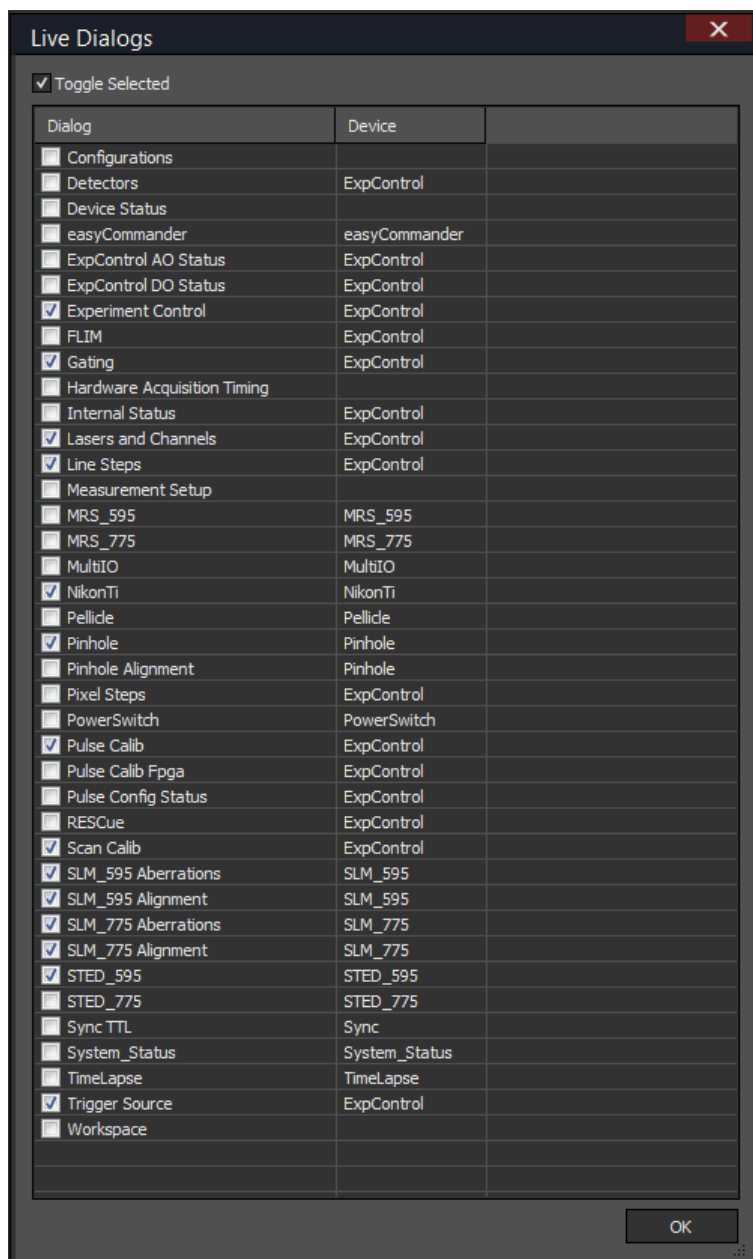


Fig. 2: 'Live Dialogs' window.

- Scan ‘Range’ in X, Y, Z, T
- Local and global offsets of the scanner i.e. the displacement from the central axis. A local offset (given in the upper half of the panel) applies only the current measurement. A global offset applies to all measurements in addition to the local offsets.
- Pixel size (‘Pix. Size’) and the number of pixels (‘# Pixel’)
- Number of scan lines to accumulate (‘Line Accu’)
- Time steps (‘T’): the T-axis in ExpControl allows for fast time-dependent scanning, i.e. for FCS measurements. This will only allow for consecutive time steps. If pauses in between the measurement steps or longer time scales, the *Time Lapse Live Dialog* has to be used.
- The ‘Orientation’ parameters allow to change the orientation of the scan field: scan field rotation (‘Rot’), tilting of the scan field (‘Tilt’) or rolling of the scan field (‘Roll’) may be applied.
- ‘Dwell Time’ for the individual pixels (only applies if ‘*Pixel Steps*’ is deactivated)

Note: The Line Frequency (‘Line Freq’) calculated from the scan parameters is given at the bottom right of the dialog. This is only an indicator and cannot be changed directly.

Check boxes

- ‘Lock Aspect’: If selected, the aspect ratio of the scan ranges is locked
- ‘Lock pixel Size’: if selected, the pixel size will be kept constant and the resolution is adjusted on changes of either scan range of pixel size. If it is disabled, the resolution will be kept constant and the pixel size will be adjusted automatically instead.
- ‘Square pixels’: When selected, the pixel sizes in x and y are kept equal.
- ‘Frame trigger’: Allows the start of the measurement to be started on an external signal. The polarity of the signal can be setup in the *Hardware Configuration*.
- ‘Use Autofocus’: Enables the autofocus module of the microscope (if available)

Note: The scan range may be rounded when the change is incompatible to pixel size and resolution.

7.4.2 Hardware Acquisition Timing Live Dialog

The measurement timing parameters as ‘Elapsed Time’ for the complete Scan, for the line, frame etc. are given in the ‘Hardware Acquisition Timing’ live dialog (*‘Hardware Acquisition Timing’ live dialog.*) .

7.4.3 Lasers and Channels Live Dialog

In the ‘Lasers and Channels’ live dialog (*‘Lasers and Channels’ live dialog.*) is one of the most important live dialogs. Here the user will configure the detection channels and lasers used. Furthermore the advanced imaging modes ‘Pixel Steps’, ‘Line Steps’, ‘Pulse Steps’ and ‘Pulse Gates’ can be activated.

Detectors

(Logical) Detection channels can be activated in the check box on the left side (‘Ch1’, ‘Ch2’, ‘Ch3’, ‘Ch4’), and the corresponding detectors (APDs / PMTs) selected from the drop-down menu. If PMT detectors are selected, the gain of the PMT may be adjusted here. Note that up to four logical (detection) channels can be used.

Lasers

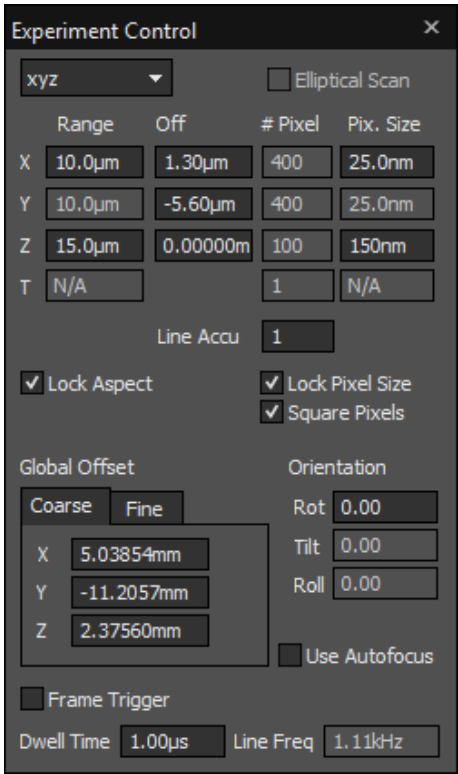


Fig. 3: ‘Experiment Control’ live dialog.

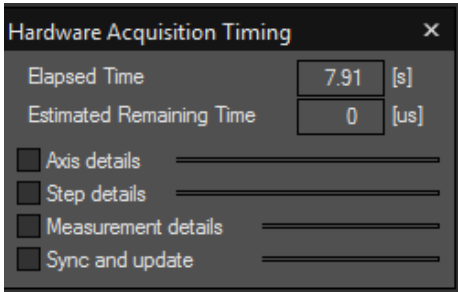


Fig. 4: ‘Hardware Acquisition Timing’ live dialog.

In the middle of the live dialog, the excitation, STED and RESOLFT lasers ('L1', 'L2', 'L3', 'L4', 'L5', 'L6', 'L7', 'L8') can be activated and their default powers adjusted using the %-values on the right. For alignment purposes, the lasers can be continuously activated using the check box on the right. (This is only possible if the user is logged in as Inspector admin.) Note that the %-values are calibrated to show a linear response in laser power.

Scan Options & Gating

On the bottom of this live dialog, following scan options can be selected: 'Pulse Steps', 'Pixel Steps', 'Line Steps'. Further the fluorescence time-gating for the different detection channels can be activated by selecting 'Pulse Gates'. Selecting an option will opens its configuration live dialog.

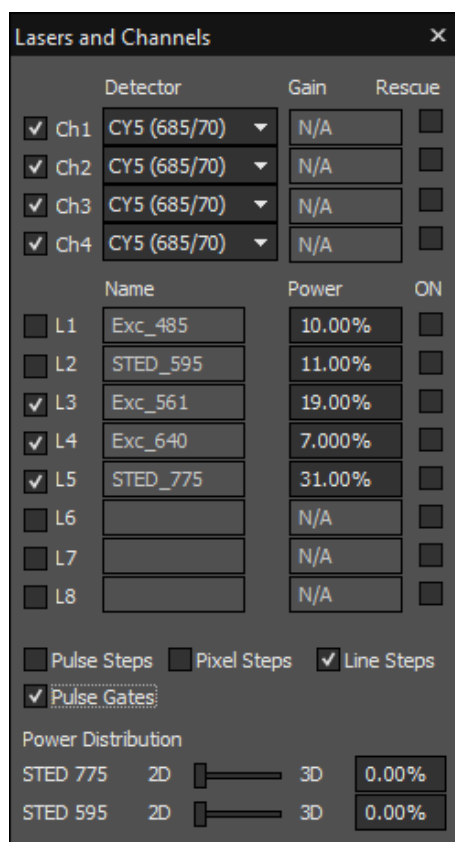


Fig. 5: 'Lasers and Channels' live dialog.

7.4.4 Microscope Control Live Dialogs

Inspector is able to communicate with the microscope stands of several vendors and all of its motorized components. It is able to drive most functions of the microscopy body (*Microscope live dialogs for Olympus IX and Nikon Ti eclipse microscopes*). Those functions include:

- switching of the observation mode: allows fast preset-based switching between different microscope configurations (i.e. for scanning, widefield imaging, etc.)
- changing the light path between different ports or the eye pieces
- setting and reading out the stage position (x, y)
- changing objective lenses
- changing filters in filter decks or motorized condensers

- setting and reading out the focus position of the objective lens (z-axis)
- switching on/off the auto-focusing device (optional upgrade)
- brightness of the halogens lamps in the transmitted illumination path (TransIllum) or intensity of fluorescence fiber illumination
- open/close shutters for illumination or auxillary ports

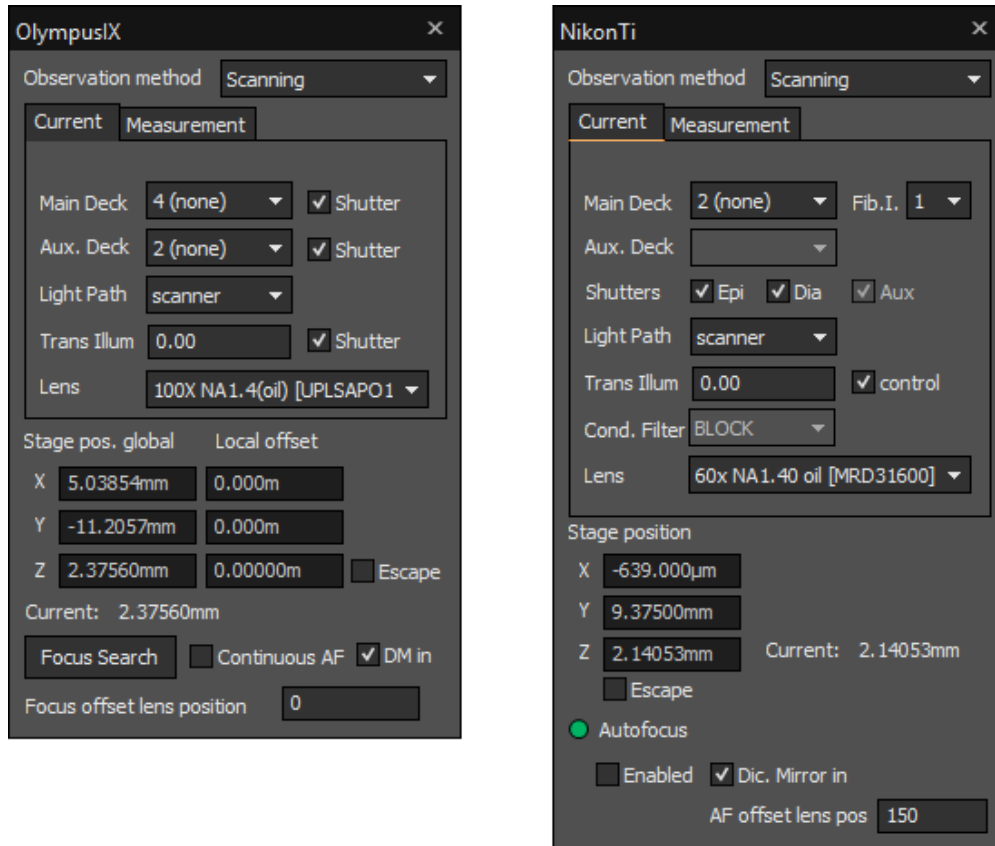


Fig. 6: Microscope live dialogs for Olympus IX and Nikon Ti eclipse microscopes.

7.4.5 Time Lapse Live Dialog

In the 'TimeLapse' live dialog (*'Time Lapse' live dialog.*), time lapse measurements with breaks in between the individual steps can be devised. More advanced settings with unequal steps can be further devised using the 'Sequence Scheduler'. 'Length' determines the interval, in which the individual measurements are to be taken, i.e. every second. 'Count' gives the number of repetitions. If the measurement might be longer than the interval, you may activate 'Ignore timer overflow' to continue the measurement even if it gets out of sync.

Note: The tool-tips on the usage of the sequence scheduler which can be opened using the question mark at the right side of the live dialog.

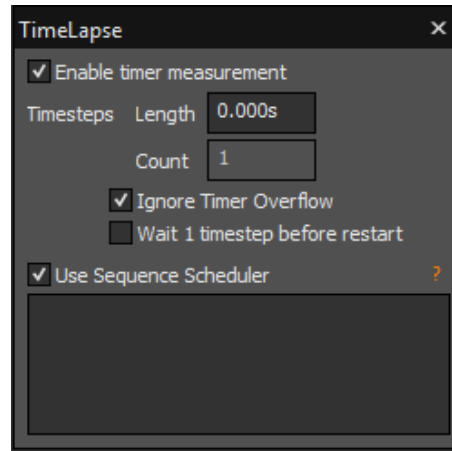


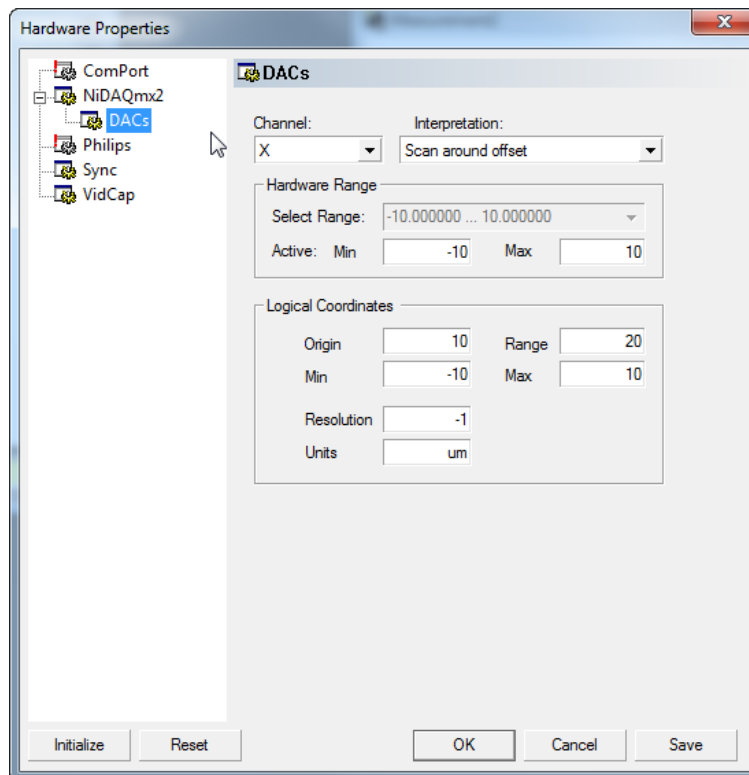
Fig. 7: 'Time Lapse' live dialog.

7.5 DAC Scanning Channels

There is a generic interface to digital-to-analog scanning channels that are e.g. provided by *NiDAQmx Cards*. It allows you to configure the scan range, fixed position during the measurement, the sync settings and line, pixel and frame clock.

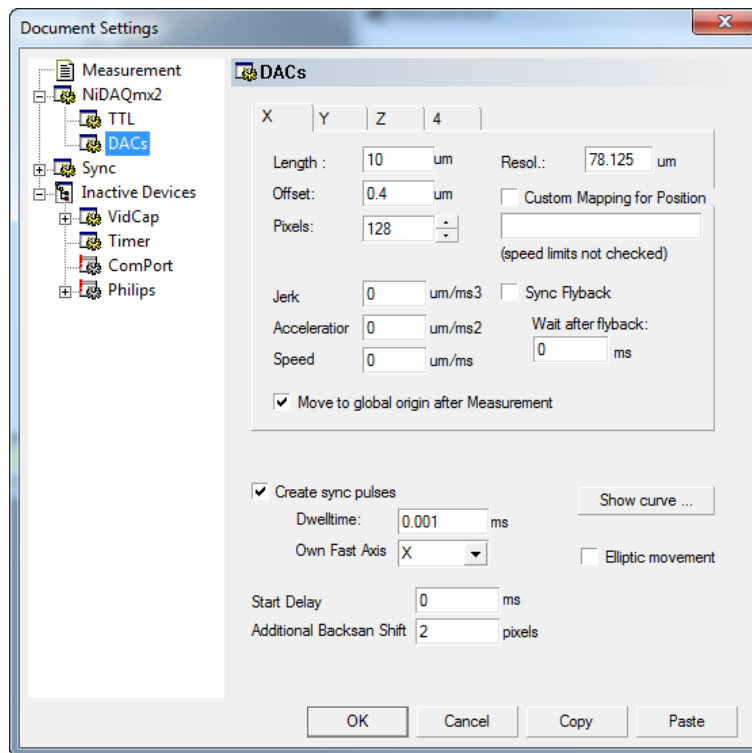
The interface currently contains a hack to allow fast 2D scanning which will disappear as soon as it is included in the main Inspector hardware control.

Devices providing such generic channels usually have sub-pages to their hardware configuration pages:

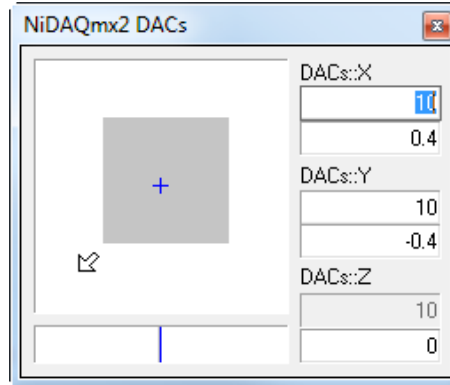


and for the per-measurement settings a page that is added below the main configuration page of the device, usually

titled 'DACs'.



Also, a Live Dialog is registered that allows adjustment and visual control of the scan range during the scan.



If you right-click on the dialog a menu appears with the commands

Maximum Range Reset the scan range to its maximum.

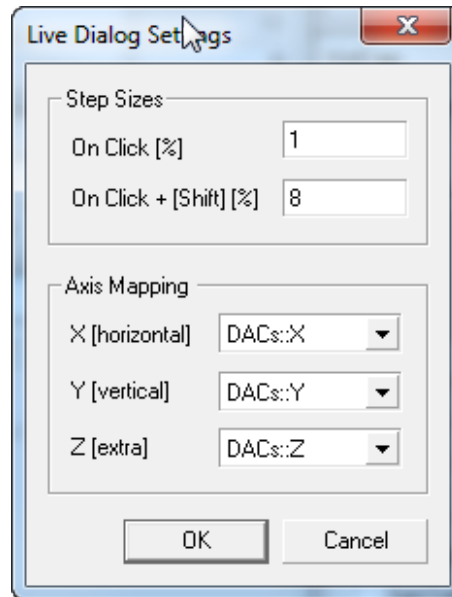
Note: If you have hardware limits in place (see above) the maximum scan range is almost always invalid as it requires infinite acceleration before flyback (the linear range reaches the edge of the valid range and there is no space to de-accelerate slowly).

Center actual scan area Center the scan area inside the valid range

Center origin Center the origin

Settings Adjust settings (see below).

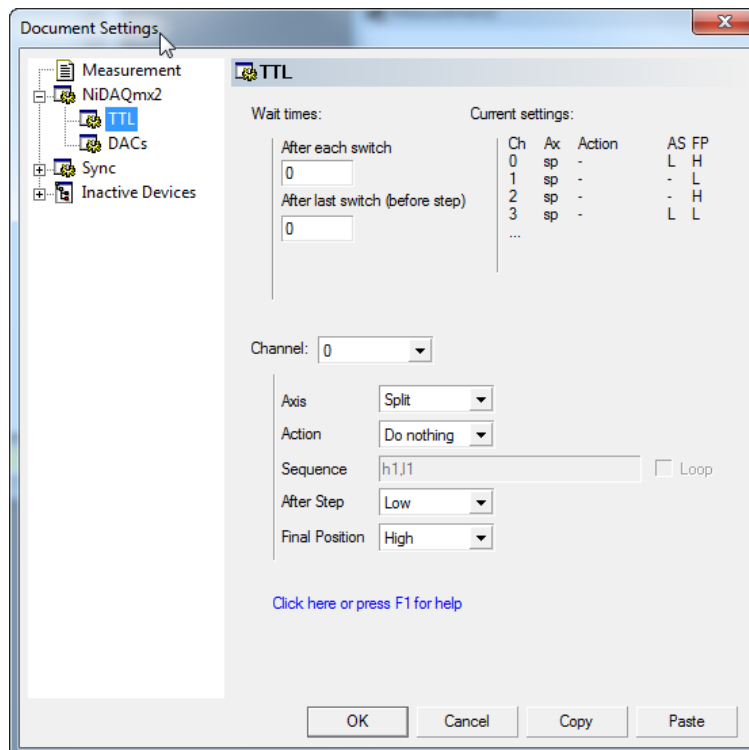
Its behaviour can be adjusted through the settings dialog



7.6 TTL Channels

There is a generic interface to TTL channels that are e.g. provided by *NiDAQmx Cards*. It allows you to configure switching sequences etc.

Devices providing such generic channels usually have sub-pages to their measurement configuration pages:



7.6.1 Wait times

All wait times refer to switching that goes on during a step on a non-synced axis. Along synced axes, all timing is governed by the dwell time and a TTL channel can only switch once per step on the fastest axis. On higher synced axes, the time between switches is given by the dwell time per pixel.

After each switch If a any of the TTL channels has selected a sequence as action that defines a pattern of several switches before the step, this is the wait time between consecutive switches. All patterns are put out simultaneously.

After last switch (before step) The time to wait after all pattern are output before the exposure / measurement of the step is started.

After last switch (after step) The time to wait after the exposure and after the TTL channels are switched to their 'After Step' state.

7.6.2 Current settings

A quick overview over the settings of all channels that are not off.

7.6.3 Channel action settings

These settings control the behaviour of a TTL channel during the measurement. You select the channel to be configured through the *Channel* combo box.

Axis If action is sequence, this defines the action axis. Upon each move along this axis, the TTL channel moves through its sequence. If a synced axis is selected different rules apply to how the sequence setting is interpreted. See [_TTLsynced_] for details. Please note that you can select the split axis which is defined in the Measurement settings. This allows you e.g. to acquire two images with different TTL settings. Frequently this is used for taking a STED and a confocal image simultaneously.

Action Valid actions are

Do nothing No action is performed. The TTL channel stays in the state it had before the measurement. If the state is changed through the live dialog (see below) this state will remain after the measurement unless the 'Final Position' parameter is set.

Keep high During each exposure the TTL channel will be high. This can be changed to *Keep high* during the measurement through the live dialog.

Keep low During each exposure the TTL channel will be low.

Sequence The TTL channel's state can change whenever a movement occurs on the selected **Axis** according to the sequence setting.

Sequence Defines a switching sequence. It is a comma separated list of states ('h' or 'l') each followed by a number defining the number of steps the state should remain unchanged afterwards. e.g.

h1,l1 Will set the state to high for one step, to low for the next. If **Loop** is selected it will thus alternate between high and low, otherwise it will remain low after the second step. The '1' can be omitted, the sequence is equivalent to 'h,l'.

h2,l2,h,l Will set the state to high for two steps and then to low for three, then to high for one and to low for one.

For some applications some switching pattern may be needed. This can be defined by a letter sequence, each defining the TTL channel to be in the corresponding state for the time defined by the wait time *After each switch*. For example

hl,hlhl2,hlhlhl Will switch to high and then to low before the first exposure. Switch to high, low, high, low before the second exposure and remain low for the third and switch three times before the third exposure.

Loop Whether to restart with the first entry in the sequence when there are more steps along axis. If not selected, the channel remains in the last state defined in the sequence.

After Step The state to put the TTL channel in after the exposure is finished. This will happen after each exposure independently of which action is selected or, if sequence is selected, independently of whether the selected axis has moved. For example, if *Keep high* is selected, but **After Step** is set to *Low* the channel will be set to high before each exposure and to low after each exposure. If *No change* is selected, the TTL state remains the same as during the exposure.

Final Position If this is set it ensures that the TTL channel is set to the specified state after the measurement irregardless of the state it was in before the measurement. If *No change* is selected, the TTL channel returns to the state it had before the measurement was started or, if the selected **Action** was *Do nothing* it remains in the current state (which may have been changed during the measurement through the live dialog).

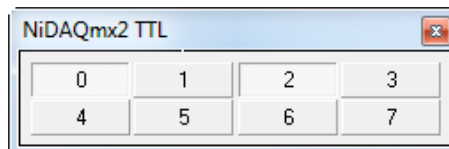
Note: The switching does not occur instantaneously as it is controlled by the computer. Also whether the device switches before or after actions by other devices depends on the order they have in the hardware list.

7.6.4 Synced axis behaviour

TTL switching along synced axes is not fully functional. Stay tuned for updates.

7.6.5 Live Dialog

Most TTL channel instances will also register a Live Dialog that allows adjustment of the TTL level during and after the measurement. In short, a button which is checked means the level of the TTL channel is high.



During a measurement, pressing buttons alters their state (they are disabled if the TTL channel runs a sequence). This state change will also modify the action setting (if it was *Keep high/low*). In this case the TTL channel will return to its pre-measurement state after the measurement finishes regardless of whether the state was changed during the measurement. See also **Final Position** above.

7.7 Generic Drivers

7.7.1 Com Driver

This is a generic comport driver. It allows you to run small scripts at the beginning and the end of the whole measurement and for each measurement step. A measurement axis is registered by the driver, so a simple stepping motor or scanning device can be programmed through this driver.

7.7.1.1 What does wait/line mean?

Line refers here to the command line, not a scan line. Wait/line is the time that Inspector waits after it has send a command line. If this waiting time is to short, part of the following commands may be lost. A value of about 300ms seems to work well at 9600baud.

7.7.1.2 What are Consts?

Consts provides 10 constants to which you can refer to in the command sequence. E.g. %f[C1] is replaced by the value of the constant C1.

7.7.2 SyncDriver

The sync driver has two functions. It serves as a provider for ‘dummy’ axes and it can do a low-accuracy sync with external devices by stopping the measurement until a certain event has occurred on a serial-port line or by triggering an external device by setting a serial-port line low or high.

7.7.2.1 Dummy Axes

Todo: empty

7.7.2.2 Passive Syncing

Todo: empty

7.7.2.3 Active Syncing

The SyncDriver cannot provide a pixel clock. It can, however set the output lines of the computers RS232 to high or low or pulse them in arbitrary sequences along the measurement axes. The triggering is set up through the ‘TTL’ property sheet in the measurement settings and follows the standard TTLChannels scheme (ref{sec:TTLChannels})

7.7.3 Timer Driver

Todo: empty

7.8 Camera Drivers

7.8.1 SpecCam Based Camera Drivers

7.8.1.1 Common Settings and Functions

Todo: Empty

7.8.1.2 Specifics of some Frequently Used Drivers

Cameras with DirectShow Drivers

Some, usually lower end, cameras come with DirectShow drivers which allow Inspector to drive them using a generic driver. The driver is tested with the following models:

mvBlueFox (Matrix Vision) Produced by Matrix Vision. Please install the appropriate driver from Matrix vision and test the camera using the software that comes with it before trying to use it with Inspector. In order to use the DirectShow driver you have to run the Matrix Vision configuration tool (mvDeviceConfigure.exe). Here you can set the DirectShow friendly name (the name which will show up in Inspector) and you HAVE TO register the device (in the Menu, there is a DirectShow entry that allows you to do this). If you have an x64 system you have to do this separately for x64 and x86 by running both the 32bit and 64bit version of the configuration utility.

A (not necessarily up-to-date) version of the drivers can be found here: [\(32bit\)](#) and [\(64bit\)](#).

Philips SPC900NC (Philips webpage) This camera is unfortunately not produced any more. If you have one: Lucky you, it has a decent chip and serves well as an overview cam.

The Imaging Source (Products) Offers a wide range of cameras. The driver runs with several of them - give it a try and if it does not work we will fix it.

Apogee Alta Cameras

Todo: Empty

Andor Ixon Cameras

Todo: Empty

Sensicam (by PCO)

Todo: Empty

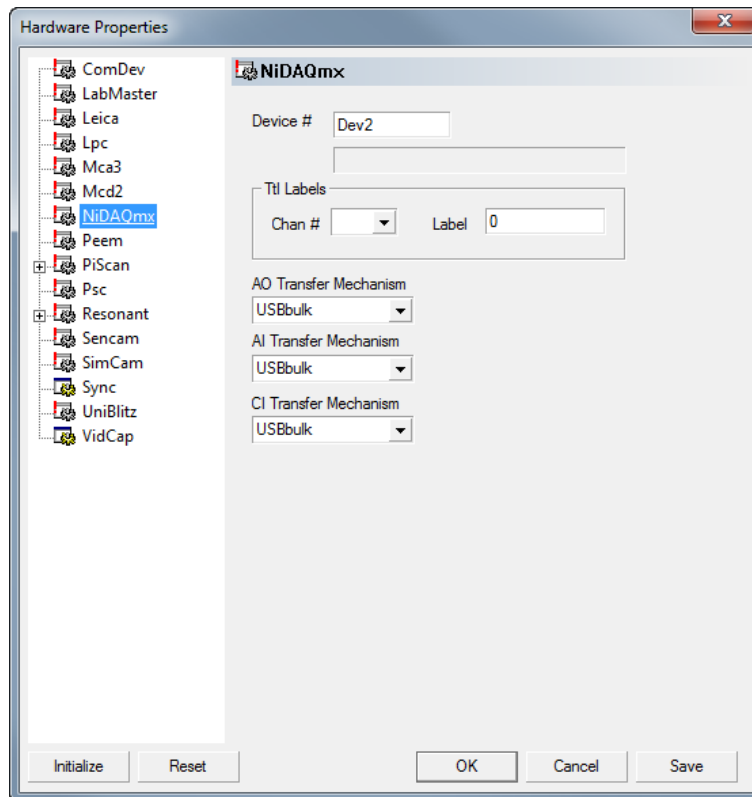
7.8.2 Deprecated non-standard Camera Drivers

7.8.2.1 Apogee Drive

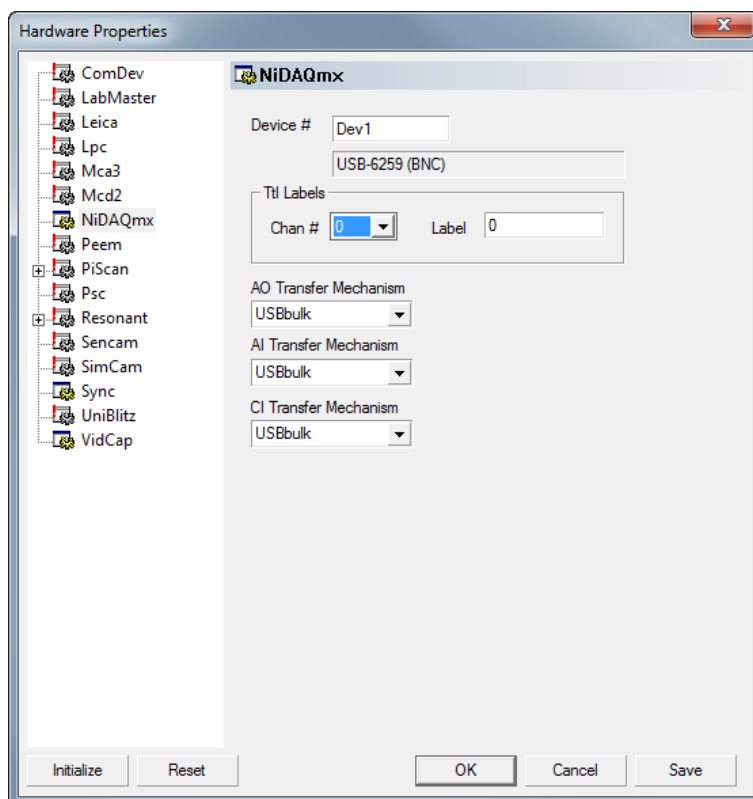
This driver is deprecated. You can use it as a fallback for old apogee cameras for which the ApogeeAlta driver does not work.

7.9 NiDAQmx Cards

Driver for National Instruments DAQ cards that can be interfaced with the NiDAQmx suite of drivers. During configuration Inspector has to be restarted once to allow the driver to register the correct amount of analog out (AO), analog in (AI), TTL (TO) and counter (CI) channels. Create a new device and enter the device id as found in the Measurement&Automation explorer into the configuration dialog.



Initialize the device by pressing the 'Init' button. Then restart Inspector and return to the configuration dialog. It should now include additional device information.



7.10 TCSPC hardware

7.10.1 Introduction

In conjunction with an AI FPGA-based scanner (generally called “Experiment Control”), that is in all AI microscopes, TCSPC hardware from both Becker&Hickl and Picoquant is supported. The documentation here is incomplete and does not replace the training you received when the package was installed on your microscope. This is for advanced users that wish to go beyond simple FLIM measurement or would like to modify the labeling.

7.10.2 Supported Devices

Supported hardware as of now are the B&H SPC-150, SPC-150N, SPC-160 and SPC-830 cards and the Picoquant TimeHarp260 and HydraHarp 400.

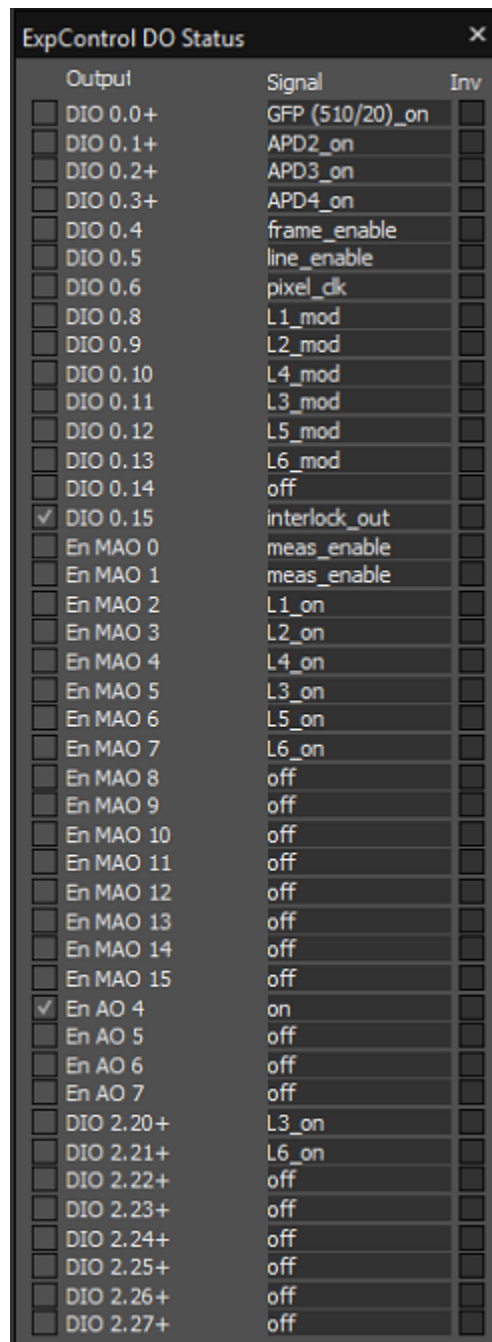
7.10.3 Operating Principle

All hardware is generally driven in FIFO mode and the FIFO data stream in hardware-specific format can generally be written to a file during measurement to allow offline analysis with tools you may have already established while using the component with its generic software. Inspector will control the hardware and supports a growing number of analysis options that should help you make sense of your measurement while it is still going on. It also ensures that you have to enter the measurement geometry only once (instead of making sure that it is synced between AI’s Inspector and the hardware vendor’s program) and that advanced features like pixel- and line-multiplexing are accounted for during channel-sorting.

Generally there will be a physical connection from the AI patch panel to the hardware's clock and detector inputs and to the marker inputs. At least the "frame enable" and "line enable" signal's have to be connected.

7.10.4 Settings to be adjusted

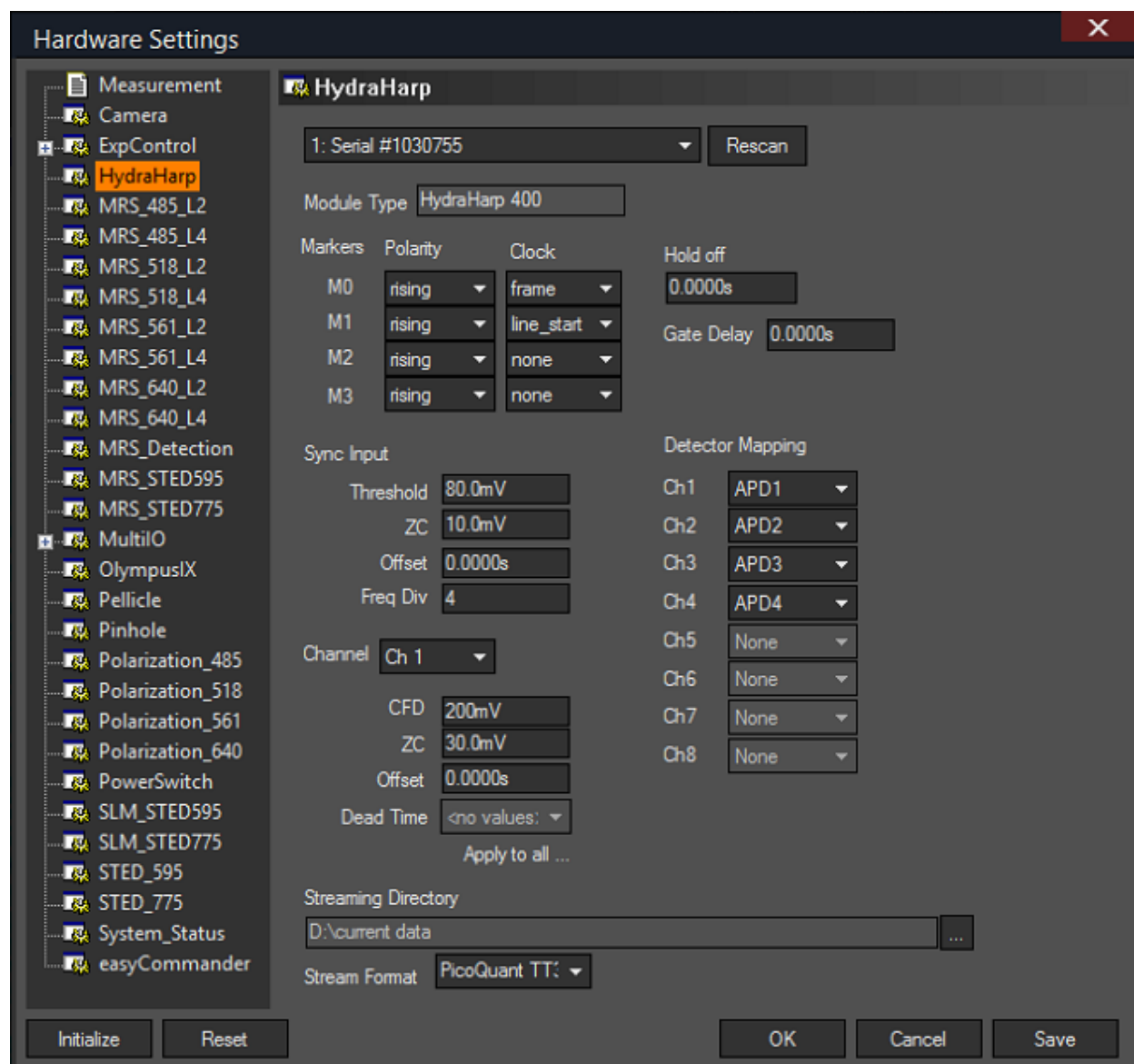
Use the "ExpControl DO Status" Live dialog to make sure the connected Patch Panel output actually carries the signal intended. Any output can be used except for those labeled "En <XXX>" which simply enable or disable the patch panel's AO outputs. All outputs marked with a "+" at the end of their label have 50 Ohm drivers. As an example:



Output	Signal	Inv
<input type="checkbox"/> DIO 0.0+	GFP (510/20)_on	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.1+	APD2_on	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.2+	APD3_on	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.3+	APD4_on	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.4	frame_enable	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.5	line_enable	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.6	pixel_clk	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.8	L1_mod	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.9	L2_mod	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.10	L4_mod	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.11	L3_mod	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.12	L5_mod	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.13	L6_mod	<input type="checkbox"/>
<input type="checkbox"/> DIO 0.14	off	<input type="checkbox"/>
<input checked="" type="checkbox"/> DIO 0.15	interlock_out	<input type="checkbox"/>
<input type="checkbox"/> En MAO 0	meas_enable	<input type="checkbox"/>
<input type="checkbox"/> En MAO 1	meas_enable	<input type="checkbox"/>
<input type="checkbox"/> En MAO 2	L1_on	<input type="checkbox"/>
<input type="checkbox"/> En MAO 3	L2_on	<input type="checkbox"/>
<input type="checkbox"/> En MAO 4	L4_on	<input type="checkbox"/>
<input type="checkbox"/> En MAO 5	L3_on	<input type="checkbox"/>
<input type="checkbox"/> En MAO 6	L5_on	<input type="checkbox"/>
<input type="checkbox"/> En MAO 7	L6_on	<input type="checkbox"/>
<input type="checkbox"/> En MAO 8	off	<input type="checkbox"/>
<input type="checkbox"/> En MAO 9	off	<input type="checkbox"/>
<input type="checkbox"/> En MAO 10	off	<input type="checkbox"/>
<input type="checkbox"/> En MAO 11	off	<input type="checkbox"/>
<input type="checkbox"/> En MAO 12	off	<input type="checkbox"/>
<input type="checkbox"/> En MAO 13	off	<input type="checkbox"/>
<input type="checkbox"/> En MAO 14	off	<input type="checkbox"/>
<input type="checkbox"/> En MAO 15	off	<input type="checkbox"/>
<input checked="" type="checkbox"/> En AO 4	on	<input type="checkbox"/>
<input type="checkbox"/> En AO 5	off	<input type="checkbox"/>
<input type="checkbox"/> En AO 6	off	<input type="checkbox"/>
<input type="checkbox"/> En AO 7	off	<input type="checkbox"/>
<input type="checkbox"/> DIO 2.20+	L3_on	<input type="checkbox"/>
<input type="checkbox"/> DIO 2.21+	L6_on	<input type="checkbox"/>
<input type="checkbox"/> DIO 2.22+	off	<input type="checkbox"/>
<input type="checkbox"/> DIO 2.23+	off	<input type="checkbox"/>
<input type="checkbox"/> DIO 2.24+	off	<input type="checkbox"/>
<input type="checkbox"/> DIO 2.25+	off	<input type="checkbox"/>
<input type="checkbox"/> DIO 2.26+	off	<input type="checkbox"/>
<input type="checkbox"/> DIO 2.27+	off	<input type="checkbox"/>

In this case relevant "pixel_clk" is on DIO 0.6, "line_enable" on DIO 0.5 and "frame_enable" on DIO 0.4. Please note that none of them has a 50 Ohm driver, so if your hardware has 500Ohm terminated marker inputs, you may want to use

i.e. the free “DIO 2.20+” ff. for this purpose. Also ensure that the cables go from these outputs to the marker inputs of the hardware. In the hardware settings you can now configure which marker carries which meaning and also whether the marker should be switched on and off. This will vary a little between different SPC hardware types but essentially the required settings are similar.



Here, the rising edge of M0 will be interpreted as frame enable, the rising edge of M1 as line enable. M2 and M3 will not be interpreted but are active and will be written to the stream (if disk streaming is enabled in the measurement). All markers that are not on “off” will in fact be written to the stream but Inspector only interprets “line_start”, “frame”, “line_end” and “pixel_clock” to correctly sort data into pixels and channels.

In your TCSPC hardware’s live dialog’s “Monitor Details” section you can see how many markers have been encountered in the current frame. This can help with debugging. In the main section, the LEDs will turn green if markers are encountered. If a marker is expected (i.e. the marker has been configured as “line_start”), the LED will turn red if no marker signal arrives.

HydraHarp

Rates

SYNC

40.00MHz

CFDs

0.000Hz

Marker Inputs

☐ frame clock

☐ line start

☐ M2

☐ M3

☐ Stream raw data to disk

Calibrate

Gate Delay

0.0000s

Monitor Details

Marker Signals Received

frame clock

1

line start

143

M2

0

M3

0

Channel Mon

1

0.000Hz

2

0.000Hz

3

0.000Hz

4

0.000Hz

5

0.000Hz

6

0.000Hz

7

0.000Hz

8

0.000Hz

FIFO usage

0.00%

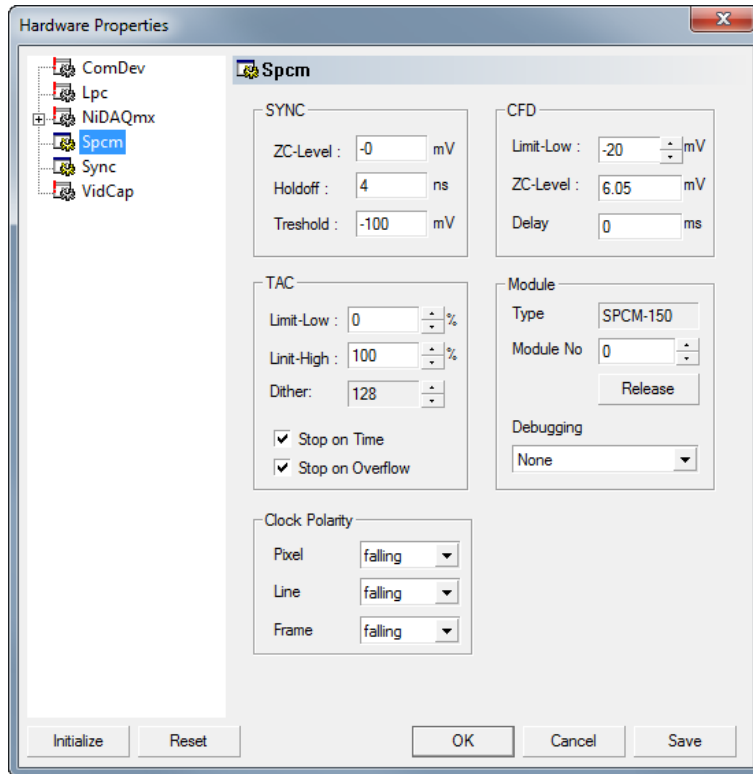
Status

idle

7.11 Becker&Hickl SPCM Cards (without AI FPGA scanner)

7.11.1 Configuring the Card

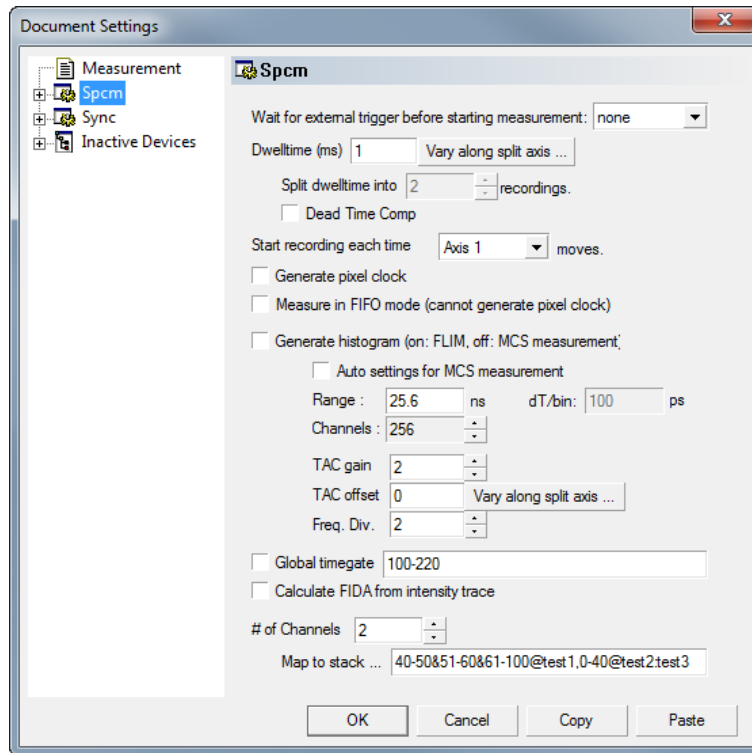
When scanning with “old style” SpecScan based drivers like NiDAQmx2 driver, time correlated single photon counting cards from Becker&Hickl are supported. When using the the FPGA card (“Experimental Control”), i.e. in all AI microscopes the new TCSPC interface should be used as described in another section. In the hardware configuration dialog the settings that concern all measurements are chosen:



Set the module number and initialize the device by pressing the 'Init' button. Check whether the Type displayed is correct. Most settings are analogous to the B&H software, please refer to their documentation for now.

7.11.2 Configuring Measurements

The settings for each measurement offer different possibilities than the B&H software. Most of them are self explanatory and this documentation may grow when time is at hand.



7.11.2.1 Multi-channel measurements

To use a router in order to record several channels, enter the appropriate channel number. In the 'Map to stacks ...' entry you can enter a list of stack names separated by ':'. You need exactly as many channels as there are router channels. For example

Ch1:Ch2:Ch3:Ch1

would be a valid string when using four channels. It would acquire 3 data stacks, mapping channels 1 and 4 to a stack named 'Ch1', channel 2 to a stack named 'Ch2' etc.

7.11.2.2 Global Timegates

Importantly, if 'Generate Histogram' is not selected and no FIDA analysis is to be applied, timegates can be chosen.

Either check 'Global Timegate'. This enables a global timegate for all channels. The configuration string is of the form '<b1>-<e1>[&<b2>-<e2>]...', i.e. a list of bin ranges to be included in the histogram separated by '&' characters (';' characters are also allowed due to backwards compatibility).

7.11.2.3 Channel-specific Timegates

Alternatively, global timegates can be de-selected and the channel mapping can be used to apply timegates. Instead of a simple name, the entry for a physical routing channel can consist of a list of assignments, separated by commas. Each assignment in turn has the form '<b1>-<e1>[&<b2>-<e2>]... @<name>' assigning one or several ranges of bins to a stack labelled with <name>. For example with one routing channel:

10-40&50-80@Ch2,90-220@Ch1

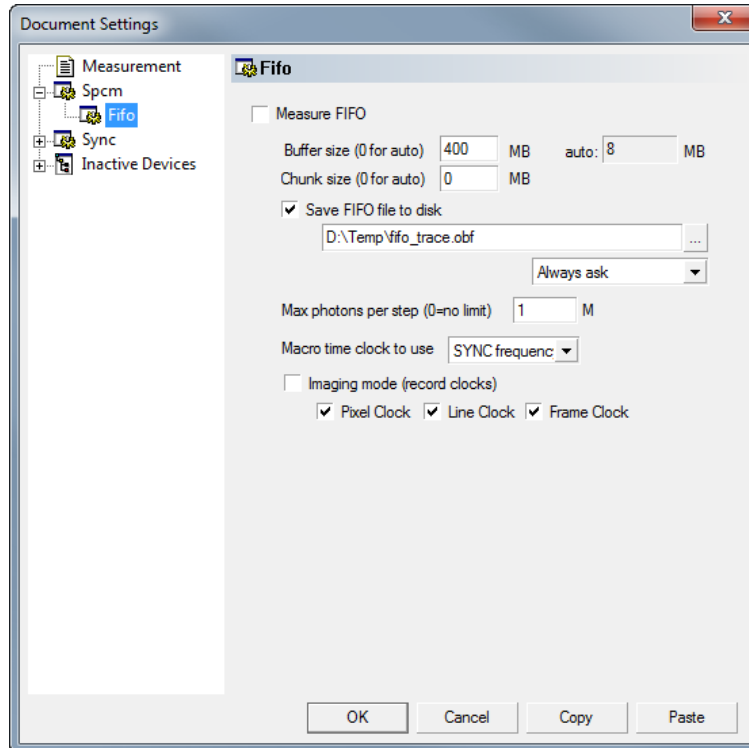
Would assign bins 10-40 and 50-80 to a stack 'Ch2' and bins 90-220 to a stack named 'Ch1'. For more physical channels:

10-40&50-80@Ch2,90-220@Ch1:10-120@Ch2

would do the same and assign bins 10-120 of routing channel 2 to the stack 'Ch2'. If several assignments to the same stack occur, they are added in the process.

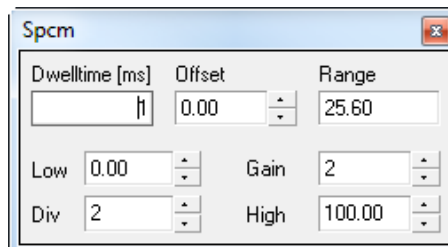
7.11.3 FIFO Measurements

The card can stream FIFO data to the disk during the measurement. Inspector then displays histograms calculated from the traces. This FIFO imaging mode is not fully functional. If an axis is synced it will ignore it and display all events added for this axis.



7.11.4 Live Dialogs

Some settings can be altered while the measurement is running through the Live dialog.



7.11.5 Rate monitor

Updated every second this displays current rates in Hz

Spcm Rates			
Sync	5.200e+005	CFD	4.680e+005
ADC	2.548e+005	TAC	3.640e+005

Inspector Python Interface

Inspector comes with a Python Interface named SpecPy which can be used either from the embedded Python console or from an external Python console running on the same computer (to enable sharing of measurement data) or even running on a different computer.

8.1 Setup

To setup SpecPy you first need to install

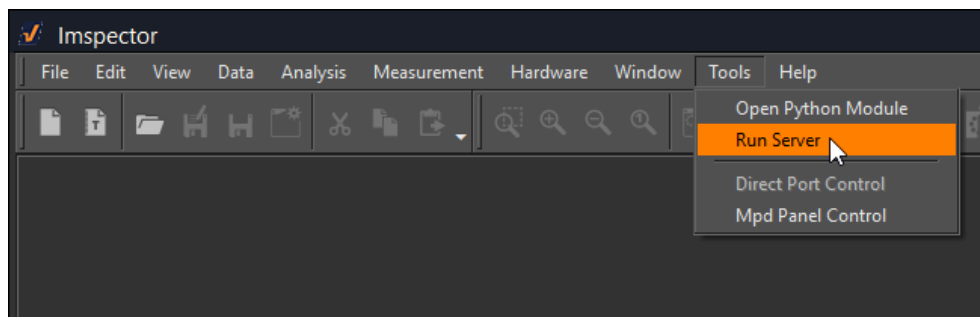
- Python ≥ 3.5 together with
- NumPy ≥ 1.10

Then from the Command Prompt run

```
python.exe -m pip install --upgrade specpy
```

8.2 Start

- Working from an external Python console you need to start the Inspector Server (which requires Administrator privileges the first time):



- To load the Python Interface just say

```
from specpy import *
```

8.3 Interface

8.3.1 SpecPy constants

```
SpecPy.__version__
```

contains a version string.

```
SpecPy.File.Read  
SpecPy.File.Write
```

constants **for** read **and** write access

8.3.2 Inspector

```
get_application()
```

first tries to return the local Inspector object (living in the same process) or else returns a proxy Inspector object connected to the Inspector Application running on *localhost*.

```
get_application(host)
```

where *host* is a host name returns a proxy Inspector object connected to the Inspector Application running on the corresponding host.

If *inspector* is an Inspector object then

```
inspector.host()
```

returns the name of the host the Inspector Application is running on or an empty string in case the Inspector object is local (living in the same process),

```
inspector.version()
```

returns the current Inspector version,

```
inspector.device_drivers()
```

returns the Inspector device drivers as a dictionary of name value pairs,

```
inspector.parameters(path)
```

where *path* is of the form *device/.../parameter_name* returns the corresponding Inspector parameter value (the empty path returns a dictionary of name value pairs of all parameters),

```
inspector.set_parameters(path, value)
```

where *path* is of the form *device/.../parameter_name* and *value* is a value, sets the corresponding Inspector parameter value (the empty path sets a dictionary of name value pairs of all parameters),

```
inspector.state(path)
```

where *path* is of the form *device/.../state_name* returns the corresponding Inspector state value (the empty path returns a dictionary of name value pairs of all states),

```
inspector.set_state(path, value)
```

where *path* is of the form *device/.../state_name* and *value* is a value, sets the corresponding Inspector state value (the empty path sets a dictionary of name value pairs of all states),

```
inspector.measurement_names()
```

returns the list of names of all open measurements in Inspector,

```
inspector.active_measurement()
```

for the currently active measurement in Inspector, returns the corresponding Measurement object or throws a `RuntimeError` if no measurement is active,

```
inspector.measurement(name)
```

where *name* is the name of an open measurement in Inspector, returns the corresponding Measurement object,

```
inspector.create_measurement()
```

creates an empty measurement in Inspector and returns the corresponding Measurement object,

```
inspector.open(path)
```

where *path* is the path to a measurement file, opens it in Inspector (if it is not already open) and returns the corresponding Measurement object,

```
inspector.activate(measurement)
```

where *measurement* is a Measurement object, activates the corresponding measurement in Inspector,

```
inspector.start(measurement)
```

where *measurement* is a Measurement object, starts the corresponding measurement in Inspector and returns immediately,

```
inspector.pause(measurement)
```

where *measurement* is a Measurement object, pauses the corresponding measurement in Inspector,

```
inspector.run(measurement)
```

where *measurement* is a Measurement object, runs the corresponding measurement in Inspector (starts it and returns when it has finished),

```
inspector.close(measurement)
```

where *measurement* is a Measurement object, closes the corresponding measurement in Inspector,

```
inspector.active_stack()
```

for the currently active stack (from the currently active measurement) in Inspector, returns the corresponding Stack object or throws a `RuntimeError` if no stack is active,

```
inspector.connect_begin(callable, flag)
```

where `callable` is a callable Python object, connects it to the corresponding begin signal in Inspector (if `flag` is 0 the begin of the whole measurement and if `flag` if 1 the begin of one measurement step),

```
inspector.disconnect_begin(callable, flag)
```

where `callable` is a callable Python object, disconnects it from the corresponding begin signal in Inspector (if `flag` is 0 the begin of the whole measurement and if `flag` if 1 the begin of one measurement step),

```
inspector.connect_end(callable, flag)
```

where `callable` is a callable Python object, connects it to the corresponding end signal in Inspector (if `flag` is 0 the end of the whole measurement and if `flag` if 1 the end of one measurement step),

```
inspector.disconnect_end(callable, flag)
```

where `callable` is a callable Python object, disconnects it from the corresponding end signal in Inspector (if `flag` is 0 the end of the whole measurement and if `flag` if 1 the end of one measurement step).

8.3.3 Measurement

If `measurement` is a `Measurement` object then

```
measurement.name()
```

returns the name of the measurement,

```
measurement.number_of_configurations()
```

returns the number of configurations in the measurement,

```
measurement.configuration_names()
```

returns the list of names of all configurations in the measurement,

```
measurement.active_configuration()
```

for the currently active configuration in the measurement, returns the corresponding `Configuration` object,

```
measurement.configuration(position)
```

where `position` is in the range from zero to the number of configurations in the measurement minus one, returns the corresponding `Configuration` object,

```
measurement.configuration(name)
```

where `name` is one of the configuration names in the measurement, returns the corresponding `Configuration` object,

```
measurement.activate(configuration)
```

where `configuration` is a `Configuration` object, activates the corresponding configuration in the measurement (if the measurement contains only one configuration, this configuration is activated by default),

```
measurement.clone(configuration)
```

where `configuration` is a `Configuration` object, clones the corresponding configuration in the measurement and activates and returns the clone,

```
measurement.remove(configuration)
```

where `configuration` is a `Configuration` object, removes the corresponding configuration in the measurement,

```
measurement.parameters(path)
```

where `path` is of the form `device/.../parameter_name` returns the corresponding measurement parameter value for the currently active configuration (the empty path returns a dictionary of name value pairs of all parameters),

```
measurement.set_parameters(path, value)
```

where `path` is of the form `device/.../parameter_name` and `value` is a value, sets the corresponding measurement parameter value for the currently active configuration (the empty path sets a dictionary of name value pairs of all parameters),

```
measurement.number_of_stacks()
```

returns the number of stacks in the measurement,

```
measurement.stack_names()
```

returns the list of names of all stacks in the measurement,

```
measurement.stack(position)
```

where `position` is in the range from zero to the number of stacks in the measurement minus one, returns the corresponding `Stack` object,

```
measurement.stack(name)
```

where `name` is one of the stack names in the measurement, returns the corresponding `Stack` object,

```
measurement.create_stack(type, sizes)
```

where `type` is a NumPy `array data type` and `sizes` is a list of exactly four sizes of dimensions, creates a new stack and returns the corresponding `Stack` object,

```
measurement.update()
```

redraws all corresponding stacks in Inspector (useful when the stack content was changed from Python),

```
measurement.save_as(path[, compression])
```

where `path` is a file path and `compression` is `True` by default or `False` saves it into a file.

8.3.4 Configuration

If `configuration` is a `Configuration` object then

```
configuration.name()
```

returns the name of the configuration,

```
configuration.parameters(path)
```

where `path` is of the form *device/.../parameter_name* returns the corresponding measurement parameter value for this configuration (the empty path returns a dictionary of name value pairs of all parameters),

```
configuration.set_parameters(path, value)
```

where `path` is of the form *device/.../parameter_name* and `value` is a value, sets the corresponding measurement parameter value for this configuration (the empty path sets a dictionary of name value pairs of all parameters),

```
configuration.number_of_stacks()
```

returns the number of stacks in this configuration,

```
configuration.stack_names()
```

returns the list of names of all stacks in this configuration,

```
configuration.stack(position)
```

where `position` is in the range from zero to the number of stacks in the configuration minus one, returns the corresponding Stack object,

```
configuration.stack(name)
```

where `name` is one of the stack names in this configuration, returns the corresponding Stack object.

8.3.5 File

```
File(path, mode)
```

where `path` is the path to an *.obf* or *.msr* file and `mode` is either `File.Read` or `File.Write` or `File.Append` opens it and returns the corresponding File object.

If `file` is a File object then

```
file.description()
```

returns the description of the file,

```
file.set_description(string)
```

where `string` is a string sets the description of the file,

```
file.number_of_stacks()
```

returns the number of stacks in the file,

```
file.read(position)
```

where `position` is in the range from zero to the number of stacks in the file minus one, reads and returns the corresponding Stack object,

```
file.write(stack[, compression])
```

where `stack` is a `Stack` object and `compression` is `True` by default or `False` writes it to the file,

```
del file
```

closes it.

8.3.6 Stack

```
Stack(array)
```

where `array` is a NumPy `array` returns a new local `Stack` object with data values from the array.

```
Stack(type, sizes)
```

where `type` is a NumPy `array data type` and `sizes` is a list of sizes of all dimensions returns a new local `Stack` object.

If `stack` is a `Stack` object then

```
stack.name()
```

returns the name of the stack,

```
stack.set_name(string)
```

where `string` is a string sets the name of the stack. If another stack in the same measurement already has the same name, suffixes of the form `[1]`, `[2]`, .. are added.

```
stack.description()
```

returns the description of the stack,

```
stack.set_description(string)
```

where `string` is a string, sets the description of the stack,

```
stack.type()
```

returns the type of the stack elements as NumPy `array data type`,

```
stack.number_of_elements()
```

returns the number of elements of the stack,

```
stack.number_of_dimensions()
```

returns the number of dimensions of the stack (including singleton dimensions, i.e. a shortcut for `len(stack.sizes())`),

```
stack.size(dimension)
```

where `dimension` is one of the dimensions returns the corresponding size of the stack (the number of steps/positions in that dimension),

```
stack.sizes()
```

returns the list of sizes of all dimensions of the stack,

```
stack.label(dimension)
```

where `dimension` is one of the dimensions returns the corresponding label of the stack,

```
stack.set_label(dimension, string)
```

where `dimension` is one of the dimensions and `string` is a string sets the corresponding label of the stack,

```
stack.labels()
```

returns the list of labels of all dimensions of the stack,

```
stack.set_labels(strings)
```

where `strings` is a list of strings for all dimensions sets the corresponding labels of the stack,

```
stack.length(dimension)
```

where `dimension` is one of the dimensions returns the corresponding length of the stack,

```
stack.set_length(dimension, number)
```

where `dimension` is one of the dimensions and `number` is a number sets the corresponding length of the stack,

```
stack.lengths()
```

returns the list of lengths of all dimensions of the stack,

```
stack.set_lengths(numbers)
```

where `numbers` is a list of numbers for all dimensions sets the corresponding lengths of the stack,

```
stack.offset(dimension)
```

where `dimension` is one of the dimensions returns the corresponding offset of the stack,

```
stack.set_offset(dimension, number)
```

where `dimension` is one of the dimensions and `number` is a number sets the corresponding offset of the stack,

```
stack.offsets()
```

returns the list of offsets of all dimensions of the stack,

```
stack.set_offsets(numbers)
```

where `numbers` is a list of numbers for all dimensions sets the corresponding offsets of the stack,

```
stack.parameters(path)
```

where `path` is of the form `.../parameter_name` returns the corresponding stack parameter value (the empty path returns a dictionary of name value pairs of all parameters),


```
stack.data()
```

returns the data of the stack as a NumPy array,

```
stack.meta_data()
```

returns the meta data of the stack as a dictionary of name value pairs (amongst others the units of the pixels are given there and are valid for the pixel sizes and well as the stack lengths, i.e. the pixel sizes are the division of the stack lengths by the stack sizes).

8.4 Examples

8.4.1 Python Interface Examples

There are a couple of examples.

8.4.1.1 Hello Inspector Example

Connects to Inspector, displays the parameters and doubles the exposure time of the sample camera if a camera is existing as device.

hello_inspector_example.py

```
"""
    Connects to Inspector, displays the parameters and doubles the exposure time of
    ↳ the sample camera
    if a camera is existing as device.

    Requirements: Latest Python 2.X or 3.X, specpy (https://pypi.python.org/pypi/
    ↳ specpy/1.0.2)
    which itself requires NumPy >= 1.8.1, Inspector with "Run Server" checked and an
    ↳ open measurement

    See also: http://inspectordocs.readthedocs.io/en/latest/specpy/examples.html
"""

# Python 2/3 compatibility
from __future__ import absolute_import, division, print_function

# import specpy
import specpy as sp

# connect to local Inspector
im = sp.get_application()

# print Inspector host and version
print('Connected to Inspector {} on {}'.format(im.version(), im.host()))

# get active measurement and print nicely
msr = im.active_measurement()
from pprint import pprint
print('Parameters of measurement {}'.format(msr.name()))
pprint(msr.parameters())
```

(continues on next page)

(continued from previous page)

```
# read and write a parameter
if 'SimCam' in msr.parameters():
    time = msr.parameters('SimCam/ExposureTime')
    msr.set_parameters('SimCam/ExposureTime', 2 * time)

# now have a look at Inspector, the parameter should have changed in the measurement
↳ properties of this measurement
```

8.4.1.2 Data Analysis Example

Connects to Inspector, opens a recorded measurement, computes statistical measures on this data and prints them in a file.

data_analysis_example.py - data_analysis_example.msr

```
"""
    Connects to Inspector, opens a recorded measurement, computes statistical
    ↳ measures on this data
    and prints them in a file.

    Requirements: Latest Python 2.X or 3.X, specpy (https://pypi.python.org/pypi/specpy/1.0.2)
    ↳ specpy/1.0.2)
    which itself requires NumPy >= 1.8.1, Inspector with "Run Server" checked and the
    ↳ file
    "data_analysis_example.msr".

    See also: http://inspectordocs.readthedocs.io/en/latest/specpy/examples.html
"""

# Python 2/3 compatibility
from __future__ import absolute_import, division, print_function

# import numpy
import numpy as np

# import specpy
import specpy as sp

# connect to local Inspector and open an example file
im = sp.get_application()
measurement = im.open("data_analysis_example.msr")

# set threshold
threshold = 410

# open output file
file = open('data_analysis_example_output.txt', 'w')

# for each stack in the measurement
for name in measurement.stack_names():
    # get the stack (as our stub object)
    stack = measurement.stack(name)
    # get the data array from the stack
    data = stack.data()
    # compute mean and std
```

(continues on next page)

(continued from previous page)

```

mean = data.mean()
standard_deviation = data.std()
# print mean and std to console and to file
print('stack {} has mean {} and std {}'.format(name, mean, standard_deviation))
print('stack {} has mean {} and std {}'.format(name, mean, standard_deviation),
↪file=file)

# apply mask (all values smaller threshold)
masked_data = np.ma.masked_less(data, threshold)

# compute mean and std on masked image
mean = masked_data.mean()
standard_deviation = masked_data.std()

# print mean and std on masked image to console and to file
print('masked stack {} with threshold {} has mean {} and std {}'.format(name,
↪threshold, mean, standard_deviation))
print('masked stack {} with threshold {} has mean {} and std {}'.format(name,
↪threshold, mean, standard_deviation), file=file)

# change pixels (you can see this in Inspector) below threshold to another value
np.putmask(data, data < threshold, 4095)

file.close()

```

8.4.1.3 Measurement Example

Example of how to traverse the measurements, configurations, stacks hierarchy of Inspector with Python. Also runs a measurement.

measurements_example.py

```

"""
    Example of how to traverse the measurements, configurations, stacks hierarchy of
↪Inspector with Python.
    Also runs a measurement.

    Requirements: Latest Python 2.X or 3.X, specpy (https://pypi.python.org/pypi/specpy/1.0.2)
↪which itself requires NumPy >= 1.8.1, Inspector with "Run Server" checked and an
↪open measurement.

    March, 2015, Jan Keller (jan.keller@mpibpc.mpg.de)

    See also: http://inspectordocs.readthedocs.io/en/latest/specpy/examples.html
"""

# Python 2/3 compatibility
from __future__ import absolute_import, division, print_function

import pprint
pp = pprint.PrettyPrinter(indent=2)

# import NumPy
import numpy as np

```

(continues on next page)

(continued from previous page)

```

# import specpy
import specpy as sp

# get inspector object and print version
im = sp.get_application()
print('Inspector {} on {}'.format(im.version(), im.host()))

# print existing device drivers
print('devices')
pp.pprint(im.device_drivers())

# print current inspector parameters
print('parameters')
pp.pprint(im.parameters())

# list all open measurements
print('names of open measurements')
pp.pprint(im.measurement_names())

# get currently active measurement
msr = im.active_measurement()
print('name of active measurement: {}'.format(msr.name()))
print('configurations in active measurement')
pp.pprint(msr.configuration_names())
print('number of stacks: {}'.format(msr.number_of_stacks()))
pp.pprint(msr.stack_names())

# get another measurement (the first one, should be at least one open)
measurement_name = im.measurement_names()[0]
msr = im.measurement(measurement_name)
print('name of chosen measurement: {}'.format(msr.name()))
print('configurations in chosen measurement')
pp.pprint(msr.configuration_names())
print('active configuration: {}'.format(msr.active_configuration().name()))
print('number of stacks: {}'.format(msr.number_of_stacks()))
pp.pprint(msr.stack_names())

# set active configuration (the first one, should contain at least one)
configuration_name = msr.configuration_names()[0]
configuration = msr.configuration(configuration_name)
msr.activate(configuration)
print('active configuration: {}'.format(msr.active_configuration().name()))

# create new stack (will be added to active measurement)
s = msr.create_stack(np.int32, [100, 100, 1, 1])
print('number of stacks: {}'.format(msr.number_of_stacks()))
pp.pprint(msr.stack_names())

# runs the measurement
print('active configuration {}'.format(msr.active_configuration().name()))
im.run(msr)
print('number of stacks: {}'.format(msr.number_of_stacks()))
pp.pprint(msr.stack_names())

# set a configuration and runs it
configuration = msr.configuration(configuration_name)

```

(continues on next page)

(continued from previous page)

```
msr.activate(configuration)
print('active configuration {}'.format(msr.active_configuration().name()))
im.run(msr)
print('number of stacks: {}'.format(msr.number_of_stacks()))
pp.pprint(msr.stack_names())
```

Inspector Matlab Interface

Inspector comes with a Matlab Interface named SpecMx which can be used by Matlab running on the same computer (to enable sharing of measurement data) or even running on a different computer.

The usage is very similar to *Inspector Python Interface*. The syntax is identical with the obvious conversions (dictionaries to structs, None to []).

The data content of a Stack in Inspector (obtained by `Stack.data()`) can only be read, not written from Matlab.

- Supported data types for Stack are single, double, int8, uint8, int16, uint16, int32, uint32.
- Dimension indices start from 0.
- `stack.set_data(array)` updates the data, updating the underlying array doesn't (for now)

CHAPTER 10

Utility applications

Inspector comes with several utility and test applications to check the functionality of hardware and ease resolving of hardware issues. In particular the direct port control utility may prove useful outside the use of Inspector.

10.1 Direct Port Control

Inspector comes bundled with a direct port control utility (MppTest.exe) which allows direct access to serial, parallel, GPIB and QuickUSB devices and can be used to test these devices using the same framework Inspector will use to drive them through its hardware drives.

If you are only interested in MppTest and not in Inspector, contact the support team.

10.2 MPD Panel Control

In theory the MPD panels can be used by several applications in parallel with access synchronized through a service process MpdCtrl.exe. If your panel is not visible from inside Inspector the most common cause is that MpdCtrl is not running or that a wrong version is running.

To start the proper version, stop MpdCtrl.exe by right-clicking the taskbar symbol and choosing Exit. If the symbol is not visible in the taskbar, MpdCtrl is probably not running but you can double-check using the TaskManager.

Then through the Inspector Tools menu choose “Mpd Panel Control” to re-start the app. Repeat this step any time you change the version of Inspector you are using.

CHAPTER 11

HOWTOs

While things can be obvious to one user, another might be completely unaware of the feature, even if it was well labeled by the programmer. Here we have compiled a list of useful tips making daily routine much more efficient.

- `ctrl + arrow left/right` moves between the last selections in a stack window. Unfortunately, selections are not kept when saving the data.
- Assigning custom shortcuts are very helpful at times! To assign custom shortcuts chose 'Toolbar Options' (the small arrows at the end of a toolbar) - 'Add Remove Buttons' - 'Customize' - 'Keyboard'.
- The function 'Change Source File', invoked by `ctrl + shift + i` comes handy when inspecting several similar files in a folder. But be aware that the window name does not change when the content is changed.

CHAPTER 12

Shortcuts

Category	Shortcut	Function
	F9 / F10	Fit maximum / minimum In a stack view this adjusts the zoom to fit the maximum or minimum value of the shown graph.
	page up / down	Go up / down one slice along the z-axis. Also pres
	ctrl + page up / down	Go up / down one layer along the hidden axis. Also
	right mouse click	Within an image: Context menu appears for selecti
	left mouse button drag	Select a ROI area (or line profile position) within t
	ctrl + left mouse button drag	Copy selected image (or ROI of the image) into a n
	ctrl + alt + left mouse button drag	Copy the view of the data (independently of the se
	Select ROI, right mouse button drag	Context menu appears and select 'use as ROI'. Thi
	tab	Toggles on/off all live dialogs. For measurement a
	alt + left mouse button drag	Zoom into selected region.
Stack view - Size & Info	ctrl + t	Shows information about the stack size (physical d
File	ctrl + n	Create new measurement.
File	ctrl + m	Edit comment.
File	ctrl + o	Open measurements. . .
File	ctrl + w	Close measurements. . .
File	ctrl + p	Print.
File	ctrl + e	Export measurements. . .
File	ctrl + x	Crop Stack / Crop Graph.
File	ctrl + v	Paste Stack / Paste Graph.
File	ctrl + i	Import measurements. . .
File	ctrl + shift + s	Save measurements. . .
File	ctrl + s	Save measurements as. . .
Graph view	ctrl + r	Rename data container / line profile.
Graph view	F9	Adapt the maximum (Y) value limit of the shown g
Graph view	F10	Adapt the minimum (Y) value limit of the shown g
Graph view	ctrl + t	Change graph size.
Graph view	ctrl + l	Toggle Limits on / off.
Graph view	ctrl + a	Select all.

Category	Shortcut	Function
Graph View	ctrl + d	Change Graph style.
Graph View	ctrl + r	Rename data container / Rename Graph.
Graph View	ctrl + shift + e	Edit graph properties.
Graph View	ctrl + shift + g	Open graph window.
Region of interest	ctrl + shift + u	Set Selection as Region of interest (ROI).
Region of interest	ctrl + shift + r	Copy Region of interest (ROI).
Stack view	F5	Fit image to frame.
Stack view	ctrl + r	Rename data container / Rename Stack.
Stack view	ctrl + shift + c	Change Pixel Value.
Stack view - Colormap	F9	Adapt the upper colormap limit to the current image.
Stack view - Colormap	F10	Adapt the lower colormap limit to the current image.
Stack view - Colormap	alt + shift + e	Equalize colormap between different measurements.
Stack view - Colormap	shift + F9	Adapt the lower colormap limit to the current image.
Stack view - Colormap	shift + F10	Adapt the lower colormap limit to the current image.
Stack view - Gallery	ctrl + 1 - ctrl + 9	Gallery layout (number of stacks in a row).
Stack view - Gallery	ctrl + g	Gallery mode / overlay images. All windows of the stack.
Stack view - Projections	ctrl + h	Hidden axis / projections.
Stack view - RGB	ctrl + shift + b	RGBize Data. When having 3 images in one window.
Stack view - Zoom	ctrl + Num-	Zoom out.
Stack view - Zoom	ctrl + Num+	Zoom in.

Contributions to the Documentation

You are welcome to correct any errors and contribute yourself to this documentation.

13.1 Correct Mistakes

The easiest way is to send in patches for existing pages. All documentation is created using [restructured text](#) which basically means that you have to edit some text files. You can get the existing source of any page by clicking on the ‘Edit on Github’ link on the left sidebar and adjust it to your liking.

Unfortunately you will not be able to see the results unless you rebuild the documentation using the [Sphinx documentation generator](#) but you are more than welcome to send in untested text fragments and we will do our best to format and include them. If you want to include screenshots, just attach them and mark the place where to include them in your text file using a restructured text directive or any other way that is understandable to an editor. We are happy about any bit of content.

Of course, proposals for new sections or documentation pages are also welcome as plain text, restructured text or any other format you are comfortable with.

13.2 Committing

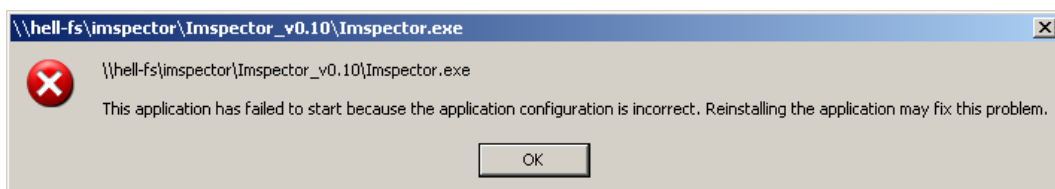
If you feel confident about your changes, open a pull request on Github. Otherwise send in modified files by email.

This is a collection of common problems people encounter when using Inspector. Some are common misconceptions or connected to bugs or shortcomings in other software / hardware, but most of them will probably be due to a bug or at least an unintuitive feature of Inspector itself. In all cases a possible workaround is suggested.

14.1 Errors During Startup

14.1.1 Inspector does not start because the ‘application configuration’ is incorrect.

If Inspector never starts and instead you see the following dialog



most probably you are missing the Visual Studio 2008 runtime. The problem is easy to fix, just download the redistributable package from Microsoft here:

[Microsoft Visual C++ 2008 SP1 Redistributable Package \(x86\)](#)

and install it on your computer. Inspector should now start.

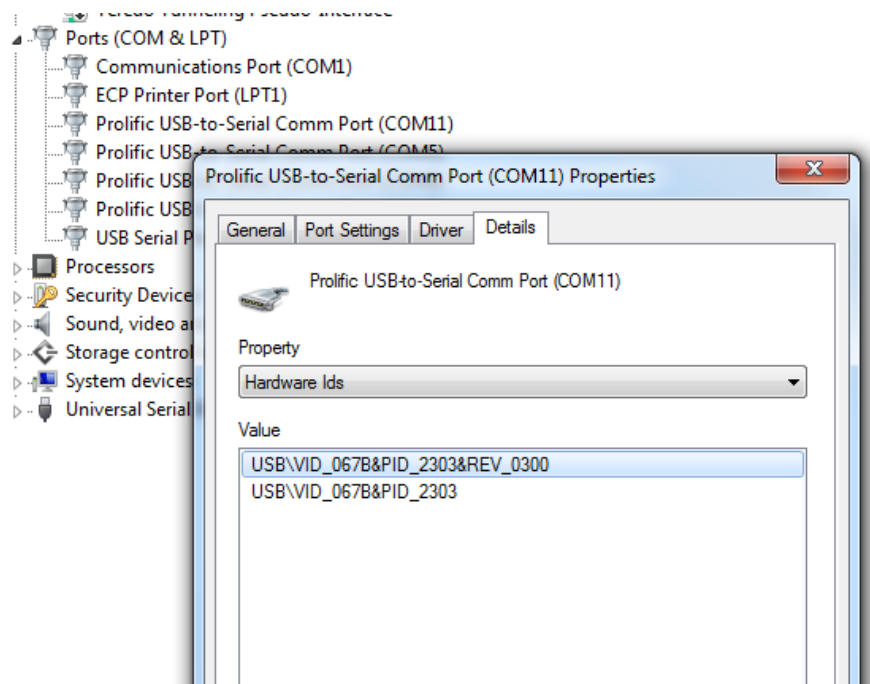
14.1.2 Registering Hardware Ports takes too long.

During startup, Inspector will take a long time before displaying ‘Registering devices’ or, in version since *v0.10rev4881*, display a message **Registering hardware ports ... [Press and hold SHIFT + ESC to cancel]** in the splash screen. The most probable cause of this is that you have some USB to serial or parallel port bridges installed that announce ports but produce errors when accessing them. The reason for the delay are long timeouts. In

older Inspector versions you have to wait and, to speed up future starts try some of the solutions below. In newer Inspector versions the problem is eliminated for serial ports, where initialization does no longer test-open the port. Thus all ports present in the Device Manager (seen by the OS) will be added to the list but if you have a bad bridge, opening them will fail when initializing a com device using them. While the problem has never been reported for parallel ports they are test-opened to find out their capabilities and thus a delay may still occur. In this case you can continue by pressing `shift + escape` and the registration will continue in the background. When you try accessing serial or parallel ports immediately after this (before initialization has finished) they may be missing from the list of available ports.

Of course it is preferable to avoid the problem altogether and often removing the bridge from the USB port and plugging it in again solves the problem temporarily. Alternatively, try to identify the offending ports by disabling them in the device manager and enabling them one by one.

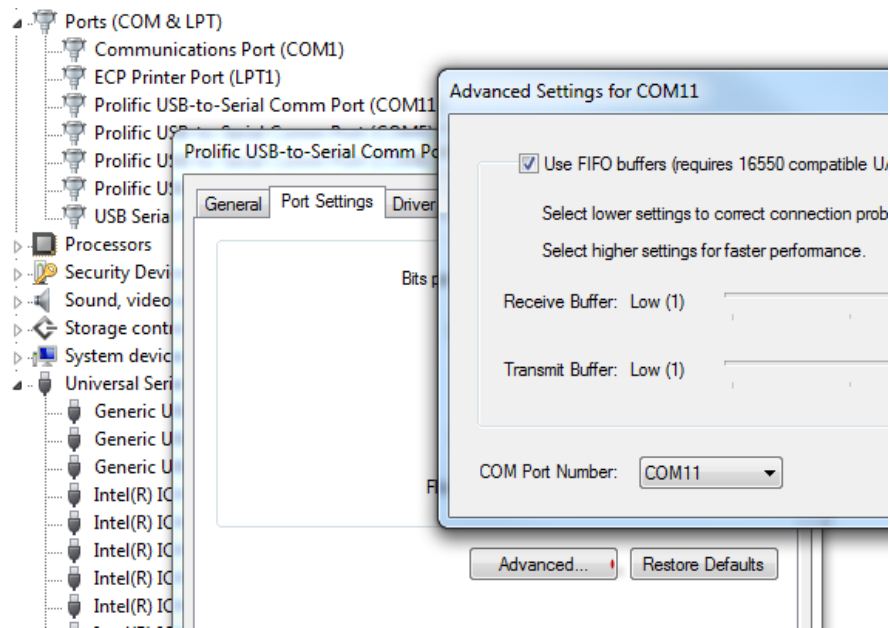
Also you should make sure that the newest drivers are installed for your device. These USB bridges are often cheaply produced and not labelled. A convenient way to identify the correct drivers is to look for the hardware ids through the device manager. Identify the device and go to the details tab. Select the **Hardware Ids** property and then google for the whole string or the ids behind **VID** and **PID** (for example 067B and 2303). Often this points you in the right direction.



14.2 Hardware Initialization

14.2.1 COM ports beyond COM9 cannot be opened.

When selecting COM ports COM10 and beyond initialization fails. This is a bug in older Inspector versions. You can either switch to a newer version or you can rename the ports in question through the device manager. For this, open port properties and click on the “Advanced Settings” button. There you may select a new port number. Often you will need to disconnect and re-connect the device afterwards to re-initialize the driver.



14.3 Measurement Configuration

14.3.1 Is it possible to time two (or more) configurations in Inspector so that one automatically starts directly after the other?

Yes it is. Create several 'Configurations' (also called 'Property Sets') in your measurement template (by pressing the STOP button, i.e. the one with the square on it). You can see available configurations in the 'Configurations' live dialog and change between them by clicking on the entries. You can then edit the settings independently for each of them. Usually this is a way to combine e.g. an overview scan and the measurement in the same template but there is an option *Measurement* → *Loop Through Property Sets* that does exactly what you want. It will run a measurement for each 'Property Set' in the order they are defined. More sophisticated control will only be available when Inspector finally has python embedded as a scripting language.

15.1 The Inspector MSR File Format

The Inspector '.msr' file format is a native binary format and now api exists to read or write it in its entirety. As window positions, data dependencies, hardware configuration and settings measurement state and configuration settings are saved in each document reading the entirety of this information is hardly desirable for other programs. The file format evolves with Inspector but old files will always remain readable with newer versions of Inspector or a conversion tool will be made available.

The data and directly associated meta-data contained in any Inspector file is organized according to the specification of the OBF (*.obf) file format and Inspector also reads OBF files in an intuitive way. There are bindings for OBF to both c++, Matlab and Java and we are working on Labview and Python bindings to allow reading of Inspector data into most applications used in scientific data analysis and to allow writing to an Inspector-readable format from Labview.

The C++ API for OBF will be rather stable as the file format evolves. The format itself is forward and backwards compatible between all versions released.

Note: The fact that OBF is compatible in both directions helps when reading a newer version .msr file with an old Inspector copy. While window positions and some analysis related data is omitted, all physical data and the associated meta-data remains readable. Just (temporally) rename the file to .obf and open it in Inspector.

Note: Some older versions of Inspector do not write OBF files and used the DBL (*.dbl) format as an exchange file format. This format is also documented below for completeness.

15.2 The OBF File Format

While it is recommended that you use the ANSI-C or C++ interface provided by the **OmasIo** dll to read and write OBF files the file structure is relatively simple and it should be straightforward to implement a reader or writer for

OBF files. A native OBF file starts with the following binary file header. Please note that all structures saved in binary format into an OBF file have packsize 1 and that all binary data is stored in little endian byte order (i.e. the byte order used on x86, x64, Itanium and Alpha platforms)

```
typedef struct _OBF_FILE_HEADER
{
    //! Must be "OMAS_BF\n" followed by 0xff 0xff
    char magic_header[10];
    //! The actual format version of the file.
    omas_UINT32 format_version;
    //! The position of the first stack header in the file
    omas_UINT64 first_stack_pos;
    //! Length of following UTF8 description (bytes).
    omas_UINT32 descr_len;
} OBF_FILE_HEADER;
```

The header is immediately followed by `descr_len` bytes of a UTF-8 string containing the file description which is preferably in xml format such that it can be parsed into a `OPropstruct` by the function `omas_xml2prop()` described in a later chapter.

From format version 2 on the file description string is followed by the meta data position

```
omas_UINT64 meta_data_position_ ;
```

The header members are

magic_header The magic header identifies compatible OBF files and no read routine should go on if it does not match.

format_version The format version is used by newer routines to read older lacking features but the file format is designed to be upwards and downwards compatible. That is, newer features and additional information is to be added such that the old routines should be able read as much information from files of a newer version as possible. This document describes format version 1.

first_stack_pos Location of the first stack header in the file. Passing this to a `seek()` function should position the file pointer to the beginning of a stack header (see below)

descr_len Length of the file description string following this header immediately in bytes (not in characters). The string is defined to be in UTF-8 encoding.

OBF is an embeddable file format. i.e. a more feature-rich file format may choose to be readable as OBF by having this file header at its beginning, writing any additional information between the header and any stacks as long as the first stack header is found where specified by `first_stack_pos`. Each stack starts with a binary header of the following structure:

```
typedef struct _OBF_STACK_HEADER
{
    //! Magic header. As long as this matches the compiled-in magic string
    //! the format is forward and backward compatible.
    //! For this version of the header the magic string is "OMAS_BF_STACK\n"
    //! followed by 0xff 0xff
    char magic_header[16]{};
    //! The version of the file format for backwards compatibility.
    omas_UINT32 format_version{};
    //! The rank of the stack.
    omas_UINT32 rank{};
    //! The number of pixels along the axes
    omas_UINT32 res[OMAS_BF_MAX_DIMENSIONS]{};
    //! The physical length of the stack axes
```

(continues on next page)

(continued from previous page)

```
double len[OMAS_BF_MAX_DIMENSIONS]{};
///! The physical offset of the stack
double off[OMAS_BF_MAX_DIMENSIONS]{};
///! The data type of the stack on disk.
omas_DT dt{};
///! The type of compression. Currently 1 for zip and 0 for none
omas_UINT32 compression_type{};
///! The compression level 0-9
omas_UINT32 compression_level{};
///! The length of the utf-8 name of the stack in bytes
omas_UINT32 name_len{};
///! The length of the utf-8 description in bytes. It should be a valid xml
///! description. However if it is not, a reading routine should still read
///! the stack and save the description as metadata containing a single
///! string.
omas_UINT32 descr_len{};
///! Do not touch! A value of 0 will indicate data-less stack to pre-versions
omas_UINT64 reserved{};
///! The length of the data on the disk. For version 6 stacks this is just the
↳ offset from the
///! end of the header plus description to the stack footer which needs to be read
↳ before reading
///! the stack.
omas_UINT64 data_len_disk{};
///! The next stack position in the file
omas_UINT64 next_stack_pos{};

} OBF_STACK_HEADER;
```

The header is immediately followed by `name_len` bytes of a UTF-8 string containing the stack name and `descr_len` bytes containing the UTF-8 encoded stack description, which is, again, preferably in xml format such that it can be parsed into a `OPropstruct` by `omas_xml2prop()`. For format versions smaller than 6, or if `num_chunk_positions` in the footer is zero, binary data follows immediately after the description and takes up exactly `data_len_disk` bytes. It may be compressed using `zlib` (see below).

newline The header members are

magic_header The magic header identifies compatible OBF stacks and a read routine should stop reading the file when it is not found at the specified position.

format_version The format version for backwards compatibility. This allows versions being set per stack if necessary. E.g. the stack footer is present only in version 1, not in version 0 and a write routine may choose to write stacks as version 0 omitting the footer and other stacks (in the same file) as version 1 including the footer. **IMPORTANT:** A reader for version `n` is allowed to read the stack of higher version as long as it found the `magic_header` and as long as it seeks to the end of the footer using its size member before reading the variably sized components.

rank The number of used dimensions. The following members are valid only up to e.g. `res[rank-1]`.

res (only the first rank members are valid) The number `res[i]` is the number of pixels the stack has along the `i`'th dimension.

len (only the first rank members are valid) The physical length along each used dimension. Units may be given as part of the dimension labels in the footer. The physical center position of the `k`*th* along the `i`*th* axis is given by `off[i] + (.5 + k)*len[i]/res[i]` where `k` runs from 0! to `\lstinline!res[i] - 1`.

off (only the first rank members are valid) The physical offset. May be used to specify relative positions of stack volumes inside a larger measurement space.

dt (see *OmasTypes.h for the actual values*) The binary data type as stored on disk. The {bf Omas} binary types are explained in detail later together with their helper routines. For the file format the constants are:

```
#define OMAS_DT_AUTO      0x00000000 // Automatically determine the data type
#define OMAS_DT_UINT8     0x00000001 // An unsigned byte
#define OMAS_DT_SINT8     0x00000002 // A signed char
#define OMAS_DT_UINT16    0x00000004 // A 16 bit word value
#define OMAS_DT_SINT16    0x00000008 // A 16 bit signed integer
#define OMAS_DT_UINT32    0x00000010 // A 32 bit unsigned integer
#define OMAS_DT_SINT32    0x00000020 // A 32 bit signed integer
#define OMAS_DT_REAL32    0x00000040 // A 32 bit floating point value (float, )
#define OMAS_DT_REAL64    0x00000080 // A 64 bit floating point value (double, )
#define OMAS_DT_RGB       0x00000400 // Byte RGB, 3 samples per pixel
#define OMAS_DT_RGBA      0x00000800 // Byte RGB, 4 samples per pixel.
#define OMAS_DT_UINT64    0x00001000 // A 64 bit unsigned integer
#define OMAS_DT_SINT64    0x00002000 // A 64 bit signed integer
#define OMAS_DT_BOOL      0x00010000 // A c++ boolean
```

Each of the numeric types has a complex counterpart by setting the following bit in dt:

```
#define OMAS_DT_COMPLEX 0x40000000 // Is set, if this is a complex array.
```

So a stack containing `std::complex<float>` values would have

```
dt = OMAS_DT_REAL32 | OMAS_DT_COMPLEX
```

compression_type The type of compression used. Currently only the values {bf 0} (no compression) and {bf 1} (ZIP compression) are supported.

compression_level The compression level used. This is whatever the library allows. For ZIP compression the levels are 0 to 9 from fastest to strongest.

name_len The length in bytes of the UTF-8 encoded stack name following this header immediately.

descr_len The length in bytes of the UTF-8 encoded stack description following the name.

reserved Out of use.

next_stack_pos Pointer to the location of the next stack header.

For stacks with `format_version >= 1` the binary data is immediately followed by the stack footer

```
//! Stack footer
typedef struct _OBF_STACK_FOOTER
{
    // VERSION 1, no footer before VERSION 1

    //! Size of the struct
    omas_UINT32 size{};

    //! Entries are != 0 for all axes that have a pixel position array following.
    omas_UINT32 has_col_positions[OMAS_BF_MAX_DIMENSIONS]{};
    //! Entries are != 0 for all axes that have a label following
    omas_UINT32 has_col_labels[OMAS_BF_MAX_DIMENSIONS]{};

    // VERSION 1A??

    //! Length of a free metadata string which has been superseded by the tag_
    ↪dictionary. It may
    //! still be there for some files of version < VERSION 4.
```

(continues on next page)

(continued from previous page)

```

///! The metadata-string immediately follows col positions and col labels.
omas_UINT32 metadata_length{};

/// VERSION 2

///! Si units of the value carried
OBF_SI_UNIT si_value{};
///! Si units of the axes
OBF_SI_UNIT si_dimensions[OMAS_BF_MAX_DIMENSIONS]{};

/// VERSION 3

///! The number of flush points
omas_UINT64 num_flush_points{};
///! The flush block size
omas_UINT64 flush_block_size{};

/// VERSION 4

///! The total length of the tag dictionary following the flush positions the
↳dictionary
///! consists of a number of entries of the following type ending with zero unit32:
///! <len of key [uint32]><key><len of val [uint32]><val>
omas_UINT64 tag_dictionary_length{};

/// VERSION 5

///! Where on disk all the meta-data ends. This is only important for formats that
↳read a
///! container in which OBF resides as a file format and for removing extra space
↳in the
///! compacting routine
omas_UINT64 stack_end_disk{};

///! Should always be 1, as we want forward- and backwards compatibility but should
↳still be
///! honored in readers as an emergency break for the future.
///! If we break forward compatibility this will be noted here and a new number !=
↳1 will be
///! named in the comment
omas_UINT32 min_format_version{};

/// VERSION 5a ("blind version increase due to bioformats")

///! The position where the stack ends on disk. The space between stack_end_disk and
///! stack_end_used_disk is unused and not allowed to be used by anything as it can
↳be
///! cleared at any time
omas_UINT64 stack_end_used_disk{};

/// VERSION 6

///! The total number of samples available on disk. By convention all remaining
↳data is
///! assumed to be zero or undefined. If this is less than the data contained of
↳the stack
///! it is safe to assume that the stack was truncated by ending the measurement
↳early.

```

(continues on next page)

(continued from previous page)

```

///! If 0, the number of samples written is the one expected from the stack size.
omas_UINT64 samples_written{ 0 };

///! The number of chunk positions in V6 chunk wise writing
omas_UINT64 num_chunk_positions{ 0 };

} OBF_STACK_FOOTER;

```

where the OBF_SI_UNIT structure is defined as follows:

```

///! A fraction, ideally should be reduced when writing to file
typedef struct _OBF_SI_FRACTION
{
    omas_SINT32 numerator;
    omas_SINT32 denominator;
} OBF_SI_FRACTION;

///! The dimensions and scaling factor of an SI unit. For each of th base and_
→supplemental
///! units the exponent is saved as a fraction.
///! Ordering for the exponents array is as follows:
///! exponents[0]: Meters (M)
///! exponents[1]: Kilograms (KG)
///! exponents[2]: Seconds (S)
///! exponents[3]: Amperes (A)
///! exponents[4]: Kelvin (K)
///! exponents[5]: Moles (MOL)
///! exponents[6]: Candela (CD)
///! exponents[7]: Radian (R)
///! exponents[8]: Steradian (SR)
typedef struct _OBF_SI_UNIT
{
    OBF_SI_FRACTION exponents[9];
    double scalefactor;
} OBF_SI_UNIT;

```

The footer contains additional meta-information that is too large to be saved as a string and/or is to be made available without the need for xml parsing. Future versions of the footer may become larger so a read routine should always read the known members and then seek to a position `footer.size` bytes after its beginning before starting to read the variable sized parts of the meta information. In detail: The header members are

size The size of this structure on disk. Read the known part of the structure and discard the `footer.size - sizeof(OBF_STACK_FOOTER)` bytes. This allows a reader written for a lower version to read stacks of a higher version simply omitting meta-data that has been added to the new version. In case breaking changes will be introduced, the magic header will be changed and the changes will be described in this document.

has_col_positions For those dimensions for which `has_col_positions[i] != 0` an array of `res[i]` (64bit) double values is appended after the label strings (see below) which signify the position of the column along its axis. If present the `len!` and `\lstineline!off` should be ignored in favor of the position values.

has_col_labels For those dimensions for which `has_col_labels[i] != 0` an array of `res[i]` label strings is appended after the column position arrays. Each label string starts in the form `(omas_UINT32)n:char[n]` where `n` is the length of the string. It is thus read out by reading a 32bit integer `n` and then reading `n` bytes forming an UTF-8 encoded string.

metadata_length Immediately after the label strings a block of memory is appended which is a string in UTF-8

format which contains meta-data interpreted on a higher level in the OmasIo xml format for properties described elsewhere. This entails e.g. the stack position and orientation in a global coordinate system etc. As it becomes important some of it may find its way into the obf specification appended to the header in a binary format. While you can use this field for your own meta-data this is not encouraged. The field is intended to be filled in a standard way that OBF readers may or may not read. Custom meta-data should be saved in the file and stack description fields, preferably also in UTF-8 xml(see below). Nevertheless, readers should not throw or report a fatal error when they do not understand the data contained in this field - they should issue a warning.

si_value For The SI units of the stack values.

si_dimensions The SI units of the stack axes.

num_flush_points For zip compressed stacks this is the number of full flush points the zlib compression has created for fast seeking. The flush point positions relative to the beginning of the zlib compressed data follow immediately after the meta data as an array `omas_UINT64 flush_positions[num_flush_points]`. When uncompressing only a window of the stack starting at `pos` the inflator may start decompressing data at the disk position `flush_positions[n]` where `n` is the largest integer with `flush_block_size*n <= pos`. Please note that there is no ZLIB header written at that position, so the inflator needs to be initialized in 'raw' format i.e. `inflateInit2(h, -15)` needs to be called in zlib.

flush_block_size The number of (uncompressed) bytes between full flush points. See above.

tag_dictionary_length The number of bytes contained in the tag dictionary that follows the flush points. The tag dictionary is organized as in the way outlined above. key names are Utf8 encoded. Most key values are xml as the description. The 'inspector' tag e.g. provides all inspector related meta-data.

stack_end_disk See comment.

min_format_version The minimum format version needed to successfully read this stack.

stack_end_used_disk See comment.

samples_written If smaller than the expected amount of samples, the stack was truncated at this point. Remaining data is zero by default (unless you want to show uninitialized data differently).

num_chunk_positions Chunk positions following the tag dictionary when stacks have been written interleaved. Each chunk position is a pair of 64bit unsigned integers:

The footer is immediately followed by `rank` label strings (encoded in the same form as the column labels) which are in turn followed by the column positions, column labels, meta data and flush positions, tag dictionary and chunk positions as outlined above.

If `num_chunk_positions != 0`, the data is not written continuously but interleaved with other stacks or other file content. It is organized as follows:

Let `start_pos` be the position following the stack header description. The data is then organized in chunks starting at `start_pos` and the `file_offset` positions of the chunks. The size of each chunk is the difference between its `logical_offset` and the `logical_offset` of the next chunk. The logical and file offset of the first (unlisted) chunk is zero, i.e. it starts at `start_pos` and its length is the logical offset of the first listed chunk. The length of the last chunk is `samples_written` minus its logical offset.

If several segments have the same first value it means that only the last is non-empty. Pseudo code to read all data:

```
struct ChunkPosition
{
    omas_UINT64 logical_offset;
    omas_UINT64 file_offset;
}

std::vector<ChunkPosition> chunk_pos = chunk_positions_read_from_footer();
```

(continues on next page)

(continued from previous page)

```
std::uint64_t pos = 0;
file.seek(start_pos);
std::size_t idx = 0;

while(pos < footer.samples_written)
{
    auto bytes_to_read = footer.samples_written - pos;
    std::uint64_t seek_pos = -1;

    if (idx < footer.num_chunk_positions)
    {
        if (pos + bytes_to_read > chunk_pos[idx].logical_offset)
        {
            bytes_to_read = chunk_pos[idx].logical_offset - pos;
            seek_pos = chunk_pos[idx].file_offset + start_pos;
            idx++;
        }
    }

    if (bytes_to_read > 0)
    {
        read_from_file(file, bytes_to_read);
    }
    if (seek_pos != -1)
    {
        file.seek(seek_pos);
    }
    pos += bytes_to_read;
}
```

Note: Backwards and forward compatibility: As outlined above, OBF files are designed to be backwards and forward compatible. However, version 6 files where `samples_written` is different from the total amount of samples in the stack or where `num_chunk_points` is not zero break forward compatibility of old readers. Readers that do not check `min_format_version` will fail miserably, those that do should disregard the stack and notify the user.

Please note that `min_format_version` is the minimum format version that needs to be implemented to successfully read all the information from the file this version knows about. The footer and the data that follows are allowed to grow in future versions without this value changing. You may want to notify the user if the stack or file format have gone up as information might be available but disregarded. Make sure to use the `size` member of the footer to jump to the variable sized data section.

Note: SI units While simply writing SI units as a string in a certain format would have been simpler and would have allowed to display the units directly in a simple reader (and have them written more easily after user input) this format was chosen as it allows bindings to existing units implementations i.e. in C/C++, Python and Matlab more easily.

For C/C++ OmasIo contains a simple formatter and parser for unit strings into this format.

15.3 The DBL File Format (*deprecated*)

The DBL format is a simple binary file containing a single up to four-dimensional data stack with some header information about physical dimensions of the sampled a volume. The header is exactly 128 bytes long

```
unsigned char header[128];
```

For historical reasons it has an mixed little endian and big endian format. The rank of the stack is not explicitly contained but the pixel number of higher dimensions are simply set to 1. The number of pixels along the four possible dimensions are given by

```
res[2] = header[0]*256 + header[1];
res[1] = header[2]*256 + header[3];
res[0] = header[4]*256 + header[5];
res[3] = header[6]*256 + header[7];
```

The physical length is

```
len[2] = *((float *) (header + 8));
len[1] = *((float *) (header + 12));
len[0] = *((float *) (header + 16));
len[3] = *((float *) (header + 20));
```

where the floats are stored in little endian format. Reading on big endian machines involves flipping bytes before casting to float. The header is followed by floating point data in little endian byte order. If `header[24] == 1` it is 32bit floating point (float), otherwise it is 64bit floating point data (double).

15.4 The Omaslo API, Bindings

The **OmasIo** library implements the OBF and DBL file format providing both a C++ interface to OBF files. There are bindings using the C++ implementation for both Matlab and Python and in addition, a pure Java implementation of a reader is in the process of becoming part of [BioFormats](#).

Note: All .msr files written by Inspector conform to the OBF specification. Additional information is stored between the stacks and before the first stack but any .msr file (except for very, very old ones) should be readable by a correctly implemented OBF reader.

Vice versa, Inspector reads .obf files. Because OBF is forward and backwards compatible this comes in handy when opening .msr files from newer versions of Inspector (as .msr is not forward compatible).

15.5 Meta information data model

Strictly speaking, the OBF file format does not specify the way meta information is to be associated with the file or data stacks within and because it can be embedded into arbitrary, more complex formats it even encourages the use of methods suitable for the task at hand.

For meta information that is to be shared by several applications it is however strongly encouraged that meta-information is saved as UTF-8 text in the file or stack description, preferably formatted as xml in a way compatible to the output of the `omas_export_xml()` function in the *OmasIo* library, described in a separate section. In C++, the easiest way to do this is to write the meta information into an `OProp` object and actually use the `omas_export_xml()` function to convert it to an xml string. For Matlab and Python, toolboxes are provided that can convert a (complex) variable into a compatible xml string and back. In fact these toolboxes, too first map the data into an `OProp` variable and then export it to xml and vice versa.

The `OProp` data model is strongly based on the Matlab data model. Data is organized in arrays of arbitrary numeric complex or real data type and arbitrary rank (with the special case of a scalar, which is a 1x1 array in Matlab), cell

arrays (where each cell can contain data of a different type), structs (where each member is addressed by its name and can contain arbitrary data) and arrays of structs (with identical fields). In Matlab strings are one-dimensional character arrays. Matlab string arrays therefore will always contain strings of equal length (with shorter strings simply padded by NULL bytes). The toolbox will convert these to cell arrays of strings tagged with a special flag. On the C++ side they will look like cell arrays of strings but as long as the tag is untouched they will be converted back to string arrays on the Matlab side. Please note that usually it is preferable to use cell arrays of strings on the Matlab side to start with. Also, `OProp` knows empty 'cells' (an `OProp` with no content) which is mapped to an empty 'double' array in Matlab. There are similar mapping issues with other bindings like Python. The general Ansatz is that variables converted to xml by one language binding will produce the same variable when read back directly but that there is no guarantee that this applies once a property tree has been converted back and forth between different languages.

16.1 A little KB

There is no comprehensive guide to the data model, the SDK and the class hierarchy in Inspector and most probably this will only become available after adding python scripting support. For now, there is a small collection of topics that we stumble upon. The following information may be incomplete, inaccurate or plain wrong depending on when it was added and what was changed afterwards.

16.1.1 SpecDll versions, revisions and module revisions

The Inspector solution and/or SDK contains a few python scripts and associated batch files that add the current subversion revision of various modules to header files as defines. Currently the rationale is as follows: A plugin's file where the `CWinApp::InitInstance` method is implemented imports either `<SpecInitialize.h>`:

```
#include "rev.h"
#include <SpecInitialize.h>
```

where `<SpecInitialize.h>` looks like this:

```
#ifndef SPECDLL_VERSION
    // The SpecDll Version. This would change with branches. Possibly
    // we have it consistent with the name.
    // The rule is: if this: if this changes, the API definitely breaks and the
    // plugin will not be loaded.
#    ifndef SPECDLL_REVISION
#        include <SpecRevision.h>
#    endif
#    ifndef MODULE_REVISION
#        define MODULE_REVISION 0
#    endif
#endif
```

(continues on next page)

(continued from previous page)

```
#    include <SPECInit.h>
#endif
```

where SPECInit.h defines the functions

```
#ifndef SPECDLL_VERSION
    // The SpecDll Version. This would change with branches. Possibly
    // we have it consistent with the name.
    // The rule is: if this: if this changes, the API definitely breaks and the
    // plugin will not be loaded.
#    define SPECDLL_VERSION (0x0010 | (SPECDLL_REVISION << 16))

    ///! Call this function to add the resource handle of the dll to the chain
    ///! in your plugin or application.
    ///! SpecDll_Init() will fail if your version is larger than that of
    ///! the loaded dll or if your major version is not binary compatible
    ///! with the dll version loaded.
    SPEC_EXT_CLASS BOOL SpecDll_Init(
        DWORD winnt = _WIN32_WINNT,
        DWORD specver = SPECDLL_VERSION,
        LPCTSTR modname = __MODULE__,
        DWORD modrev = MODULE_REVISION);

    ///! One simple line for the lazy ones. Put this into InitInstance() of
    ///! your dll's CWinApp object.
    #define SpecDll_Initialize() \
        if (! SpecDll_Init()) return FALSE;

    ///! Get the version of the dll actually loaded in memory. You can
    ///! use SPECDLL_VERSION in your code anywhere for the version you
    ///! compiled against. e.g. SPECDLL_VERSION == SpecDll_GetVersion()
    ///! is TRUE if they are the same.
    SPEC_EXT_CLASS DWORD SpecDll_GetVersion();

    ///! Get the human-readable form of the version
    SPEC_EXT_CLASS LPCTSTR SpecDll_GetVersionName(DWORD specver);
#endif
```

The file <SpecRevision.h> includes a define for SPECDLL_REVISION which is the last revision when any relevant code of SpecDll including SpecLib was updated. This importantly includes all the helper libraries, too. This revision is likely to mark a change in the way the library behaves. The file SpecRevision.h is written by the WriteSpecRev.bat script which does NOT FAIL if there is no valid python installation accessible. In this case the file remains unchanged. The SPECDLL_VERSION should manually be updated any time the header files change, i.e. when the binary compatibility is sure to break. The above section is subject to change.

A plugin simply calling SpecDll_Initialize() will thus tell the library at runtime against which revision of it it has been compiled but it will not tell which revision of the repository it was compiled from itself.

Alternatively a plugin can call 'WriteRev.bat' as a pre-build step. If there is a valid python configuration it will write a file "rev.h" including the current SpecDll revision (the same as written in <SpecRevision.h>) and a definition for MODULE_REVISION which is passed into SpecDll at runtime. Instead of including <SpecInitialize.h> such a plugin should then

```
#include "rev.h"
```

Why so complicated? Well, the problem is that we may well change some of the cpp code in SpecDll and therefore the revision of SpecDll will change but no re-compile of the plugins is necessary. If the plugin would include

<SpecRevision.h> or includes <SpecInitialize.h> it will then be re-compiled due to the change in <SpecRevision.h>. This is unacceptable in a development cycle. “rev.h” on the other hand will change only when the pre-build step for the module is triggered, i.e. when the build system has in fact determined that the module needs updating. It will then ensure that the correct value for SPEC_DLL_REVISION and MODULE_REVISION is put in. Please note that MODULE_REVISION is NOT the last changed revision but the revision of the whole repository at compile time.

Inspector treats a module compiled against a different version of SpecDll as an error but if the SPEC_DLL_REVISION of a module (revision when module was compiled) and that of SpecDll (revision when SpecDll was compiled) are different only a warning is logged to the log file.

16.1.2 Exception Handling and Crash Reporting

At the moment all code is compiled in Microsoft C++ with the /EHsc option. That is, C++ exceptions are of course enabled but win32 exceptions are not handled by the catch(...) block and clean-up code is not compiled into any try-block that can throw a C++ exception. Whether or not clean-up code is compiled into it and the stack is properly unwound may depend on compiler optimization. There is an excellent article at <http://www.thunderguy.com/semicolon/2002/08/15/visual-c-exception-handling/> about this. In short there are three solutions to the problem

1. Compile with /EHs or /EHsc. Your win32 exceptions will end up in a __catch(translator) {} block if you have it. the stack will not be unwound and from there you can do whatever you want. (Mostly you can pass the exception up or you can enter the handler. Continuing execution is not really an option for most exceptions.
2. Compile with /EHa and make sure you have catch(...) handlers wherever needed to ensure proper stack unwinding (that means that you need a try .. catch wherever a win32 exception is expected).
3. Compile with /EHa and throw a C++ exception in your default translator. This does not play well when compiling with /EHs because the compiler does not know that functions without a throw() statement can throw c++ exceptions and will omit the clean-up code and the handler in these functions.

Currently we have option 3 for handling unexpected exceptions but not with the necessary /EHa compiler option (at least not for all modules). The reason why this is OK IMHO is that we do not set the default translator but rather SetUnhandledExceptionFilter() which will be called only if there is not appropriate __catch() statement. Therefore whenever the filter is called a serious occurred and we will live with the stack not being properly unwound before we reach our error handling code. In fact this may even be good as we don't know whether the program is in some bad condition. In addition there is a problem as MFC includes some exception handling in the windows callback functions:

On 64bit systems running a 32bit application the process will usually swallow exceptions behind a window callback. This is a windows bug which may stay in place because it has been there for a long time. See [Microsoft KB 976038](#) and [this forum post](#) for details.

Calling this function will enable the hotfix and cause an application error dialog to appear instead. Importantly the exception is still not propagated upwards, i.e. there is no stack unwinding to a possible exception handler before the callback. I have still to test whether this is true for all exceptions and whether it depends on the exception model we compile with. The behaviour of the hotfix is to enter an exception handler that ultimately crashes the program.

For some applications, enabling the hotfix is not enough. They rely on being able to catch the exception and do e.g. some data recovery before exiting or they roll their own diagnostics. There is no known way to make the exception go past the callback but with this function you can wrap the default windows procedure in MFC with a procedure that handles exceptions. The easiest way to do this is shown in the comment below.

Note: Subclassing in MFC is a little complicated as there is both the windows procedures (and the default MFC windows procedure calls the MFC CWnd::WindowProc implementation which can be overwritten). So basically MFC windows can be subclassed by (1) overwriting the virtual WindowProc function, (2) adding entries to the message map handled by the standard implementation and (3) subclassing in the windows sense, that is by calling SetWindowLong(). We do not want to call SetWindowLong on each and every window, maintain a map of previous functions etc. Rather

we replace the function pointer that is used by `_AfxCbtFilterHook()` to set the subclass the window making it call some windows procedure that will actually call the member function.

Note: This explanation, even the way MFC handles this may change and then the code may stop working. If that happens, find out where the new MFC version calls `SetWindowLong()` to subclass the windows and replace the pointer it uses by the installed windows procedure.

Note: One should also look at context switching. Exceptions occurring behind a context switch will be caught and rethrown, obscuring their origin. In order to avoid this, call `AfxSetAmbientActCtx(FALSE)`; in every `InitInstance()` function.

See [this forum post](#) for details.

Todo: Removed outdated Imspector changelog. If needed, copy another more updated changelog automatically and include it.

- [search](#)

Compiled: Nov 09, 2019