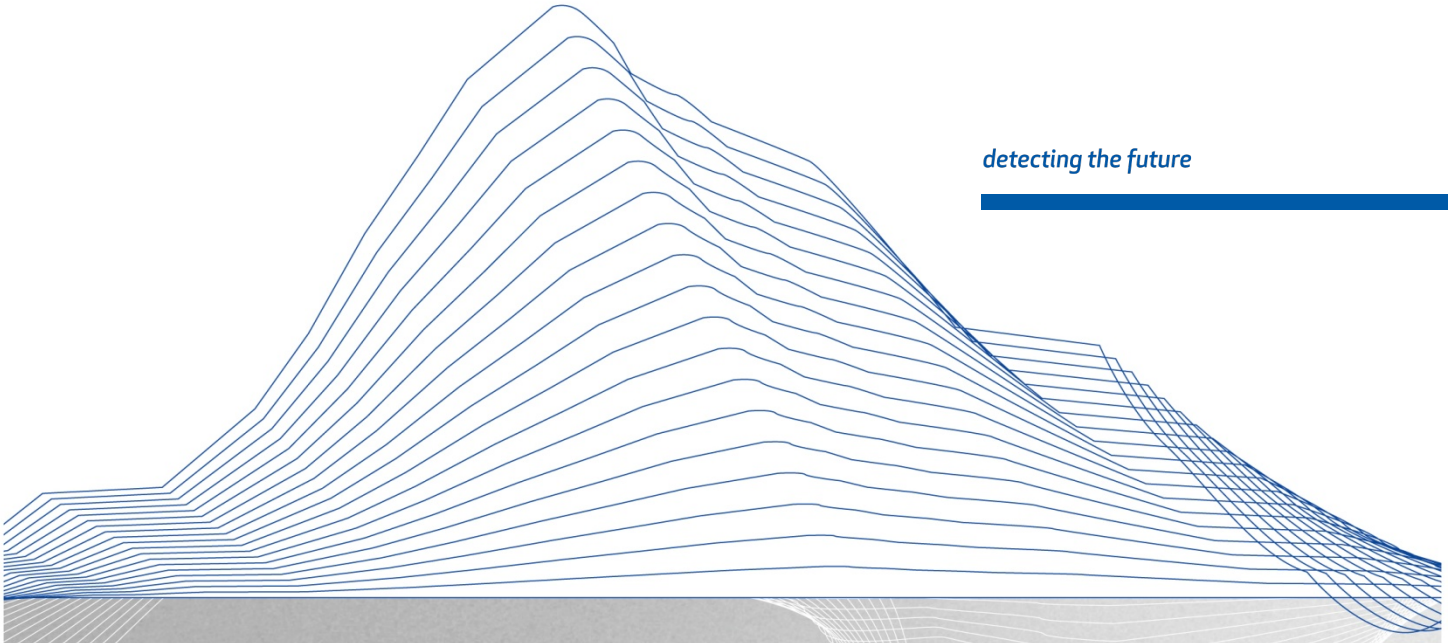
DECTRIS®

detecting the future



SIMPLON API Reference

SIMPLON API Version: 1.5

Document Version: 1 draft

DECTRIS Ltd.
5400 Baden
Switzerland
www.dectris.com

Table of Contents

1. Document History

1.1. Current Document

Version	Date	Status	Prepared	Checked	Released
1	31.08.2015	In work	AM/SB/MH/VP	MM	

1.2. Changes

Version	Date	Changes
1	31.08.2015	First version

2. Introduction – RESTlike API

The main objective of the SIMPLON API is to provide platform-independent access to the EIGER detector based on a well-established standardized protocol. Neither shall any special software be installed on the detector control unit nor shall access to the detector be restricted to a special programming language. The HTTP server running on the detector control unit provides a control interface to the detector that fulfills all of these requirements. In order, e.g., to define the state of the detector, trigger an exposure or request an image file, an HTTP request needs to be transmitted to the server and the requested data may be received within the HTTP response.

Let's for instance consider the common detector control parameter count_time, which defines the time the detector is exposed to X-rays. You may request the current state of this parameter by entering its URL "`http://<IP_of_DCU>/detector/api/1.0.0/config/count_time`" in your favorite web browser's address field, where `<IP_of_DCU>` needs to be replaced by the IP address of the detector computer. Analogously, the URL of the frame time is "`http://<IP_of_DCU>/detector/api/1.0.0/config/frame_time`".

In particular, we call this HTTP-based API RESTlike, because every detector resource is uniquely identified by its URL. A comprehensive definition of what is RESTful goes beyond the scope of this documentation. We confine ourselves to describing how to work with the SIMPLON API, which we indeed call RESTlike rather than RESTful, because it does not fulfill all requirements of a RESTful API.

Let's look at the sample request of count_time in more detail. We have learned that it is mapped to a unique URL, which is transferred via HTTP to the server. Besides the URL, the HTTP request contains extra data. The HTTP verb or method defines which kind of action has to be performed on the server. The SIMPLON API uses the verbs GET, PUT and DELETE. When entering `http://<IP_of_DCU>/detector/api/1.0.0/config/count_time` into your browser, your browser will send a GET request to the server, which is meant to return a representation of the resource but, by definition, must not change the resource itself. In our case, we receive the value of count_time. The value of count_time may be changed by a PUT request on the same URL. The data itself that is requested from the server or uploaded to the server, i.e. the value of count_time, is encoded in the message body of the HTTP request. The SIMPLON API chooses json as its main exchange format for message data. For instance, your browser may display the following string:

```
{"min": 0.00032999899121932685, "max": 1800, "value": 0.5, "value_type":
"float",     "access_mode": "rw", "unit": "s"}
```

This is a json dictionary that contains the keys "min", "max", "value", "value_type", "access_mode" and "unit". From the value of the keys "value" and "unit," we find the value of the count_time to be 0.5 seconds.

Note:

If you try this example and the browser prints instead "Parameter count_time does not exist", your detector may not have been initialized properly. You need to initialize the detector because the HTTP server needs to read a configuration from the detector in order to know to which kind of detector it is connected to. This is basically what is done when the detector is initialized. In order to initialize the detector, you must send a PUT request to "`http://<IP_of_DCU>/detector/api/1.0.0/command/initialize`".

A PUT request cannot be sent from a web browser. For testing, you may either use the plugin HttpRequester for Mozilla Firefox (`https://addons.mozilla.org/de/firefox/addon/httprequester/`) or the command line tool cURL. HttpRequester (version 2.0) opens a window with a field "URL", where you have to enter "`http://<IP_of_DCU>/detector/api/1.0.0/command/initialize`". Again, substitute `<IP_of_DCU>` by the IP of the EIGER detector control unit. Press PUT and wait for the reply, which may take some time. You could also use cURL, in a terminal window (Linux, Mac, Windows).

Now we want to set count_time to 1.0 seconds. We just upload the float value 1.0, because count_time has the unit second (Note: There is no way to change the unit of a parameter). In HttpRequester we set the URL to `http://<IP_of_DCU>/detector/api/1.0.0/config/count_time`. Below there is a field "Content to Send". The content type must be changed to "application/json" and we paste into the content field the string `{"value": 1.0}`, i.e., we upload a json dictionary with its only key "value" set to 1.0. After pressing PUT, in the return window on the right hand, we receive the list:

```
["count_time", "frame_count_time", "frame_period", "nframes_sum",
"frame_time"]
```

This is the list of parameters that have been implicitly changed. The SIMPLON API always keeps the configuration in a consistent state. So if the count time has been changed, the frame time (time between two successive images) needs to be changed as well, because obviously it must be greater than the count time. A GET request on `http://<IP_of_DCU>/detector/api/1.0.0/config/frame_time` tells us that frame_time is now slightly greater than count_time.

So far we have seen two examples of addressing a detector resource via a URL. Parameters are configured via GET/PUT requests on "`http://<IP_of_DCU>/detector/api/1.0.0/config/<some parameter>`", detector commands are transferred via PUT requests "`http://<IP_of_DCU>/detector/api/1.0.0/command/<some command>`". Finally the status of the detector may be queried via "`http://<IP_of_DCU>/detector/api/1.0.0/status/<status parameter>`". In addition to the detector interface (the URLs starting with /detector), there is a monitor interface ("`http://<IP_of_DCU>/monitor/api/1.0.0/...`") and a filewriter interface ("`http://<IP_of_DCU>/filewriter/api/1.0.0/...`"). The monitor interface lets you watch the currently running measurement; the filewriter interface lets you control how the data is stored in hdf5 files. In addition the hdf5 files may be received from /data/. There are two hdf5 files. The master file contains header data and links to the image data, which reside in series_1_data_000001.h5. Image series that contain more than one dataset may be distributed over multiple data files, each containing a block of (e.g., 1000) images.

3. Operating the EIGER Detector System

In order to acquire data with an EIGER detector system, these steps need to be performed:

- Initialize the detector
  - Mandatory only once, after power-up of either the detector or the detector control unit
  - Depending on System configuration, this may take up to 2 Minutes
  - Blocking operation, no other API operation may be performed until successful completion
  - [See API → Commands → Initialize]

- Configuration
  - Although this does not result in error if not performed, the user should set the required parameters for the experiment
  - If nothing is configured, defaults will be used
  - [See API → Configuration]

- Arm the detector (mandatory)
  - This uploads the configuration to the modules in the detector and prepares the system for data acquisition, but does not yet activate acquisition
  - Depending on System configuration, this may take up to 17 seconds. If the configuration is not changed, arm will be performed very quickly.
  - [See API → Commands → Arm]

- Trigger the detector
  - *This is mandatory in software trigger mode (ints) and has to be omitted in external enabled modes (exts, exte)*
  - This activates the actual data acquisition.
  - [See API → Commands → Trigger]

- Disarm the detector
  - Disables the trigger unit
  - Note: The last acquired image (or the image, if only one image was configured) is available only after this command
  - [See API → Commands → Disarm]

If a new acquisition is required, repeat these steps in the given order:

configure	(optional)
arm	(mandatory)
trigger	(mandatory for internal trigger, omit for external trigger/enable)
disarm	(mandatory)

- The status of the EIGER detector system may be optionally checked during acquisition, but the values are only updated if there is no actual acquisition active.

- For receiving the data, two options are available:
  - Download the HDF5 files

    See API: Filewriter for details.
  - Direct stream-like access on the images

    See API: Monitor interface for details

3.1. Interface: http/REST

The interface to the EIGER detector system is defined through its protocol. The protocol is based on the http/REST framework. This definition helps to cleanly isolate the detector system. Thus, no DECTRIS software is needed on the user control computer. The main idea behind the http/RESTful is the following:

- A configuration parameter, a status message, a detector command etc. correspond to a RESTful resource. Each resource has a URL.
- A user can perform a **get** or **put** operations on the URL. Special resources in the filewriter and monitor subsystem can also be **deleted**.
- A **get** request returns the current value of the configuration parameter.
- A **put** request sets the value of a configuration parameter.
- A **delete** request deletes the resource.
- Every configuration parameter has a corresponding data type (e.g. float or string). The data type in a **put** request must agree. The type of the value of a parameter can be requested with a **get** operation.
- For every configurable parameter with numeric data type, the minimum and maximum value can be requested if available. For enumerated data types, the available values can be requested.
- Any **put** request changing parameters may implicitly change dependent parameters. **Put** will always return a list of all parameters implicitly and explicitly changed.
- If an invalid resource is requested, an HTTP error code is returned.
- The serialization format of the configuration parameter values is, by default, in the JSON format. The syntax is described below. Larger datasets can be received in the hdf5 format.

3.1.1. URLs

Note:

Please replace <IP_of_DCU> with the IP-Address at which you can reach your detector control unit (e.g. 10.42.42.20 if you are using the preconfigured fixed address on interface em2).

To represent the resources of the SIMPLON API, URLs are used:

detector: Used to configure the detector and the readout system, to control data acquisition and request the detector status
`http://<IP_of_DCU>/detector/api/<version>`

monitor: Used to receive single frames at a low rate.
`http://<IP_of_DCU>/monitor/api/<version>`

filewriter: Configuration of the HDF5 filewriter.
`http://<IP_of_DCU>/filewriter/api/<version>`

stream: Configuration of the stream interface.
`http://<IP_of_DCU>/stream/api/<version>`

The URLs to configure the detector, to send a command to the detector and to request its status are:

`http://<IP_of_DCU>/detector/api/<version>/config`

`http://<IP_of_DCU>/detector/api/<version>/command`

`http://<IP_of_DCU>/detector/api/<version>/status`

A configuration parameter resource has the following URL:

`http://<IP_of_DCU>/detector/api/<version>/config/<parameter_name>`

For **get** requests, the image format can be chosen with the header item

`accept=<format>`

Possible formats are JSON and, for data arrays, hdf5 and tiff as well. (MIME types `application/json`, `application/hdf5` and `application/tiff`). The header item "`content-type`" is set respectively in all responses. Default is `application/json`. For small datasets `application/json` is recommended. For larger datasets, in particular 2d arrays (flatfield and pixel_mask), only `application/tiff` is well defined, whereas other formats may work as well.

4. SIMPLON API

Note:

Please replace <IP_of_DCU> with the IP-Address at which you can reach your detector control unit (e.g. 10.42.42.20 if you are using the preconfigured fixed address on interface em2).

4.1. Detector Subsystem

The detector subsystem has the base URL:

```
http://<IP_of_DCU>/detector/api/
```

It is used to configure the detector, to request its status and send control commands.

4.1.1. Detector Configuration Parameters

The user can set the parameters listed below. The base path to the resource is always:

```
<base_path> = http://<IP_of_DCU>/detector/api/<version>/config/
```

configuration parameter	resource	datatype	access	remarks
	<base_path>/	configuration	rw	If the html header item accept=hdf5 is used, an hdf5 file containing the configuration is returned. **Put**ting an hdf5 file will set all values in the file recursively. For accept=JSON, the configuration is returned in the JSON format. Remark: Images will NOT be included (flatfield, pixel_mask) due to their size. Note:The flatfield and pixel_mask are included in the *master.h5 file for each acquisition.
count_time	<base_path>/count_time	float	rw	Exposure time per image.
frame_time	<base_path>/frame_time	float	rw	Time interval between start of image acquisitions. This defines the speed of data collection and is the inverse of the frame rate, the frequency of image acquisition.
nimages	<base_path>/nimages	uint	rw	Number of acquired images per trigger event.
ntrigger	<base_path>/nimages	uint	rw	Number of triggers. See the manual for details.
photon_energy	<base_path>/photon_energy	float	rw	Energy of incident X-rays.
element	<base_path>/element	string	rw	Sets parameter 'photon_energy' to the K-alpha fluorescence radiation energy of an element.

threshold_energy	<base_path>/f_energy	float	rw	Threshold energy for X-ray counting. Photons with an energy below the threshold are not detected. See the detector manual for details.
flatfield	<base_path>/flatfield	float[][]	rw	Flatfield correction factors used for flatfield correction. Pixel data are multiplied with these factors for calculating flatfield corrected data.
flatfield_correction_applied	<base_path>/flatfield_correction_applied	bool	rw	Enables (True) or disables (False) flatfield correction. Should always be enabled.
pixel_mask	<base_path>/pixel_mask	uint[][]	rw	A bit mask that labels and classifies pixels which are either defective, inactive or not exhibit non-standard behavior. 0 gap (pixel with no sensor) 1 dead 2 under responding 3 over responding 4 noisy 5-31 -undefined-
number_of_excluded_pixels	<base_path>/number_of_excluded_pixels	uint	r	Total number of defective, disabled or inactive pixels.
countrate_correction_applied	<base_path>/countrate_correction_applied	bool	rw	Enables (True) or disables (False) countrate correction. Should always be enabled. See the detector manual for details.
auto_summation	<base_path>/auto_summation	bool	rw	Enables (True) or disables (False) auto-summation. Should always be enabled.
trigger_mode	<base_path>/trigger_mode	string	rw	Mode of triggering image acquisition. See the manual for details.
data_collection_date	<base_path>/ data_collection_date	string		Date and time of data collection. Specifically this is the time when the ARM command was issued.
description	<base_path>/description	string	r	Detector model and type.
detector_number	<base_path>/detector_number	string	r	Serial number of the detector system.
software_version	<base_path>/software_version	string	r	Software version used for data acquisition and correction.
sensor_material	<base_path>/sensor_material	string	r	Material used for direct detection of X-rays in the sensor.
sensor_thickness	<base_path>/sensor_thickness	float	r	Thickness of the sensor material.
x_pixel_size	<base_path>/x_pixel_size	float	r	Size of a single pixel along x-axis of the detector.
y_pixel_size	<base_path>/y_pixel_size	float	r	Size of a single pixel along y-axis of the detector.
x_pixels_in_detector	<base_path>/x_pixels_in_detector	uint	r	Number of pixels along x-axis of the detector.

y_pixels_in_detector	<base_path>/y_pixels_in_detector	uint	r	Number of pixels along y-axis of the detector.
beam_center_x	<base_path>/beam_center_x	float	rw	Beam position on detector.
beam_center_y	<base_path>/beam_center_y	float	rw	Beam position on detector.
detector_distance	<base_path>/detector_distance	float	rw	Sample to detector distance.
countrate_correction_count_cutoff	<base_path>/countrate_correction_count_cutoff	uint	r	Maximum number of possible counts after count rate correction.
bit_depth_readout	<base_path>/bit_depth_readout	int	r	Bit depth of the detector's image data.
detector_readout_time	<base_path>/detector_readout_time	float	r	Readout dead time between consecutive detector frames.
wavelength	<base_path>/wavelength	float	rw	Wavelength of incident X-rays. See the detector manual for details.

For a description of the parameters, see:

`https://www.dectris.com/nexus.html#main_head_navigation`

4.1.1.1. JSON Serialization

Meta information in the body of the request and in the reply from the HTTP server are serialized in the JSON format and described in the table below.

The returned JSON of a **get** request string contains a subset of the fields below. For hdf5 objects, the JSON metadata is stored with hdf5 attributes.

JSON key	JSON value	description
“value”	<parameter_value>	The value of the configuration parameter. Data type can be int, float, string or a list of int or float. 2 dimensional arrays are returned as darrays (see text below)
“value_type”	<string>	Returns the data type of a parameter. Data types are bool, float, int, string or a list of float or int.
“min”	<minimal parameter_value>	Returns the minimum of a parameter (for numerical datatypes).
“max”	<maximal parameter_value>	Returns the maximum of a parameter (for numerical datatypes).
“allowed_values”	<list of allowed values>	Returns the list of allowed values. An empty list indicates there are no restrictions.
“unit”	<string>	The unit of the parameter.
“access_mode”	<string>	String, describing read, and/or write access to resource. When not available, the access_mode is “rw”.

put requests send a body serialized in the JSON format. Arrays may be **put** as hdf5 objects. The HTTP header item “`content-type`” must be set appropriately. The JSON string may contain the following keywords:

JSON key	JSON value	description
“value”	<parameter_value>	The value of the configuration parameter. Data type can be int, float, string or a list of int or float. 2 dimensional arrays shall be uploaded as darrays (see text below)

Alternatively you can **put** larger datasets and images as hdf5 files.

The return body of a **put** request is:

JSON key	JSON value	description
None	<changed_parameters>	A list of all resources that are also affected by the **put** configuration parameter.

darray

2 dimensional arrays (pixel_mask, flatfield) are exchanged as darrays as defined below:

`{"__darray__": <version>, "type": <type>, "shape": [<width>,<height>], "filters":["base64"],"data": <base 64 encoded data> }` where

`<version>` is the darray version `([major, minor, patch])`, `<type>` is either `"<u4"` or `"<f4"` (little endian encoded 4 byte unsigned int or float), the filters is always `["base64"],` and `<base 64 encoded data>` contains the base 64 encoded data.

Remarks:

- It is highly recommended to change a configuration by **put**ting each parameter separately. Do NOT upload a full configuration in one step to the config url. This will only succeed if the configuration is valid and consistent. Uploading an inconsistent configuration (e.g., element is configured to “Cu” and energy should be set to 7400keV) leads to undefined behavior.
- The order in which parameters are **put** is important, as parameters can influence each other.
- A base configuration can be stored on the user computer by using the HTTP header item `accept=application/hdf5` on the base url. This configuration is consistent and valid and can be uploaded in one step.

4.1.1.2. Example – Setting photon_energy

As already mentioned, when setting a value, the DCU sends the names of the parameters, which were subsequently changed (to have a valid configuration) in the answer of the set request.
If the photon energy is, e.g., set to 8040 keV, then the answer tells which other values were also changed.

The following example uses some python libraries as a web client to send HTTP requests:

Code:

```
import json
# Imports "JSON" library
import requests
# Imports "requests" library
dict_data = {'value':8040.0}
# Prepare the dictionary (a "value" with the value 8040.0)
data_json = json.dumps(dict_data)
# Convert the dictionary to JSON
payload = {'json_playload': data_json}
# Set the request payload (the prepared JSON dictionary)
r =
requests.put('http://<IP_of_DCU>/detector/api/<version>/config/photon_energy'
, data=payload)
# Execute the request on the config value "photon_energy" (REPLACE
<IP_of_DCU> and <version> with the values of YOUR system)
print r.status_code
# Print the http status code (Note: Only http code 200 is OK, everything else
is an error)
print r.json()
# Print the returned JSON string. (Containing the names of the subsequently
changed values)
```

Output:

```
200
{["threshold_energy", "flatfield"]}
```

The returned HTTP code "`200`" indicates successful completion of the put request.

The JSON string "`{["threshold_energy", "flatfield"]}`" indicates that, resulting from the photon energy change, the threshold energy and the applied flatfield were also changed.

4.1.2. Detector Status Parameters

Status parameters are read only. The base path to the resource is:
`<base_path> = http://<IP_of_DCU>/detector/api/<version>/status/`

Status parameters are “measured values” which might change without interaction due to operation conditions.

4.1.2.1. JSON Serialization

get requests have no body. The returned JSON string contains the fields below.

JSON key	JSON value	description
“value”	<parameter_value>	The value of the configuration parameter. Data type can be single type or list of int, float or string.
“value_type”	<string>	Returns the data type of a parameter.
“unit”	<string>	Returns the unit of the parameter.
“time”	<date>	Timestamp for when the value was updated.
“state”	<state>	invalid, normal, critical, disabled
“critical_limits”	<list containing minimal and maximal parameter_value>	Returns the minimum and maximum error threshold for a parameter if it is a numerical value type.
“critical_values”	<list of critical values>	Returns the list of values treated as error conditions. An empty list indicates there are no states causing an error condition.

4.1.2.2. Status Information

status parameter	resource	data type	access	remarks
state	<base_path>/state	string	r	Possible states: na (not available), ready, initialize, configure, acquire, test, error Note: State is “na”, when the DCU is booted or the acquisition service was restarted.
error	<base_path>/error	list of strings	r	Returns list of status parameters causing error condition.
time	<base_path>/time	date	r	Returns actual system time.
board_000/th0_temp	<base_path>/board_000/th0_temp	float	r	Temperature reported by temperature sensor.
board_000/th0_humidity	<base_path>/ board_000/th0_humidity	float	r	Relative humidity reported by humidity sensor.

4.1.3. Detector Command Parameters

Command parameters are write only. The base path to the resource is:

```
<base_path> = http://<IP_of_DCU>/detector/api/<version>/command/
```

command parameter	resource	return value	access	remarks
initialize	<base_path>/initialize	-	w	Initializes the detector.
arm	<base_path>/arm	sequence_id: int	w	Loads configuration to the detector and arms the trigger unit.
disarm	<base_path>/disarm	sequence_id: int	w	Writes all data to file and disarms the trigger unit.
trigger	<base_path>/trigger	-	w	Starts data taking with the programmed trigger sequence.
cancel	<base_path>/cancel	sequence_id: int	w	Stops taking data, but only after the next image is finished.
abort	<base_path>/abort	sequence_id: int	w	Aborts all operations and resets the system immediately.
status_update	<base_path>/status_update		w	Update detector status.

If an error occurs, the HTTP error code “400” is returned. In this case, please download the API log, which can be accessed by the web interface of the DCU and contact support@dectris.com.

The trigger command can also accept an argument in the put request – the count_time - if used in trigger_mode inte (internal enable).

4.2. Monitor Interface

The monitor interface is used to inspect single frames. This is a low performance and low bandwidth interface, and thus should only be used at low frame rates. For high frame rates (>10Hz), either use of the filewriter or the streaming interface is recommended to. In order to use the monitor interface, it must first be configured. The base URL for the Monitor API is:

`http://<IP_of_DCU>/monitor/api/<version>`

4.2.1. Monitor Configuration

The configuration is applied at the URL:

`<base_path> = http://<IP_of_DCU>/monitor/api/<version>/config`

with the following commands:

configuration parameter	resource	data type	access	remarks
mode	<base_path>/mode	bool	rw	Operation mode of the monitor, which can be 'enabled' or 'disabled'. When enabled, a number of 'buffer_size' images are stored in the monitor buffer. The monitor keeps old and drops new images if the buffer is running full.
buffer_size	<base_path>/buffer_size	int	rw	Number of images that can be buffered by the monitor interface.

4.2.2. Data Access

During data taking, the frames can be accessed with a **get** operation at the url:
`<base_path> = http://<IP_of_DCU>/monitor/api/<version>/images`

data	resource	data type	access	remarks
	<base_path>	Json List	r	List of available images [[seriesId, [imfId, imgId, ...]], ...].
monitor	<base_path>/monitor	tif	r	Gets latest image and removes it from the buffer. Default waits for 500 ms for image. Timeout via ?timeout=[ms] adjustable. Returns 408 if no image available.
next	<base_path>/next	tif	r	Gets next image and removes it from the buffer. Default waits for 500 ms for image. Timeout via ?timeout=[ms] adjustable. Returns 408 if no image available.
<series>/<id>	<base_path>/<series>/<id>	tif	r	Gets corresponding image, if not available, returns HTTP 404 Not Found.

4.2.3. Monitor Status Parameters

The status of the monitor can be requested at the address:

`<base_path> = http://<IP_of_DCU>/monitor/api/<version>/status`

status parameter	resource	data type	access	remarks
state	<base_path>/state	string	r	State can be 'normal' or 'overflow' if images have been dropped.
error	<base_path>/error	string	r	Returns list of status parameters causing error condition.
buffer_fill_level	<base_path>/buffer_fill_level	int	r	Returns a tuple with current number of images and maximum number of images in buffer.
dropped	<base_path>/dropped	Int	r	Number of images which were dropped as not requested.
next_image_number	<base_path>/next_image_number	int	r	seriesId, imageId of the last image requested via images/next.
monitor_image_number	<base_path>/monitor_image_number	int	r	seriesId, imageId of the last image requested via images/monitor.

4.2.4. Monitor Control Parameters

To clear the buffer of images, the following command can be executed at the address

`<base_path> = http://<IP_of_DCU>/monitor/api/<version>/command`

control parameter	resource	data type	access	remarks
clear	<base_path>/clear	-	-	Drops all buffered images and resets status/dropped to zero.
initialize	<base_path>/initialize	-	-	Resets the monitor to its original state.

4.3. File Writer

The data itself, the frames, are by default written to HDF5 files, where the meta data is stored in the NeXus compliant metadata standard. These files can be accessed through the SIMPLON API. Frames can also be obtained through the SIMPLON API with the monitor interface, but at a low performance (see point 4.2). This is usually used to monitor the data.

In a future version, the data stream can be also accessed directly through its own API.

The filewriter subsystem writes the frames and the metadata in the NeXus format to an HDF5 file. The base URL for the filewriter is:

`http://<IP_of_DCU>/filewriter/api/<version>/`

To configure the filewriter, this url is used:

`<base_path> = http://<IP_of_DCU>/filewriter/api/<version>/config`

configuration parameter	resource	data type	access	remarks
mode	<base_path>/mode	string		Operation mode of the filewriter, which can be 'enabled' or 'disabled'. When disabling the filewriter, data loss may occur if data is not retrieved via the stream or the monitor.
transfer_mode	<base_path>/transfer_mode	string	rw	Transfer mode for files written by the filewriter. Currently, only http is supported.
nimages_per_file	<base_path>/nimages_per_file	int	rw	Maximum number of images stored in each <name_pattern>_data_<file_nr>.h5 file in the HDF5 file structure created by the filewriter.
image_nr_start	<base_path>/image_nr_start	int	rw	Sets the 'image_nr_low' metadata parameter in the first HDF5 data file <name_pattern>_data_000001.h5. This parameter is useful when a data set is collected in more than one HDF5 file structures. If you collect image number m to n in the first file strutcture, you can set image_nr_start to n+1 in the subsequent file structure.
name_pattern	<base_path>/name_pattern	string	rw	The basename of the file. The pattern $id will include the sequence number in the file name. 'series_$id' is the default name pattern, resulting in the following names of the HDF5 file structure created by the filewriter: "series_<sequence_nr>_master.h5, series_<sequence_nr>_data_<filenr>.h5 WARNING: The filewriter will overwrite existing files with identical names of the files to be written

compression_enabled	<base_path>/compression_enabled	bool	rw	Enables (True) or disables (False) LZ4 compression of detector data written to HDF5 files. Compression is required for full detector performance, disabling compression may lead to data loss at high frame rates.

The files are created locally on the detector server and have to be transferred to the user computer. The master file is accessible at the URL:

`http://<IP_of_DCU>/data/<name_pattern>_master.h5`

and the data files at:

`http://<IP_of_DCU>/data/<name_pattern>_data_<filenr>.h5`

A **get** request to the URL:

`http://<IP_of_DCU>/filewriter/api/<version>/files/`

returns a list with all available files.

The filewriter is automatically started when data taking is started. The status of the filewriter can be accessed at:

`<base_path> = http://<IP_of_DCU>/filewriter/api/<version>/status`

The following filewriter status variables are accessible:

status parameter	resource	data type	access	remarks
state	<base_path>/state	string	r	Possible states: disabled, ready, acquire, error
error	<base_path>/error	string	r	List of status parameters causing error state.
time	<base_path>/time	date	r	Current system time.
buffer_free	<base_path>/buffer_free	int	r	The remaining buffer space in KB.

4.3.1. Filewriter Control Parameters

To clear the buffer of images, the following command can be executed at the address

`<base_path> = http://<IP_of_DCU>/filewriter/api/<version>/command`

control parameter	resource	data type	access	remarks
clear	<base_path>/clear	-	-	Drops all data (image data and directories) on the DCU.
initialize	<base_path>/initialize	-	-	Resets the monitor to its original state.

4.4. Stream

The RESTful interface lets you configure and read out the status of the stream.

The base URL for the stream is:

`http://<IP_of_DCU>/stream/api/<version>/`

To configure the stream, this url is used:

`<base_path> = http://<IP_of_DCU>/stream/api/<version>/config`

configuration parameter	resource	data type	access	remarks
mode	<base_path>/mode	string	rw	Operation mode of the stream, which can be 'enabled' or 'disabled'. When disabling the stream, data loss may occur if data is not retrieved via the filewriter or monitor.
header_detail	<base_path>/header_detail	string	rw	Detail of header data to be sent: Either "all" (all header data), "basic" (no flatfield nor pixel mask) or "none" (no header data).
header_appendix	<base_path>/header_appendix	string	rw	Data that is appended to the header data as zeromq submessage
image_appendix	<base_path>/image_appendix	string	rw	Data that is appended to the image data as zeromq submessage

Please see section 4.1.1 for details about the GET and PUT requests.

4.4.1. Data Interface

Image and header data are transferred via zeromq sockets. The port is 9999, the scheme is Push/Pull, i.e. the server opens a zeromq push socket, whereas the client needs to open a zeromq pull socket.

Protocol	Zeromq
Port	9999
Scheme	Push/Pull
Direction Connection	Receiver connects to detector (this enables automatic load balancing if more than 1 client is required to receive/process the data)

There are 3 types of messages, which are defined below in more detail: ***Global Header Data***, ***Image Data*** and ***End of Series*****.** After passing the "arm" command to the detector one message containing *Global Header Data* is sent over the zeromq socket. After passing "trigger" one messages per image containing *Image Data* is sent. After passing "disarm", "cancel" or "abort", one message containing *End of Series* is sent.

4.4.1.1. Global Header Data

Zeromq multipart message consisting of the following parts:

- ***Part 1*** : Json Dictionary, reading {"htype":"dheader-1.0", "series": <id>, "header_detail": "all" | "basic" | "none"}. <id> denotes the series id of the present image series.
- ***Part 2*** *(only if header_detail is "all" or "basic"):* Detector configuration as json dictionary, reading { <config parameter>: <value> }. The keys are the configuration parameters as defined in the detector API. The values are the current configuration values. There are maximum 1 dim arrays, which are stored as json array. Flatfield and Pixelmask and countrate_correction_table are not part of the dictionary.
- ***Part 3*** *(only if header_detail is "all"):* Flatfield Header. Json Dictionary reading {"htype": "dflatfield-1.0", "shape": [x,y], "type": <data type> }. <data type> is always "float32" (32 bit float) for a flatfield.
- ***Part 4*** *(only if header_detail is "all"):* Flatfield data blob.
- ***Part 5*** *(only if header_detail is "all"):* Pixel Mask Header. Json Dictionary reading {"htype": "dpixelmask-1.0", "shape": [x,y], "type": <data type> }. <data type> is always "uint32" (32 bit unsigned integer) for a pixel mask.
- ***Part 6*** *(only if header_detail is "all"):* Pixel Mask data blob.
- ***Part 7*** *(only if header_detail is "all"):* Countrate Table Header. Json Dictionary reading {"htype": "dcountrate_table-1.0", "shape": [x,y], "type": <data type> }. <data type> is always "float32" (32 bit float) .
- ***Part 8*** *(only if header_detail is "all"):* Countrate Table data blob.
- ***Appendix*** *(only if header_appendix contains non*-empty string): Content of API parameter header_appendix

Example:

{"htype":"dheader-1.0", "series": 1, "header_detail": "all"}
{"auto_summation": true, "photon_energy": 8000, ...}
{"htype": "dflatfield-1.0", "shape": [1030,1065], "type": "float32" }
DATA BLOB (Flatfield)
{"htype": "dpixelmask-1.0", "shape": [1030,1065], "type": "uint32" }
DATA BLOB (Pixel Mask)
{"htype": "dcountrate_table-1.0", "shape": [2,1000], "type": "float32" }
DATA BLOB (countratecorrection table)

4.4.1.2. Image Data

Zeromq multipart message consisting of the following parts:

- ***Part 1*** : Json Dictionary, reading {"htype":"dimage-1.0",”series”: <series id>, "frame": <frame id>, "hash": <md5>}, <series id> is the number identifying the series, <frame id> is the frame id, i.e. the image number. <md5> is the md5 hash of the next message part.

- ***Part 2:*** {"htype":"dimage_d-1.0", "shape":[x,y,(z)], "type": <data type>, "encoding": <encoding>, "size": <size of data blob> }.
    - <data type>: "uint16" or "uint32".
    - <encoding>: String of the form "[n<BIT>][-lz4[-lz4]][<|>]" that encodes bit shufflling filter with number of bits, compression algorithm and endianness. *n<BIT>* stands for bit shuffling with *<BIT>* bits, *lz4* for lz4 compression and < (>) for little (big) endian. E.g. "n8-lz4<" stands for 8bit bitshuffling, lz4 compression and little endian. lz4 data is written as defined at https://code.google.com/p/lz4/ without any additional data like block size etc.
    - <size of data blob>: Size in bytes of the following data blob
- ***Part 3:*** Data Blob
- ***Part 4:*** {"htype":"dconfig-1.0", "start_time": <start_time>, "stop_time", <stop_time>, "count_time": <count_time>}. Begin, end and duration of the exposure of the current image in nano seconds. The start time of first image of the series is by definition zero.
- ***Appendix*** *(only if image_appendix contains non*-empty string): Content of API parameter image_appendix

Example:

{"htype":"dimage-1.0", "frame": 324, "hash": "fc67f000d08fe6b380ea9434b8362d22"}
{"htype":"dimage_d-1.0", "shape":[1030,1065], "type": "uint32", "encoding": "lz4<", "size": 47398247}
DATA BLOB (Image Data)
{"htype":"dconfig-1.0", "start_time": 834759834260, "stop_time", 834760834280, "real_time": 1000000}

4.4.1.3. End of Series

Zeromq message consisting of one part containing the json string:

{"htype": "dseries_end-1.0", "series": <id>}

The status of the stream can be accessed at:

`<base_path> = http://<IP_of_DCU>/stram/api/<version>/status`

The following stream status variables are accessible:

status parameter	resource	data type	access	remarks
state	<base_path>/state	string	r	"disabled", "ready", "acquire" or "error". After the detector has been armed the state becomes acquire, after disarm, abort or cancel the

				state becomes ready. There are currently no error conditions.
error	<base_path>/error	string	r	Returns list of status parameters causing error condition (currently only "state").
dropped	<base_path>/dropped	int	r	Number of images that got dropped as not requested. After "arm" this number is reset to zero.

4.5. System

The service providing the SIMPLON API may be restarted by posing a PUT request at the URL

`http://<IP_of_DCU>/system/api/<version>/command/restart`