SERVAL User manual

AMSTERDAM
SCIENTIFIC
INSTRUMENTS

SERVAL: TimePix3 Server software manual

In case of any technical question please contact ASI

Telephone: +31 207235200

Email: support@amscins.com

Postal address:
Amsterdam Scientific Instruments B.V.
Science Park 106
1098 XG Amsterdam
The Netherlands

Date of release 31 October, 2022
Version 3.1

Contents

SERVAL: TimePix3 Server software manual ... 2
Contents ... 3
1. Introduction ... 5
1.1 Usage of SERVAL ... 5
2. Installing SERVAL ... 7
2.1 Requirements ... 7
2.1.1 Operating System requirements ... 7
2.1.2 Software environment requirements ... 7
2.2 Connection to Camera / Detector ... 8
2.3 Installation ... 8
3. Using SERVAL ... 8
3.1 Running SERVAL ... 8
3.2 Connecting the client to SERVAL ... 8
3.3 Example: load (SoPhy) calibration files and set config ... 9
3.4 Example: start measurement ... 10
3.5 Example: collect previews and save measurement data ... 11
4. API description ... 14
4.1 Introduction ... 14
4.1.1 Response ... 15
4.2 API structure ... 16
4.2.1 Namespaces ... 16
4.2.2 Commands, readable and writable parameters ... 16
4.3 Welcome and Root: / and /* requests ... 16
4.4 /server requests ... 16
4.4.1 Destination JSON specification ... 17
4.5 /detector requests ... 25
4.5.1 GET request retrieving settings ... 26
4.5.2 GET action layout requests: rotate and flip image ... 27
4.5.3 PUT request to set detector settings by uploading data ... 28
4.5.4 Detector Info JSON structure ... 29
4.5.5 Detector Health JSON structure ... 31
4.5.6 Detector Layout JSON structure ... 32
4.5.7 Detector Config JSON structure ... 34
4.5.8 DACs JSON structure ... 37

4.6 /measurement requests 38
4.6.1 GET request to get measurement information 38
4.6.2 PUT request to set measurement settings by uploading data 38
4.6.3 Measurement config JSON structure 39
4.7 /config requests 41
4.8 /dashboard requests 41
4.8.1 dashboard JSON structure 43
4.8.2 DiskSpace JSON structure 45
4.8.3 Notification JSON structures 46
5. Example software 47
6. Appendix: file formats 48
6.1 TPX3 raw file format 48
6.2 Image File 50
7. Versions 51
About Amsterdam Scientific Instruments 52

1. Introduction

SERVAL is the software developed by Amsterdam Scientific Instruments (ASI) to integrate control and perform data acquisition from ASI's Timepix3 based cameras like TPX3Cam, LynX (X-ray) and Cheetah (Electron Microscopy) within your experimental environment. The Timepix3 camera is controlled (locally or remotely) by TCP/IP commands and therefore independent of the program language and operation system used. Measurement data is saved on the local acquisition workstation, while preview images are optionally sent over the network to allow for real-time data review. In this manual the software is described in detail, including several practical examples to get started. Also, the complete Application Programing Interface (API) of SERVAL is made available for developers. For technical details and functionality about the Timepix3 chip and the Timepix3 cameras we refer to the hardware manual.

1.1 Usage of SERVAL

The SERVAL software allows you to control all functionalities of the camera via an HTTP interface to the Timepix3 camera. This means that you can let the camera take measurements, control camera settings or request results such as images. Any programming language that can work with HTTP connections can be used. Examples are: Python, C/C++ and LabView.
The SERVAL software can replace SoPhy with the exception of detector calibration and generation of the factory settings files. SoPhy and SERVAL should not run simultaneously.
The intended use of SERVAL is to operate the camera within a more complex experimental environment with for example mechanical stages, electronic triggers, lasers, environmental sensors, microscopes, beam lines, etc. Additionally, it can be used for an application specific user interface or data viewer.

There are two approaches:
Local client
The client and server are on the same computer. This is the compact method. Figure 1 displays a setup in which you install the SERVAL software on a control PC. On this PC you can run your own software written in languages such as Python, C/C++ or LabView to control your experiment. The ASI camera is connected via Ethernet. In this method you control all your devices from one PC.

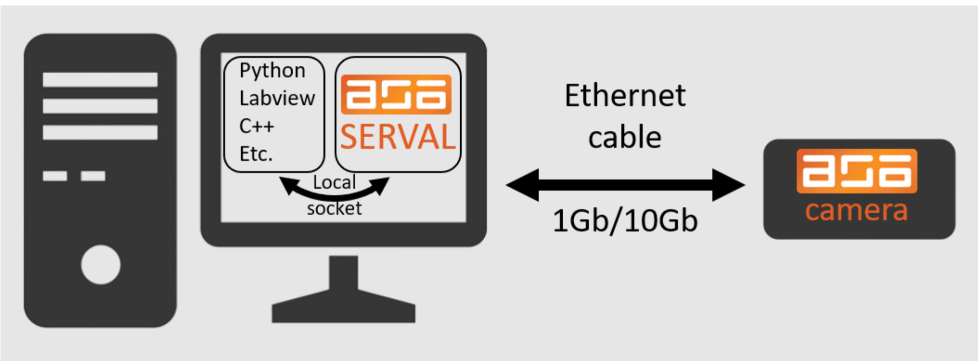


Figure 1, Using SERVAL as a local server, on your own acquisition PC. Both your software and SERVAL are running on your PC using a local socket to communicate (left) and SERVAL controls the camera (right).

Remote client
The client is on a different computer, connected over Ethernet. Here, a separate PC is the client and the control PC is the server. Via the Ethernet connection commands and data can be sent to control the camera. Because SERVAL runs on a dedicated system you can achieve maximum performance. For

maximum performance the hardware of the server is important, and you should consider acquiring the server from ASI with your camera.

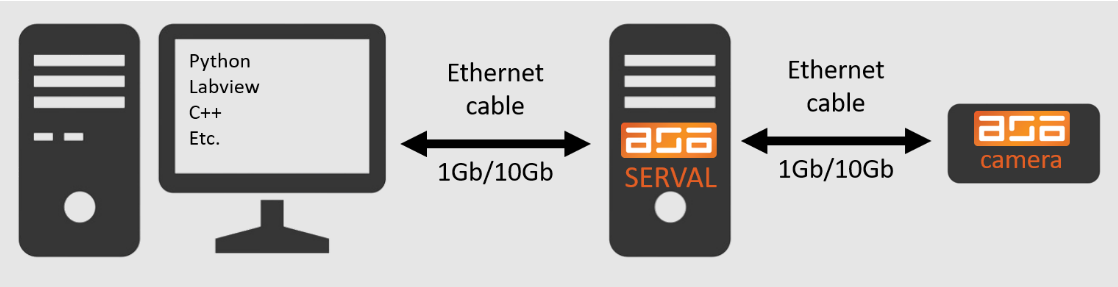


Figure 2, Using SERVAL from a remote server. SERVAL is on a physically separate server to get maximum performance. Here the client PC (left) instructs the SERVAL server (middle) to control the camera (right).

The setup in which you want to integrate the camera will typically consist of more components than described in the simplified figures 1 and 2. Figure 3 gives a more general overview of the components that are relevant for software integration.

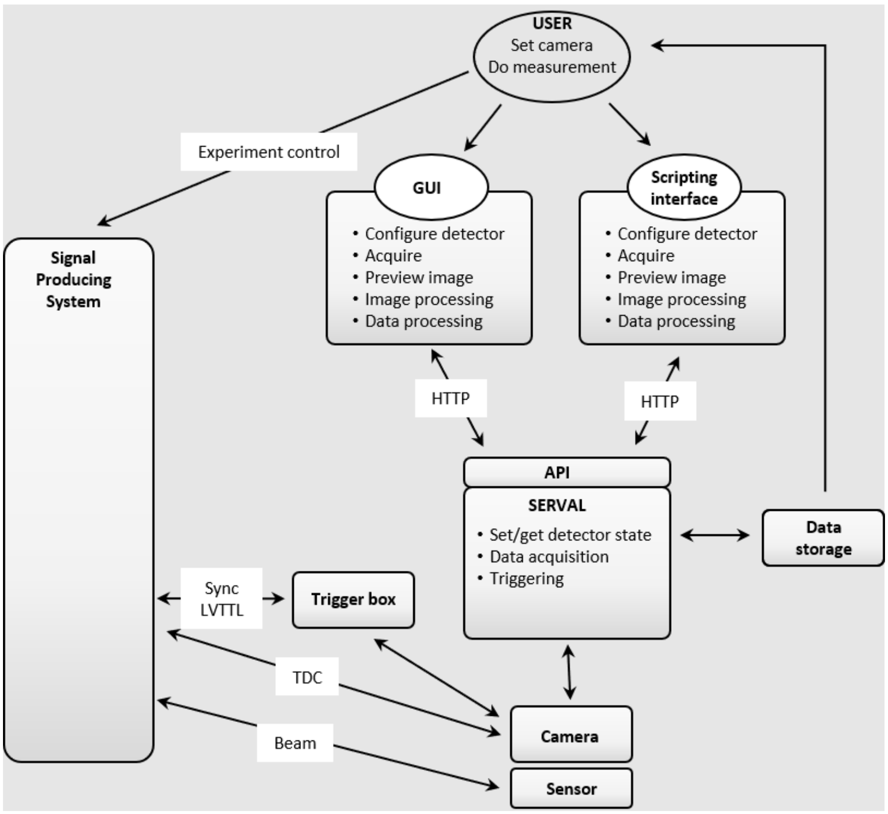


Figure 3. Block diagram for software integration with SERVAL.

2. Installing SERVAL

2.1 Requirements

We recommend using SERVAL on ASI delivered acquisition computers.

2.1.1 Operating System requirements

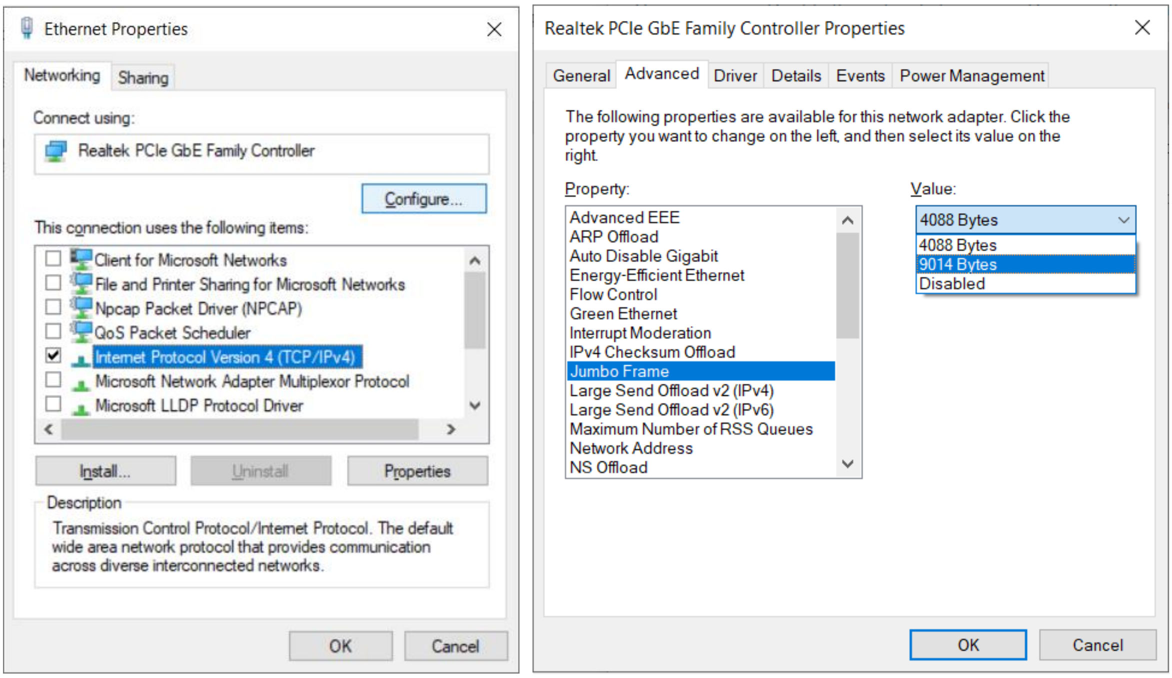
On ASI workstation computers Ubuntu Linux 20.04 LTS is installed.

2.1.2 Software environment requirements

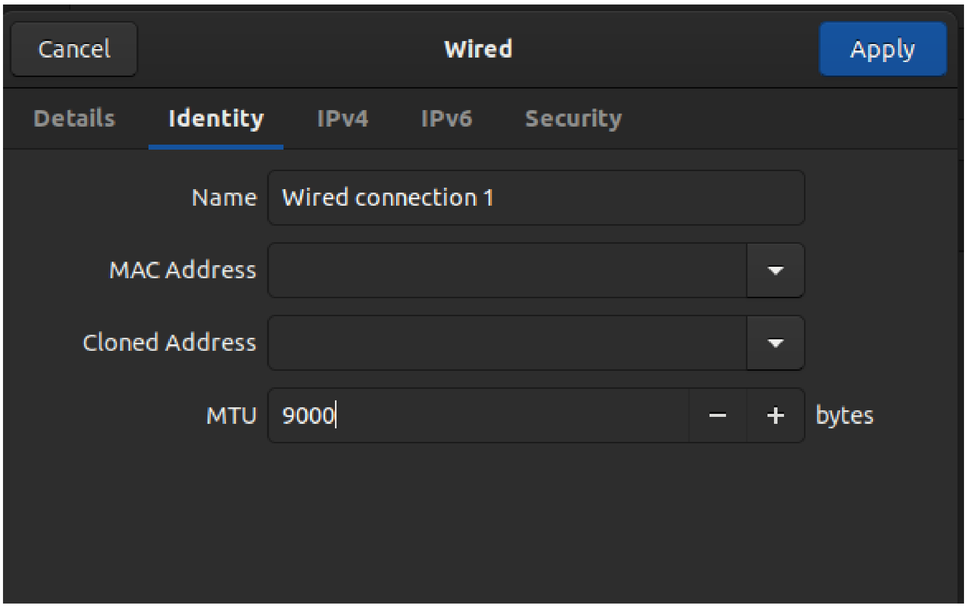
On ASI workstation computers Java 11 is installed.

If the PC is not from ASI, the user must set the network firewall correctly. Otherwise, it may have problems setting up a connection between SERVAL and the camera.

Make sure that:
1. firewalls allow outgoing and incoming TCP traffic on port 8080 (the default port Serval runs on),
2. firewalls allow outgoing and incoming TCP traffic on port 50000 (the port SERVAL uses to connect to the detector),
3. firewalls allow incoming UDP communication on ports 8192, 8193, 8194 and 8195,
4. no other programs use these ports (close SoPhy if it is running),
5. on Windows: the Jumbo Frame value on all relevant interfaces is set to 9014 Bytes.



6. On Ubuntu: the MTU value on all relevant interfaces is set to 9000.



7. The IPv4 address of the network interface that is connected with the detector is set to 192.168.100.1 (for 10Gb/s connections) or 192.168.1.1 (for 1Gb/s connections).

2.2 Connection to Camera / Detector

Please check your hardware manual to read how to connect to the detector and test the connection.

2.3 Installation

SERVAL automatically checks which cameras are available. Each time that Serval connects to the detector, it is best to load the factory setting files. See §4.7 on how to do this.

3. Using SERVAL

3.1 Running SERVAL

SERVAL can be **started** by executing the following command (the file name contains a version number that may differ):

```
$ java -jar serval-3.1.0.jar
```

To see a list of additional options that you can run Serval with, use

```
$ java -jar serval-3.1.0.jar --help
```

3.2 Connecting the client to SERVAL

SERVAL is controlled by sending commands over an IP socket. It will respond with a status code and text, data or error message.

When the client is on a remote computer the availability of the connection with the server can be tested by sending a command to or requesting data from the server.

For example, with the Python requests library (https://requests.readthedocs.io):

```
import requests
# ACTION: replace the IP-address with the address that serval runs on
serverurl = 'http://192.168.x.x:8080'
response = requests.get(serverurl)
print(response.text)
```

For a good connection, the response status code will be 200, and you will see the welcome message of Serval. More details on the response status codes are in table 4.1, §4.2.
The command `/dashboard` gives a summary of relevant information on the current state of the server, detector and measurement.

You can use the following Python snippet to request the dashboard from the server. The libraries used are the ‘requests’ and the ‘json’ modules. For a full example, see tpx3_example_01.py

```
import json

def get_dashboard(serverurl):
    response = requests.get(url = serverurl + '/dashboard')
    return json.loads(response.text)

get_dashboard(serverurl)
```

This snippet returns a JSON object which among others contains the software version, measurement information and detector type. The advantage of the JSON object is that parameters can be addressed by their names.

3.3 Example: load (SoPhy) calibration files and set config

For this we need the `/config/load` and `/detector/config` commands. We use `GET /config/load` to load the DACs and the BPC files that are stored on the server. It is possible to upload a remote BPC file and remote DACs through JSON, but this is out of scope for this example. The detector configuration is set through a `PUT /detector/config` request.

In the example, the file names of the BPC file and the DACs file are: tpx3-demo.bpc and tpx3-demo.dacs. These file paths should be replaced with the files that were delivered with your system.

All examples were written for Python 3.8.10.
In Python we can do this as follows, see tpx3_example_02.py:

```
import os

def init_cam(serverurl, detector_config, bpc_file, dacs_file):
    # load a binary pixel configuration exported by SoPhy, the file should exist on 
the server
    response = requests.get(url=serverurl + '/config/load?format=pixelconfig&file=' 
+ bpc_file)
    data = response.text
    print('Response of loading binary pixel configuration file: ' + data)

    #  .... and the corresponding DACs file
    response = requests.get(url=serverurl + '/config/load?format=dacs&file=' + 
dacs_file)
    data = response.text
    print('Response of loading DACs file: ' + data)
```

```
    # set the detector configuration
    response = requests.put(url=serverurl + '/detector/config',
data=json.dumps(detector_config))
    data = response.text
    print('Response of loading Detector Configuration: ' + data)

# Main part
# ACTION: replace files with the files delivered with your system:
bpcFile = os.path.join(os.getcwd(), 'tpx3-demo.bpc')
dacsFile = os.path.join(os.getcwd(), 'tpx3-demo.dacs')

# Example of getting the detector configuration from the server in JSON format
response = requests.get(url=serverurl + '/detector/config')
data = response.text
print('Response of getting the Detector Configuration from SERVAL: ' + data)

# Converting detector configuration data from JSON to Python dictionary and
modifying values
detectorConfig = json.loads(data)
detectorConfig["BiasVoltage"] = 100
detectorConfig["BiasEnabled"] = True

# Detector initialization with modified detector configuration values
init_cam(serverurl, detectorConfig, bpcFile, dacsFile)
```

Note that GET commands can be sent with or without argument and a PUT command is sent with data, in this case the JSON object in a text string format. Check the output of the execution whether all three uploads have succeeded. If a step fails, a corresponding error message will be shown, which should help you solve the problem.

3.4 Example: start measurement

The example below demonstrates how to set up the configuration (`/detector/config`), set up a destination that writes the files to disk (`/server/destination`), and start (`/measurement/start`) a measurement. For a full example, see tpx3_example_03.py.

```
import pathlib


def init_acquisition(serverurl, detector_config, ntriggers=10, trigger_period=0.1,
exposure_time=0.05):
    # Sets the number of triggers.
    detector_config["nTriggers"] = ntriggers

    # Set the trigger mode to be automatically triggered.
    detector_config["TriggerMode"] = "AUTOTRIGSTART_TIMERSTOP"

    # Sets the trigger period (time between triggers) in seconds.
    detector_config["TriggerPeriod"] = trigger_period

    # Sets the exposure time (time the shutter remains open) in seconds.
    detector_config["ExposureTime"] = exposure_time

    # Upload the Detector Configuration defined above
    response = requests.put(url=serverurl + '/detector/config',
data=json.dumps(detector_config))
    data = response.text
    print('Response of updating Detector Configuration: ' + data)
```

```
def init_destination(serverurl):
    # Example of destination configuration (Python dictionary) for the data output
    destination = {
        "Image": [{
            # URI to a folder where to place the image files.
            "Base": pathlib.Path(os.path.join(os.getcwd(), 'data')).as_uri(),
            # How to name the files for the various frames.
            "FilePattern": "f%Hms_",
            # What (image) format to write the files in.
            "Format": "tiff",
            # What data to build a frame from
            "Mode": "count"
        }]
    }

    # Upload the Destination defined above
    response = requests.put(url=serverurl + '/server/destination',
data=json.dumps(destination))
    data = response.text
    print('Response of updating Destination: ' + data)


def acquisition_test(serverurl):
    # Starting acquisition process
    response = requests.get(url=serverurl + '/measurement/start')
    data = response.text
    print('Response of acquisition start: ' + data)

    # Example of measurement interruption
    taking_data = True
    while taking_data:
        dashboard = json.loads(requests.get(url=serverurl + '/dashboard').text)
        # Stop measurement once Serval has collected all frames
        if dashboard["Measurement"]["Status"] == "DA_IDLE":
            taking_data = False


init_acquisition(serverurl, detectorConfig)
init_destination(serverurl)
acquisition_test(serverurl)
```

With this example, we set SERVAL up to acquire 10 frames (nTriggers=10), with a frame rate of 10 (TriggerPeriod=0.1), each frame having an exposure time of 0.05 seconds (ExposureTime=0.05). These frames will be stored in the “data” folder under the current working directory (as specified in the destination object).

3.5 Example: collect previews and save measurement data

In the above example, we saw how to write files directly to disk. If we want to also obtain a subset of the images on a Preview stream over TCP, we should edit the data destinations in `/server/destination`. With the destination we configure below, we can obtain the real time preview with the HTTP GET call to `/measurement/image`. See also tpx3_example_04.py. Here we set up an example of a destination JSON object for obtaining previews and to save the raw data (raw binary list of event data (.tpx)).

```
# Example of destination configuration (Python dictionary) for the data output
destination = {
    "Raw": [{
        # URI to a (server) folder where to place the raw files.
        "Base": pathlib.Path(os.path.join(os.getcwd(), 'data')).as_uri(),
        # How to name the files for the various frames.
        "FilePattern": "raw%Hms_",
```

```
    }],
    "Preview": {
        # In what interval and sampling mode to grab the previews.
        "SamplingMode": "skipOnFrame",
        "Period": 0.2,

        "ImageChannels": [{
            # Where to place the preview files.
            # We selected the HTTP scheme.
            "Base": "http://localhost",
            # What (image) format to provide the files in.
            "Format": "tiff",
            # What data to build a frame from (tot, toa, tof, count, count_fb)
            "Mode": "tot"
        }]
    }
}

# Main part
response = requests.put(url = serverurl + '/server/destination', data =
json.dumps(destination))
print('Response of uploading Destination: ' + response.text)
response = requests.get(url = serverurl + '/server/destination')
print('Uploaded destination is: ' + response.text)
```

Here the preview stream is made available through HTTP on the server and the raw data stream is saved in the data folder in the current working directory. For the preview images, we selected the TIFF image format. We also selected the mode `tot` (Time over Threshold). Other preview modes are `count`, `count_fb` (counting hits in frame-based mode), `toa` (Time of Arrival) and `tof` (Time of Flight). Remember that previously we set a TriggerPeriod of 0.1, which means the Image stream (which we do not save here) will produce 10 frames per second. The preview stream specifies a sample "Period" of 0.2, which means that it will sample 5 frames per second from the Image stream. We will therefore only see a subset of the images in the preview stream. This functionality makes it possible to, for example, save 500 frames per second to disk, but only show 30 frames per second to the user.

Now that we have set up the destination to queue up the preview images on the image HTTP end-point, we can retrieve the data and convert it to a python image using the following example code. You will need the python pillow library (https://pillow.readthedocs.io/) to run this example.

```
from PIL import Image
from io import BytesIO


def simple_acquisition(serverurl):
    response = requests.get(url=serverurl + '/measurement/start')
    data = response.text
    print('Response of acquisition start: ' + data)


def preview(serverurl, ntrig=1):
    for i in range(ntrig):
    # Getting preview data. This is blocking, so it will wait until an image is
ready.
    response = requests.get(url=serverurl + '/measurement/image')
    image = Image.open(BytesIO(response.content))
    # Show the data in the image
    image.show()
    # Save a preview image to the current directory (<cwd>/data/test-preview...).
    image.save(pathlib.Path(os.path.join(os.getcwd(), 'data', 'test-
preview{}.png'.format(i))))
```

```
# Main part
# Running acquisition process and preview
simple_acquisition(serverurl)
preview(serverurl, 10)
```

4. API description

4.1 Introduction

SERVAL has a built-in webserver that enables controlling its functions. The communication uses the standard HTTP request-response protocol. We use the GET request method to retrieve data or send a command, and PUT to send data to SERVAL, for example to change camera settings.

Requests take the form:

```
<method>   <URL>   [<data>]
```

The standard URL components scheme, host, port, path and query are used to specify server and command or setting.

```
http:// 127.0.0.1:8080/load?format=dacs&file=demo.dacs

<scheme>://<host>:<port><  path >?<      query          >

<          server          ><command>?<parameter=value>
```

Figure 4.1: request structure

We will use `<server>` to describe scheme, host and port. `<command>` to designate a command or setting, and `<parameter=value>` to specify arguments.

The general structure of a full command looks like:

```
<method> <server> / <command> ? <param1=value1&param2=value2>
```

Note 1: The `/` between server and command is strictly speaking part of the path of the URL. Commands could contain multiple path separators. For clarity we frequently include the first `/` when we discuss commands.

Note 2: Commands are NOT case sensitive. However, if a parameter like ‘file’ is specified the value IS case sensitive. For example, the following two commands are equal:

```
GET http://127.0.0.1:8080/measurement/start

GET http://127.0.0.1:8080/mEAsuremEnt/StaRt
```

A full GET request to load the DACs values from a file will look like:

```
GET http://127.0.0.1:8080/config/load?format=dacs&file=demo.dacs
```

The GET method may return data. For example, it could return a JSON structure describing the camera’s pixel configuration, or could return a tiff image.

A full PUT request to set the destination with a JSON object will look like:

```
PUT    http://127.0.0.1:8080/server/destination       <data:JSON object>
```

4.1.1 Response

After processing the request, SERVAL typically responds with the standard HTTP “200 OK” status code (table 4.1), and if applicable, a data object is returned of the type indicated by the HTTP Content-Type; e.g., text, image, file, JSON etc. In cases where the processing is not successful, one of the other status codes can be produced.

HTTP code	HTTP text	Description
200	OK	The request has succeeded, but an application error can still occur, which will be returned as an application error code.
204	No Content	The server has fulfilled the request, but there is no new information to send back
302	Moved Temporarily	The server redirects the request to the URI given in the Location header.
400	Bad Request	The request had bad syntax or was impossible to fulfill.
401	Unauthorized	The request requires user authentication, or the authorization has been refused.
404	Not Found	The server has not found anything matching the request.
409	Conflict	The request could not be completed due to a conflict with the current state of the resource.
500	Internal Error	The server encountered an unexpected condition that prevented it from fulfilling the request.
503	Service Unavailable	The server is unable to handle the request due to temporary overload.

Table 4.1: HTTP responses

4.2 API structure

4.2.1 Namespaces

The API requests are grouped into namespaces:

Namespace	Description
/	Welcome message
/*	Root, shows the entire JSON hierarchy
/server	Server related
/detector	Detector related
/measurement	Measurement related
/config	Configuration via loading and saving or im- and export of files.
/dashboard	Overview of readily (immediately) available information concerning all areas.

4.2.2 Commands, readable and writable parameters

The management and control of the detector and SERVAL involves setting and retrieving parameters like `ExposureTime` or `ElapsedTime`, and sending commands like `measurement/start` (start acquisition).

Parameters are grouped in JSON structures that are either only readable, or read- and writeable. Commands are named with verbs to indicate action. Retrieving and setting JSON structures use end-points named with nouns.

4.3 Welcome and Root: / and /* requests

For convenience SERVAL returns a welcome message when its base URL is requested. The full JSON tree is available by the `/*`, the root request.

Syntax:

```
GET <server><command>
```

Note: PUT methods for `/` and `/*` are not supported.

method	command	Response	remark
`GET`	`/`	Returns a welcome page.	
`GET`	`/*`	Returns a JSON object containing the entire JSON tree.	

4.4 /server requests

The /server requests cover action requests and the destination setting. The “destination” consists of parameters and values that control the way SERVAL outputs its data. For example, you can direct SERVAL to write captured image files to disk or to provide a live preview. The destination is retrievable and configurable by a JSON structure.

Syntax:

```
<method> <server><command>
```

With the following commands:

method	command	Response	remark
GET	/server	Returns JSON object with joint server info	Info: Returns all JSON objects under the server namespace.
GET	/server/shutdown	"Shutting down."	Action: Terminate SERVAL
GET	/server/destination	JSON object (§4.4.1)	Info: Retrieve the current destination
PUT	/server/destination	"Successfully uploaded destination configuration."	Set: Upload destination JSON object (§4.4.1)

Example 1: request shut down SERVAL

```
GET http://127.0.0.1:8080/server/shutdown
```

Example 2: request set destination

```
PUT http://127.0.0.1:8080/server/destination      <data:JSON object>
```

4.4.1 Destination JSON specification

During measurements SERVAL collects raw data from the detector and can process this to images and histograms, which are written to output channels like disk or network. The detector acquisition framerate can be considerably larger than the screen refresh rate. Therefore, live Preview channels are generated at separate, lower framerate. For flexible visualization, options are provided to integrate images. This can be useful to capture a summary of the images that were skipped due to a lower preview framerate than the detector acquisition framerate, or summarize all images from the start of the measurement.

Main categories

There are 3 main categories of output channels that SERVAL supports to write data to while measurement is active:

1. **Raw**
   Unprocessed data straight from the detector in 'tpx3' format, see Appendix in §6.1.
2. **Image**
   Two-dimensional images at detector acquisition framerate.
3. **Preview**
   Live preview of two-dimensional images and histograms at preview framerate with configurable integration over frames.

Output channel configuration

Each output channel has a `Base` to specify the destination. For the `file` scheme, the `Base` includes file path, and a separate `FilePattern` field determines the file name sequence the files will be written to. For the `HTTP` and `TCP` schemes, the `Base` includes an IP address and a port. `HTTP` indicates that the image (or histogram data) can be retrieved with `GET /measurement/image` (see example §3.5.1), or respectively `GET /measurement/histogram`. And TCP indicates that the image or histogram will be sent over the TCP pipe. At the start of a measurement, the client can connect to this TCP address and start listening for data.

The Image and Preview output channels can be configured:

1. Pixel mode: COUNT, TOT, TOA, TOF or COUNT_FB
2. Format: for the file-based 2D images and HTTP images, tiff and pgm formats are supported. For the TCP scheme, pgm, jsonhisto and jsonimage are supported, providing a dedicated easy to decode format.

Raw has a fixed mode and format (tpx3).
Preview output channels are divided into `ImageChannels` and `HistogramChannels`.

TCP: Listen mode & connect mode

The TCP pipe can be configured in two ways 1) listen mode and 2) connect mode.
In listen mode Serval opens a socket at the specified port at the start of a measurement. Hereafter the client can connect to this Serval's TCP address and start listening for data.
In connect mode the client has to open a TCP socket at the specified address and port. On starting the measurement Serval will connect to the client socket. To specify either mode, we use the userinfo part of the URL: tcp://connect@127.0.0.1:8080 or tcp://listen@127.0.0.1:8080.

NOTE: Limitations

The configuration of the destination is very flexible. Some configuration combinations can limit achievable performance. This may depend on the system hardware setup.
In addition, the current version only supports the same "`Mode`" for all output channels. Thus, in all output channels the "`Mode`" field should be the same. Raw does not take a "Mode", so it is excluded.

Example destination configuration

A destination configuration that writes raw and image files to disk and sends two preview image TCP streams (current frame and integrated image from the start of the measurement), may look like:

```
{
  "Raw" : [ {
    "Base" : "file:/data/raw",
    "FilePattern" : "f%Hms_",
    "SplitStrategy" : "FRAME",
    "QueueSize" : 16384
  } ],
  "Image" : [ {
    "Base" : "file:/data/image",
    "FilePattern" : "f%Hms_",
    "Format" : "tiff",
    "Mode" : "tot",
    "Thresholds" : [ 0, 1, 2, 3, 4, 5, 6, 7 ],
    "IntegrationSize" : 0,
    "StopMeasurementOnDiskLimit" : true,
    "QueueSize" : 1024,
    "Corrections" : [ ]
  } ],
  "Preview" : {
```

```
    "Period" : 0.1,
    "SamplingMode" : "skipOnFrame",
    "ImageChannels" : [ {
      "Base" : "tcp://connect@127.0.0.1:8088",
      "Format" : "jsonimage",
      "Mode" : "tot",
      "Thresholds" : [ 0, 1, 2, 3, 4, 5, 6, 7 ],
      "IntegrationSize" : 0,
      "StopMeasurementOnDiskLimit" : false,
      "QueueSize" : 16,
      "Corrections" : [ ]
    }, {
      "Base" : "tcp://listen@127.0.0.1:8089",
      "Format" : "jsonimage",
      "Mode" : "tot",
      "Thresholds" : [ 0, 1, 2, 3, 4, 5, 6, 7 ],
      "IntegrationSize" : -1,
      "IntegrationMode" : "last",
      "StopMeasurementOnDiskLimit" : false,
      "QueueSize" : 16,
      "Corrections" : [ ]
    } ],
    "HistogramChannels" : [ {
      "Base" : "tcp://localhost:8451",
      "Format" : "jsonhisto",
      "Mode" : "tof",
      "Thresholds" : [ 0, 1, 2, 3, 4, 5, 6, 7 ],
      "IntegrationSize" : 0,
      "StopMeasurementOnDiskLimit" : false,
      "QueueSize" : 1024,
      "NumberOfBins" : 100,
      "BinWidth" : 1.0,
      "Offset" : 0,
      "Corrections" : [ ]
    } ]
  }
}
```

Parameter	Value	Description
Raw	Array of {Base, FilePattern, SplitStrategy, QueueSize} objects. See table 4.3.	Defines the raw output. Currently only a single raw output entry is supported.
Image	Array of OutputChannel objects. See table 4.3	Defines the image output.
Preview	Preview object. See table 4.4.	Defines the preview output.

Table 4.2: destination top level parameters

Parameter	values	description

Base	URI Examples: file:/data http://localhost:8082 tcp://localhost:8088 tcp://listen@localhost:8088 tcp://connect@localhost:8088	Send data to a file path or socket like tcp://localhost:8088 TCP listen mode or connect mode can be specified with userinfo: listen@ or connect@. If userinfo is unspecified listen mode is used.
FilePattern	String For example: "f%Hms_"	The "prefix" is prepended to the frame number. If it contains a % it should be followed by a date/time format, to indicate the start time of the run. If this format contains "Hms" and then each of MdHms is replaced by one character out of 0-9A-Za-z, so at Feb 3 4:05:06 am "MdHms" becomes "23456" and Dec 31 2:30:40 pm becomes "CVEUe". Otherwise, a standard DateTimeFormatter is used.
SplitStrategy	String: single_file, frame	Only supported with Raw file channels. Optional, defaults to single_file. Specifies how the raw file output stream is split into files. SINGLE_FILE: splitting is disabled, a single file generated. FRAME: creates one file per frame. Note: events within a millisecond before the end time of the current file may end up in the next file.
QueueSize	Int: > 0 Optional, defaults to 16384 (for Raw channels), 1024 (for Image channels) or 16 (for Preview channels).	The size of the queue per channel. if maximum is reached: - Preview channels: the oldest frame in the queue will be dropped. - Non-Preview channels: pipeline will wait until a spot in queue is free.
Format	String: tiff, pgm, png, jsonimage, jsonhisto	See below for format description.
Mode	String: count, tot, toa, tof, count_fb	COUNT: number of hits above threshold TOT: Time over Threshold TOA: Time of Arrival TOF: Time of Flight COUNT_FB: number of hits in frame-based mode
Thresholds	List of Integers 0 .. 7 Optional, defaults to [0, 1, 2, 3, 4, 5, 6, 7]	A field used by the MPX3, for TPX3 it is unused.
IntegrationSize	Integer: -1, 0, 1 ... 32, Optional: defaults to 0.	0 or 1: no integration

		-1: integration of all images from the start of the measurement > 1 up to 32: integration over the last n images.
IntegrationMode	String: sum, average, last, Optional, defaults to omitted. Required when IntegrationSize is specified.	Sum: sum of images clipped at 32 bits (for each pixel). Average: arithmetic mean of the images clipped at 32 bits (for each pixel). Last: any non-zero value of the new image will overwrite a pre-existing value.
StopMeasurementOnDiskLimit	Boolean: true, false Optional, defaults to true (for file channels), false (for other channels).	If true, the entire measurement will be stopped when a critical disk limit has been reached. If false, the channel will temporarily stop writing to disk when a critical disk limit has been reached. It will attempt to continue writing when enough space has been freed again.
Corrections	List of String: multiply Optional, defaults to [] (no corrections)	Selects the corrections that the images should undergo before being sent to this channel. Multiply: a pixel-wise multiplication with the array specified in /measurement/config/corrections/multiply. If empty, no corrections will apply. Order of this list has no effect; the corrections will be applied in a predetermined order.
NumberOfBins	Integer Optional, defaults to 0	Specifies the number of bins to use in the histogram output (only specified for HistogramChannels).
BinWidth	Float Optional, defaults to 1.0	Specifies the width of the bins to use in the histogram output (only specified for HistogramChannels).
Offset	Int Optional, defaults to 0	Specifies which bin to use as the first bin. Shifts the bins to the left in BinWidth steps (only specified for HistogramChannels).

Table 4.3: OutputChannel object parameters

Parameter	**Value**	**Description**
Preview/Period	Float in [seconds]	Period of the preview refresh rate
Preview/SamplingMode	[“skipOnFrame”, “skipOnPeriod”]	Specifies the method to sample the frames from the Image stream. skipOnFrame: Samples frames based on the number of frames. skipOnPeriod: Samples frames based

		on time (1/period). See below for details.
Preview/ImageChannels	Array of OutputChannel objects. See table 4.4	
Preview/HistogramChannels	Array of OutputChannel objects. See table 4.4	

Table 4.4: destination Preview parameters

Sampling mode

The detector acquisition framerate can be considerably larger than the screen refresh rate. Therefore, live Preview channels are generated at separate, lower framerate. For both modes, the first and last frames are always sent.

skipOnFrame:
Sample frames based on frame count. The Preview/Period is then interpreted as relative to TriggerPeriod.
Every [Preview/Period] / [TriggerPeriod] frames, a frame is sampled and sent.

skipOnPeriod:
Sample frames based on wall time: after the given Preview/Period elapses, the next frame is sampled and sent.

PGM image format

PGM, Portable Gray Map, is a simple image format with a compact header. The P5 magic number is used for 16 bits images, see for reference: https://en.wikipedia.org/wiki/Netpbm#File_formats

TIFF, PNG image format

Industry standard TIFF and PNG.

Jsonimage and jsonhisto format

The jsonimage consists of a JSON header followed by image data. This header contains specific information about each frame sent during measurement. It may look like:

```
{
   "timeAtFrame": 1655990130.181,
   "frameNumber": 14,
   "measurementID": "None",
   "dataSize": 524288,
   "bitDepth": 16,
   "isPreviewSampled": true,
   "thresholdID": 0,
   "pixelEventNumber": 0,
   "tdcEventNumber": 0,
   "integrationSize": 0,
   "integrationMode": "None",
```

```
    "width": 512,
    "height": 512,
    "corrections": []
}
```

The jsonhisto consists of a similar JSON header followed by image data. This header contains specific information about each frame sent during measurement. It may look like:


```
{
    "timeAtFrame": 1656058579.9,
    "frameNumber": 0,
    "measurementID": "None",
    "dataSize": 400,
    "bitDepth": 32,
    "isPreviewSampled": true,
    "thresholdID": 0,
    "pixelEventNumber": 0,
    "tdcEventNumber": 0,
    "integrationSize": 0,
    "integrationMode": "None",
    "binSize": 0,
    "binWidth": 0,
    "binOffset": 0,
    "mean": 0.0,
    "sigma": 0.0
}
```


Parameter	**Values**	**description**
timeAtFrame	Float in seconds	System timestamp since epoch when the frame was taken.
frameNumber	Integer	Frame number. Zero at the start of the measurement
measurementID	String	Context of the measurement. Currently always None.
dataSize	Integer in bytes	Size of the following image or histogram data
bitDepth	Integer	Number of bits used per pixel or item in the image or histogram data.
isPreviewSampled	Boolean	Whether this image is selected for preview or not.
thresholdID	Integer	A field used by the MPX3, for TPX3 it is always 0.
pixelEventNumber	Integer	The number of pixel events fired during this frame.
tdcEventNumber	Integer	The number of TDC events fired during this frame.
integrationSize	Integer	The number of images that were integrated into this image. If this is a non-integrated image, the value will be 0.

integrationMode	String: sum, average, last	The mode the image was integrated with. If this is a non-integrated image, the value will be “None”. Otherwise, the value will be equal to the value set in the Destination.
width	Integer	Width of the image
height	Integer	Height of the image
corrections	List of String: multiply	The list of corrections that were applied to this image. If no corrections were applied, the list will be empty.
binSize	Integer	Histogram: Number of bins, and also the number of items of the array. Currently it is always 0.
binWidth	Integer in steps of 1.5625 nanoseconds (tpx3 native pixel time resolution)	Histogram: width of the bins. Currently it is always 0
binOffset	Integer	Histogram: offset, position of the first bin. Currently it is always 0.
mean	Float	Mean value in the histogram. Currently it is always 0.
sigma	Float	Standard deviation in the histogram. Currently it is always 0.

Table 4.5: Jsonimage and jsonhisto format parameters

4.5 /detector requests

Settings are structured as paths with levels. Figure 4.1 outlines the settings tree setup. The top level `detector` object consists of `info`, `health`, `layout`, `config` and `chips`, an array of chip configurations, one per detector chip. Each chip configuration consists of `DACs`, `adjust` and `PixelConfig` settings.

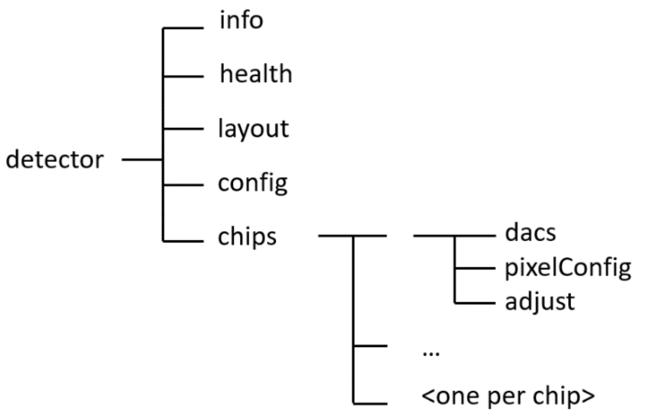


Figure 4.2: JSON tree

By specifying the path to a specific parameter or a group of parameters, one can retrieve the actual value(s) or set these.

Example to retrieve the DAC values of the third chip:

```
GET http://127.0.0.1:8080/detector/chips/2/dacs
```

This will return a JSON structure with the DAC values of chip 3. The first chip has chip ID 0.

Example to set detector configuration values:

```
PUT http://127.0.0.1:8080/detector/config       <data:JSON object>
```

Note that the parameter values are passed by uploading the dump of a JSON structure with a specific format (check paragraph 4.4.1).

To disable a pixel, set the mask value to 1. For example, to set the mask value of row 127, column 63 of the second chip, use:

```
PUT http://127.0.0.1:8080/detector/chips/1/mask/127/63      <data:'1'>
```

Note that the value is passed by uploading the string ‘1’. In this case, the html <query> field is not used to pass parameters. The mask property is part of pixelConfig.

In addition to manipulate settings with JSON, SERVAL can load and save directly settings or configuration files (see section 4.6).

4.5.1 GET request retrieving settings

Syntax:

```
GET <server><command>
```

With the following settings:

command	description
/detector/list	Returns a JSON list of objects containing a field "Address" with the respective IPs.
/detector/connect	If no parameter is provided, the first camera of the list is used.; (parameter is not implemented yet).
/detector/disconnect	Will disconnect the current camera.
/detector	Returns the top-level JSON object (§4.5.0).
/detector/info	Returns the detector info as a JSON object (§4.5.3).
/detector/health	Returns the detector health as a JSON object (§4.5.4).
/detector/layout	Returns the detector layout as a JSON object (§4.5.5).
/detector/config	Returns the detector config as a JSON object (§4.5.6).
/detector/chips	Returns a list of JSON objects representing the chips (§4.5.0).
/detector/chips/<chip number>	Returns a JSON object representing the chip (containing DACs, PixelConfig and adjust) (§4.5.0). Note: <chip number> in the command starts at 0. For a quad tpx3, the valid values are: [0,3]
/detector/chips/<chip number>/dacs	Returns a JSON object with the DAC parameter values (§4.5.7).
/detector/chips/<chip number>/pixelconfig	Returns a JSON object with the PixelConfig as a String.
/detector/chips/<chip number>/pixelconfig?format=bpc	Returns the PixelConfig in a binary (bpc) format.
/detector/chips/<chip number>/mask/<row>/<column>	Returns a String object with the mask value of the pixel of at position (row, column).

Example: request detector health

```
GET http://127.0.0.1:8080/detector/health
```

Example: request PixelConfig of the 3rd chip

```
GET http://127.0.0.1:8080/detector/chips/2/pixelconfig
```

Example: request the full detector data (§4.5.3)

```
GET http://127.0.0.1:8080/detector
```

4.5.2 GET action layout requests: rotate and flip image

The rotate command requests to apply an orientation transform to the images as defined by the destination.

Syntax:

```
GET <server>/detector/layout/rotate?flip=<flip>&direction=<direction>
```

With the following commands:

Command	Response	Remark
`/detector/rotate?reset& flip=horizontal& direction=right`	Successfully reset layout, then rotated layout <direction>, then flipped layout <flip>.	Action: apply orientation transformation. This will update the DetectorOrientation field of the /detector/config, see (§4.5.6).

And parameters:

Format	Values
flip	horizontal, vertical
direction	left, right, 180
reset	Not applicable

Example: request image to rotate right

```
GET http://127.0.0.1:8080/detector/layout/rotate?direction=right
```

Example: request image to rotate right and then flip horizontally

```
GET
http://127.0.0.1:8080/detector/layout/rotate?flip=horizontal&direction=right
```

Example: request to reset to default orientation

```
GET http://127.0.0.1:8080/detector/layout/rotate?reset
```

4.5.3 PUT request to set detector settings by uploading data

Syntax:

```
PUT <server>/<path>      <data object>
```

With the following setting and data:

path	data	description
`/detector`	JSON dump of config structure as specified in §4.3.6	Uploads all settable data to SERVAL. Updates the connected detector with the new configuration.
`/detector/config`	JSON dump of detector config structure as specified in §4.3.7	Uploads the detector config data to SERVAL. Updates the connected detector with the new configuration.
`/detector/chips`	List of JSON dump of chips config structure as specified in §4.2	Uploads the settings for all chips in bulk. Updates the connected detector with the new configuration.
`/detector/chips/<chip number>`	JSON dump of chips config structure as specified in §4.2	Uploads the settings for a chip. Updates the connected detector with the new configuration.
`/detector/chips/<chip number>/dacs`	JSON dump of DACs structure as specified in §4.2	Uploads the DACs for a specific chip. Updates the connected detector with the new configuration.
`/detector/chips/<chip number>/pixelconfig`	JSON dump of PixelConfig structure as specified in §4.2	Uploads the PixelConfig in JSON String format for a specific chip. Updates the connected detector with the new configuration.
`/detector/chips/<chip number>/pixelconfig?format=bpc`	Binary dump of BPC structure.	Uploads the PixelConfig in binary format for a specific chip. Updates the connected detector with the new configuration.
`/detector/chips/<chip number>/mask/<row>/<column>`	String, [‘0’, ‘1’]	‘0’: pixel not masked ‘1’: pixel masked

Example: request set detector config

```
PUT http://127.0.0.1:8080/detector/config    <data:JSON object>
```

4.5.4 Detector Info JSON structure


```
{
  "IfaceName" : "Spidr",
  "SW_version" : "21052719",
  "FW_version" : "18052510",
  "PixCount" : 262144,
  "RowLen" : 2,
  "NumberOfChips" : 4,
  "NumberOfRows" : 512,
  "MpxType" : 6,
  "Boards" : [ {
    "ChipboardId" : "41000039",
    "IpAddress" : "127.0.0.10",
    "FirmwareVersion" : "18052510",
    "Chips" : [ {
      "Index" : 0,
      "Id" : 680,
      "Name" : "W0002_H10"
    }, {
      "Index" : 1,
      "Id" : 681,
      "Name" : "W0002_I10"
    }, {
      "Index" : 2,
      "Id" : 682,
      "Name" : "W0002_J10"
    }, {
      "Index" : 3,
      "Id" : 683,
      "Name" : "W0002_K10"
    } ]
  } ],
  "SuppAcqModes" : 63,
  "ClockReadout" : 125.0,
  "MaxPulseCount" : 2147483647,
  "MaxPulseHeight" : 1.0,
  "MaxPulsePeriod" : 34.35973836,
  "TimerMaxVal" : 34.35973836,
  "TimerMinVal" : 8.0E-9,
  "TimerStep" : 8.0E-9,
  "ClockTimepix" : 125.0
}
```


Parameter	Value	Description
IfaceName	string	Interface name
SW_version	string	Readout Software version
FW_version	string	Readout Firmware version
PixCount	integer	Total number of pixels of all chips

RowLen	integer	Row length
NumberOfChips	integer	Number of chips
NumberOfRows	integer	Number of pixel rows of whole chip layout
MpxType	integer	Medipix type
Boards	array	
ChipboardId	string	Unique id of the chipboard
IpAddress	string	Address of the chipboard
FirmwareVersion	string	Version of the firmware of the chipboard
Chips	array	
Index	integer	Chip index
Id	integer	Chip ID
Name	string	Chip name (from wafer and position)
SuppAcqModes	integer	
ClockReadout	float	
MaxPulseCount	integer	
MaxPulseHeight	float	
MaxPulsePeriod	float	
TimerMaxVal	float	
TimerMinVal	float	
TimerStep	float	
ClockTimepix	float	

Table 4.6: detector info parameters

4.5.5 Detector Health JSON structure


```
{
  "LocalTemperature" : 30.713,
  "FPGATemperature" : 50.57,
  "ChipTemperatures" : [ 52, 47, 53, 25 ],
  "Fan1Speed" : 1200,
  "Fan2Speed" : 1200,
  "AVDD" : [ 1.44, 1.796, 2.586 ],
  "VDD" : [ 1.444, 0.708, 1.022 ],
  "BiasVoltage" : 49.951171875,
  "Humidity" : 20
}
```


Parameter	**Value**	**Description**
LocalTemperature	Float in °C	Acquisition board temperature
FPGATemperature	Float in °C	FPGA temperature
ChipTemperatures	Array of integers in °C	Chip temperatures
Fan1Speed	Integer	rpm of Fan 1
Fan2Speed	Integer	rpm of Fan 2
VDD	Array of floats	Supply readings in [V, A, W]
AVDD	Array of floats	Analog supply readings in [V, A, W]
BiasVoltage	Float	Bias voltage in Volts
Humidity	Integer	Board humidity

Table 4.7: Detector health parameters

4.5.6 Detector Layout JSON structure

```
{
  "DetectorOrientation" : "UP",
  "Original" : {
    "Width" : 512,
    "Height" : 512,
    "Chips" : [ {
      "Chip" : 0,
      "X" : 256,
      "Y" : 0,
      "Orientation" : "RtLBtT"
    }, {
      "Chip" : 1,
      "X" : 0,
      "Y" : 0,
      "Orientation" : "RtLBtT"
    }, {
      "Chip" : 2,
      "X" : 0,
      "Y" : 256,
      "Orientation" : "LtRTtB"
    }, {
      "Chip" : 3,
      "X" : 256,
      "Y" : 256,
      "Orientation" : "LtRTtB"
    } ]
  },
  "Rotated" : {
    "Width" : 512,
    "Height" : 512,
    "Chips" : [ {
      "Chip" : 0,
      "X" : 256,
      "Y" : 0,
      "Orientation" : "RtLBtT"
    }, {
      "Chip" : 1,
      "X" : 0,
      "Y" : 0,
      "Orientation" : "RtLBtT"
    }, {
      "Chip" : 2,
      "X" : 0,
      "Y" : 256,
      "Orientation" : "LtRTtB"
    }, {
      "Chip" : 3,
      "X" : 256,
      "Y" : 256,
```

```
        "Orientation" : "LtRTtB"
      } ]
    }
}
```

Parameter	Value	Description
DetectorOrientation	String, 8 possible values UP, RIGHT, DOWN, LEFT UP_MIRRORED, RIGHT_MIRRORED, DOWN_MIRRORED, LEFT_MIRRORED	Determines the orientation to take in the “Rotated” field. UP is the original position. The MIRRORED orientations are horizontally flipped.
Original	{Width, Height, Chips}	Represents the non-rotated layout. Any changes in this layout will be propagated into the Rotated layout, after rotation.
Rotated	{Width, Height, Chips}	Represents the rotated layout. Obtained by rotating Original by DetectorOrientation. Any changes in this layout will be propagated into the Original layout, by applying the inverse rotation.
Width	integer	Represents the horizontal width of the canvas.
Height	integer	Represents the vertical height of the canvas.
Chips	List of {Chip, X, Y, Orientation} objects.	Represents the positions of the chips on the canvas.
Chip	integer	Chip index
X	integer	X position of the chip
Y	integer	Y position of the chip
Orientation	String, 8 possible values LtRBtT, RtLBtT, LtRTtB, RtLTtB, BtTLtR, TtBLtR, BtTRtL, TtBRtL	Represents the readout order of the chip onto the canvas. All 8 possible rotation and mirroring combinations. LtR: Left to Right BtT: Bottom to Top

Table 4.8: detector layout parameters

4.5.7 Detector Config JSON structure

The detector configuration consists of parameters and values. These are retrievable or configurable by a JSON structure.

```
{
  "LogLevel" : 1,
  "Fan1PWM" : 100,
  "Fan2PWM" : 100,
  "BiasVoltage" : 50,
  "BiasEnabled" : true,
  "Polarity" : "Positive",
  "PeriphClk80" : false,
  "ChainMode" : "NONE",
  "TriggerIn" : 0,
  "TriggerOut" : 0,
  "TriggerPeriod" : 0.016666666666666666,
  "ExposureTime" : 0.0002,
  "TriggerDelay" : 0.0,
  "TriggerMode" : "AUTOTRIGSTART_TIMERSTOP",
  "nTriggers" : 100,
  "Tdc" : [ "PN0123", "PN0123" ],
  "GlobalTimestampInterval" : 10.0,
  "ExternalReferenceClock" : false
}
```

Parameter	**values**	**description**
LogLevel	0, 1, 2	Logging level
Fan1PWM	[0, 100] integer	Pulse Width Modulation value of Fan 1
Fan2PWM	[0, 100] integer	Pulse Width Modulation value of Fan 2
BiasVoltage	[0, 140]	Bias voltage in [Volts]
BiasEnabled	false, true	Enable bias
Polarity	[Negative, Positive]	Usually Positive
PeriphClk80	false, true	Enables 80MHz readout for single-chip TPX3
ChainMode	NONE LEADER	Default setting Sets device as *leader* enabling synchronization of multiple TPX3 devices or synchronization with other instruments. *For synchronization of multiple TPX3 devices* connect HDMI2 of LEADER to HDMI1 on a FOLLOWER; *For synchronization of other devices:* LEADER mode will output on a pre-configured triggerbox connected to HDMI2 (available on request from ASI) output 1: the T0 sync start of a measurement

	FOLLOWER	output 2: the shutter signal of the TPX3 detector output 4: the internal clock of the TPX3 detector This setting overrides TriggerIn/TriggerOut; External shutter control of the leader is still possible with HDMI1 and an ASI triggerbox HDMI1 connected to HDMI2 on the LEADER or another FOLLOWER this overrides TriggerIn, TriggerOut, ExposureTime, TriggerMode, nTriggers
TriggerIn	[1, 6]	Sets the HDMI channel numbers (1-3 = HDMI1, 4-6 = HDMI2)
TriggerOut	[1, 6]	Sets the HDMI channel numbers (1-3 = HDMI1, 4-6 = HDMI2)
TriggerPeriod	[0, 50.0] float	Trigger period [seconds]
ExposureTime	[0, 10.0] float	Exposure time in [seconds]
TriggerDelay	[0, 1.0] integer	Trigger delay in [seconds]
TriggerMode	PEXSTART_NEXSTOP NEXSTART_PEXSTOP PEXSTART_TIMERSTOP NEXSTART_TIMERSTOP AUTOTRIGSTART_TIMERSTOP CONTINUOUS SOFTWARESTART_TIMERSTOP SOFTWARESTART_SOFTWARESTOP	Acq. is started by positive edge external trigger input, stopped by negative edge Acq. is started by negative edge external trigger input, stopped by positive edge Acq. is started by positive edge external trigger input, stopped by HW timer Acq. is started by negative edge external trigger input, stopped by HW timer Acq. is started by trigger from HW, stopped by HW timer. For this mode the shutter must be closed for at least the required deadtime. Required deadtime is 2ms or 1ms when PeriphClk80 is enabled. TriggerPeriod - ExposureTime > deadtime Acq. is started by software, stopped by software Acq. is started by API call /measurement/trigger/start, stopped by HW timer

		Acq. Is started by API call /measurement/trigger/start, stopped by API call /measurement/trigger/stop Note: this setting controls the shutter behavior. For a more detailed description, consult the trigger manual.
nTriggers	[0, max] integer	Number of triggers to acquire
Tdc	[<tdc1 string>, <tdc2 string>] Example single chip camera: ['P0', 'N0'] Records the positive edge of tdc1 & negative edge of tdc2 Example quad chip camera: ['PN0123', 'PN0123'] Example TDC recording off: ['', '']	Specifies TDC recording by an array of strings P: positive edge N: negative edge PN: both edges will be recorded 0..3: chip number Note: TDC recording is gated by the shutter.
GlobalTimestampInterval	Float <= 0, or [0.001, 10E6]	Specifies the timeinterval between global timestamps in the raw data stream. <= 0 : disabled [0.001, 10E6] : interval in seconds
ExternalReferenceClock	false, true	Whether to use an external clock. Please contact support if you want to use this option, because it requires a special detector setup. The external clock should be connected to the 4th channel of the first HDMI slot.

Table 4.9: Detector config parameters

4.5.8 DACs JSON structure

```
{
  "Ibias_Preamp_ON" : 128,
  "Ibias_Preamp_OFF" : 8,
  "VPreamp_NCAS" : 128,
  "Ibias_Ikrum" : 10,
  "Vfbk" : 128,
  "Vthreshold_fine" : 216,
  "Vthreshold_coarse" : 6,
  "Ibias_DiscS1_ON" : 128,
  "Ibias_DiscS1_OFF" : 8,
  "Ibias_DiscS2_ON" : 128,
  "Ibias_DiscS2_OFF" : 8,
  "Ibias_PixelDAC" : 230,
  "Ibias_TPbufferIn" : 128,
  "Ibias_TPbufferOut" : 128,
  "VTP_coarse" : 128,
  "VTP_fine" : 256,
  "Ibias_CP_PLL" : 128,
  "PLL_Vcntrl" : 128
}
```

Parameter	values
Ibias_Preamp_ON	integer
Ibias_Preamp_OFF	integer
VPreamp_NCAS	integer
Ibias_Ikrum	integer
Vfbk	integer
Vthreshold_fine	integer
Vthreshold_coarse	integer
Ibias_DiscS1_ON	integer
Ibias_DiscS1_OFF	integer
Ibias_DiscS2_ON	integer
Ibias_DiscS2_OFF	integer
Ibias_PixelDAC	integer
Ibias_TPbufferIn	integer
Ibias_TPbufferOut	integer
Vtp_coarse	integer
Vtp_fine	integer
Ibias_CP_PLL	integer
PLL_Vcntrl	integer

Table 4.10: DACs parameters

4.6 /measurement requests

4.6.1 GET request to get measurement information

Syntax:
```
<method> <server><command>
```

With the following commands:

method	command	Response	Remark
GET	/measurement/start	Successfully started measurement.	Action: start measurement
GET	/measurement/stop	Successfully stopped measurement.	Action: stop measurement
GET	/measurement/preview	Successfully started previewing.	Action: start preview Same as start but only the preview destination will output data. No file recording.
GET	/measurement	Returns JSON object with joint measurement info	Combined measurement data.
GET	/measurement/config	JSON dump of config structure as specified in §4.6.2	The configuration options for the measurement.
GET	/measurement/image	Returns image Data: image	The output image, as specified in the HTTP setup in the destination config.
GET	/measurement/histogram	Returns histogram, array of int Data: histogram	Info: the histogram as specified as the HTTP setup in the destination config.

Example 1: request start measurement
```
GET http://127.0.0.1:8080/measurement/start
```

Example 2: request get image from HTTP queue if configured in destination configuration. Returns an image blob as specified by the destination configuration image format.
```
GET http://127.0.0.1:8080/measurement/image
```

4.6.2 PUT request to set measurement settings by uploading data

Syntax:
```
PUT <server>/<path>      <data object>
```

With the following setting and data:

Path	Data	Description
/measurement	JSON dump of full measurement structure.	Uploads all settable measurement data to SERVAL.

`/measurement/config`	JSON dump of measurement config structure as specified in §4.6.3	Uploads the measurement config data to SERVAL.
`/measurement/corrections`	JSON dump of corrections config structure as specified in Table 4.12	Uploads the corrections config data to SERVAL.
`/measurement/config/timeofflight`	JSON dump of time-of-flight config structure as specified in Table 4.13	Uploads the time-of-flight config data to SERVAL.

Example: request set measurement config

```
PUT http://127.0.0.1:8080/measurement/config     <data:JSON object>
```

4.6.3 Measurement config JSON structure

```
{
  "Corrections" : {
    "Multiply" : [1.5, 2.0, 1.23, 1.1, 0.89, …, 0.1],
    "Gapfill" : {
      "Distance" : 1,
      "Strategy" : "NEIGHBOUR"
    }
  },
  "TimeOfFlight" : {
    "TdcReference" : [ "PN0123", "PN0123" ],
    "Min" : 0.0,
    "Max" : 1.0E99
  }
}
```

Parameter	values	description
Corrections	Corrections configuration (see Table 4.12)	The full JSON structure of correction parameters
TimeOfFlight	TimeOfFlight configuration (see Table 4.13)	The full JSON structure of time-of-flight parameters

Table 4.11: Measurement config parameters

Parameter	values	description
Multiply	List of Float	Per-pixel multiplication values to apply to the image in “multiply” corrections.
Gapfill	{Distance, Strategy}	The full JSON structure of gapfill parameters

Distance	Integer	How far the strategy will look outward from a source pixel to find gap pixels. For example, a distance of 1 will only affect directly neighbouring pixels (for corner pixels, this includes the diagonal).
Strategy	NEIGHBOUR, SPLIT	NEIGHBOUR: copy the neighbouring value, averaging if there are multiple neighbours in the “Distance” range of the pixel. SPLIT: splits the value of the source pixel evenly to itself and the gap pixels in range.

Table 4.12: Corrections config parameters

Parameter	values	description
TdcReference	[<tdc1 string>, <tdc2 string>] Example single chip camera: ['P0', 'N0'] Records the positive edge of tdc1 & negative edge of tdc2 Example quad chip camera: ['PN0123', 'PN0123'] Example TDC recording off: ['', '']	Specifies TDC recording by an array of strings P: positive edge N: negative edge PN: both edges will be recorded 0..3: chip number Note: Which TDCs come in are defined by the Tdc field in /detector/config.
Min	Float	Filters events with a TOF equal or higher than this value to be included in the image.
Max	Float	Filters events with a TOF equal or lower than this value to be included in the image.

Table 4.13: Time of flight config parameters

4.7 /config requests

Configuration via loading and saving of request data and im- and export of files.

Syntax:

```
<method> <server><command>?format=<format>&file=<filepath>
```

With the following commands:

method	command	Response	remark
GET	`/config/load`	Successfully uploaded config.	Action: load config file of type indicated by format
GET	`/config/store`	Successfully loaded config from <filepath>	Action: store config file of type indicated by format

Note: PUT methods for /config/* are not supported.

With the following format:

format	remark
serval	Type: JSON Content: an overall configuration file to facilitate persistency of SERVAL's state.
pixelconfig	Type: 'bpc' fileformat Content: a proprietary binary format specifying for all chip pixels row by column: the equalisation trim value, maskingbit, and testbit.
dacs	Type: JSON or 'dacs' fileformat Content: DAC values of the detector.

Example 1: request save config to file with (server) path `~/experiment/myconfig.json`

```
GET
http://127.0.0.1:8080/config/store?format=serval&file='~/experiment/myconfig.json'
```

Example 2: request load pixel configuration from file with path `~/tpx3Detector_asi.bpc`

```
GET
http://127.0.0.1:8080/config/load?format=pixelconfig&file='~/tpx3Detector_asi.bpc'
```

4.8 /dashboard requests

Overview of readily (immediately) available information concerning all areas.

Syntax:

```
<method> <server><command>
```

With the following commands:

method	command	Response	remark

GET	`/dashboard`	Returns JSON object with joint info see table (§4.9)	This request is designed to respond as fast as possible by returning only data that is immediately available and does not require hardware polling. An example of its usage can be live retrieval of the number of captured frames (frameCount).

Example 1: request dashboard

```
GET http://127.0.0.1:8080/dashboard
```

4.8.1 dashboard JSON structure


```
{
  "Server" : {
    "SoftwareVersion" : "3.0.0",
    "SoftwareTimestamp" : "2022/06/21 14:04",
    "SoftwareCommit" : "60be309",
    "SoftwareBuild" : "311",
    "DiskSpace" : [ ... ],
    "Notifications" : [ ... ]
  },
  "Measurement" : {
    "StartDateTime" : 1656070697593,
    "TimeLeft" : 5992.0,
    "ElapsedTime" : 189.855,
    "FrameCount" : 379617,
    "DroppedFrames" : 0,
    "Status" : "DA_RECORDING"
    "PixelEventRate" : 1455120,
    "TdcEventRate" : 100,
  },
  "Detector" : {
    "DetectorType" : "Tpx3"
  }
}
```


Parameter	Values	description
SoftwareVersion	String	Software Version
SoftwareTimestamp	String	Timestamp of the software build
SoftwareCommit	String	Additional version information
SoftwareBuild	String	Additional version information
DiskSpace	Array of diskspace info	DiskSpace per destination file channel, see below for details.
Notifications	Array of notifications	Notifications, see below for details.
StartDateTime	UNIX timestamp	Timestamp of the start of the measurement
TimeLeft	Float	(Predicted) Time left in seconds during measurements to reach the specified number of frames
ElapsedTime	Float	Elapsed time in seconds from the start of the measurement
FrameCount	Integer	Number of captured frames during measurement
DroppedFrames	Integer	Total number of dropped frames
PixelEventRate	Integer	Total number of extrapolated pixel events per second: EventRate = numEventsInLastSecond / openTimeInLastSecond
TdcEventRate	Integer	Total number of extrapolated TDC events per second: EventRate = numEventsInLastSecond / openTimeInLastSecond
Status	Enum: [DA_IDLE,	Indicating data acquisition state: : idle

	DA_PREPARING, DA_RECORDING, DA_STOPPING]	: busy to setup recording : busy recording and output data to destinations : busy to stop the recording process
DetectorType	String	Detector type, for example Tpx3 or Mpx3

Table 4.14: dashboard parameters

4.8.2 DiskSpace JSON structure


```
[{
  "Message" : "Free Space: 670.8 GB. Space left to lower limit
(100.0 MB): 670.7 GB.",
   "Path" : "/data/Measurement_apr_01_2021_11h05m10s/image",
   "FreeSpace" : 670758268928,
   "WriteSpeed" : 1.050576E9,
   "LowerLimit" : 100000000,
   "DiskLimitReached" : false
}]
```


Parameter	Values	description
Message	String	Summary
Path	String	File path of the file channel
FreeSpace	Integer	Usable free space in bytes for the path of the file channel
LowerLimit	Integer	Required free space when diskspace limit is reached.
DiskLimitReached	Boolean	True if disk space limit is reached.

Table 4.15: diskspace parameters

4.8.3 Notification JSON structures


```
[{
   "Type" : "severe",
   "Domain" : "server",
   "Message" : "Stopped writing to file channel because free disk
space limit was reached (100.0 MB) in directory:
/home/pluto2/data/Measurement_apr_01_2021_12h17m54s/image",
   "ReferenceID" : "REF_ID_DISK_FULL",
   "Timestamp" : 1617277710219
}, {
   "Type" : "severe",
   "Domain" : "server",
   "Message" : "Noticed freed disk space of directory:
/home/pluto2/data/Measurement_apr_01_2021_12h17m54s/image. Resuming
writing to file channel. Usable (free) disk space limit (100.0 MB)",
   "ReferenceID" : "REF_ID_DISK_SPACE_FREED",
   "Timestamp" : 1617277757320
}]
```


Parameter	**Values**	**description**
Type	String: update, info, severe, error	-
Domain	String: server, detector, chip	-
Message	String	-
ReferenceID	String: REF_ID_DISK_FULL REF_ID_DISK_SPACE_FREED REF_ID_GENERAL	Identifies: when diskspace limit is reached. when diskspace is freed. general notification occurred.
Timestamp	UNIX timestamp	Time when the notification occurs.

Table 4.16: notification parameters

5. Example software

A collection of python code examples is available on request. Contact your support contact for more information.

6. Appendix: file formats

6.1 TPX3 raw file format

The .tpx3 raw data files contain the original data as sent by the readout board. It consists of chunks with an 8-byte chunk header prepended (see Table 6.1).
The chunk content consists of 8-byte words (little endian); the type of which is determined by the most significant nibble (i.e., the high nibble of the **last** byte):

- **TPX3 pixel data** with type 0xa for count_fb, or 0xb for the other modes, the maximum timestamp is 26.8435456 s;
- **SPIDR TDC data** with type 0x6, the maximum timestamp is 107.3741824 s;
- **TPX3 global time** with type 0x4, the maximum is ~81 days;
- **SPIDR control** with type 0x5.
- **TPX3 control** with type 0x7.

Sample code on how to decode this information from the raw data can be requested from your support contact.

Table 6.1. Chunk header

63 - 48 bit	47 - 40 bit	39 - 32 bit	31 – 0 bit			
Chunk size (bytes)	Reserved	Chip index	'3'	'X'	'P'	'T'
			"TPX3"			

Table 6.2. Pixel data packet (0xa, 0xb)

63 - 60 bit	59 - 44 bit	43 - 30 bit	29 - 20 bit	19 – 16 bit	15 – 0 bit
0xa (in count_fb)	PixAddr (see Table 6.6)	Integrated ToT (25ns)	EventCount	HitCount	SPIDR time (0.4096ms)
0xb (in other modes)	PixAddr (see Table 6.6)	ToA (25ns)	ToT (25ns)	FToA (-1.5625ns)	SPIDR time (0.4096ms)

Table 6.3. TDC data packet (0x6)

63 - 60 bit	59 – 56 bit	55 - 44 bit	43 - 9 bit	8 - 5 bit	4 – 0 bit
0x6	0xf = TDC1 Rise 0xa = TDC1 Fall 0xe = TDC2 Rise 0xb = TDC2 Fall	Trigger count	Timestamp (3.125ns)	Fine timestamp Takes value 1-12 (260.41666ps*)	Reserved

* Value 0 is an error state. Therefore: value 1 = 0ps, 2 = 260.41666ps, 3 = 520.83332ps, … 12 = 2.86458ns

Table 6.3. Global time data packet (0x4)

63 - 56 bit	55 – 48 bit	47– 16 bit	15 – 0 bit
0x44 = Time Low	Reserved	Timestamp (25ns)	SPIDR time (0.4096ms)

63 - 56 bit	55 – 32 bit	31 – 16 bit	15 – 0 bit
0x45 = Time High	Reserved	Timestamp (107.374182s)	SPIDR time (0.4096ms)

Table 6.4. SPIDR control packet (0x5)

63 - 56 bit	55 – 48 bit	47 – 0 bit
0x50 = Packet ID	Reserved	Packet count

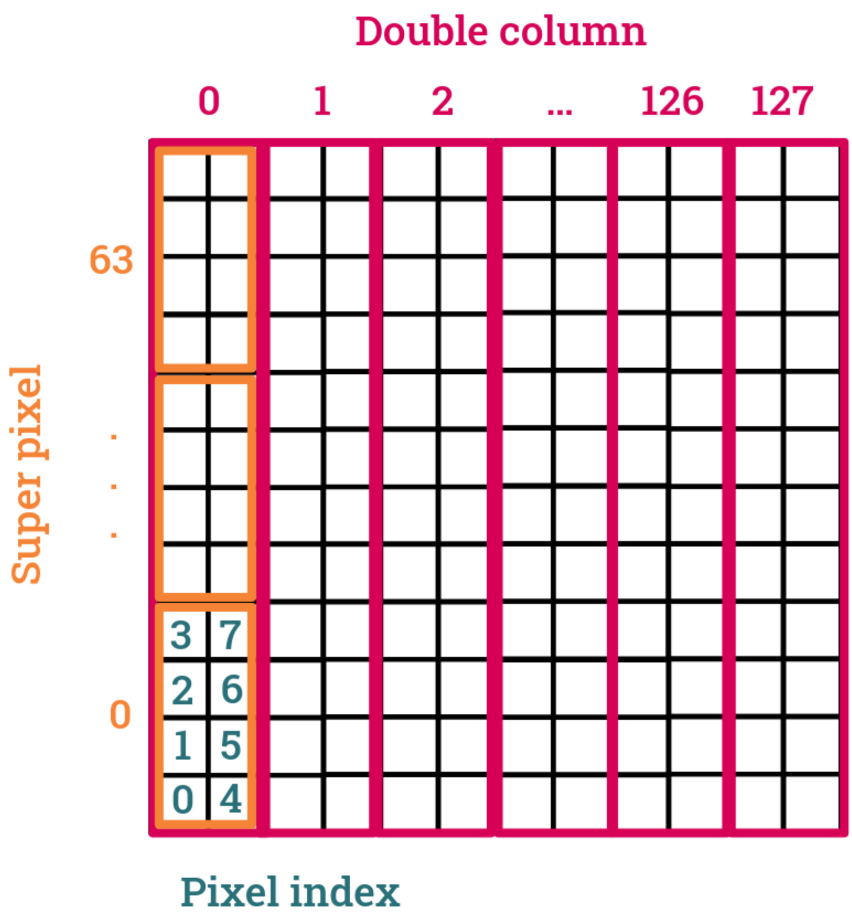
64 – 60 bit	59 – 56 bit	55 – 46 bit	45 – 12 bit	11 – 0 bit
0x5	0xf = open shutter 0xa = close shutter 0xc = heartbeat	Reserved	Timestamp (25ns)	Reserved

Table 6.5. TPX3 control packet (0x7)

63 - 56 bit	55 – 48 bit	51 – 0 bit
0x71	0xa0 = End of sequential readout 0xb0 = End of data driven readout	Reserved

Table 6.6. Pixaddr format

15 - 9 bit	8 - 3 bit	2 – 0 bit
Double column There are 128 double columns on the chip, indexed left to right.	Super pixel There are 64 super pixels in each double column, stacked vertically, indexed bottom to top.	Pixel index There are 8 pixels in each super pixel. Indexed bottom to top, left to right. 0-3 = left column, 4-7 = right column.



6.2 Image File

In SERVAL, image files can be saved as .tiff format, .png format and .pgm format. They take the dimensions as defined in /detector/layout in the Rotated field. Depending on the acquisition mode (count, tot, toa, tof) and image format, they can come in 8, 16 and 32-bit ranges. If specified in the Destination, image files and raw data can be saved at the same time.

7. Versions

Version	Date	Serval version
V0.87	20.01.2020	1.0
V1.00	03.12.2020	2.0
V1.10	08.01.2021	2.1.0
V1.11	03.03.2021	2.1.2
V1.20	01.04.2021	2.1.2
V1.21	13.04.2021	2.1.4
V1.22	06.05.2021	2.1.5
V1.3	02.09.2021	2.3.0
V1.31	11.11.2021	2.3.4
V3.0	23.06.2022	3.0.0
V3.1	31.10.2022	3.1.0

About Amsterdam Scientific Instruments

Amsterdam Scientific Instruments (ASI) is a fast-growing high-tech company located in Amsterdam, the Netherlands. The company was founded in 2011 as a spin-off of the Dutch National Institute for Subatomic Physics (Nikhef) and the Dutch Research Council Institute for the physics of functional complex matter (AMOLF).

Our mission is to progress the pace and scope of scientific research by enabling more sensitive and accurate measurement of high-energy particles. We empower researchers with sophisticated instruments that spark breakthrough discoveries and generate new insights into the functional dynamics and properties of X-rays, photons, electrons, neutrons and ions.

ASI produces hybrid-pixel detectors specialized for a broad range of analytical imaging techniques in fundamental physics, materials science and life science research. Our detectors are based on the Medipix/Timepix sensor ASIC developed and marketed by CERN, the European Organization for Nuclear Research based in Geneva, Switzerland. Primary applications supported by ASI's devices involve diffraction-based electron microscopy (e.g., EBSD, MicroED, Single Particle Analysis and 4D STEM), time-resolved microscopy (e.g., EELS, Molecular Imaging), and reaction microscopy (e.g., VMI, PEPICO, TOF-PEEM). ASI's detectors offer unprecedented performance, with attributes including single particle sensitivity, noiseless detection, event-based time stamping, wide dynamic range, and ultra-fast frame rates.

At ASI, we take great care to satisfy our customers by providing customizable hardware solutions, powerful and intuitive software tools, and dedicated installation and support services. We are passionate about innovation and strive to deliver the best quality products to our customers.