DECTRIS
detecting the future

User Manual
DECTRIS QUADRO®

Document Version v1.7.2

DECTRIS Ltd.

5405 Baden-Daettwil
Switzerland
www.dectris.com

Document Version v1.7.2

CONTENT

1. GENERAL INFORMATION

1.1. Contact and Support

Address:	DECTRIS Ltd. Taefernweg 1 5405 Baden-Daettwil Switzerland
Phone:	+41 56 500 21 02
Fax:	+41 56 500 21 01
Homepage:	http://www.dectris.com/
Email:	support@dectris.com

Should you have questions concerning the system or its use, please contact us via telephone, mail or fax.

1.2. Explanation of Symbols

Warning #0

Warning blocks are used to indicate danger or risk to personnel or equipment.

Caution #0

Caution blocks are used to indicate danger or risk to equipment.

Information #0

Information blocks are used to highlight important information.

1.3. Warranty Information

Caution	#1

Do not ship the system back before you receive the necessary transport and shipping information.

1.4. Disclaimer

DECTRIS® has carefully compiled the contents of this manual according to the current state of knowledge. Damage and warranty claims arising from missing or incorrect data are excluded.

DECTRIS® bears no responsibility or liability for damage of any kind, also for indirect or consequential damage resulting from the use of this system.

DECTRIS® is the sole owner of all user rights related to the contents of the manual (in particular information, images or materials), unless otherwise indicated. Without the written permission of DECTRIS® it is prohibited to integrate the protected contents in this publication into other programs or other websites or to use them by any other means.

DECTRIS® reserves the right, at its own discretion and without liability or prior notice, to modify and/or discontinue this publication in whole or in part at any time, and is not obliged to update the contents of the manual.

2. SAFETY INSTRUCTIONS

Caution	#2

Please read these safety instructions before operating the detector system.

- Before turning the power supply on, check the supply voltage against the label on the power supply. Using an improper main voltage will destroy the power supply and damage the detector.
- Power down the detector system before connecting or disconnecting any cable.
- Make sure the cables are connected and properly secured.
- Avoid pressure or tension on the cables.
- The detector system should have enough space for proper ventilation. Operating the detector outside the specified ambient conditions could damage the system.
- Ensure that the detector is switched off before pumping or venting the detector head. Operating the detector outside the specified pressure range could damage the system.
- Place the protective cover on the detector when it is not in use to prevent the detector from accidental damage.
- Opening the detector or the power supply housing without explicit instructions from DECTRIS® will void the warranty.
- Do not install additional software or change the operating system.
- Do not touch the sensor or contaminate any vacuum related part.

3. SYSTEM DESCRIPTION

3.1. Components

The QUADRO detector system consists of the following components:

- QUADRO detector
- Power supply for the detector[1]
- Detector control unit
- Thermal stabilization unit[2]
- Accessories
- Documentation

3.2. Hybrid Pixel Technology

3.2.1. Basic Functionality

DECTRIS® detectors provide direct detection of electrons with optimized solid-state sensors and CMOS readout ASICs in hybrid pixel technology. Well-proven standard technologies are employed independently for both the sensor and the CMOS readout ASIC. The detectors operate in single-electron counting mode and provide outstanding data quality. They feature very high dynamic range, zero dark signal and zero readout noise and hence achieve optimal signal-to-noise ratio at short readout time and high frame rates.

Key Advantages

- Direct detection of electrons
- Single-electron counting
- Excellent signal-to-noise ratio and very high dynamic range (zero dark signal, zero noise)
- Two 16 bit digital counters
- Short readout time and high frame rates
- Shutterless operation

The QUADRO hybrid pixel detector is composed of a sensor, a two-dimensional array of pn-diodes processed in a high-resistivity semiconductor, connected to an array of readout channels designed in advanced CMOS technology.

3.2.2. Continuous Readout

One of the hallmark features of QUADRO is its continuous readout that enables high frame rates at high duty cycles. Every pixel of an QUADRO ASIC features two digital counters. After acquisition of a frame, the pixels switch from counting in one digital counter to the other. While one counter is being read out, data acquisition continues in the other counter.

3.2.3. Auto-Summation

QUADRO auto-summation mode is a further benefit of continuous readout with high duty cycle. While a single frame is limited to the 16 bit of the digital counter, auto-summation extends the data depth up to 32 bit, or more than 4.2 billion counts per pixel, depending on the number of summed frames in an image. At short exposure times and high frame rates, all counts are captured in the digital counter of a pixel and directly read out as an image. If long exposure times are requested, frames are still acquired at high rates on the pixel level, effectively avoiding any overflows. The detector system sums the frames to images on the fly, extending the bit depth of the data by the number of summed frames.

[1] Some systems might be delivered without an external power supply unit. Please consult the Technical Specifications for more information.
[2] Some systems might be delivered without a thermal stabilization unit. Please consult the Technical Specifications for more information.

3.2.4. Region-of-Interest (ROI)

QUADRO provides a readout mode that enables high frame rates by reducing the readout area to a region-of-interest (ROI) and/or reducing the number of bits per pixel. The user can set the number of lines that will be read out symetrically around the horizontal center line (roi_y_size). The resulting y-size of the ROI image is determined by y-size = 2 x roi_y_size + 2 with a maximum of 512 pixels. The x-size of the image is kept constant. A schematic representation of the readout area is give in figure 3.1.

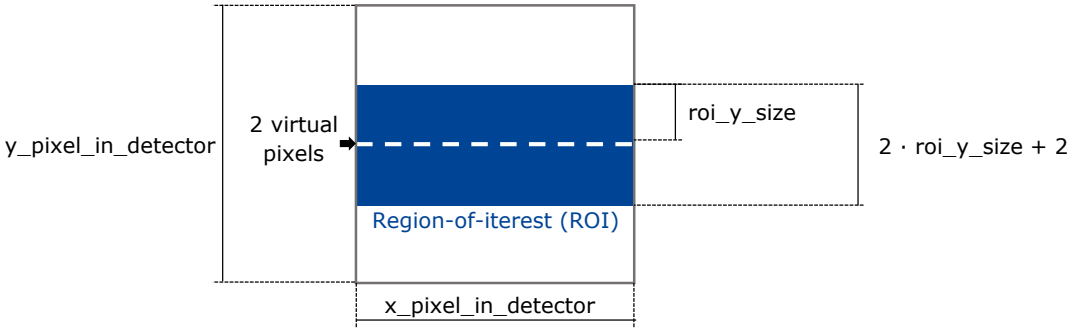

Figure 3.1: Readout area for ROI mode (not to scale)

A list of the highest achievable framerates at specific settigs is given in the table below.

Table 3.1: Highest achievable framerates in ROI mode

roi_y_size	image size [w x h]	roi_bit_depth	highest framerate
256	512 pixel x 512	16	2250 fps
256	512 pixel x 512	8	4500 fps
128	512 pixel x 258	16	4500 fps
128	512 pixel x 258	8	9000 fps
64	512 pixel x 130	16	9000 fps
64	512 pixel x 130	8	18000 fps

For more information on how to enable this feature please refer to section 7.3.2.

3.3. Software

3.3.1. Overview of SIMPLON

The QUADRO detector system is controlled via the SIMPLON API, which relies on a http/REST interface. The API Reference is provided as a separate document "SIMPLON API Reference".
The detector's web interface (chapter 6) gives access to fundamental settings and status parameters and also enables a first test to see if the detector system has been set up properly (after installation and startup as described in chapters 4 and 5).
The QUADRO detector writes images in the HDF5 file format (chapter 9). DECTRIS® provides the image viewer ALBULA, which is able to handle the HDF5 images, with the primary aim to display them. ALBULA is available free of charge for the platforms Linux and Windows (for download please go to http://www.dectris.com). The ALBULA version for Linux and Windows comes with a Python API for handling the HDF5 files that allows performing arithmetic operations on image data as well as basic analysis. Furthermore, the API enables seamless integration of the viewer into an existing infrastructure. More information on HDF5 and ALBULA is given in chapter 9.

4. QUICK START GUIDE

4.1. Accessing the Detector Control Unit

The QUADRO detector is controlled via the network interface of the detector control unit. Hence, the IP network address of the detector control unit has to be known to be able to connect to the API. Depending on the network structure, there are several ways of determining the IP network address, which are described below.

- See the Technical Specifications for the default network port configuration of your detector control unit.
- The default network port configuration may be changed through the detector's web interface (see section 6.2).

4.1.1. Using DHCP

If there is a DHCP server available on the network, plug the network cable into a port of the detector control unit pre-configured for DHCP. See the Technical Specifications for the default network port configuration of your detector control unit.

To determine the IP of your detector, plug in a keyboard and monitor to the detector control unit and power it up. Then follow the following steps:

- Once the detector control unit is fully booted, a command line login prompt will appear.
- Type **`recovery`** and press return.
- If a password prompt appears, leave it empty and press return. The login name was most likely misspelled, restart from point 1.
- Type **`i`** to select option **`(i) show ip addresses`**.
- The IP address will be displayed on the screen.
- If no IP is displayed, make sure that the DCU is properly connected to your network.

Alternatively, the IP network address can be retrieved by searching for the MAC address on the network. For network safety reasons, please ask the network administrator for assistance in obtaining the IP address. If you are the network administrator or have the required permission, the following Linux command can be used to retrieve the IP network address:

[]$_ Linux Command

```
sudo nmap -sP xxx.xxx.xxx.xxx/24 | awk '/^Nmap/{ip=$NF}/yy:yy:yy:yy:yy:zz/{print ip}'\\
```

where xxx.xxx.xxx.xxx/24 is the network address range to be scanned (e.g. 192.168.0.1/24) and yy:yy:yy:yy:yy:zz is the MAC address of the DHCP network port in the back of the detector control unit. You can find the MAC address of the port using the method described below.

Determining the MAC Address of a Port

The MAC address of the first network port can be found on the bottom of the service tag label (pull-out label in the front of the detector control unit, the correct address is the **`EMBEDDED NIC 1 MAC ADDRESS`**). The MAC addresses of the second, third and fourth ports are the same as the first one, but with the last two digits incremented by zz+2, zz+4 and zz+6, respectively. E.g. if the first port is 01:23:45:67:89:ab, then the second port is 01:23:45:67:89:ad (make sure to use the hexadecimal system).

4.1.2. Using a Fixed IP

If you want to access the detector control unit using a fixed IP network address, plug the network cable into the service port of your detector control unit pre-configured for a fixed IP. See the Technical Specifications for the network port configuration of your detector control unit and configure your network accordingly.

If you use e.g., a laptop to access the detector control unit directly for the initial configuration, you can use the following network settings on the laptop:

Table 4.1: Network Settings

IP Adress	169.254.254.100
Subnet Mask	255.255.0.0
Default Gateway	not required

5. GETTING STARTED

Information #1

Instructions on properly installing the detector system on a Transmission Electron Microscope can be found in the Technical Specifications. Before operating the detector, read the complete documentation.

Warning #1

Pay attention to POWER OFF the detector by disconnecting the power supply before pumping or venting the detector chamber.

5.1. Startup Procedure

- Make sure the detector is properly mounted on the Transmission Electron Microscope and the detector chamber is evacuated.
- Connect and turn on the thermal stabilization unit. Set the operation temperature as specified in the Technical Specifications.
- Connect all the required cables for power, data, and trigger signals (if applicable) to the detector and the detector control unit.
- Power on the detector. Pay attention not to power on the detector outside the specified temperature and pressure range.
- Turn on the detector control unit. Please allow at least 5 minutes for the BIOS test procedures and startup of software to complete.
- Check the web interface for system status or start using the SIMPLON API.

6. WEB INTERFACE

6.1. Overview

The QUADRO web interface (figure 6.1) provides simple access to basic functions and settings of the detector system for installation, debugging, and system updates. For productive operation of the detector, please refer to the API Reference.

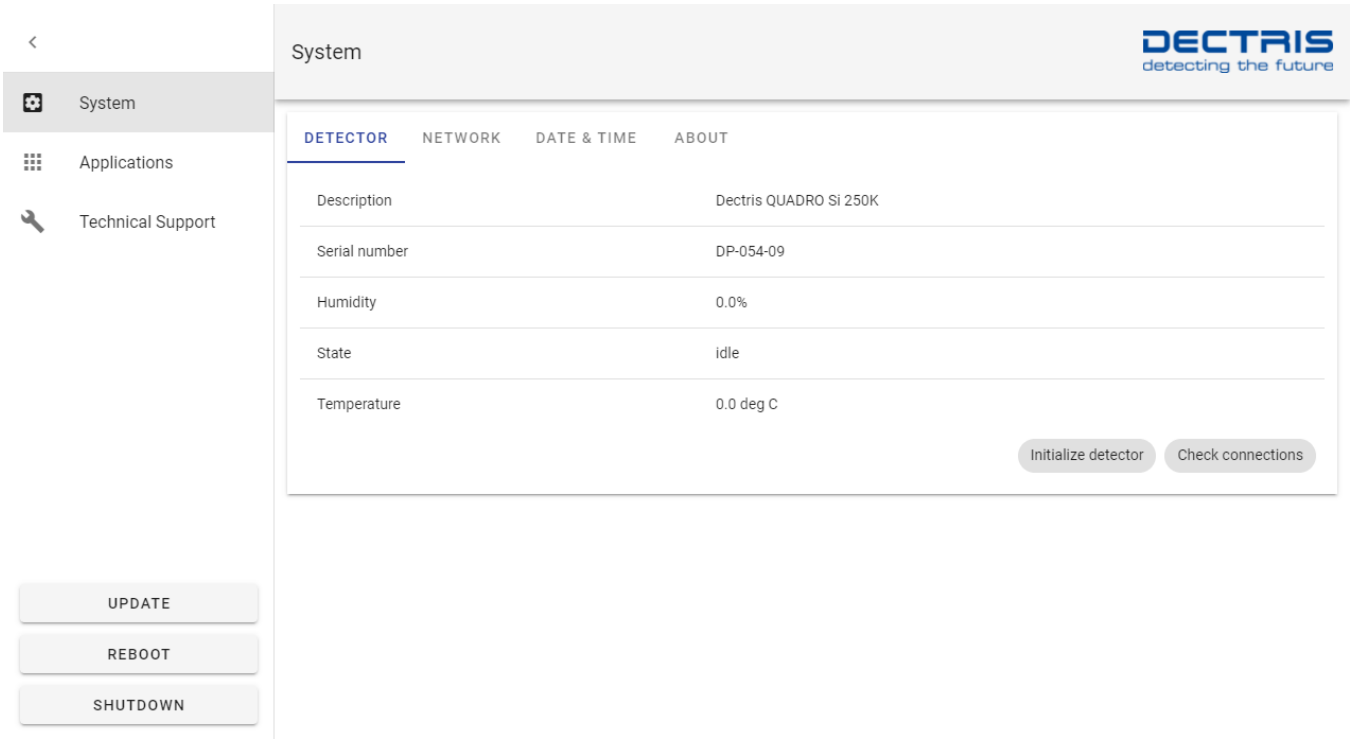

Figure 6.1: QUADRO home page.

Table 6.1 summarizes the functions available through the left panel of the web system interface.

Table 6.1: Menu items of the QUADRO web interface.

Menu Point	Content
System	View the system information and access the detector control unit system settings (see section 6.2)
Applications	Manage the detector and calibration applications. This tab provides functions to start and stop the detector software, change software versions and upgrade the detector software to a new version.
Technical Support	Simple interface to create a bug-report. The bug-report creates a tarball that can be downloaded and sent to DECTRIS® support at support@dectris.com. The bug report is not sent automatically.

6.2. System Settings and Administration

To access the QUADRO system settings, click on the corresponding tab on the homepage. The system tab allows to configure the detector control unit network settings and get some status and system informations.

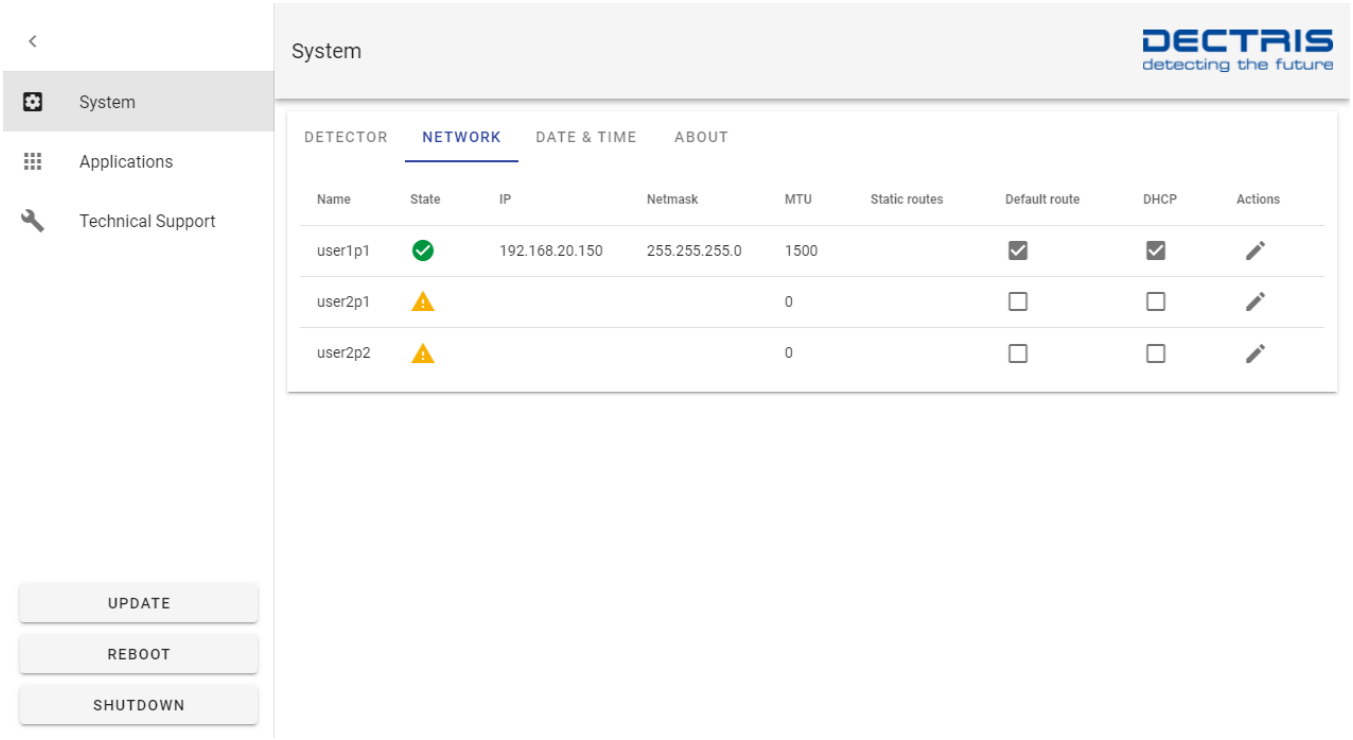

Figure 6.2: Screenshot of the system settings showing the network configuration page.

Table 6.2 summarises menu items available for configuration

Table 6.2: Menu Items for Configuration/Testing

Detector	See the detector status and initialize the detector. The detector status can only be read out when the detector is initialized and the detector application is running. This tab also provides a way to quickly check the quality of the detector connection using the "Check connections" button. Typical values are above 0.15 mW for the TX power and above 0.3 mW for the RX power.
Network	Configure the user accessible network interfaces.
Date & Time	Configure date and time on the detector control unit. Provides the possibility to add an NTP server.
About	View system information.

7. GENERAL USAGE OF THE DETECTOR SYSTEM

7.1. Detector Control and Output

The QUADRO detector system is controlled through the SIMPLON API, an interface to the detector that is based on the http protocol and implemented on the detector control unit. The API Reference supplied with the system describes this interface in detail and allows for easy integration of detector control into instrument control or similar software. Please refer to the API Reference for details.

The data recorded by the detector can be accessed in different ways. Images can be stored by the filewriter on the detector control unit as HDF5 files (see chapter 9). HDF5 files include meta-data in a NeXus-compatible format. Buffered files have to be regularly fetched and subsequently deleted on the detector control unit as buffer space is limited[1]). Data can also be fetched through the stream API interface. The stream interface relies on ZeroMQ[2], a distributed messaging protocol. The stream has a low latency and offers utmost flexibility. The meta-data is transferred as part of the header. Streamed data is not buffered and will be lost if not fetched or incompletely fetched.

7.2. Recording an Image or an Image Series

Caution	#3

Data might have to be fetched concurrently to a running image series. The lifespan of the data on the detector control unit is dependent on the configuration of your system as well as the interface used for collecting data. Data not fetched within this lifespan is permanently lost.

To record images or image series, the following steps need to be performed through the SIMPLON API (see the API Reference for details).

1. Make sure the detector has been set up according to the steps described in chapter 5.
2. Initialize the detector if this has not yet been done. The detector does not need to be reinitialized unless the detector entered an error status.
3. Set the detector parameters for data acquisition and specify and configure the desired output interface (file writer and/or stream interface). A list of essential configuration parameters can be found in section 7.3.1.
4. Arm the detector
5. Record the image or image series.
    - Send trigger(s) to record the image or image series as previously configured.
    - Fetch data through the data interface(s).
6. Disarm the detector (to ensure files are finalized and closed).
7. Repeat from step 2 for further data acquisition with different settings or to step 3 for identical settings.

[1] Buffer space varies dependent on the configuration of your system, buffer overflow will cause loss of data. See API Reference for further details.
[2] ZeroMQ distributed messaging (http://zeromq.org/)

7.3. Control of the Detector from a Specific Environment

Integrating the detector into a specific environment requires understanding of the necessary detector functions. The API reference will list all possible commands and features, but it does not give an explanation of the required functionality. Sections 7.3.1 and 7.3.2 cover a selection of essential and situational parameters respectively.

7.3.1. Main Configuration Parameters

The parameters described in this section allow control of the detector and data acquisition. Data will be acquired, however further configuration of the interface for data retrieval might be necessary depending on your set up. For starting a data acquisition, only the following parameters need to be adjusted.

- detector | config | nimages
- detector | config | count_time
- detector | config | frame_time
- detector | config | incident_energy

The detector configuration parameter incident_energy has to be set to the electron energy used for the experiment. The difference between the timing parameters frame_time and count_time has to be greater than the detector_readout_time. The detector_readout_time can be read back from the API. Count_time refers to the actual time the detector counts electrons and frame_time is the interval between acquisitions of subsequent frames (i.e. period). The number of images in a series of images, after a trigger, is configured with the parameter nimages. The detector always considers a trigger as the start of a series of n images. For example a single image is considered as a series of images containing 1 image. Once the detector has been armed a series can be started by issuing a trigger command or triggering the detector using an electronic pulse on the external trigger input (ExtIn). To switch between the trigger modes (see chapter 8) one can use the configuration parameter trigger_mode.

- detector | config | trigger_mode
- detector | config | ntrigger

Setting values greater than 1 for ntrigger allows several trigger commands or external trigger pulses per arm/disarm sequence. This mode allows recording several series of nimages with the same parameters. The resulting number of frames is product of ntrigger and nimages. In external enable modes the parameter nimages is ignored (i.e. always 1) and the number of frames therefore has to be configured using the detector configuration parameter ntrigger.

Information	#2

Please note that data be retrieved in different ways. For details, please see the API Reference.

With the filewriter enabled, the acquired data is written into HDF5 files. The filewriter has the following important configuration parameters:

- filewriter | config | name_pattern
- filewriter | config | nimages_per_file
- filewriter | config | compression_enabled

The filewriter parameter name_pattern sets the name template/pattern for the HDF5 files. The pattern "$id" is replaced with a sequence identification number and therefore can be used to discriminate between subsequent series. The sequence identification number is reset after initializing the detector. The parameter nimages_per_file sets the number of images stored per data file. A value of 1000 (default) means that for every 1000th image, a data file is created. If for example, 1800 images are expected to be recorded, the arm, trigger, disarm sequence means that a master file is created in the data directory after arming the detector. The trigger starts the image series and after 1000 recorded images one data container is made available on the buffer of the detector control unit. No further files will made available until the series is finished either by completing the nth image (nimages) of the nth trigger (ntrigger) or by ending the series using the detector command disarm. As soon as either criteria is met the second data container is closed and made available for fetching.

7.3.2. Additional Configuration Parameters

Information	#3

The following parameters are for special conditions and should be set with care and with understanding of the consequences. Changing these parameters to non-default values can have a substantial negative impact on data quality!

Corrections are enabled by default, but can be turned off using the following detector configuration parameters. In the vast majority of experiments data quality benefits from the data corrections. Therefore, disabling either correction will likely result in inferior data quality.

- detector | config | countrate_correction_applied
- detector | config | flatfield_correction_applied
- detector | config | pixel_mask_applied

The ROI mode (see section 3.2.4) offers high framerate by reducing the readout area and/or the number of bits per pixel. It is disabled by default. For using the ROI mode the following configuration parameters need to be adjusted:

- detector | config | roi_mode
- detector | config | roi_bit_depth
- detector | config | roi_y_size

To enable the ROI mode, the parameter roi_mode has to be set to "lines". The roi_bit_depth parameter defines the number of bits per pixel. It can be either set to 8 or 16. Operating the detector in 8 bit mode is necessary to reach the highest specified framerate. The resulting y-size of the image is determined by the parameter roi_y_size and can be calculated by y-size = 2 x roi_y_size + 2. The x-size of the image is the same as with disabled ROI mode. To disable the ROI mode, the configuration parameter roi_mode has to be set to "disabled".

Information	#4

Note: The counter in 8 bit mode does not have an overflow protection. When exceeding the maximum value of 255, the counter overflows to zero and starts counting again. This might lead to uninterpretable results.

A Python example on how to enable the ROI mode and take a single image with the filewriter interface is given below.

[]$_ Example taking an image with ROI mode enabled

```
import json
import requests

# simple detector client
class DetectorClient:
    def __init__(self, ip , port):
        self._ip = ip
        self._port = port

    def set_config(self, param, value, iface = 'detector'):
        url = 'http://%s:%s/%s/api/1.8.0/config/%s' % (self._ip, self._port, iface, param)
        self._request(url, data = json.dumps({'value': value}))

    def send_command(self, command):
        url = 'http://%s:%s/detector/api/1.8.0/command/%s' % (self._ip, self._port, command)
        self._ip, self._port, command
        self._request(url)

    def _request(self, url, data={}, headers={}):
        reply = requests.put(url, data=data, headers=headers)
        assert reply.status_code in range(200, 300), reply.reason


if __name__ == '__main__':
        # create detector instance and initialize the detector
    client = DetectorClient('169.254.254.1', '80')
    client.send_command('initialize')

    # enable the filewriter interface
    client.set_config('mode', 'enabled', 'filewriter')
    client.set_config('name_pattern', 'roi_test_$id', 'filewriter')

    # apply settings for roi feature
    client.set_config('roi_mode', 'lines')
    client.set_config('roi_bit_depth', 8)
    client.set_config('roi_y_size', 128)

    # set count and frame time
    client.set_config('count_time', 1/9000)
    client.set_config('frame_time', 1/9000)

    # arms, triggers and disarm the detector
    client.send_command('arm')
    client.send_command('trigger')
    client.send_command('disarm')
```

Further parameters and their function are described in the API Reference.

7.4. Interdependency of Configuration Parameters

7.4.1. Interdependency of Calibration Parameters

The following calibration parameters have an implied or direct dependency. Changing either of the parameters might influence other parameters in the list.

detector | config | incident_energy

detector | config | counting_mode

Changing incident_energy or changing the counting_mode causes a reset of the flatfield to unity.

detector | config | flatfield

The detector is shipped without flatfields. To ensure optimal data quality the user is encouraged to upload custom flatfields on the detector. More information on how to upload custom flatfields is given in chapter 11.

7.4.2. Interdependency of Timing Parameters

The following parameters are essential for exposure timing. Changing either values might influence other values in the list.

detector | config | detector_readout_time

detector_readout_time may change if incident_energy is changed.

detector | config | frame_time

If frame_time conflicts with the current count_time, count_time is set to the difference of frame_time and detector_readout_time. The auto summation parameter frame_count_time may be updated as well.

detector | config | count_time

If count_time conflicts with frame_time, frame_time is set to the sum of count_time and detector_readout_time. The auto summation parameters frame_count_time, frame_period and nframes_sum may be updated as well. To acquire images with a certain frame rate and best possible duty cycle, a simple procedure is to first set count_time to the inverse of the frame rate and subsequently frame_time to the inverse of the frame rate.

8. TRIGGER USAGE

8.1. Introduction

Information	#5

If the settings or the external trigger/enable pulses applied are out of specification, acquisitions will not be performed and the measurement obtained with the detector might be incomplete.
All values used in the example are for demonstrational purposes only and should be adapted to meet the requirements of your application.

In order to record an image or a series of images, the QUADRO detector has to be initialized, configured, armed, and the exposure(s) started by a trigger signal. The steps necessary to record an image series are comprehensively described in chapter 5 and section 7.2. The detector can be triggered through software (internal trigger) or by an externally applied trigger signal (external trigger). Four different trigger modes are available and described in the following sections 8.2 to 8.5.

8.2. INTS - Internal (Software) Triggering

An exposure (series) can be triggered by using a software trigger. This is the default mode of operation.

Example detector configuration for internally triggered exposure series:

detector | config | trigger_mode	{"value": "ints"}
detector | config | frame_time	{"value": 1}
detector | config | count_time	{"value": 0.7}
detector | config | nimages	{"value": 10}

The detector starts the first exposure after the trigger command has been received and processed[1]. All subsequent frames are triggered according to the configuration of the frame_time and count_time parameters. The detector records nimages frames per trigger and stays armed until ntrigger are received. Figure 8.1 depicts an internally triggered series defined by frame_time, count_time and nimages.

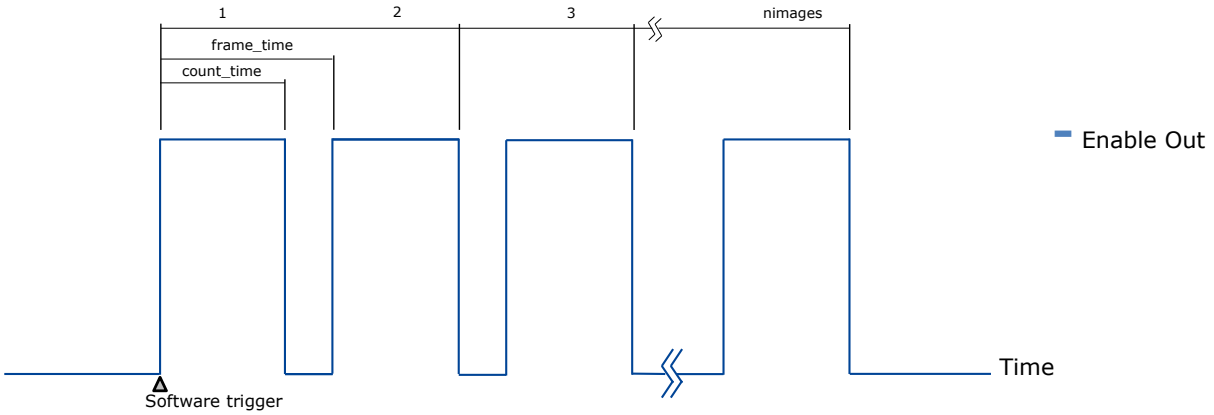

Figure 8.1: Series of exposures, defined by frame_time, count_time and nimages, triggered by a software trigger.

[1] As the trigger command is sent over an TCP/IP connection the exact latency of the start of the exposure is hard to predict.

8.3. INTE – Internal (Software) Enable

In the trigger_mode 'inte' a single exposure or series of individual exposures can be started by issuing a number of (ntrigger) trigger commands. Unlike trigger commands in the trigger_mode 'ints', 'inte' trigger commands take an (optional) argument containing the count_time for the subsequent frame. In all enable modes the detector configuration parameter nimages is implied to be 1. The number of frames in a series therefore is solely based on the value of the parameter ntriggger.

	Information	#6
	The configured count_time and frame_time should be close to the count time and frame time of the shortest expected exposure in the configured series. The set count_time will be used to calculate internal auto-summation configuration values (section 3.2.3). In most situations a reasonable estimate of these values is sufficient.	

Example detector configuration for an internally enabled exposure series:

detector | config | trigger_mode	{"value": "inte"}
detector | config | nimages	{"value": 1}
detector | config | ntrigger	{"value": 3}
detector | config | frame_time	{"value": 1.0} (see 6)
detector | config | count_time	{"value": 0.7} (see 6)

The detector starts the first exposure after a trigger command has been received and processed[2]. All subsequent frames have to be triggered by individual trigger commands with an (optional) argument containing the count_time of the triggered frame. The detector stays armed until ntrigger are issued or the detector is disarmed. Figure 8.2 depicts an internally enabled exposure series defined by count_time (payload of the trigger command) and ntrigger. Table 8.3 summarises the commands issued to record the same series.

Table 8.3: Command sequence for an internally enabled (inte) series.

Method	Parameter	Payload
PUT	detector | command | arm	
PUT	detector | command | trigger	{"value": 0.7}
PUT	detector | command | trigger	{"value": 2.1}
...		
PUT	detector | command | trigger	{"value": 0.7}

[2] As the trigger command is sent over an TCP/IP connection the exact latency of the start of the exposure is hard to predict.

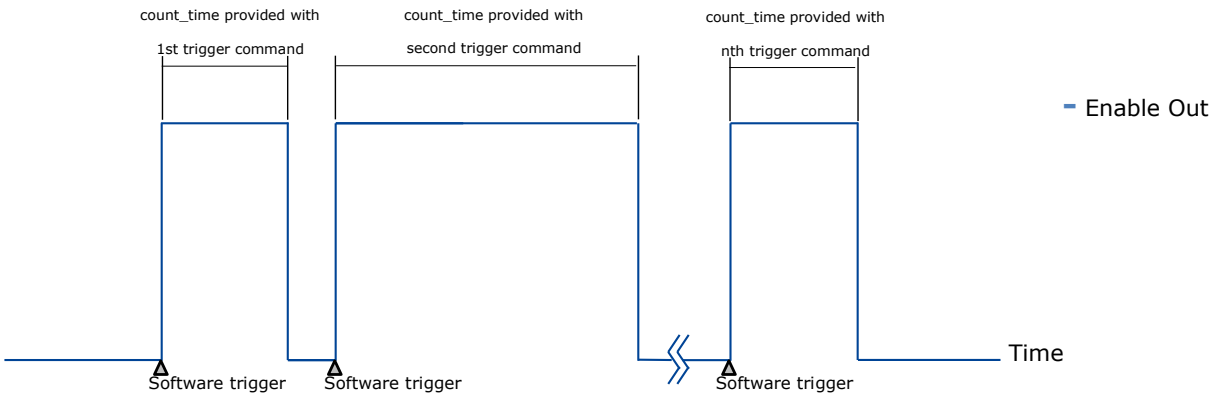

Figure 8.2: Series of exposures, defined by count_time (payload of the trigger command) and ntrigger, triggered by a software trigger.

8.4. EXTS - Externally Triggered Exposure Series

Caution	#4

Consult the Technical Specifications for details about the required electrical characteristics of the trigger signal.

The QUADRO detector systems also support external triggering. In the trigger_mode 'exts', nimages are recorded per trigger until ntrigger are received. Both count_time as well as frame_time are defined by the configuration.
Example detector configuration for externally triggered exposure series:

detector | config | trigger_mode	{"value": "exts"}
detector | config | frame_time	{"value": 1.0}
detector | config | count_time	{"value": 0.7}
detector | config | nimages	{"value": 10}
detector | config | ntrigger	{"value": 1}

After the detector has been initialized, configured, and armed the acquisition can be triggered by a single external trigger pulse. The detector starts exposing after the (electrical) trigger signal has been issued. All subsequent frames are internally triggered according to the information previously configured by the frame_time and count_time parameters. The detector records nimages frames and stays armed until ntrigger are received. Figure 8.3 depicts an externally triggered series defined by frame_time, count_time and nimages.

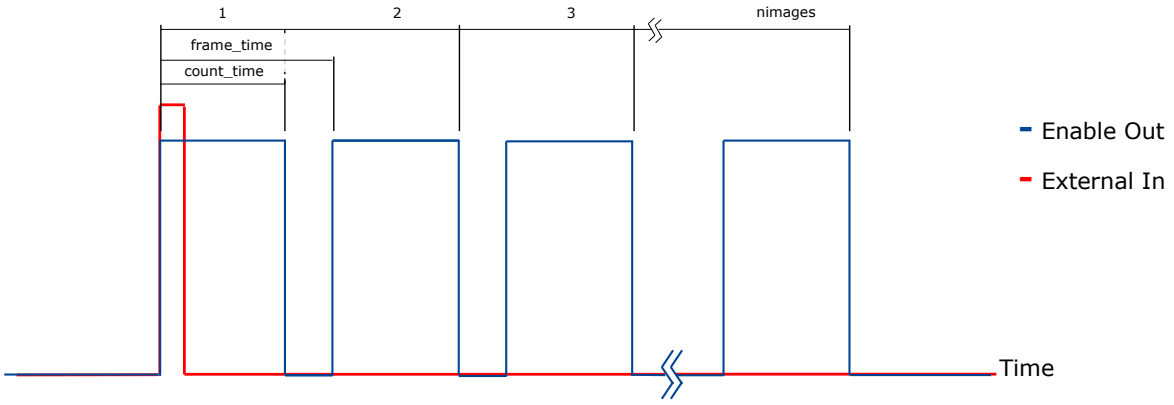

Figure 8.3: Exposure series defined by frame_time, count_time and nimages, triggered by a single external trigger pulse. Note that the periods are not drawn true to scale.

8.5. EXTE - Externally Enabled Exposure Series

Caution #5

Consult the Technical Specifications for details about the required electrical characteristics of the trigger signal.

The QUADRO detector systems also support external enabling. In the external enable mode 'exte' a series of ntrigger frames can be recorded. The count time as well as the period of individual frames of a series are defined by the duration of the high state of the external trigger/enable signal. In all enable modes the detector configuration parameter nimages is implied to be 1. The number of frames in a series therefore is solely based on the value of the parameter ntriggger.

Information #7

The configured count_time and frame_time should be close to the count time and frame time of the shortest expected exposure in the configured series. The set count_time will be used to calculate internal auto-summation configuration values (section 3.2.3). In most situations a reasonable estimate of these values is sufficient.

Example detector configuration for externally enabled exposure series:

detector | config | trigger_mode	{"value": "exte"}
detector | config | nimages	{"value": 1}
detector | config | ntrigger	{"value": 10}
detector | config | frame_time	{"value": 1.0} (see 7)
detector | config | count_time	{"value": 0.7} (see 7)

After arming the detector, the acquisition can be enabled by an external signal. The value ntrigger defines how often this can be repeated. The detector starts exposing the first image after the rising edge and stops after the falling edge of the external trigger signal. In the same manner, all subsequent frames are externally enabled. The count time and period are therefore solely determined by the external enable signal and the limitations of your detector system. The detector records

as many frames as valid (according to the specifications) enable pulses are received until the value set for ntrigger is reached. Figure 8.4 illustrates a externally enabled series.

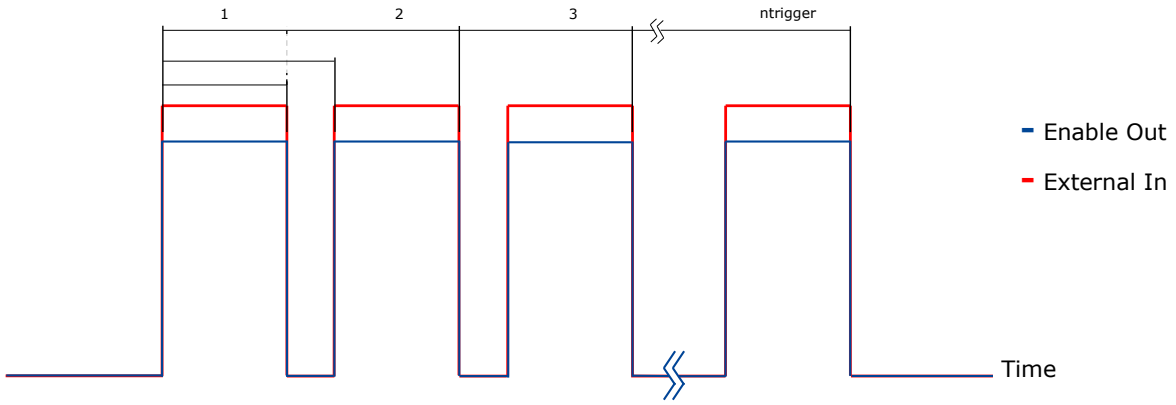

Figure 8.4: Exposures defined by external enable

9. HDF5 AND ALBULA

9.1. ALBULA Overview

ALBULA is a cross-platform image viewer developed and maintained by DECTRIS®. The Linux and Windows versions also provide an image library for the Python language.

ALBULA can be downloaded for free[1] at www.dectris.com. Scripts written in Python using ALBULA can be used to read, display and store data taken by the QUADRO detectors.

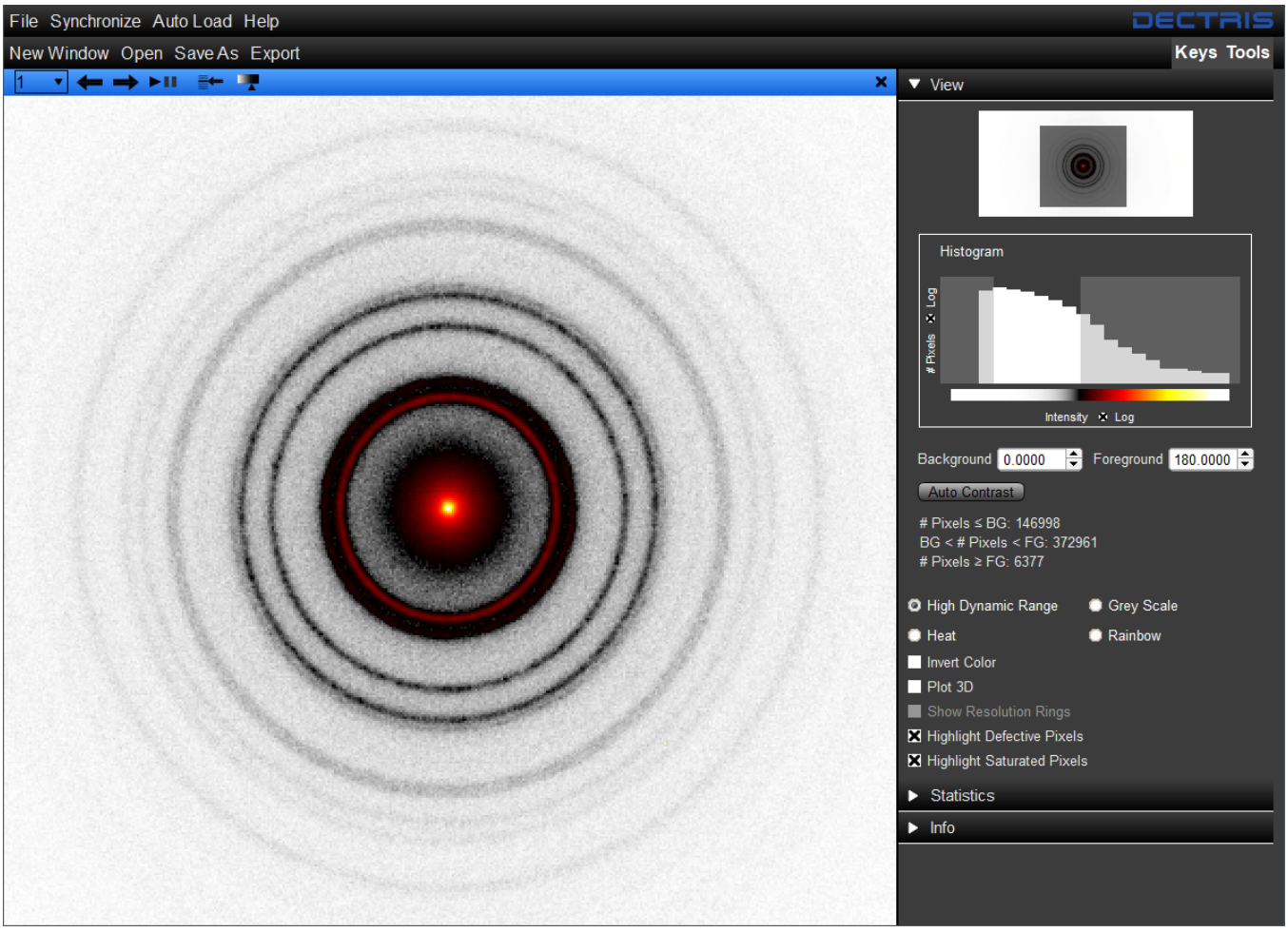

Figure 9.1: Screenshot of ALBULA showing an electron ring diffraction pattern acquired by an ELA

[1] Registration required

9.2. ALBULA HDF5 Python Library (Linux and Windows only)

The following examples illustrate how the data stored in HDF5 files by the QUADRO detector can be manipulated with ALBULA.

9.2.1. Getting Started

[]$_ Example ALBULA

```
# import the dectris.albula image library
import sys
sys.path.insert(0,'/usr/local/dectris/python')
import dectris.albula as albula

# open the albula viewer
m = albula.openMainFrame()
s = m.openSubFrame()

# load image series
s.loadFile('series_16_master.h5')

# waits for an input
raw_input('Press any key to exit the program...')
```

9.2.2. Reading data

[]$_ Example ALBULA

```
# read the compressed (or uncompressed) container through the master file
h5cont = albula.DImageSeries('series_16_master.h5')

# loop over the frames and read out optional data
for i in range(h5cont.first(), h5cont.last() + 1):
    # read image
    img = h5cont[i]

    # read header items using convenience functions
    optData = img.optionalData()
    # e.g. count time
    count_time = optData.count_time()

# Read the header item directly without convenience functions
neXusHeader = h5cont.neXus()

# print all header item names with path
neXusRoot = neXusHeader.root()

def iterateChildren(parent):
    if not hasattr(parent, 'children'):
        return set([parent.path()])
    else:
        paths = set()
        for child in parent.children():
            paths.update(iterateChildren(child))
        return paths

for kid in iterateChildren(neXusRoot):
    print(kid)

# extract count_time
count_time = neXusRoot.childElement('/entry/instrument/detector/count_time')

# print value
print('count_time value: %s' % count_time.value())
```

9.2.3. Writing Data

[]$_ Example ALBULA

```
# create additional metadata (e.g. camera_length)
camera_length = neXusHeader.newPath('/entry/instrument/camera_length', albula.DNeXusNode.
    DATA)
camera_length.setValue('420 mm')

# write the (uncompressed) images and the neXus header to a new HDF5 file
HDF5Writer = albula.DHdf5Writer('testContainer.h5', 1000, neXusHeader)
for i in range(h5cont.first(), h5cont.last() + 1):
    img = h5cont[i]
    HDF5Writer.write(img)

# flushing closes the master and the data files
HDF5Writer.flush()

# write the images in the cbf format. Careful: Information from the header will be lost!
for i in range(h5cont.first(), h5cont.last() + 1):
    img = h5cont[i]
    albula.DImageWriter.write(img, 'testImage_{0:05d}.cbf'.format(i))

# write the images in the tif format. Careful: Information from the header will be lost!
for i in range(h5cont.first(), h5cont.last() + 1):
    img = h5cont[i]
    albula.DImageWriter.write(img, 'testImage_{0:05d}.tif'.format(i))
```

9.3. Third Party HDF5 Libraries

The QUADRO HDF5 data can also be directly read with programs using the HDF5 library. By default the QUADRO data is compressed using the BSLZ4[2] algorithm. In order to decompress the data, the HDF5 plug-in filter[3] can be used, see https://github.com/dectris/HDF5Plugin. By setting the environment variable HDF5_PLUGIN_PATH to the path where the compiled plug-in filter can be found, the HDF5 library will decompress the data compressed with LZ4 by itself. If you want to use proprietary software like Matlab, IDL or similar, make sure that the HDF5 library version used by this software is at least v1.8.11 in order for the plug-in mechanism to work.

For developers using C++, example code can be found on the DECTRIS® website, after registration and login.

[2] See: https://code.google.com/p/lz4/, https://github.com/kiyo-masui/bitshuffle

[3] In order to use the filter plug-in mechanism, HDF5 v1.8.11 or greater must be used. See also http://www.hdfgroup.org/HDF5/doc/Advanced/DynamicallyLoadedFilters/HDF5DynamicallyLoadedFilters.pdf

10. PIXEL MASK

10.1. Applying the pixel mask

The detector configuration parameter pixel_mask_applied enables (True) or disables (False) applying the pixel mask on the acquired data. For different incident_energy values these masks may differ from one another. If true (default), pixels which have any bit set in the pixel_mask are flagged with $(2^{\text{image bit depth}} - 1)$. Please consult the API Reference for details on the detector configuration parameter pixel_mask.

10.2. Updating the pixel mask

10.2.1. Overview

Updating the pixel mask of a QUADRO detector system involves four basic steps:

1. Retrieving the current pixel mask from the detector system via the SIMPLON API.
2. Manipulating the pixel mask to add or update pixels.
3. Uploading the updated pixel mask to the detector system via the SIMPLON API.
4. Persistently storing the updated pixel mask on the detector system by sending the detector command arm.

10.2.2. Retrieving the current mask from the detector system

The pixel mask can be retrieved from the detector system by a GET request on the detector configuration parameter pixel_mask. The data of the pixel mask is retrieved either as tiff or in JSON serialization by choosing application/tiff or application/json in the get request accordingly.

10.2.3. Manipulating the pixel mask

Information	#8
For details about the pixel values in the pixel mask and their meaning, consult the API Reference.	

TIFF

If the pixel mask is retrieved and stored as tiff, the uint32 data in the tiff file can be manipulated with ALBULA API.

JSON

If the pixel mask is retrieved as JSON, the HTTP reply has to be parsed correctly into an array. Please see the example below and the API Reference for details. The values in this array can then be manipulated to reflect the required updates of the pixel mask. After updating the array, it has to be serialized again in JSON according to the specifications in the API Reference.

10.2.4. Uploading and storing the pixel mask

The pixel mask is uploaded by sending a PUT request on the detector configuration parameter pixel_mask with the new mask as data. After sending the detector command arm, the updated pixel mask is permanently stored on the detector system.

10.2.5. Python Example

The following Python code using common libraries provides a simple example for updating the pixel mask:

[]$_ Python Code

```
import json
import numpy
import requests
from base64 import b64encode, b64decode

# simple detector client
class DetectorClient:
    def __init__(self, ip , port):
        self._ip = ip
        self._port = port

    def send_command(self, command):
        url = 'http://%s:%s/detector/api/1.8.0/command/%s' % (self._ip, self._port, command)
        self._request(url)

    def get_mask(self, mask):
        url = 'http://%s:%s/detector/api/1.8.0/config/%s' % (self._ip, self._port, mask)
        darray = requests.get(url).json()['value']
        return numpy.frombuffer(b64decode(darray['data']),
            dtype=numpy.dtype(str(darray['type']))).reshape(darray['shape'])

    def set_mask(self, ndarray, mask):
        url = 'http://%s:%s/detector/api/1.8.0/config/%s' % (self._ip, self._port, mask)
        data_json = json.dumps({'value': {
                                '__darray__': (1,0,0),
                                'type': ndarray.dtype.str,
                                'shape': ndarray.shape,
                                'filters': ['base64'],
                                'data': b64encode(ndarray.data).decode('ascii') }})
        self._request(url, data=data_json, headers={'Content-Type': 'application/json'})

    def _request(self, url, data={}, headers={}):
        reply = requests.put(url, data=data, headers=headers)
        assert reply.status_code in range(200, 300), reply.reason

if __name__ == '__main__':
    # create detector instance and initialize the detector
    client = DetectorClient('169.254.254.1', '80')
    client.send_command('initialize')

    # get the pixel mask
    pixel_mask = client.get_mask('pixel_mask')
    # copy the mask to writeable buffer, necessary for numpy>=1.16.0
    pixel_mask = numpy.copy(pixel_mask)
    # set a new dead pixel [y,x]
    pixel_mask[123, 234] = 2
    # set a new noisy pixel [y,x]
    pixel_mask[234, 123] = 8
    # upload the new pixel mask
    client.set_mask(pixel_mask, 'pixel_mask')

    # arm and disarm to test the system
    client.send_command('arm')
    client.send_command('disarm')
```

11. FLATFIELD

11.1. Applying the flatfield

The user is encouraged to use flatfields to counteract variations in the pixel-to-pixel sensitivity. Each flatfield consists of pixel-wise correction factors that are applied on the acquired data. The detector configuration parameter flatfield_correction_applied enables (True) or disables (False) applying the flatfield on the acquired data. Please consult the API Reference for details on the detector configuration parameter flatfield.

Information	#9
The factory calibration of the detector does not contain any flatfields. Therefore, it is the user's responsibility to provide meaningful flatfields in order to ensure optimal data quality.	

11.2. Creating a flatfield

11.2.1. Overview

Creating a custom flatfield for the QUADRO detector system involves two basic steps:

1. Acquire uniformly-illuminated image via the SIMPLON API.
2. Upload pixel-wise correction factors via the SIMPLON API.

11.2.2. Acquire uniformly-illuminated image

Initialize the detector and set the detector configuration parameters to match the intended imaging conditions. Set up the microscope to produce a homogeneous illumination on the detector and acquire an image with at least 10,000 counts per pixel.

Information	#10
Flatfields are valid only for the given set of configuration parameters. Using flatfields recorded with different configuration parameters will have a negative impact on data quality.	

The following Python code using common libraries provides a simple example on how to acquire a single flatfield image:

[]$_ Python Code

```
import json
import requests

# simple detector client
class DetectorClient:
    def __init__(self, ip , port):
        self._ip = ip
        self._port = port

    def set_config(self, param, value, iface = 'detector'):
        url = 'http://%s:%s/%s/api/1.8.0/config/%s' % (self._ip, self._port, iface, param)
        self._request(url, data = json.dumps({'value': value}))

    def send_command(self, command):
        url = 'http://%s:%s/detector/api/1.8.0/command/%s' % (self._ip, self._port, command)
        self._request(url)

    def monitor_image(self, param):
        url = 'http://%s:%s/monitor/api/1.8.0/images/%s' % (self._ip, self._port, param)
        return requests.get(url, headers = {'Content-type': 'application/tiff'}).content

    def _request(self, url, data={}, headers={}):
        reply = requests.put(url, data=data, headers=headers)
        assert reply.status_code in range(200, 300), reply.reason


if __name__ == '__main__':
    # create detector instance and initialize the detector
    client = DetectorClient('169.254.254.1', '80')
    client.send_command('initialize')

    # enable monitor interface
    client.set_config('mode', 'enabled', 'monitor')

    # set detector parameters like e.g. count_time
    client.set_config('count_time', 5)

    # arms, triggers and disarm the detector
    client.send_command('arm')
    client.send_command('trigger')
    client.send_command('disarm')

    # fetch the latest tif
    data = client.monitor_image('monitor')

    # save tif to disk
    with open('flatfield.tif', 'wb') as file:
        file.write(data)
```

11.2.3. Upload pixel-wise correction factors

The flatfield data on the detector is stored as a floating point array of pixel-wise correction factors that are multiplied with the recorded data. In case on an homogeniously-illuminated image, pixel-wise correction factors can be calculated by dividing the counts of each pixel by the average counts in the image. The flatfield on the detector is updated by sending a PUT request on the detector configuration parameter flatfield with the array of correction factors as data.

Information	#11
	Uploaded flatfields are lost after (re-)initializing the detector or changing configuration parameters like e.g. incident_energy or counting_mode.

The following Python code using common libraries provides a simple example for updating the flatfield:

[]$_ Python Code

```
import json
import numpy
import requests
from base64 import b64encode, b64decode
import tifffile

# simple detector client
class DetectorClient:
    def __init__(self, ip , port):
        self._ip = ip
        self._port = port

    def send_command(self, command):
        url = 'http://%s:%s/detector/api/1.8.0/command/%s' % (self._ip, self._port, command)
        self._request(url)

    def set_mask(self, ndarray, mask):
        url = 'http://%s:%s/detector/api/1.8.0/config/%s' % (ip, port, mask)
        data_json = json.dumps({'value': {
                                '__darray__': (1,0,0),
                                'type': ndarray.dtype.str,
                                'shape': ndarray.shape,
                                'filters': ['base64'],
                                'data': b64encode(ndarray.data).decode('ascii') }})
        _request(url, data=data_json, headers={'Content-Type': 'application/json'})

    def _request(self, url, data={}, headers={}):
        reply = requests.put(url, data=data, headers=headers)
        assert reply.status_code in range(200, 300), reply.reason

if __name__ == '__main__':
    # create detector instance and initialize the detector
    client = DetectorClient('169.254.254.1', '80')
    client.send_command('initialize')

    # set detector configuration paramters here

    # read image with homogeneous illumination (> 10,000 counts per pixel)
    flatfield = tifffile.imread('flatfield.tif')
    # calculate pixel-wise correction factors
    flatfield = numpy.median(flatfield) / flatfield.astype('float32')
    # upload the new flatfield
    client.set_mask(flatfield, 'flatfield')

    # arm and disarm to test the system
    client.send_command('arm')
    client.send_command('disarm')
```