

LAMBDA

LARGE AREA MEDIPIX3 BASED DETECTOR ARRAY

60k/250k

User Manual



LAMBDA 60k/250k User Manual

Version: 0.8

Please read this manual carefully before operating the LAMBDA detector system.

If you have any questions concerning the system, please contact us:

Address: **X-Spectrum GmbH**
 Notkestr. 85
 22607 Hamburg
 Germany

Phone: +49-40-8998 3959

E-Mail: info@x-spectrum.de

Internet: www.x-spectrum.de

CAUTION

Protective Cover

The sensitive area of the detector has a thin carbon fibre cover to protect it from accidental damage without preventing X-rays from reaching the sensor. When moving or operating the detector, avoid touching this cover.



Figure 1: The Lambda detector. The black region is the carbon fibre cover.

During operation, the ventilation slits located on the side of the detector should not be covered.

Table of Contents

1. Technical information	6
1.1. Technical specifications	6
1.2. System overview	7
1.3. Mechanical dimensions	9
1.4. Detector Cooling	11
1.5. Electrical connections	12
1.6. Connection to control PC	14
1.7. Power supply	15
1.8. Pixel layout	16
2. Software control	17
2.1. Control PC	18
2.2. The TANGO control system	19
2.3. Starting and stopping the LAMBDA software	20
2.4. Tango commands and attributes and the "Jive" user interface	21
2.5. Detector command list (pull-down menu)	23
2.6. Detector attributes (i.e. acquisition settings) list	24
2.7. Live viewing	27
2.8. Hardware Trigger	28
2.9. Saving files during the acquisition	30
2.10. Tango and Python scripting	31
3. HDF5 / Nexus file format	34
3.1. HDF5 / Nexus file browsing	35
Pixel Mask	36
3.2. The "nexpy" image viewer tool	37
Using nexpy	38
3.3. Typical HDF5 file reading methods	42
PYTHON	42
IDL	43
MATLAB	44
C++	44
4. Appendices	45
4.1. Quick reference guide:	45
4.2. Troubleshooting guide	46
4.3. The Python Interface	48

pyxsp	48
lambdacontrol	50
4.4. Detailed Operating Mode Behaviour	53

1. Technical information

1.1. Technical specifications

LAMBDA is a single-photon-counting X-ray detector with 55 μm pixel size and high-frame-rate capability (2000 fps). The specifications below are for 60k / 250k modules with Si sensors.

	60k	250k
Number of modules	1 module with 1 Si sensor bonded to 1 readout chip	1 module with 1 Si sensor bonded to 4 readout chips
Sensor	Si photodiode array	
Sensor thickness	300 μm	
Quantum efficiency	95% efficiency at 8 keV, 70% at 12 keV, 10% at 25 keV	
Readout chip	Medipix3RXv2	
Pixel size	55 x 55 μm^2	
Sensor size	14.1 x 14.1 mm ²	28.2 x 28.2 mm ²
Format	256 x 256 pixels (65536)	512 x 512 pixels (262144)
Dynamic range	24 bits maximum (dependent on readout mode)	
Count rate per pixel	200,000 counts / pixel / s (without count rate correction) 800,000 counts / pixel / s (if count rate correction measured and applied)	
Energy range	6 keV – 25 keV*	
Adjustable threshold range	3 – 40 keV	
Energy resolution	1 keV	
Max. frame rate	2000 Hz (12-bit mode)	
Readout time	No readout time in 12-bit mode, 1 ms in 24-bit mode and dual threshold mode	
Point spread function	1 pixel FWHM	
Data format	Hdf5 (Nexus standard)	
External trigger / gate	3.3V TTL (low-voltage TTL)	
Software interface	Open source hardware library w. Python bindings TANGO drivers available	
Cooling	Air-cooled, water-cooled	
Dimensions	150.5 mm long, 85 mm wide and 40 mm high	
Weight	0.9 kg	
Overvoltage category	II	
Pollution degree	II	

1.2. System overview

The silicon LAMBDA system is specifically designed for synchrotron experiments at moderate X-ray energies (6-20 keV) requiring high spatial resolution, high sensitivity and extremely high speed. This makes it well-suited to applications such as XPCS (X-ray Photon Correlation Spectroscopy), time-resolved measurements, ptychography, SAXS (Small Angle X-ray Scattering) and imaging. LAMBDA is based on the Medipix3 readout chip, developed at CERN. By using single-photon-counting circuitry, this provides effectively noise-free operation, which is particularly critical for achieving a high image quality during fast measurements and discriminating against fluorescence. Thanks to CERN's expertise in microelectronics, this photon-counting feature is combined with a small pixel size (55 μm) for high-resolution imaging, and flexible in-pixel circuitry.

The standard LAMBDA system can be operated in two modes. In continuous read-write mode, the detector can take images at up to 2000 images per second with no time gap between images and 12-bit counter depth. This can allow efficient measurement at high speeds. In 24-bit mode, images are taken with 24-bit depth, allowing a signal range from 0 to 16 million photon hits per pixel within a single image. This mode requires a delay of 1ms between images, making it suitable for lower-speed experiments.

The LAMBDA readout electronics, developed at DESY, make it possible to read out large detector modules at 2000 frames per second rate using high-speed optical links. The detector unit is provided with a compact power supply and requires only air cooling, making it convenient to mount on movable stages. The detector can be externally triggered during operation.

During operation, the LAMBDA system is controlled by a server PC, which processes and stores images received by the detector. The detector can be controlled and monitored at the beamline using the Tango control system. Images are saved using the HDF5 format, which saves an entire image series to a single file along with image metadata. This approach makes it possible to perform high-speed imaging without creating excessive numbers of files. The file can then be accessed by standard functions in C, Python, MATLAB, IDL and other languages.

Further information on the LAMBDA technology can be found in the following references:

Pennicard, D., et al. "The LAMBDA photon-counting pixel detector." *Journal of Physics: Conference Series*. Vol. 425. No. 6. IOP Publishing, 2013. doi:10.1088/1742-6596/425/6/062010 <http://iopscience.iop.org/1742-6596/425/6/062010>

D. Pennicard et al., "High-speed readout of high-Z pixel detectors with the LAMBDA detector", JINST 9 C12014, 2014. doi:10.1088/1748-0221/9/12/C12014 <http://iopscience.iop.org/1748-0221/9/12/C12014>

D Pennicard et al., "The LAMBDA photon-counting pixel detector and high-Z sensor development", JINST 9, C12026, 2014. doi:10.1088/1748-0221/9/12/C12026 <http://iopscience.iop.org/1748-0221/9/12/C12026>

1.3. Mechanical dimensions

The drawings below show the dimensions, mounting hole positions and sensor position for the detector. There are four screw holes on the base plate for mounting (M4 size).

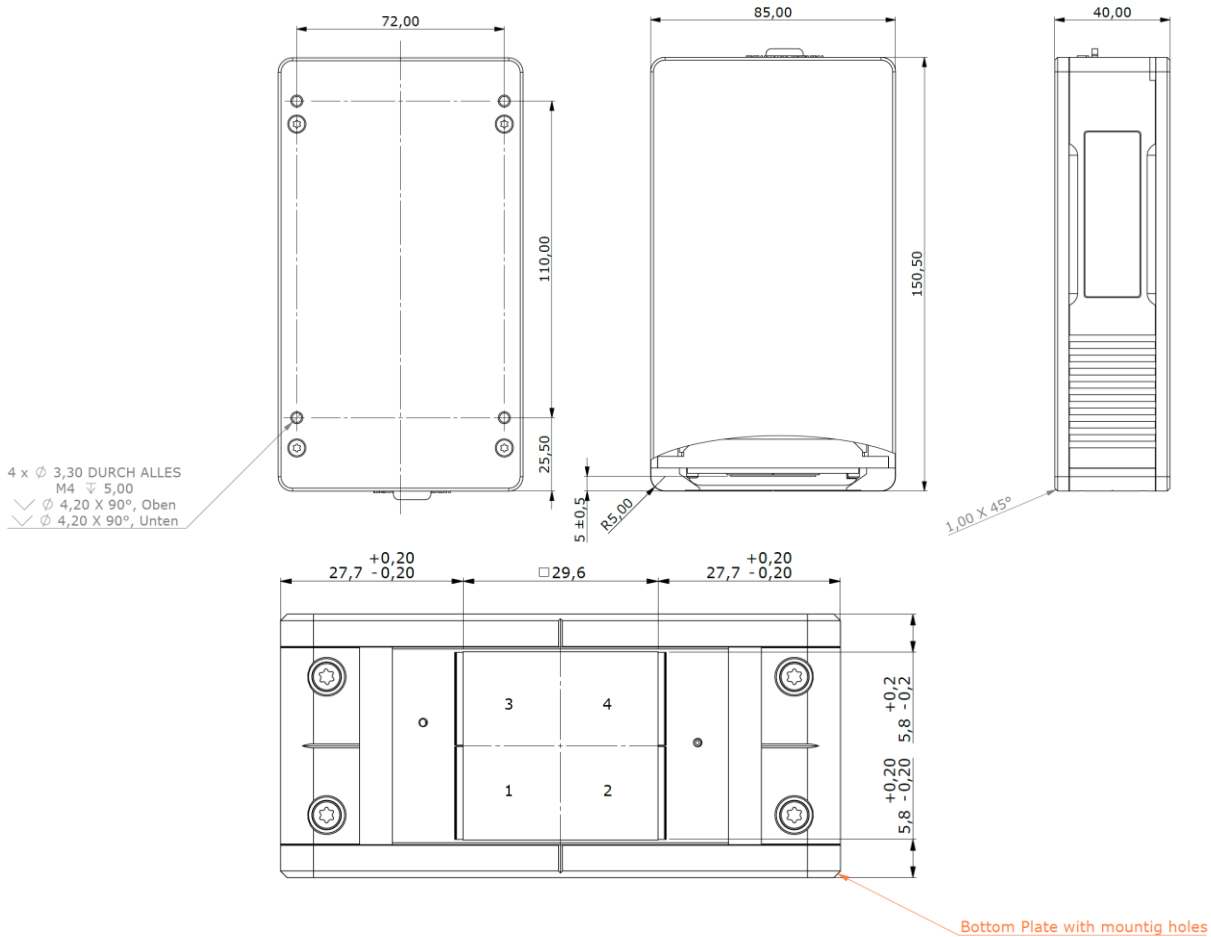


Figure 2: Technical drawing of LAMBDA 60K housing

Please note that the position of the sensor (sensitive surface) differs from model to model. In a 60k system the light-sensitive sensor is mounted in position 3 as standard (Single chips can also be mounted in position 2 upon request), whereas in a 250k system all four chip positions are covered. The centre of the chip for a 250k system is indicated by a pair of crosslines on the front of the detector.

The firmware is installed on a Micro SD card in the detector. To swap the card (e.g. to update firmware) the card can be accessed by unscrewing the side panel, as shown below.



Figure 3: Micro SD card access

1.4. Detector Cooling

The detector has two built-in fans, which can be disconnected from power if operation with minimal vibrations is desired. In this case, the detector can be cooled an external water-cooling supply. The water-cooling connectors are located on the back of the detector and covered by a rubber protective cover, as shown in Figure 4. These connectors take 6 mm hose pipe, polyurethane or similar. We recommend using a 50% glycol/water mixture, with minimum flow rate of 0.035 l/min at a maximum pressure of 2 bar. Important: The connectors are not self-sealing. Therefore, the system should be flushed with compressed air to remove any remaining liquid before disconnecting the piping.

1.5. Electrical connections

The module's connectors are on the back, and are shown below:

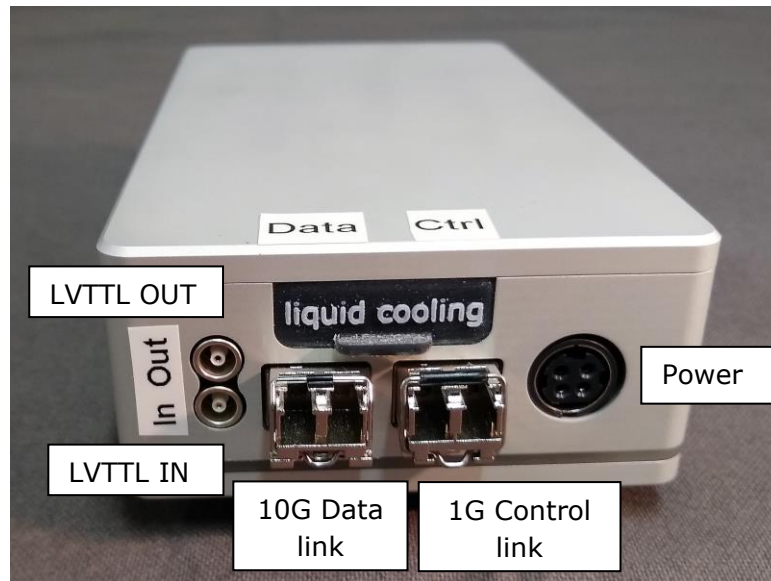


Figure 4: LAMBDA connectors

1G optical – optical patch cable connected via 1G SFP transceivers. These should be duplex multimode cables with 2 x LC duplex connectors (see Figure 5). They should be compatible with 10G Ethernet, e.g. OM3 fibre (multimode, 50/125 μm).

10G optical – optical patch cable connected via 10G SFP transceivers. These should be duplex multimode cables with 2 x LC duplex connectors (see Figure 5). They should be compatible with 10G Ethernet, e.g. OM3 fibre (multimode, 50/125 μm).

Power (12V) – This should be connected to the power supply provided with the detector.

Trigger in and out – Trigger in takes a TTL signal (3.3 V) via a LEMO input. Likewise, trigger out provides a TTL output signal. The behaviour of trigger in and out depends on the trigger mode selected, as described in the Hardware Trigger section of the manual on page 28 and the **Fehler! Verweisquelle konnte nicht gefunden werden.** appendix on page **Fehler! Textmarke nicht definiert..**



Figure 5: LC duplex optical connector

1.6. Connection to control PC

The back of the control PC is labelled to indicate the following connections (see Figure 6):

- Ethernet cable to network
- 1G Ethernet fibreoptic cable to detector
- 10G Ethernet fibreoptic cable to detector

Please note that the configuration files for the LAMBDA detector specify the source and destination IP and MAC addresses for the 10G and 1G optical links. This means that (for a given configuration) these links are not interchangeable – each fibre from the detector must be plugged into a specific network interface on the control PC. Likewise, for control over the 1G link, the detector has a fixed IP address set by the firmware (typically 169.254.1.2), and the 1G link on the computer must be configured to use the same subnet.

Please note that the correct assignment of the plugs on the control PC is labelled on them. It might look a little different on each PC. In case of doubt, always follow the labels on the PC.



Figure 6: Connections on back of control PC

1.7. Power supply

The detector is powered with 12 V DC. A laptop-style power supply is provided with the detector. This power supply is designed to work with 90-260 V AC at frequencies of 47-63 Hz.

1.8. Pixel layout

A LAMBDA module of a 250k system is composed of 4 Medipix3 chips in a 2 by 2 layout, connected to 1 sensor, whereas in a 60k system only one of the four chip positions is covered. As a result, a 250k detector does not have uniform 55 μm pixels across its full area; instead, gaps between the Medipix3 chips are bridged by larger pixels (165 μm). This is shown in Figure 7.

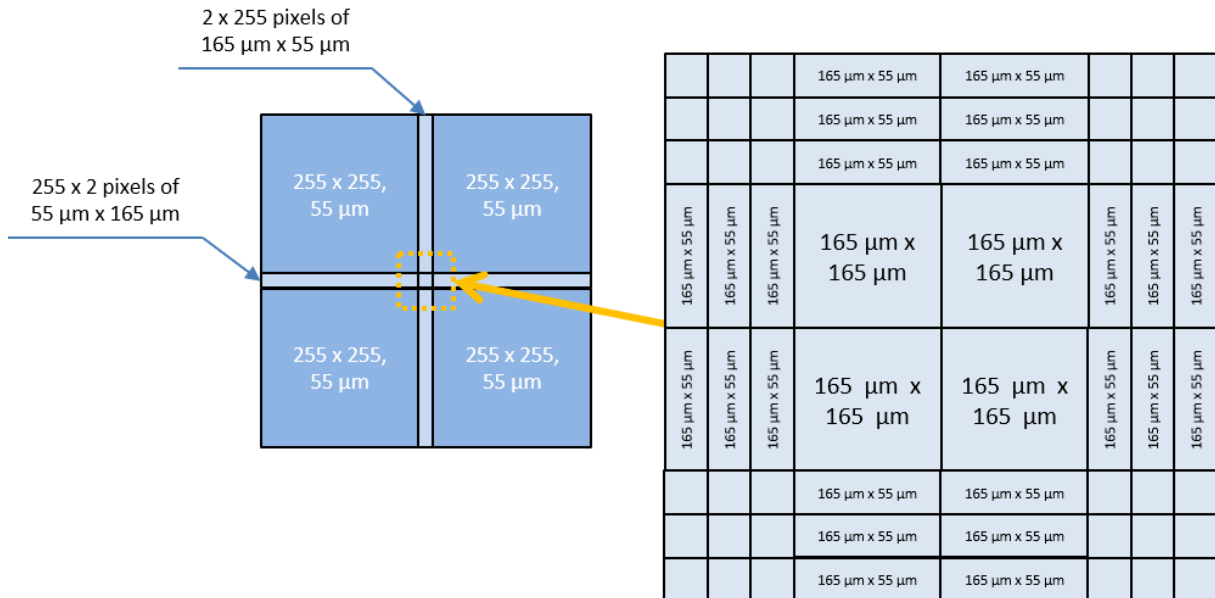


Figure 7: 250k detector pixel layout

Images saved by the LAMBDA software deal with these issues as follows:

- Larger pixels are interpolated to 3 or 9 normal-sized pixels as appropriate, with the pixel value being divided by the relative pixel area. For example, a 165 μm x 55 μm pixel with 100 counts will be interpolated to three 55 μm x 55 μm pixels each with 33 counts. There is one exception: if a pixel is at the maximum value (i.e. 4095 in 12-bit readout mode) the interpolated pixels will also be at this value, to indicate saturation.

2. Software control

The detector is sold with a control PC. There are two systems installed on the PC for controlling the detector. Firstly, there is detector software based on the “Tango” control system. This provides various ways of running the detector, including a GUI and Python scripting. We recommend the GUI as a starting point when getting familiar with the detector. Secondly, there are Python bindings that allow the detector to be run without using Tango. This approach may be helpful for integrating the detector into other control systems, and is described in the appendix.

2.1. Control PC

The detector control PC runs Debian 8. A user account has been set up on the PC, the details of which can be found on the "LAMBDA Server" document enclosed.

This account has sudo rights to allow administrative tasks.

The primary network interface uses DHCP to get an IP address. If you need to change this to a static IP address, edit the file `/etc/NetworkManager/system-connections/Wired\ connection\ 1`, or use the tools `nmcli` or `nmtui` to interactively change the settings.

Since Tango requires a valid IP address on the primary network interface, even if the link is down, a static IP address alias `192.168.0.1` has been added for `eth0`.

2.2. The TANGO control system

One way of using the LAMBDA detector is with the Tango control system:

<http://www.tango-controls.org/>

The Tango control system is open-source, and is widely used at synchrotron beamlines. It provides a variety of ways to control the detector, for example via a GUI or using scripting in Python.

The control libraries for the LAMBDA detector are written in C++, and are open-source, so they can be built into alternative control systems such as EPICS.

This section describes detector control within Tango. Most of the functionality described in this section corresponds fairly directly to detector control functions available within the LAMBDA control libraries. However, the user interface and file writing in Nexus format are provided by Tango. The source code for the LAMBDA Tango device server is also open-source.

The source code is available as follows.

Libraries for LAMBDA detector: Two libraries are needed; one LAMBDA-specific, and the other containing common detector functionality developed by DESY. The download "detsoftware" below contains scripts for building the libraries from the source code.

<https://stash.desy.de/scm/fsdsdet/detsoftware.git>

<https://stash.desy.de/scm/fsdsdet/liblambda.git>

<https://stash.desy.de/scm/fsdsdet/libfsdetcore.git>

Libraries for Tango server:

<https://stash.desy.de/projects/TAN/repos/lambda/browse>

2.3. Starting and stopping the LAMBDA software

Once the detector is connected to the control computer, the LAMBDA control software (Tango device server) can be started.

Firstly, log in to the LAMBDA control PC. This can either be done directly or, if it is connected to a network, this can be done remotely using ssh.

The program itself is started by opening a terminal window and executing the following script:

```
startlambda.sh
```

The Tango server can be killed e.g. by Ctrl-C in the relevant window, or the kill command in Linux. The name of the program controlling the detector is "Lambda", so the software can be killed from another terminal as follows:

```
killall -9 Lambda
```

2.4. Tango commands and attributes and the “Jive” user interface

The Tango server can be controlled using “commands” that can be executed (e.g. to start the acquisition) and “attributes” that can be read and written (e.g. the shutter time). These can be accessed in a variety of ways, e.g. using Python scripting. One option is to use the “Jive” user interface within Tango, which can provide a GUI (“ATK panel”) for each device controlled by Tango.

Starting Jive:

- (a) Log in to the detector control computer
- (b) Open a terminal window and type: *jive*
- (c) In the “Server” section, click on the tabs for the device, e.g. Lambda → xsp → Lambda, then double-click on /xsp/lambda/01

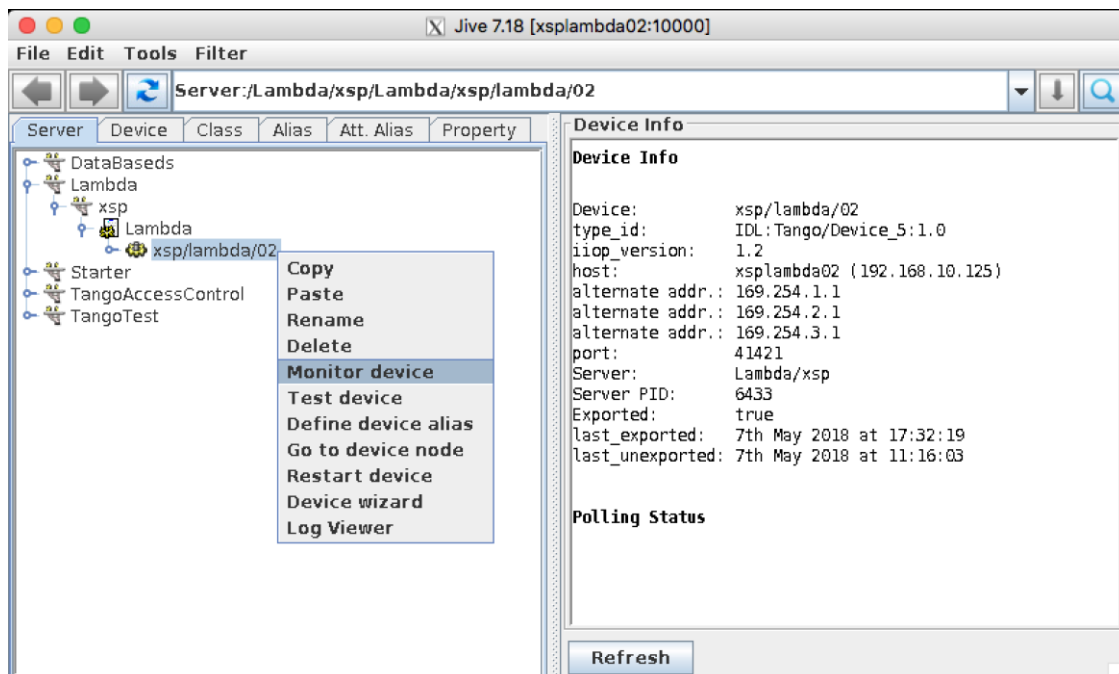
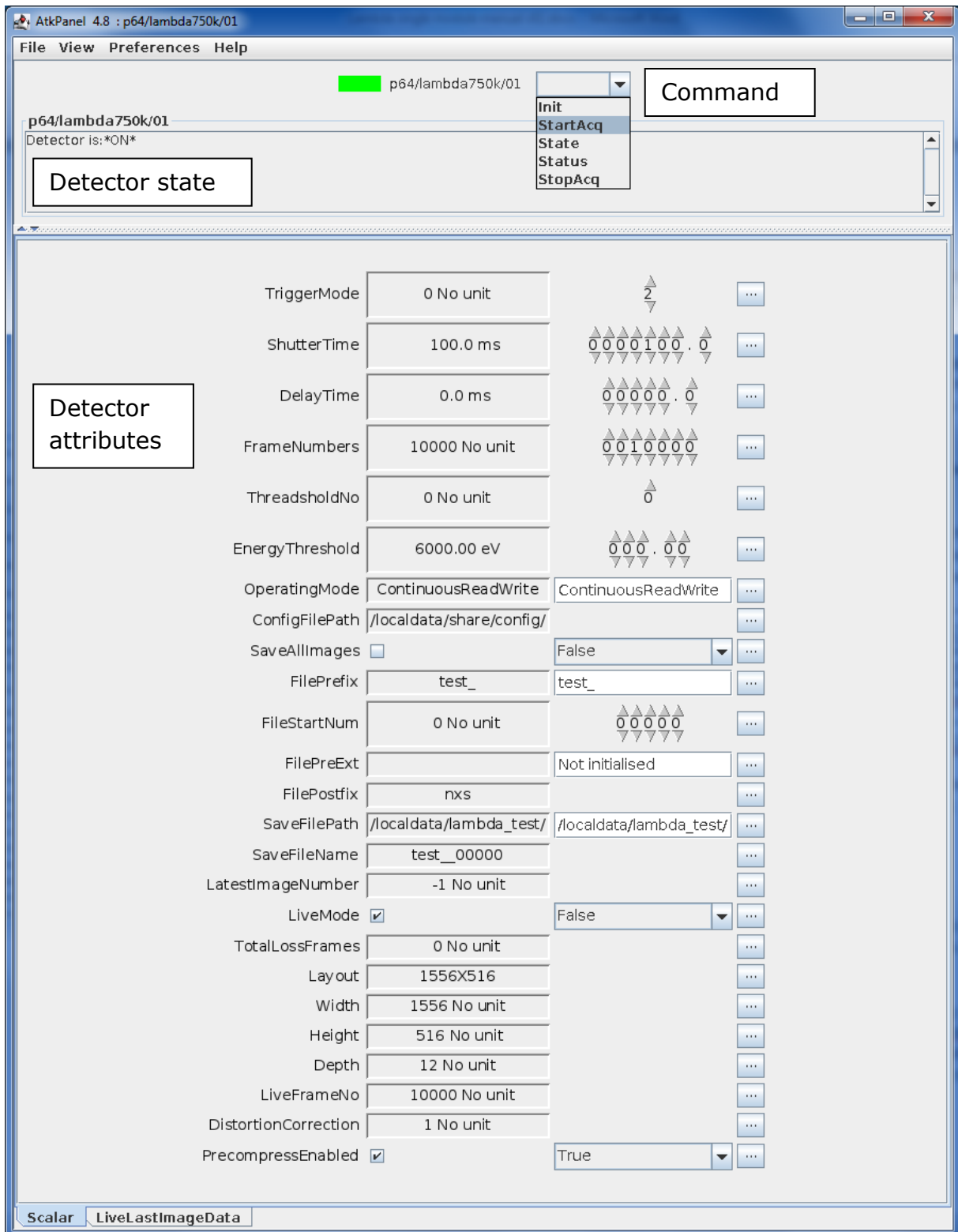


Figure 8: starting the ATK panel for Lambda (note: specific name of detector may vary)



2.5. Detector command list (pull-down menu)

StartAcq starts taking images. The box will go blue to indicate the detector is working, then return to green once all the images have been written to disk. All the images taken during a single acquisition are saved to a single .nxs file.

StopAcq can also be used to end an acquisition. Any images that have already been written to the .nxs output file will still be saved OK. Please note that the image *currently* being taken is cancelled (i.e. it's not possible to take a single image by starting an image with a very long shutter time then manually stopping the acquisition).

Please do not use **init** – this is automatically used when the Tango server starts up, but using it subsequently could cause problems.

State and **Status** commands both report on the state of the detector, e.g. whether it is on, taking images etc. This information is also displayed continuously in the GUI. "State" returns a numerical value corresponding to an enum in Tango, and "Status" a string. The LAMBDA Tango server has the following possible states/statuses:

<i>State name</i>	<i>State value</i>	<i>Explanation</i>
Tango::ON	0	Detector is ready for use, and not currently executing commands
Tango::MOVING	6	Detector is taking a series of images or executing other commands such as loading a configuration
Tango::RUNNING	10	Detector has finished acquiring images, but is still processing and writing to disk. (This can occur at high frame rates.)
Tango::FAULT	8	Fault detected
Tango::DISABLE	12	Detector disabled
Tango::UNKNOWN	13	Detector state not known

As well as these two commands, the status is shown in the dialog box in the GUI, and both state and status are listed as attributes that can be read.

2.6. Detector attributes (i.e. acquisition settings) list

The following lists all the detector attributes.

Note also that restarting the Tango server will reset most of these attributes, so you will need to set the *EnergyThreshold*, *SaveFilePath* etc. again.

TriggerMode – Sets up the use of an external electronic signal to control the acquisition. This is discussed in more detail in the next section. *TriggerMode 0* does not use any external signal. *TriggerMode 1* uses a trigger pulse to start the acquisition. *TriggerMode 2* uses a trigger pulse to start each individual image in an acquisition. *TriggerMode 3* uses the external signal as a “gate” or “electronic shutter”, to control when the detector is sensitive (e.g. selecting specific X-ray bunches during pump-probe experiments).

ShutterTime specifies the time per image in milliseconds. In continuous read-write mode, there is no gap between images, so this number sets both the time the shutter is open and the frame rate. The shortest possible shutter time in this case is 0.5 ms. In twenty-four-bit mode, there is an additional delay of 1 ms when reading out an image, in addition to the *ShutterTime*, though it is possible to set *ShutterTime* to as little as 0.01 ms. (The *DelayTime* variable is currently not used.) Please note that at higher frame rates (>100 Hz, i.e. <10 ms shutter time) the images from the detector need to be stored in RAM then written to file at a slower rate, so the time for the Tango server to finish this will be longer than the time to actually take the images. The space remaining in the RAM buffer is reported by the *FreeBuffer* attribute.

FrameNumbers sets the number of images to take during the acquisition. All the images taken during a single acquisition are written to a single file, as described later.

ThresholdNo is used to specify which threshold we want to change using *EnergyThreshold*. Depending on operating mode, the detector has either one energy threshold (numbered 0) or two (numbered 0 for the lower threshold, and 1 for the upper one).

EnergyThreshold sets the photon counting threshold in eV. Whenever energy is deposited in a pixel exceeding this value, a hit will be counted in the pixel. Please note that the energy from a single photon hit may be shared between neighbouring pixels, so a typical recommendation is to set this to approximately half the photon energy (or a bit below) to avoid missing charge-shared hits.

OperatingMode: Currently, this can be used to switch between two operating modes.

- *Twenty-four-bit mode* (type in *TwentyFourBit*) has 24-bit counter depth, but also has a time gap between images of 1 ms. A single threshold is used.
- *Continuous read-write mode* (type in *ContinuousReadWrite*) has 12-bit counter depth but no time gap between images (up to 2 kHz). A single threshold is used.

ConfigFilePath: This is a read-only variable that describes where the detector configuration files are stored. This variable is set by the Tango database.

SaveAllImages: Images will only be saved if *SaveAllImages* is checked.

FilePrefix, FileStartNum, FilePreExt, FilePostFix, SaveFilePath, SaveFileName: These attributes are involved in setting the name of save files. Firstly, *SaveFilePath* sets the directory where files should be saved. When this is changed, the server checks whether the directory exists and gives a warning method if it does not. However, it does not automatically create new directories; instead, if you want a new directory you should create it first and then change *SaveFilePath*.

The overall file name has the following format:

FilePrefix_FileStartNum_FilePreExt.nxs

FilePrefix and *FilePreExt* are straightforwardly set by the user. *FileStartNum* is a counter which increases by 1 each time a new acquisition is made, giving a series of files with increasing number, e.g.

ExampleFilePrefix_00000_preext.nxs, ExampleFilePrefix_00001_preext.nxs,
ExampleFilePrefix_00002_preext.nxs...

As well as being automatically incremented, *FileStartNum* can be set by the user. The full filename is then listed as *SaveFileName*: this attribute can be read, but not directly set. Finally, please note that if the full filename matches that of an already existing file, the existing file will be overwritten. This could happen, for example, if the user manually resets *FileStartNum* to 0 without changing the *FilePrefix*.

LatestImageNumber: This is a read-only variable used when taking a series of images (i.e. *FrameNumbers* > 1). This gives the number of the image most recently written to disk. Please note that at extremely high frame rates, images may be taken more quickly than they can be written to disk, in which case they are buffered in RAM before they are written. In this case, the state of the Tango server will change from MOVING to RUNNING once all the images have been received.

LiveMode: When this is checked, the most recently-received image can be viewed using the *LiveLastImageData* tab, and the attribute *LiveLastImageData* contains the content of this image. See below for more details.

PrecompressEnabled: This speeds up image saving by using a new parallelised image compression approach. In future this will be default, but currently this is provided as an option in case compatibility problems occur when using analysis software.

The following are all read-only attributes:

TotalLossFrames: This is the total number of lost or incomplete images from the detector. This should normally be zero. Lost or incomplete images are most likely the result of errors in receiving data from the detector; for example, if the detector is run at a high frame rate on a computer whose specs are not good enough, the software might not be able to store the received image data in RAM as quickly as it arrives.

Layout, width, height: The dimensions of the image (after interpolation of large pixels)

Depth: Bit depth of current images

LiveFrameNo: Number of most recently displayed live frame (at high frame rates, not every frame is displayed live).

DistortionCorrection: Indicates how the detector deals with large and gap pixels – see the section of the manual on detector layouts.

2.7. Live viewing

The *LiveLastImageData* tab at the bottom of the ATK panel (see Figure 9) allows you to look at live images. This is refreshed periodically, so at high frame rates it will not attempt to show you every image. Right-clicking on the image gives you various options – in particular, under the “settings” option it’s possible to change the range and image scaling, and choose a colour map.

There is also a Tango attribute *LiveLastImageData* which can be used to access the most recent image – this can be used for monitoring the detector in a more flexible way (e.g. grabbing each image and calculating a ROI).

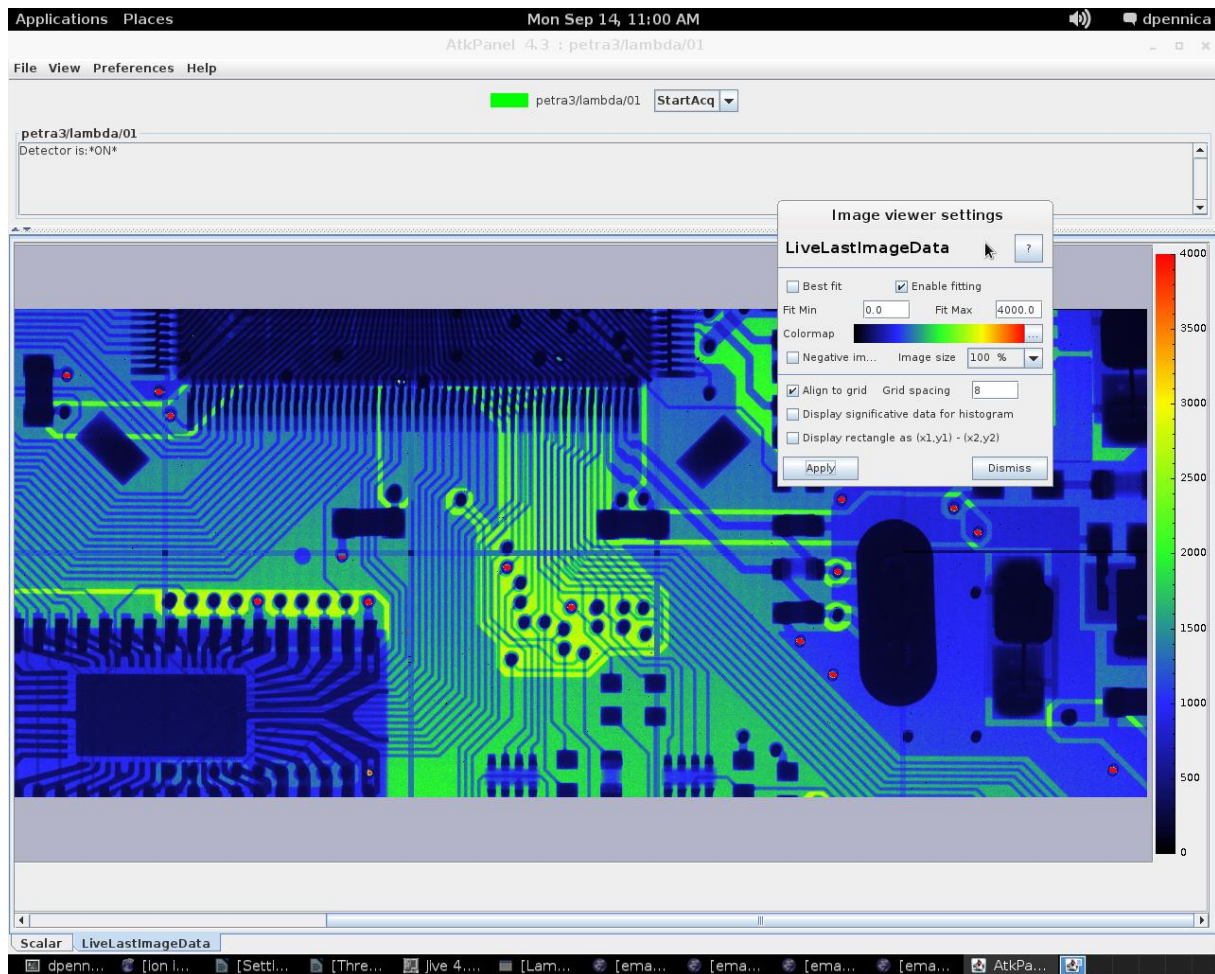


Figure 9: Live viewing tab of the ATK panel

2.8. Hardware Trigger

The system is equipped with two Trigger connections, one incoming and one outgoing. They are connected with mini-LEMO connectors, shown in Figure 4.

Hardware-wise, these are Low-voltage TTL (LVTTTL) signals with a 3.3 V max. Both the input and output signals have high input or output impedance ($\sim 1\text{k}\Omega$). So, when connecting other devices to the input and output, it may be useful to take steps to match impedances (e.g. connect a 50 Ω terminating resistor in parallel with the input, or change the device impedance settings). In our experience, although the official min HIGH voltage is 2 V, the voltage should be 3.3 V to ensure reliability.

Symbol	Parameter	min	max
U_{IH}	High-Level Input Voltage	2.0 V	3.3 V
U_{IL}	Low-Level Input Voltage	-0.3 V	0.8 V
U_{OH}	High-Level Output Voltage	3.3 V	
U_{OL}	Low-Level Output Voltage		0.4 V

The trigger behaviour depends both on operating and trigger mode. Regardless of mode, the *FrameNumbers* attribute should be set to the required number of images, and the *StartAcq* command sent to the detector; in modes 1 and 2, the *StartAcq* command will "arm" the detector so that it can receive the trigger signal. [The appendix](#) gives more detail on this, including diagrams of trigger timing and oscilloscope outputs.

Trigger mode 0 – The external trigger signal is not used, and the detector starts taking images immediately when the *StartAcq* command is used.

Trigger mode 1 - *StartAcq* gets the detector ready. One rising edge trigger signal starts the acquisition, following which the full series of images is taken (with the shutter time and number of images controlled by *ShutterTime* and *FrameNumbers* as usual).

Trigger mode 2 – *StartAcq* gets the detector ready. A rising trigger signal must be sent to start taking each individual image. Typically used for running scans. In 24-bit mode, when a trigger signal is sent, the detector takes an image for the pre-programmed *ShutterTime* and reads it out, and only after this can the detector be triggered again.

In continuous read-write mode, the first rising edge trigger signal will make the detector begin taking the first image, and subsequent rising edges will end the current image and immediately begin taking the next one. This has a few implications: the duration of each image is controlled by the time elapsed between rising edge trigger signals; and a total of *FrameNumbers* + 1 trigger

pulses are required to complete the acquisition (with the final pulse controlling the end time of the final image).

Mode 3 (gating mode) – This is only available in *TwentyFourBit* mode. In this mode, the detector immediately starts taking images when the *StartAcq* command is used, with the time per image and number of images controlled by *ShutterTime* and *FrameNumbers*. However, when the trigger input signal is logic HIGH, any photon hits in the detector will be ignored; only with logic LOW will hits be registered. This makes it possible to rapidly switch the sensitivity of the detector on and off, for example in pump-probe experiments.

In each case, the acquisition automatically stops after ***FrameNumbers*** of frames are taken. Acquisition can be stopped earlier by sending the ***StopAcq*** command.

When using modes 1 or 2, please note the following:

1. The system needs at least 300 ms delay time after the *StartAcq* signal in order to get ready.
2. The trigger signal is a RISING edge.
3. The system indicates its "*ReadyforTrigger*" state through a High Output on the output trigger. This signal is at 3.3 V when the detector is waiting to be triggered, i.e. the command has been given to start the acquisition and the detector is waiting for the initial trigger (or has finished taking an image and is ready to be triggered again, in the case of trigger mode "2" and *Twenty-four-bit mode*).

Thus, you can either incorporate a 300 ms delay into your setup or use the *TriggerOut* signal for more precise timing information.

Considerations when using 5V TTL signals:

Please note that Low-voltage TTL (LVTTTL) is designed to work with 3.3 V and significantly differs from "normal" TTL working at 5 V. It is possible to trigger the detector with 5 V, however the different logic levels of a 5 V TTL system are a possible source of error.

2.9. Saving files during the acquisition

The X-Spectrum PC has a second 1TB disk, which is mounted as /extdisk. It can be used to store data locally during acquisition.

The files are saved in HDF5 / Nexus format (.nxs). HDF5 is a general format for storing large amounts of data in a single file with appropriate metadata. Nexus is then a standardised way of using HDF5 for scientific data.

When a set of images is taken, a single output file is produced to store all the images in a compressed format with metadata (e.g. the image dimensions, bit depth etc). Matlab, IDL, Python, C++ etc. have routines for accessing this file type, as described below.

2.10. Tango and Python scripting

More information on Tango is available from www.tango-controls.org,

By using the PyTango library, which needs to be imported at the start of the script with "*import PyTango*", it is possible to connect to the detector, issue commands to it, and monitor its status via the Tango control system. (Controlling the detector via Python bindings without the Tango control system is described in the appendix.)

Code examples are available on the PC under `/localdata/pythonscripts`

Key commands are as follows:

```
lmbd = PyTango.DeviceProxy("//<hostname>:10000/xsp/lambda/01")
```

This will connect to the detector. The part in brackets identifies the detector to connect to, first by PC name and port (`<hostname>:10000`) and then by the detector name (`/xsp/lambda/01`). These can both be checked in jive. This function returns an object named "lmbd" which can then be used to control the detector.

Every parameter visible in the GUI can be checked or changed by using (e.g. for `FrameNumbers`):

```
lmbd.FrameNumbers = 10
    to set a value of 10
print(lmbd.FrameNumbers)
    to read the current value and print it
```

Drop-down menu commands like *StartAcq* can be accessed by:

```
lmbd.command_inout("StartAcq")
```

The state of the detector can be checked with:

```
lmbd.state()
```

This will return a value such as *PyTango.DevState.ON*, *PyTango.DevState.MOVING* etc., as detailed earlier.

As an example, a simple Python script for controlling LAMBDA is shown below. This script will take a series of images at regular steps in threshold energy.

```
import PyTango      # Library for Tango control in Python
import time
import os
import sys

# Connect to Tango device - in Tango this is done by creating a
# PyTango.DeviceProxy object
found = False
```

```

cnt = 0
while not found and cnt < 20:
    try:
        lmbd =
PyTango.DeviceProxy("//mycomputer.institute.de:10000/whatever/devicename/is")
        time.sleep(0.5)
        if lmbd.state() == PyTango.DevState.ON:
            found = True
    except:
        found = False
        cnt +=1

if not found:
    print("Unable to connect to LAMBDA server")
    sys.exit(0)

# Set up some attributes for image taking

expo_time = 1.0 # in seconds
expo_time_ms = (expo_time*1000) # - detector uses milliseconds
# Commands below can set attributes
lmbd.ShutterTime=expo_time_ms
lmbd.SaveFilePath="/localdata/test/"
lmbd.FrameNumbers=1
lmbd.FilePrefix="EnergyScan"
lmbd.FileStartNum=0

# start acquisition

startE = 5000
endE = 1000
stepE = 1.0
stepsNeeded = 1 + (endE-startE)/stepE

time.sleep(1)

for i in range(0,stepsNeeded):
    currentE = startE + i * stepE
    print("Energy setting "+str(currentE)+" eV")
    # Tango server increments file no automatically
    lmbd.EnergyThreshold=currentE
    # Commands like StartAcq use .command_inout
    lmbd.command_inout("StartAcq")

    found = False
    cnt = 0
    time.sleep(expo_time)
    # Poll the state of the detector till it returns to "ON"
    while not found and cnt < 100:

```



```
try:
    time.sleep(0.01)
    cnt +=1
    if lmbd.state() == PyTango.DevState.ON:
        found = True
except:
    found = False
if cnt >= 100:
    print("\b Detector timeout - exiting loop")
    break

print 'Done....'
```

3. HDF5 / Nexus file format

The LAMBDA Tango server saves data using HDF5 / Nexus.

HDF5 is a general-purpose container format, which makes it possible to save large, multi-dimensional datasets plus metadata to a single file in a structured way:

<https://www.hdfgroup.org/HDF5/>

Nexus is then an international standard for storing X-ray, neutron and muon data from experiments:

<http://www.nexusformat.org/>

The LAMBDA detector structure is then based on the NXDetector specification:

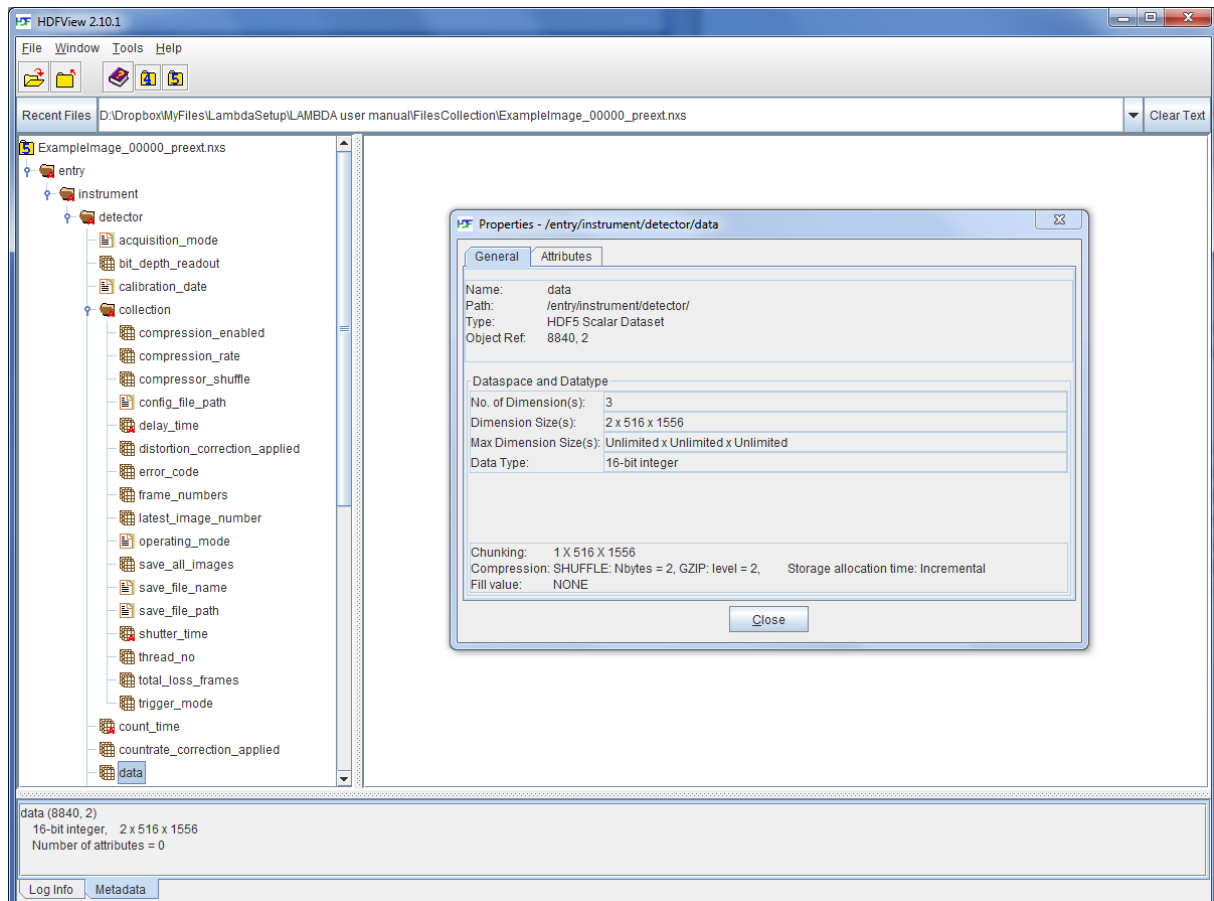
http://download.nexusformat.org/doc/html/classes/base_classes/NXdetector.html

More information on the LAMBDA file format can be found using the sources above and by browsing files created by the detector. However, some key points are as follows.

3.1. HDF5 / Nexus file browsing

These files have a directory-like structure. There are a variety of ways of looking at the structure of the file and extracting data, e.g. in Python, IDL, C++. As an example, below is a screenshot taken using a HDF5 viewer tool:

<https://www.hdfgroup.org/products/java/hdfview/index.html>



The image data from the detector is stored under entry/instrument/detector/data, and consists of a 3D array containing each image taken during the acquisition; the dimensions of the array correspond to image number, y-position and x-position respectively.

The metadata contains information about the detector in general, and the particular settings (e.g. shutter time, threshold) during the acquisition.

Pixel Mask

Of particular importance is the **pixel mask** which can be found under entry/instrument/detector/pixel_mask. This is an array, the same size as the image, which indicates which pixels are non-functional or extra-large (determined at 5 keV threshold before delivery). For each pixel, there is a 32-bit value defined as follows: bit 0, gap (pixel with no sensor); bit 1, dead; bit 2, under-responding; bit 3, over-responding; bit 4, noisy; bit 31: virtual pixel (corner pixel with interpolated value)

The Nexus file uses data compression to reduce the file size; this is particularly useful when operating at high frame rates, since the compression ratio is high when many of the pixels have zero hits. In this file format, individual images within the file are compressed, rather than the file as a whole. The decompression process is effectively invisible to the user; the user can simply use standard HDF5 libraries to access the data. Because the images are compressed individually, images can be read from within a large file relatively quickly, since it is only necessary to decompress the relevant images rather than the whole file.

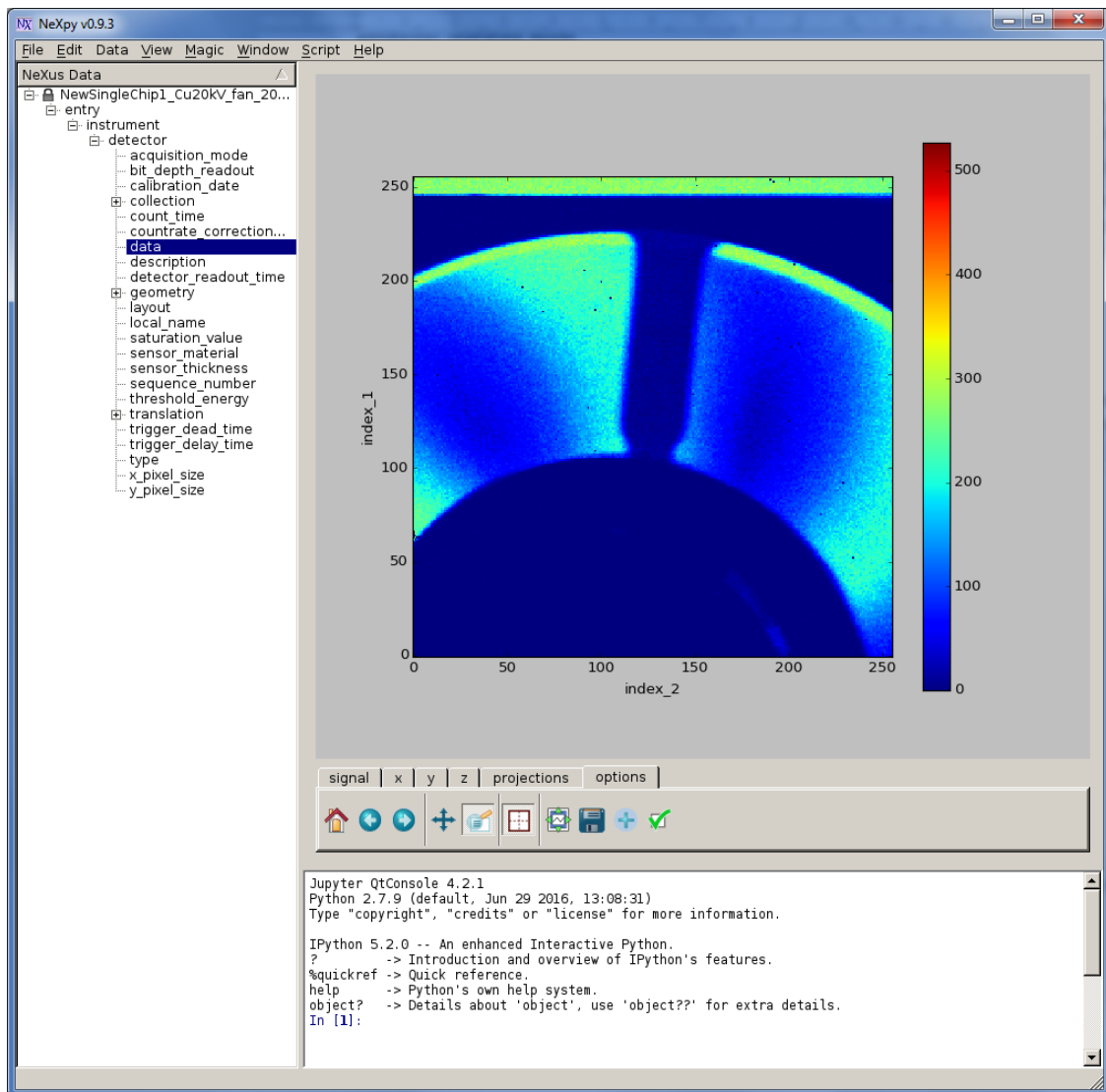
3.2. The “nexpy” image viewer tool

Frames stored inside the Nexus files can be viewed using NeXpy, which is described in detail under:

<http://nexpy.github.io/nexpy/>

This tool is already installed on the detector PC, and can be started from the command line with:

```
nexpy
```

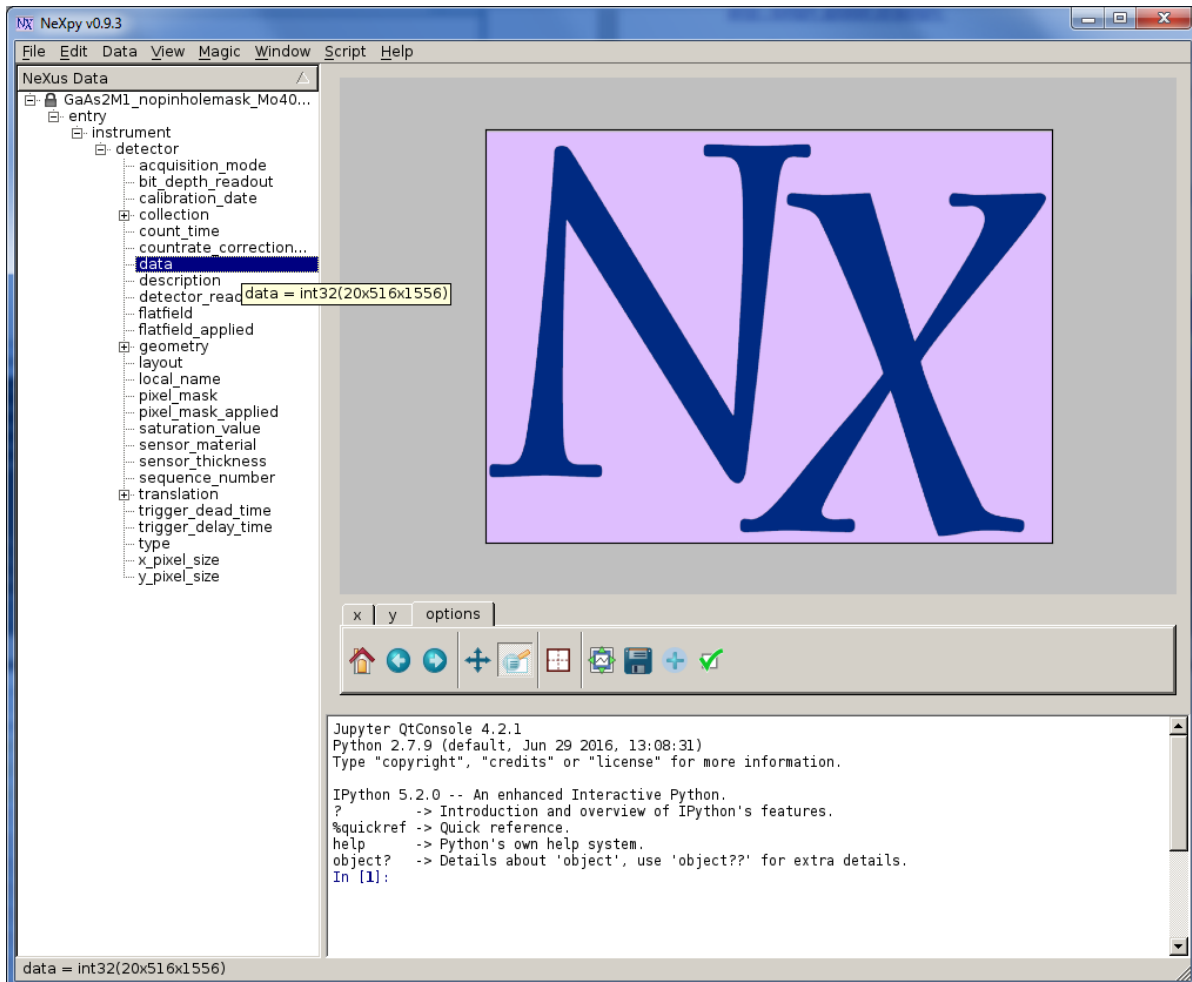


If you want to install this tool on a Windows PC, we recommend installing the “Anaconda” python distribution and using the “conda” installation process described on the webpages.

Using nexpy

Files can be opened by using *File* → *Open*.

When a file is first opened, the structure of the file appears in the left-hand window as shown below. The file contains various metadata on the acquisition. Right-clicking on a variable and choosing “view” gives information on this metadata.



The screenshot shows the NeXpy v0.9.3 application window. The title bar reads "NeXpy v0.9.3". The menu bar includes "File", "Edit", "Data", "View", "Magic", "Window", "Script", and "Help".

The left-hand pane, titled "NeXus Data", shows a tree view of the file structure. The selected file is "GaAs2M1_nopinholemask_Mo40...". Underneath, the "entry" folder is expanded, showing "instrument" and "detector". The "detector" folder is further expanded, listing various metadata fields. The "data" field is highlighted, and a tooltip shows "data = int32(20x516x1556)".

The main window displays a large blue "NX" logo on a purple background. Below the logo is a toolbar with icons for home, back, forward, zoom in, zoom out, and other navigation functions.

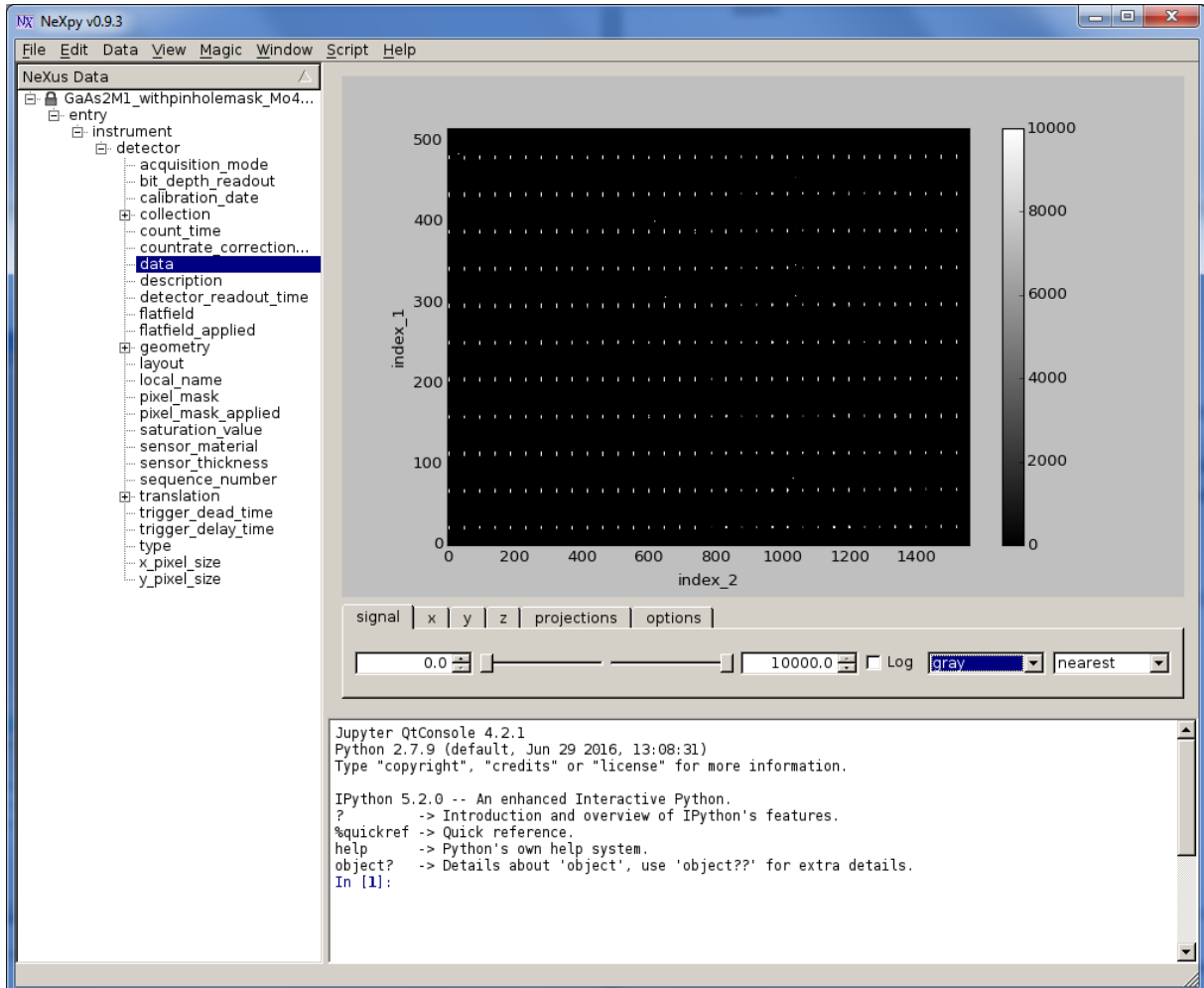
The bottom pane is a Jupyter QtConsole 4.2.1 window. It shows the following text:

```
Jupyter QtConsole 4.2.1
Python 2.7.9 (default, Jun 29 2016, 13:08:31)
Type "copyright", "credits" or "license" for more information.

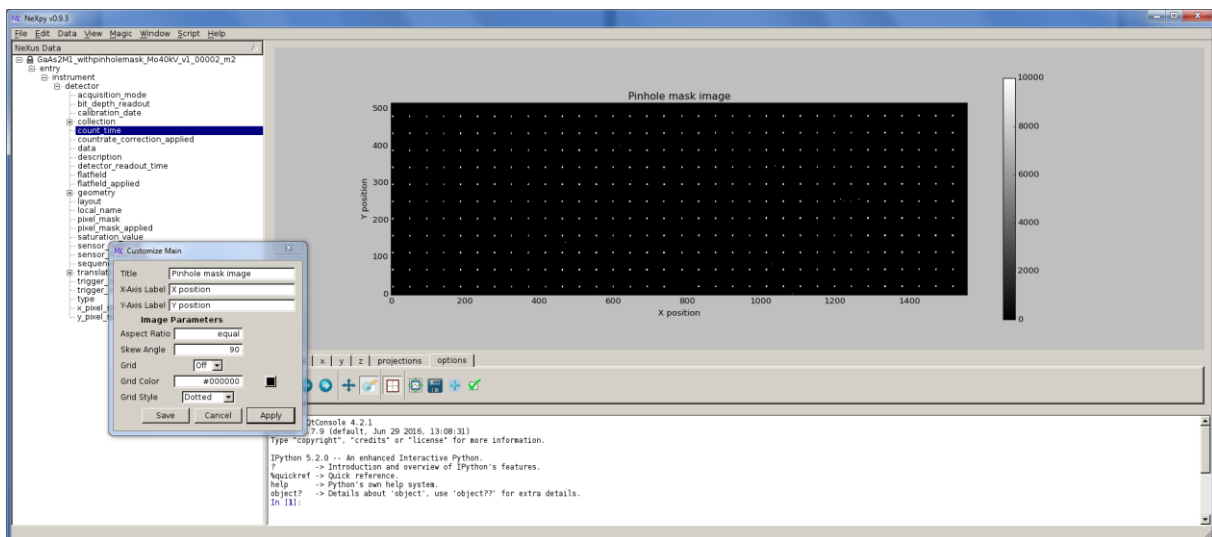
IPython 5.2.0 -- An enhanced Interactive Python.
?                -> Introduction and overview of IPython's features.
%quickref        -> Quick reference.
help             -> Python's own help system.
object?         -> Details about 'object', use 'object??' for extra details.
In [1]:
```

The status bar at the bottom of the application shows "data = int32(20x516x1556)".

The image data is stored under *entry/instrument/detector/data*. Double-clicking on this, then clicking "OK" activates the viewer in the top-right. The image scaling can be changed by clicking on the "signal" panel and choosing options such as the minimum and maximum of the scale, the colour map etc. The "z" panel can be used to browse through the series of images in the file. Left-clicking and dragging can change the field of view, and right-clicking goes back to the full image view.



The "options" panel gives options such as saving the plot, or customizing it (e.g. adding axis labels and a title).



The bottom panel can be used as in interactive Python environment.

A few particular points:

- Any loaded file can be referred to by its filename. For example, having loaded the file called:
"GaAs2M1_withpinholemask_Mo40kV_v1_00002_m2"
 its pixel size can be read with:

```
print(GaAs2M1_withpinholemask_Mo40kV_v1_00002_m2.entry.instrument.detector.x_pixel_size)
```
- Scripts can be loaded and run using *Script* → *Open Script* as shown in Figure 10.

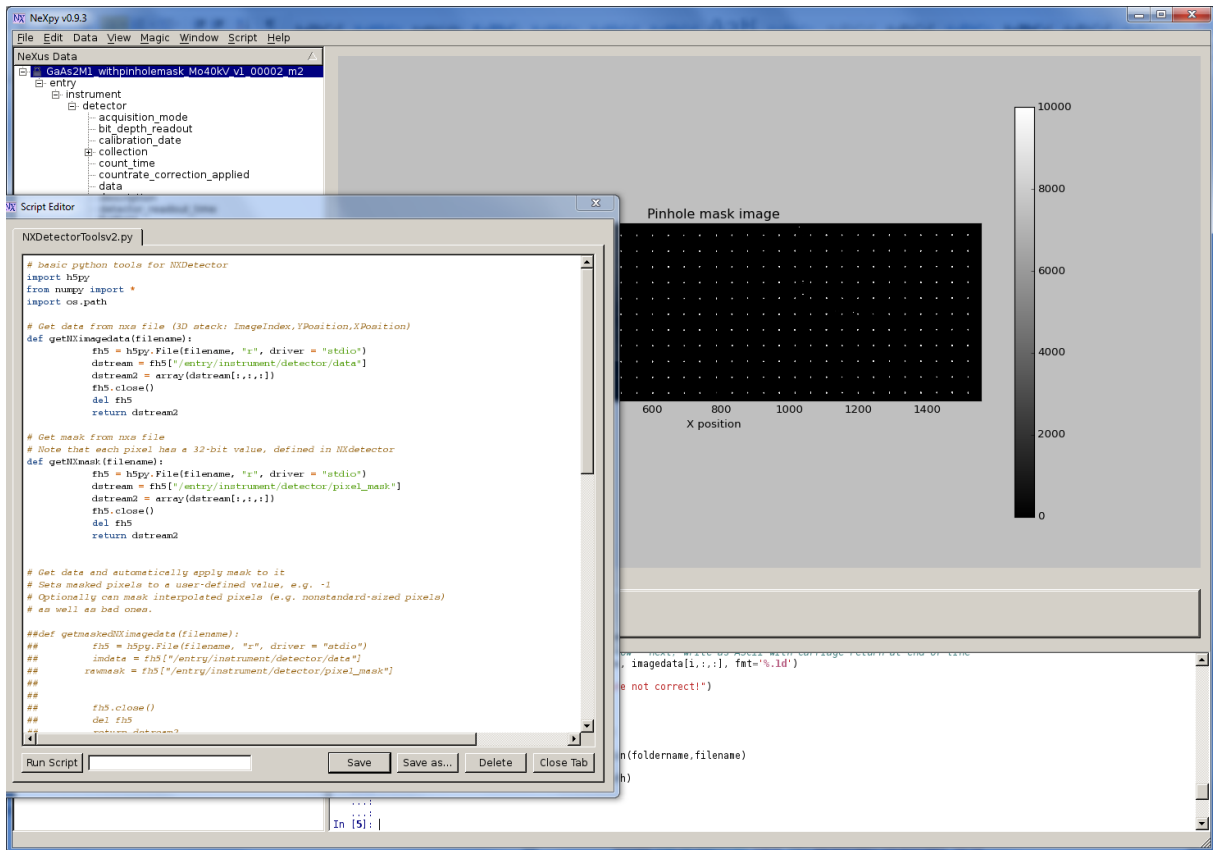


Figure 10: Example of running a script

3.3. Typical HDF5 file reading methods

PYTHON

The h5py library contains methods for reading hdf5 files: www.h5py.org

To use this in a script, it needs to be imported:

```
import h5py
```

HDF5 files can be opened with a command like follows, giving an object (fh5) which can be used to access datasets.

```
fh5 = h5py.File(filename, "r", driver = "stdio")
```

Then, accessing the "data" array within the file:

```
dstream = fh5["/entry/instrument/detector/data"]
```

For example, a function for reading the 3D image data stack from a file can be defined as follows. This function works by opening a file with h5py, copying the appropriate dataset to a variable, and then closing the file.

```
import h5py
# Get data from nxs file (3D stack: ImageIndex,YPosition,XPosition)
def getNXimagedata(filename):
    fh5 = h5py.File(filename, "r", driver = "stdio")
    dstream = fh5["/entry/instrument/detector/data"]
    dstream2 = dstream[:, :, :]
    fh5.close()
    del fh5
    return dstream2
```

IDL

There is an hdf5 file browser in IDL, which includes the option to copy data from the file to the local workspace.

```
result = h5_browser()           or           result=h5_browser(filename)
```

A more sophisticated example follows. (Please note that this approach is probably overkill – simpler implementations should be possible.)

```
function read_nx_multiimage, fname, start, wantedimages=wantedimages, hd=hd
```

```
res=H5_PARSE(fname)
```

```
;Q&D check for Nexus
```

```
check=where(tag_names(res) eq 'NEXUS_VERSION')
```

```
if check[0] eq -1 then return,(-1)
```

```
if (~KEYWORD_SET(wantedimages)) then begin
```

```
    wantedimages = 1
```

```
endif
```

```
last = start + wantedimages - 1
```

```
nframe=res.entry.instrument.detector.data._dimensions[2]
```

```
dimx=res.entry.instrument.detector.data._dimensions[0]
```

```
dimy=res.entry.instrument.detector.data._dimensions[1]
```

```
if (start lt 0) || (start ge nframe) then begin
```

```
    print, "start index out of range -- resetting to 0"
```

```
    start = 0
```

```
endif
```

```
if (last gt nframe) then begin
```

```
    print, "last image index out of range -- resetting to end of file"
```

```
    last = nframe-1
```

```
    wantedimages = last - start + 1
```

```
endif
```

```
; open hdf5 file
```

```
file_id = H5F_OPEN(fname)
```

```
dset_id = H5D_OPEN(file_id, '/entry/instrument/detector/data')
```

```
; Open up the dataspace associated with the image.
```

```
dspace_id = H5D_GET_SPACE(dset_id)
```

```
; Choose our hyperslab. We will pick out the chosen frame
```

```
startpos = [0, 0, start]
```

```
count = [dimx, dimy, wantedimages]
```

```
; Be sure to use /RESET to turn off all other selected elements.
```

```
H5S_SELECT_HYPERSLAB, dspace_id, startpos, count, /RESET
```

```
; Create a simple dataspace to hold the result. If we didn't supply the
```

```
; memory dataspace, then the result would be the same size as the image
```

```
; dataspace, with zeroes everywhere except our hyperslab selection.
```

```
memspace_id = H5S_CREATE_SIMPLE(count)
```

```
; Read in the current image data.
```

```
imdata = H5D_READ(dset_id, FILE_SPACE=dspace_id,  
MEMORY_SPACE=memspace_id)  
; Close identifiers to prevent resource leaking.  
H5S_CLOSE, memspace_id  
H5S_CLOSE, dspace_id  
H5D_CLOSE, dset_id  
; close file  
H5F_CLOSE, file_id  
  
if n_elements(hd) ne 0 then hd=[dimx,dimy,nframe]  
  
return,imdata  
end
```

MATLAB

Likewise, hdf5 functionality is available in MATLAB.
See <http://de.mathworks.com/help/matlab/high-level-functions.html>

C++

HDF5 API: https://www.hdfgroup.org/HDF5/doc/cplusplus_RM/

4. Appendices

4.1. Quick reference guide:

Starting the detector:

Option 1: Connect to the detector control PC, open a terminal, then `startlambda.sh`

Controlling the detector:

“jive” can be used: connect to the control computer, open a terminal, then execute `jive &`. Expand the “Lambda” tab fully, and double-click on the lowest level.

Key attributes to set up the acquisition: *ShutterTime* (time per image in ms), *FrameNumbers* (no of images to take), *EnergyThreshold* (threshold energy in eV – half the beam energy is normally recommended), *SaveFilePath* (path of folder to save images), *FilePrefix* (base file name to use). The *SaveAllImages* box should be checked to enable image saving.

Please note that these variables are set to default values when starting the Tango server – if you need to kill the software and restart, you must re-enter the values you need.

Files can be saved on the local disk in folders under `/extdisk`.

Additional configuration: *OperatingMode* can be set to *ContinuousReadWrite* (12-bit counter depth, no time gap between images) or *TwentyFourBit* (24-bit counter depth, 1ms delay between images). *TriggerMode* can be set to 0 – no trigger, 1 – trigger start of image series, or 2 – trigger each individual image.

Viewing images:

The *LiveLastImageData* tab in the GUI shows the most recently-taken image. This variable can also be accessed by scripts, etc.

Files can be viewed from the nexpy tool by opening a terminal then `nexpy`

Data access:

Data is saved using the HDF5/Nexus file format – more information on using this format is provided elsewhere in the manual.

4.2. Troubleshooting guide

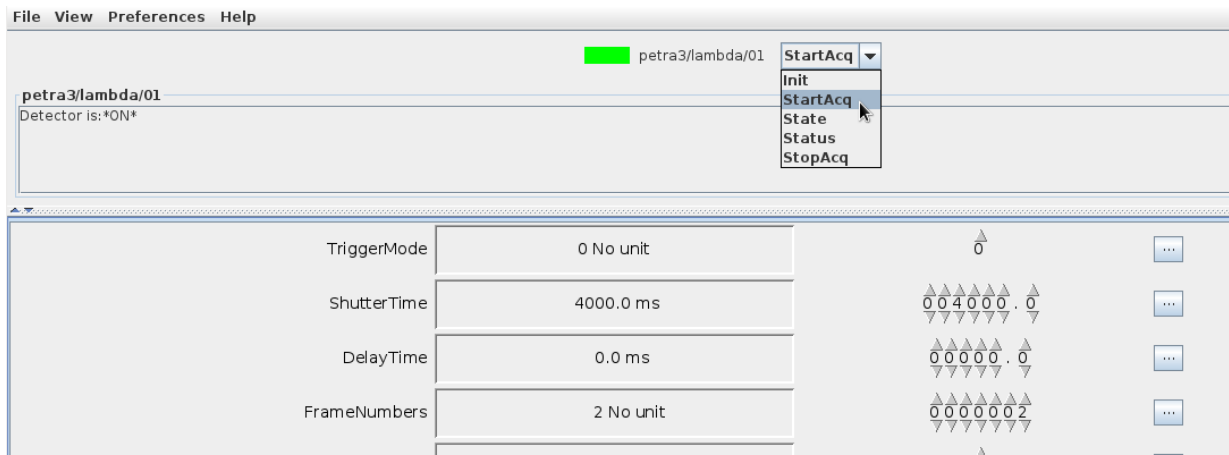
This troubleshooting guide assumes that the software and computer setup has already been performed correctly.

When starting the Tango server, it fails with the message "TCP cannot connect to host! Unable to connect to detector - please check that it is powered and correctly connected": This message appears if the Lambda software was unable to establish a connection with the detector over the control link. This may be for the following reasons: (A) The detector is not powered, (B) the detector has not been connected to the server PC correctly using the 1G Ethernet link.

The Tango server starts up OK, but when I try to take images, none are received: Firstly, if the 10G data link between the PC and detector is not connected correctly, the software can send commands to the detector, but image data cannot be received back from the detector. Therefore, check whether both of the fiberoptic cables to the detector are properly connected. Secondly, if the detector is set up to use an external trigger signal, then after *StartAcq* is used, the detector will wait for the trigger signal before taking images. Check whether external trigger is enabled (*TriggerMode* is nonzero). If you intend to use the trigger, check that the external trigger input is correctly connected.

I took images with the detector, but no files were saved: Firstly, the detector will only save images if the *SaveAllImages* box is checked. (This attribute is provided so that users can avoid saving unnecessary images – for example, during adjustment of the beamline it can be useful to take images continuously to see if any signal is seen at the detector.) Secondly, check that the attributes *FilePrefix* and *SaveFilePath* are correctly chosen – the files might be saved to a different folder or filename.

The detector doesn't use the settings I intended (e.g. wrong number of images or shutter time): Check these attributes are set properly. In the GUI, the left column gives the current value of each attribute, and the right column is used to change this value. To apply the change, it is necessary to type in the new value in the right column, and then hit *enter* – if *enter* is not pressed, the value will not be set. Also, on restarting the software, the current value will be set to safe defaults, so whatever settings you wish to use will need to be re-entered.



The software crashes when I attempt to save images: If a folder is write-protected, attempting to write images to this folder may crash the software. Check whether you have write access to the folder.

Images are saved, but are blank (or extremely noisy): Check that the *EnergyThreshold* attribute is set properly. If this is too high (e.g. greater than the beam energy) then photons will not be detected, and if it is very low (e.g. significantly below 4 keV) the many false hits will occur due to noise fluctuations. Please note that the *EnergyThreshold* value is in eV.

4.3. The Python Interface

The detector can be used with Python using the packages `pyxsp` and `lambdacontrol`. Both packages are open source, and the sources are installed into `$HOME/src/` directory of the `xspadmin` account.

pyxsp

`pyxsp` allows users to readout X-Spectrum detector systems consisting of one or more physical detectors. It is built on top of a C++ library, which handles the low-level data exchange with the detectors.

The first step is to create an instance of the `System` class

```
import pyxsp as xsp
s = xsp.System('/path/to/SystemConfig.txt')
```

The `System` class reads the system configuration from a file, which can be specified as an argument to the constructor.

The detectors, configured for a system, can be listed with

```
s.list_detectors()
```

which will return a tuple with detector IDs, such as `('lambda',)`. The ID is a string, which is then used to open a specific detector

```
d = s.open_detector('lambda')
```

Detector parameters are properties of the object that is returned by `open_detector()`. The following example shows how to set the threshold for the Lambda detector. The unit is [eV]. There are 8 thresholds, but usually only one threshold is set. The property expects a list, even if only one value is specified

```
d.thresholds = [6000.0]
```

After detector setup, the acquisition of the whole system can be started with

```
s.start_daq(10, 400.0)
```

It takes two optional arguments: the number of frames to acquire and the shutter time. Both can also be set directly on the detector as in the following example


```
d.number_of_frames = 10
d.shutter_time = 400.0
s.start_daq()
```

The acquisition can also be started on a specific detector

```
d = s.open_detector('lambda')
d.start_daq(10, 100.0)
```

The acquired data is read out using a loop like

```
import numpy as np
width = d.frame_width
height = d.frame_height
d.start_daq(100, 20.0)
for _ in range(100):
    d.wait_for_frames()
    f = d.frame # frame is a named tuple (id, error_code, data)
    a = np.asarray(f.data).reshape(height, width)
    # process the data
```

`d.frame` returns a tuple with an id (the frame number), an error code, and the data itself. The data can be transformed into a `numpy.ndarray` using `asarray()` method. Data is a contiguous array of values, starting with the values from the first row, then the values from the second row and so on. `reshape()` can be used to convert this into a 2D array with appropriate shape.

If the error code `f.error_code` is nonzero, the frame is considered incomplete and should be discarded.

The data is guaranteed to be valid, until the next `d.frame` is called.

lambdacontrol

lambdacontrol extends pyxsp with file saving capabilities.

The following is a simple example of how to use this package to start an acquisition and to store data into files.

```
import lambdacontrol as lc
ctrl = lc.Controller('/etc/opt/xsp/ SystemConfig.txt')
det = ctrl.open_detector('lambda')
det.thresholds = [6000.0]
w = lc.Writer()
w.save_to_file = True
ctrl.set_writer('lambda', w)
ctrl.start_daq(100, 600.0)
```

After importing lambdacontrol, a Controller needs to be created. The constructor takes an argument to specify the location of the detector system configuration file. If it is not given, then the default file ``/etc/opt/xsp/system.yml`` is read. You can check the location of the configuration file simply by

```
print(ctrl)
```

The configuration file contains a list of detectors that are part of the detector system. The detectors can be listed by

```
ctrl.list_detectors()
```

which returns a tuple with detector IDs. The ID is a string, which is then used to open a specific detector

```
det = ctrl.open_detector('lambda')
```

Detector parameters are properties of the object that is returned by `open_detector()`. The following example shows how to set the threshold for the Lambda detector. The unit is [eV]. There are 8 thresholds, but usually only one threshold is set. The property expects a list, even if only one value is specified

```
det.thresholds = [6000.0]
```

In order to save acquired data into a HDF5 file in Nexus format, a Writer needs to be created, which is then added to a specific detector

```
w = lc.Writer()
w.save_to_file = True
w.save_directory = '/path/to/files/'
w.save_file_prefix = 'lambda-test'
ctrl.set_writer('lambda', w)
```

The Writer does not save to files by default, so this has to be enabled explicitly. The filename is constructed from the specified directory and prefix, plus a five digit run number and the extension `.nxs`. The resulting path can be read out from the property `save_file_name`, in this case, it would be `/path/to/files/lambda-test_00000.nxs`.

If a file already exists, there are three possibilities to continue:

- abort the acquisition
- increment run number
- overwrite existing file

The behaviour can be defined by setting the `save_mode` property of the Writer to either `SaveMode.ABORT_IF_EXISTS`, `SaveMode.USE_RUN_NUMBER`, or `SaveMode.OVERWRITE`

```
w.save_mode = lc.SaveMode.USE_RUN_NUMBER
```

If `SaveMode.USE_RUN_NUMBER` is used, then the file system is checked for the highest existing number, which is then simply incremented. If the highest existing number is already 99999, then the acquisition is aborted with an exception.

Now, acquisition can be started with

```
ctrl.start_daq(10, 400.0)
```

The `start_daq()` method takes two optional arguments: the number of frames to acquire and the shutter time. Both can also be set directly on the detector as in the following example

```
det.number_of_frames = 10
det.shutter_time = 400.0
ctrl.start_daq()
```

The Controller will run a separate thread to readout the frames from the detector and send them to the writer. If no writer has been added, then no thread is started and `start_daq()` will return immediately.

`start_daq()` is blocking. The non-blocking variant is `nb_start_daq()`. In that case, the end of acquisition can be determined by calling `daq_finished()` in regular intervals

```
import time
ctrl.nb_start_daq(10, 400.0)
while not ctrl.daq_finished():
    time.sleep(0.5)
```

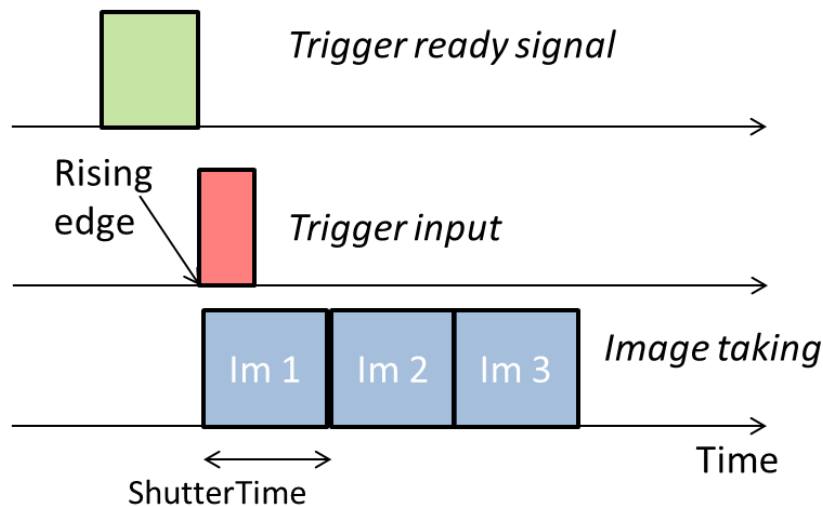
4.4. Detailed Operating Mode Behaviour

This section describes how combinations of operating and trigger modes interact with each other.

<i>OperatingMode</i>	<i>ContinuousReadWrite</i>	<i>TriggerMode</i>	0
<i>StartAcq</i>			Starts the Acquisition
<i>ShutterTime</i>			Min value: 0.5 ms
<i>FrameNumbers</i>			Sets the number of exposures of <i>ShutterTime</i> length each
HW Trigger IN			No function
HW Trigger OUT			No function

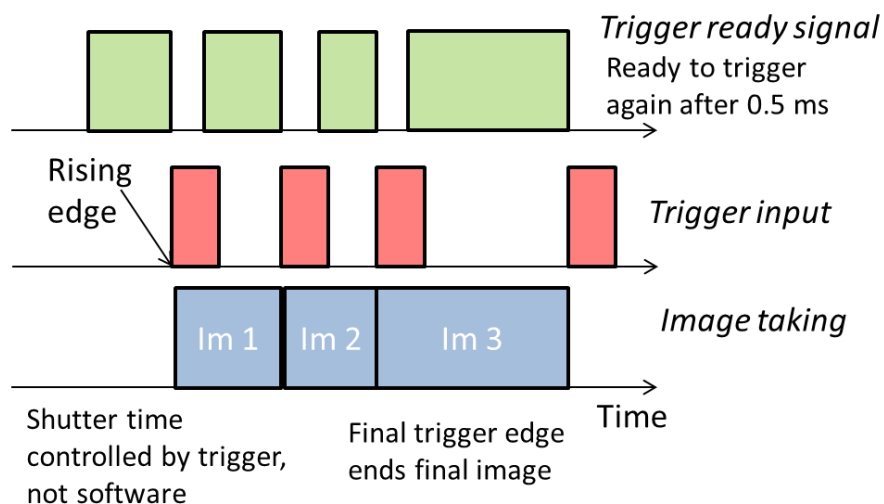
<i>OperatingMode</i>	<i>ContinuousReadWrite</i>	<i>TriggerMode</i>	1
<i>StartAcq</i>			Gets the detector ready for Acquisition
<i>ShutterTime</i>			Min value: 0.5 ms
<i>FrameNumbers</i>			Sets the number of exposures of <i>ShutterTime</i> length each
HW Trigger IN			Starts the Acquisition of <i>FrameNumbers</i> exposures at rising edge
HW Trigger OUT			Shows that the detector is ready for Trigger in after <i>StartAcq</i> command (typically 300 ms delay)

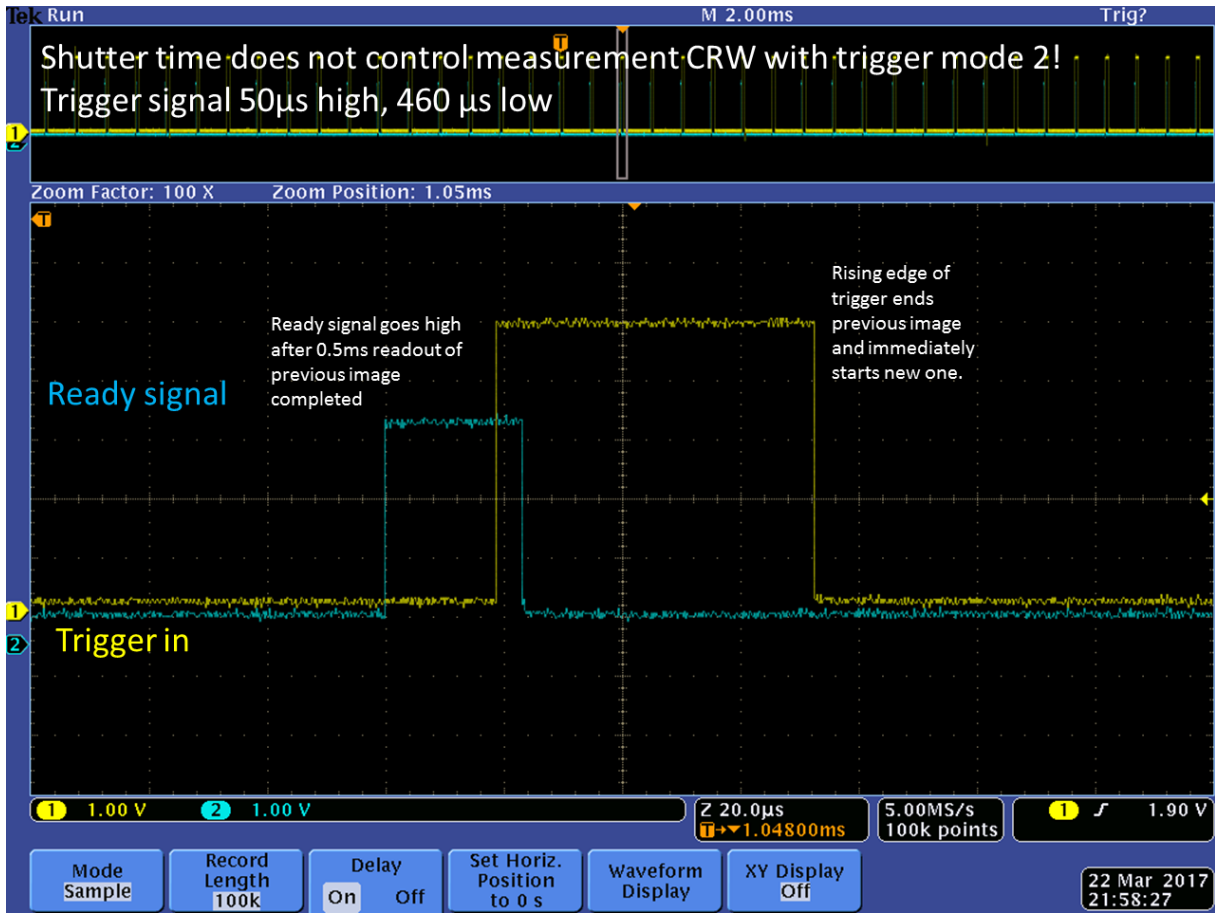
ContinuousReadWrite, trigger mode 1



<i>OperatingMode</i>	<i>ContinuousReadWrite</i>	<i>TriggerMode</i>	2
<i>StartAcq</i>		Gets the detector ready for Acquisition	
<i>ShutterTime</i>		No function	
<i>FrameNumbers</i>		Sets the number of exposures	
HW Trigger IN		Ends the current exposure and immediately starts the next exposure. This results in a variable exposure time per frame. Note that you must have at least 0.5 ms per frame. Please note that for N images, $N+1$ trigger pulses are required; the first pulse starts the first image, and the $N+1$ th pulse ends the final image.	
HW Trigger OUT		Logic high shows that the detector is ready to receive the next trigger signal. This occurs (a) after <i>StartAcq</i> command when waiting for first trigger, and (b) 0.5 ms after the previous exposure was triggered.	

ContinuousReadWrite, trigger mode 2



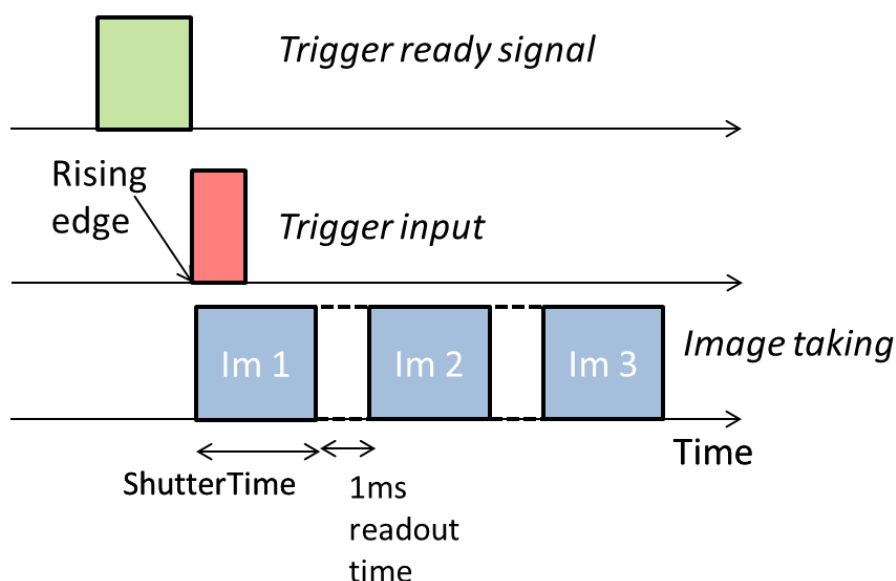


<i>OperatingMode</i>	<i>ContinuousReadWrite</i>	<i>TriggerMode</i>	3
THIS COMBINATION OF READOUT AND TRIGGER MODES DOES NOT WORK			

<i>OperatingMode</i>	<i>TwentyFourBit</i>	<i>TriggerMode</i>	0
<i>StartAcq</i>	Starts the Acquisition		
<i>ShutterTime</i>	Min value: 0.01 ms		
<i>FrameNumbers</i>	Sets the number of exposures of <i>ShutterTime</i> length each		
HW Trigger IN	No function		
HW Trigger OUT	No function		
Note: In <i>TwentyFourBit</i> mode, it takes 1 ms to read out the frame. Thus, the period of an exposure is the sum of <i>ShutterTime</i> and 1 ms.			

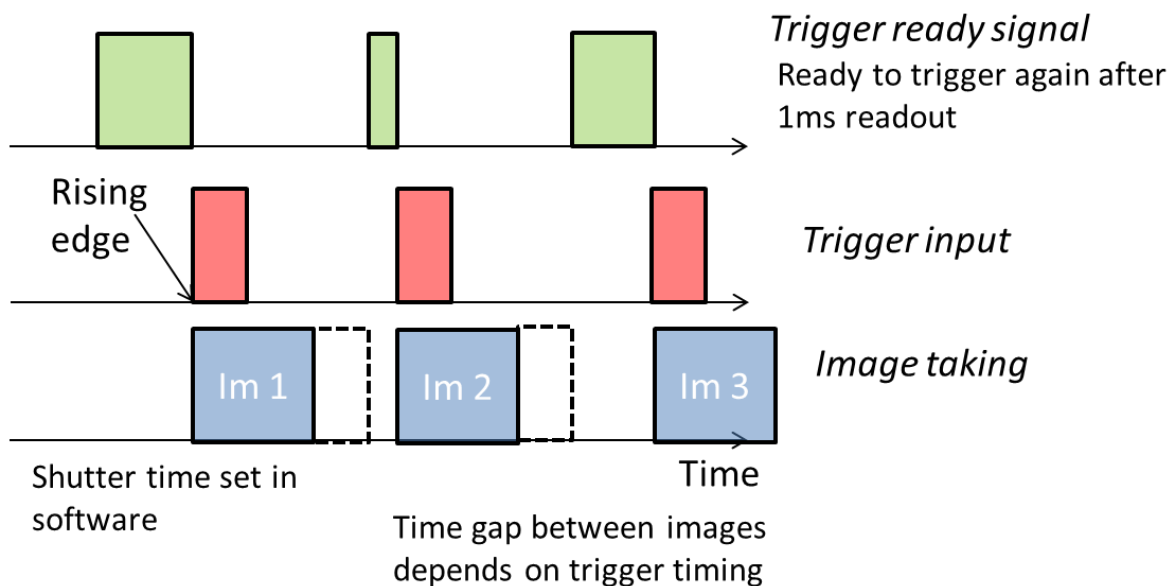
<i>OperatingMode</i>	<i>TwentyFourBit</i>	<i>TriggerMode</i>	1
<i>StartAcq</i>	Starts the Acquisition		
<i>ShutterTime</i>	Min value: 0.01 ms		
<i>FrameNumbers</i>	Sets the number of exposures of <i>ShutterTime</i> length each		
HW Trigger IN	Starts the Acquisition of <i>FrameNumbers</i> exposures at rising edge		
HW Trigger OUT	Shows that the detector is ready for Trigger in after <i>StartAcq</i> command (typically 300 ms delay)		
Note: In <i>TwentyFourBit</i> mode, it takes 1 ms to read out the frame. Thus, the period of an exposure is the sum of <i>ShutterTime</i> and 1 ms.			

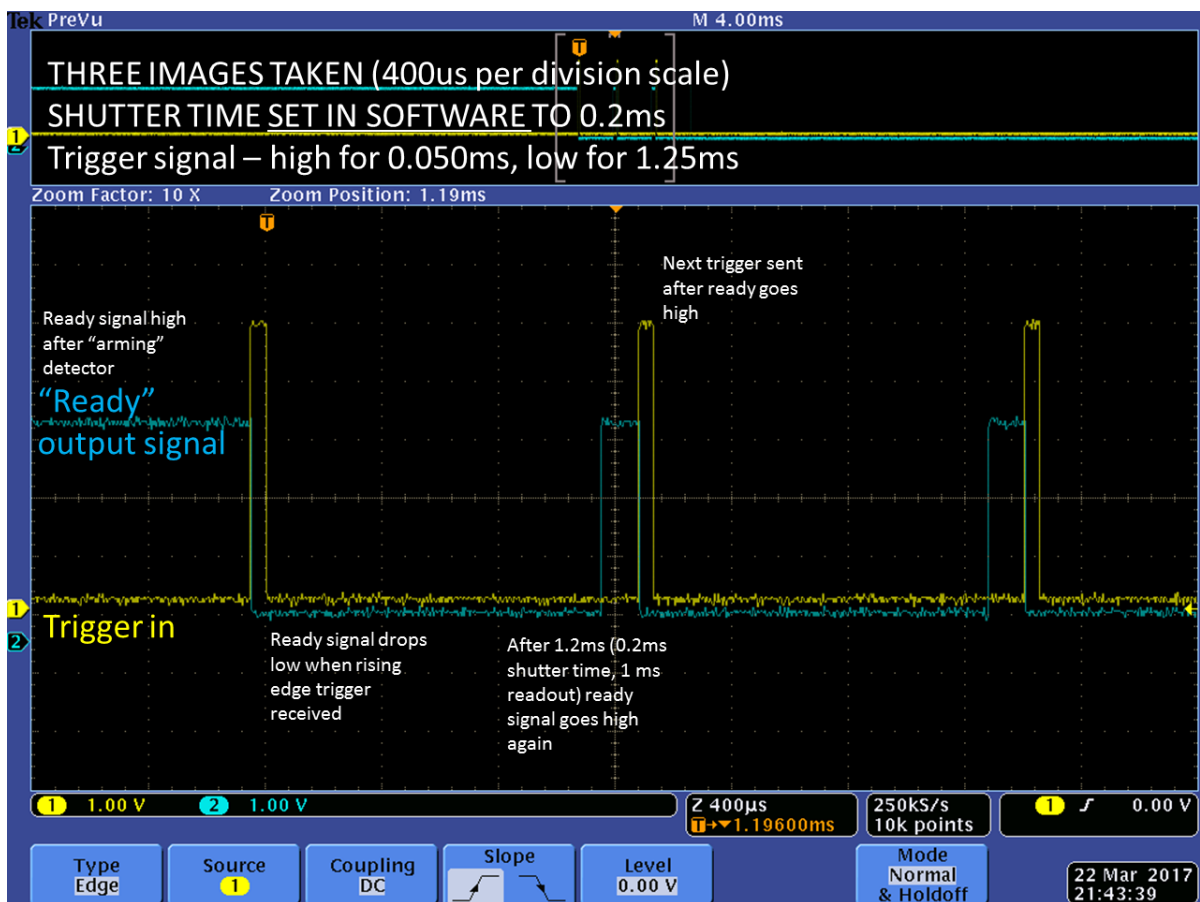
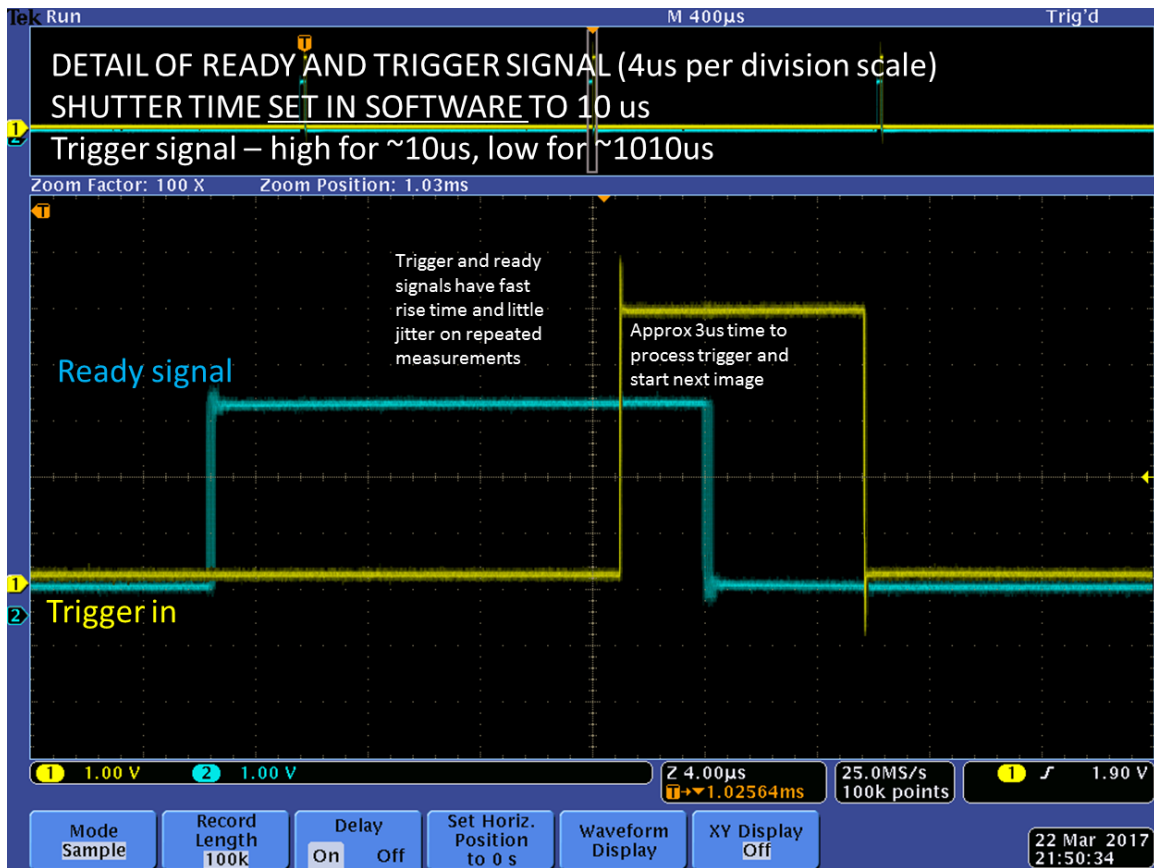
TwentyFourBit, trigger mode 1



<i>OperatingMode</i>	<i>TwentyFourBit</i>	<i>TriggerMode</i>	2
<i>StartAcq</i>		Starts the Acquisition	
<i>ShutterTime</i>		Min value: 0.01 ms	
<i>FrameNumbers</i>		Sets the number of exposures of <i>ShutterTime</i> length each	
HW Trigger IN		Triggers start of next exposure at rising edge	
HW Trigger OUT		Shows that the detector is ready for next Trigger.	
<p>Note: In <i>TwentyFourBit</i> mode, it takes 1ms to read out the frame. Thus, the period of an exposure is the sum of <i>ShutterTime</i> and 1 ms. You need to take care of this when triggering.</p>			

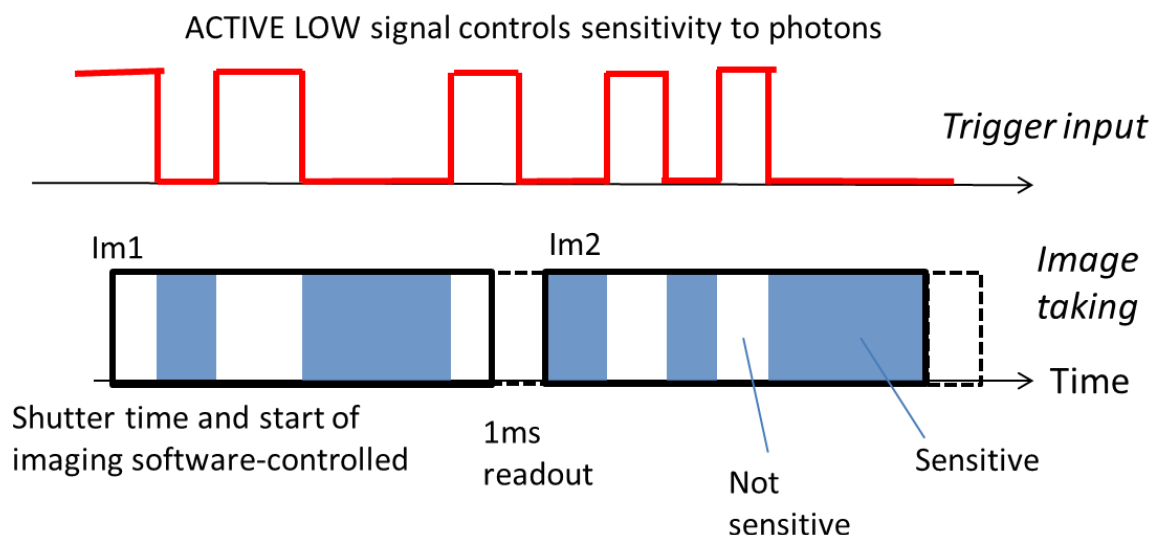
TwentyFourBit, trigger mode 2





<i>OperatingMode</i>	<i>TwentyFourBit</i>	<i>TriggerMode</i>	3
<i>StartAcq</i>		Starts the Acquisition	
<i>ShutterTime</i>		Min value: 0.01 ms	
<i>FrameNumbers</i>		Sets the number of exposures of <i>ShutterTime</i> length each	
HW Trigger IN		Initially, rising edge starts acquisition (like in mode 1). After this, logic HIGH will make the detector sensitive to photons, and LOW will make it insensitive.	
HW Trigger OUT		Shows that the detector is ready for initial Trigger.	
<p>This uses the trigger signal as a <i>gate</i> signal, to control when the detector is sensitive to photons. This gating is independent of the time per exposure, i.e. it is possible to make the detector sensitive and insensitive arbitrarily within the exposure time, for example in pump-probe experiments.</p>			

TwentyFourBit, trigger mode 3 (gating mode)



The plot below shows measurements performed with the LAMBDA detector with a 92 ns-wide gate (thanks to PETRA-III P08 beamline and the University of Kiel). The synchrotron was operating with 200 ns bunch spacing, and by varying the gate timing the intensity on the detector varied, due to the gate being applied either in time with the bunch or between the bunches.

