SERVAL User manual

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 13 April. 2021
Version 1.21

Contents

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 or LabView to control your experiment. The ASI camera is connected via Ethernet. In this method you control all your devices from one PC.

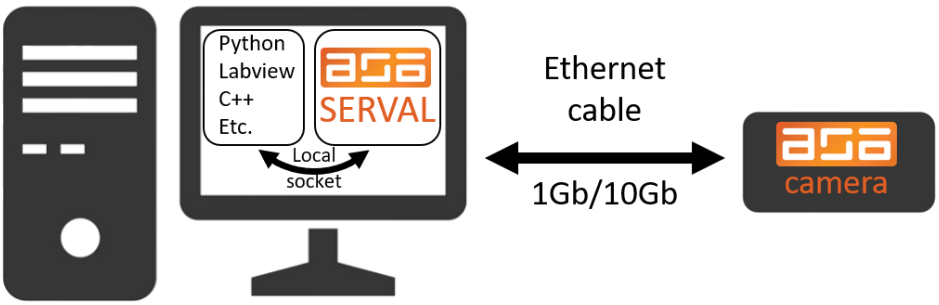


Figure 1, Using SERVAL as a local server, on your own acquisition PC. Both your software and SERVAL are running on your PC using a local socket to communicate (left) and SERVAL controls the camera (right).

Remote client
The client is on a different computer, connected over Ethernet, this is the high-performance method. 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 the server with SERVAL is normally 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.

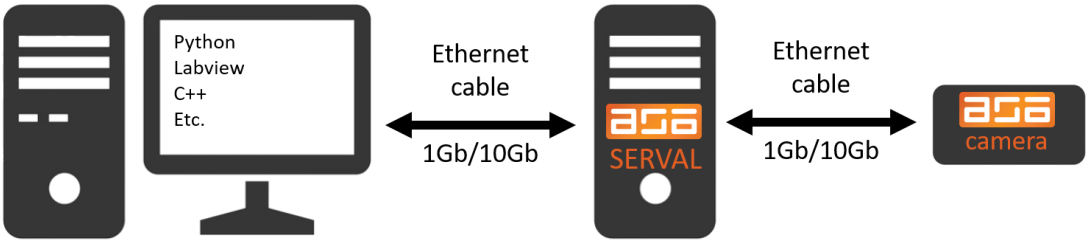


Figure 2, Using SERVAL from a remote server. SERVAL is on a physically separate server to get maximum performance. Here the client PC (left) instructs the SERVAL server (middle) to control the camera (right).

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.

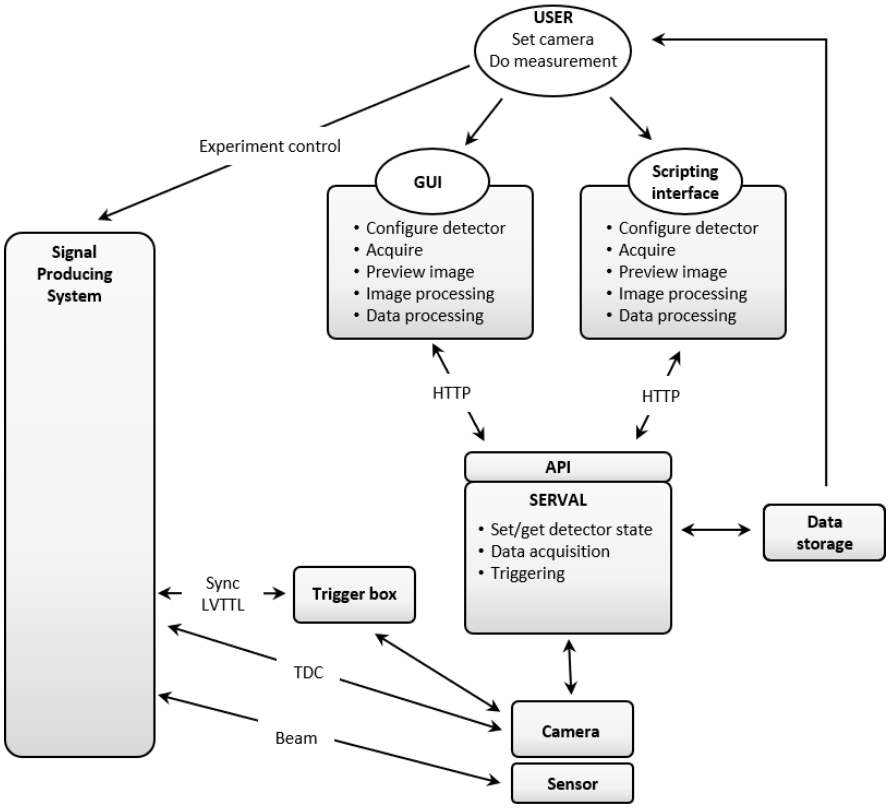


Figure 3. Block diagram for software integration with SERVAL.

2. Installing SERVAL

2.1 Requirements

We recommend using SERVAL on ASI delivered acquisition computers.

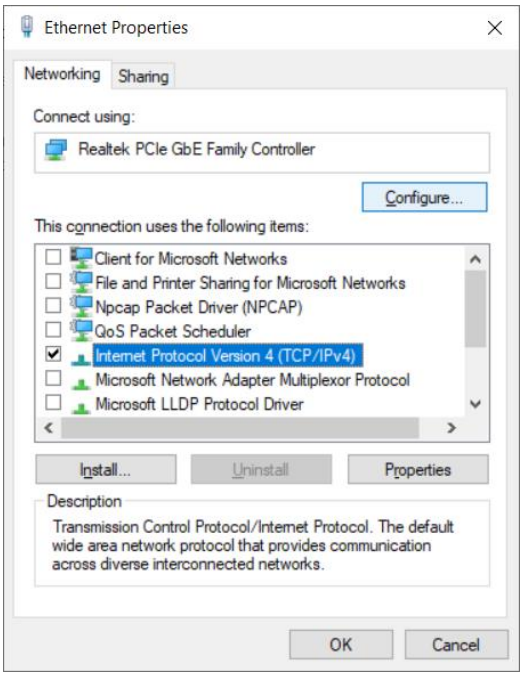
2.1.1 Operating System requirements

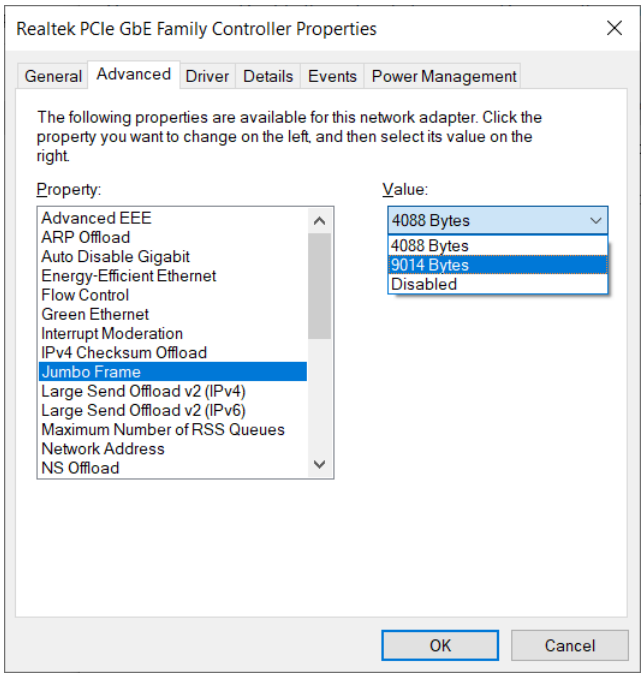
On ASI workstation computers Ubuntu Linux 18.04 LTS is installed.

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.
For Windows systems, make sure that 1. firewalls are switched off, 2. all other network connections are disconnected before starting SERVAL (close SoPhy if it is running), 3. the Jumbo frame value is set correctly (to 9014 Bytes). See below.





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

When connecting to a detector, one needs to load the factory setting files. SERVAL automatically checks which cameras are available.

3. Using SERVAL

3.1 Running SERVAL

SERVAL can be **started** by executing the following command:

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

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 response class (https://requests.readthedocs.io):

```
serverurl = 'http://192.168.x.x:8080'
response = requests.get(serverurl)
```

For a good connection, the response status code will be 200. More details are in table 4.1, §4.2.
The port number 8080 is used to communicate with SERVAL.

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

```
def tpx3_dashboard(serverurl):
    resp = requests.get(url = serverurl + '/dashboard')
    return json.loads(resp.text)
```

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.

All examples were written for Python 3.7.3.
In Python we can do this as follows, see tpx3_example_02.py:

```
def tpx3_cam_init(serverurl, detector_config, bpc_file, dacs_file):
     # load a BPC exported by SoPhy, the file should exist on the server
    resp = requests.get(url=serverurl + '/config/load?format=pixelconfig&file=' +
bpc_file)
```

```
    data = resp.text
    print('Response of loading binary pixel configuration file: ' + data)

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

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


# Main part
# Locate the files on the server:
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
resp = requests.get(url=serverurl + '/detector/config')
data = resp.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
tpx3_cam_init(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.

3.4 Example: start measurement

The example below demonstrates how to set up (`/detector/config`), start (`/measurement/start`) and stop (`/measurement/stop`) a measurement. For a full example, see tpx3_example_03.py.

```
def tpx3_acq_init(serverurl, detector_config, ntrig=1, shutter_open_ms=490,
shutter_closed_ms=10):
    # Sets the number of triggers.
    detector_config["nTriggers"] = ntrig

    # Set the trigger mode to be software-defined.
    detector_config["TriggerMode"] = "AUTOTRIGSTART_TIMERSTOP"

    # Sets the trigger period (time between triggers) in seconds.
    detector_config["TriggerPeriod"] = (shutter_open_ms + shutter_closed_ms) / 1000

    # Sets the exposure time (time the shutter remains open) in seconds.
    detector_config["ExposureTime"] = shutter_open_ms / 1000

    # Upload the Detector Configuration defined above
    resp = requests.put(url=serverurl + '/detector/config',
data=json.dumps(detector_config))
    data = resp.text
```

```
    print('Response of updating Detector Configuration: ' + data)


def tpx3_acq_test(serverurl):
    # Starting acquisition process
    resp = requests.get(url=serverurl + '/measurement/start')
    data = resp.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 the TimeLeft is smaller than 3 seconds, and we are
not still preparing.
        if dashboard["Measurement"]["Status"] != "DA_PREPARING" and
dashboard["Measurement"]["TimeLeft"] < 3.0:
            taking_data = False

            # Send stop
            resp = requests.get(url=serverurl + '/measurement/stop')
            data = resp.text
            print('Acquisition was stopped with response: ' + data)
```

3.5 Example: collect previews and save measurement data

For this we should set the data destinations: `/server/destination` and obtain the real time preview with `/measurement/image`.

Before we can retrieve the data and store it into a python Image object, we must first set up the data destination, see tpx3_example_04.py. An example of a destination JSON object and how to set it for obtaining previews and to save the raw data (raw binary list or event data (.tpx)) is shown below.

```
# 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": "png",
            # What data to build a frame from (tot, toa, tof, count)
            "Mode": "tot"
        }]
    }
}

# Main part
resp = requests.put(url = serverurl + '/destination', data =
json.dumps(destination))
resp = requests.get(url = serverurl + '/destination')
print('Response : ' + resp.text)
```

Here the preview data is made available through HTTP on the server and only the raw data is saved in files. We selected the PNG image format and the mode is ToT (Time over Threshold). Other preview modes are Count, ToA (Time of Arrival) and ToF (Time of Flight).

Now that we set up the destination to queue the images on the image HTTP end-point, we can retrieve the data and convert to a python image using the following example code:

```
def tpx3_preview(serverurl, ntrig=1):
    for i in range(ntrig):
    # Getting preview data. This is blocking, so it will wait until an image is
ready.
    resp = requests.get(url=serverurl + '/measurement/image')
    image = Image.open(BytesIO(resp.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
# Setting triggers and timing for the acquisition
numTriggers = 5
tpx3_acq_init(serverurl, detectorConfig, numTriggers, 990, 10)

# Running acquisition process and preview
tpx3_simple_acq(serverurl)
tpx3_preview(serverurl, numTriggers)
```

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 have the form:

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

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

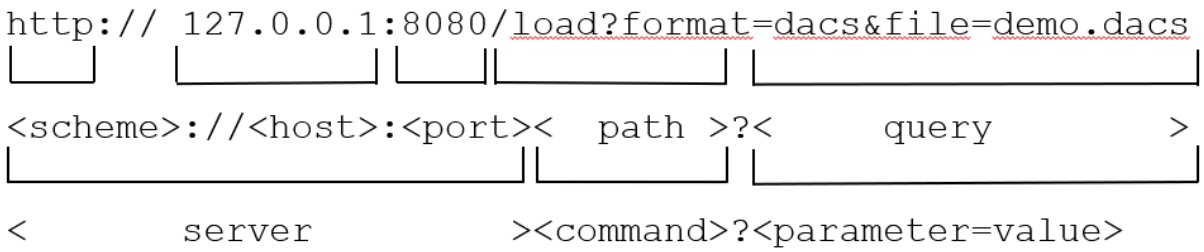


Figure 4.1: request structure

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 in 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 controls the way SERVAL for example writes captured image files to disk or provides 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: In this version the mainly the destination is provided.
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 a 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
2. Format: for the file-based 2D images, tiff, png, pgm are supported. For the HTTP / TCP scheme, pgm, jsonimage and jsonhisto are supported, providing a dedicated easy to decode format.

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

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 `Mode` should be the same. Raw 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:

```
destination = {
    "Raw" : [ {
      "Base" : "file:/data/raw",
      "FilePattern" : "f%Hms_"
    } ],
    "Image" : [ {
      "Base" : "file:/data/image",
      "FilePattern" : "f%Hms_",
      "Format" : "tiff",
      "Mode" : "tot"
    } ],
    "Preview" : {
      "Period" : 0.1,
      "SamplingMode" : "skipOnFrame",
      "ImageChannels" : [ {
        "Base" : "tcp://127.0.0.1:8088",
        "Format" : "jsonimage",
        "Mode" : "tot"
      }, {
        "Base" : "tcp://127.0.0.1:8089",
        "Format" : "jsonimage",
        "Mode" : "tot",
        "IntegrationSize" : -1,
        "IntegrationMode" : "last"
      } ],
      "HistogramChannels" : [ ]
  }
```

Parameter	Value	Description
Raw	Array of {Base, FilePattern} objects. See table 4.6.	Defines the raw output. Currently only a single raw output entry is supported.
Image	Array of OutputChannel objects. See table 4.6	Defines the image output.
Preview	Preview object. See table 4.5.	Defines the preview output.

Table 4.2: destination top level parameters

Parameter	Value	Description
Preview/Period	Float in [seconds]	Period of the preview refresh rate
Preview/SamplingMode	[“skipOnFrame”, “skipOnPeriod”]	Skip frames based on the number of frames, or on duration (1/period).
Preview/ImageChannels	Array of OutputChannel objects. See table 4.6	
Preview/HistogramChannels	Array of OutputChannel objects. See table 4.6	

Table 4.3: destination Preview parameters

Parameter	values	description
Base	URI Examples: file:/data http://localhost:8082 tcp://localhost:8088	Send data to a file path or socket like tcp://localhost:8088
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.
Format	String: tiff, pgm, png, jsonimage, jsonhisto	See below for format description.
Mode	count, tot, toa, tof	COUNT: count of hits above threshold TOT: Time over Threshold TOA: Time of Arrival TOF: Time of Flight
IntegrationSize	Integer: -1, 0, 1 ... 32	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	Sum: sum of images clipped at 16 bits (for each pixel). Average: arithmetic mean of images clipped at 16 bits (for each pixel). Last: any non-zero value of the new image will overwrite a pre-existing value.

Table 4.4: OutputChannel object parameters

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 bitdepth 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 and jsonhisto consist of a JSON header followed by image or histogram data. This header contains specific information of each frame sent during measurement.

Jsonimage header:

```
Header =
{
    'timeAtFrame': 3.5,
    'frameNumber': 35,
    'measurementID': None,
    'dataSize': 524288,
    'bitDepth': 16,
    'width': 512,
    'height': 512
}
```

Jsonhisto header:

```
Header =
{
    'timeAtFrame': 38.0,
    'frameNumber': 5,
    'measurementID': None,
    'dataSize': 2000,
    'bitDepth': 32,
    'binSize': 500,
    'binWidth': 20,
    'binOffset': 0
}
```

Parameter	Values	description
timeAtFrame	Float in seconds	Elapsed time from the start of the measurement when the frame was taken.
frameNumber	Integer	Frame number. Zero at the start of the measurement
measurementID	String	Context of the measurement. Can optionally be set per measurement. [Note: not implemented yet]
dataSize	Integer in bytes	Size of the image or histogram data
bitDepth	Integer	Number of bits used per pixel or item in the histogram array.
width	Integer	Width of the image
height	Integer	Height of the image
binSize	Integer	Histogram: Number of bins, and also the number of items of the array.
binWidth	Integer in steps of 1.5625 nanoseconds (tpx3 native pixel time resolution)	Histogram: width of the bins.
binOffset	Integer	Histogram: offset, position of the first bin

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.

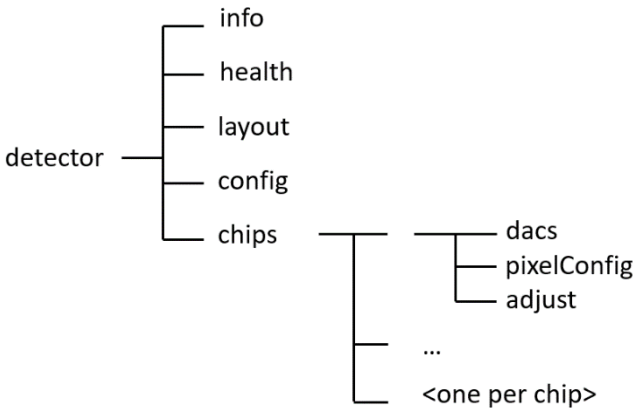


Figure 4.2: JSON tree

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 a orientation transform to the images as defined by the destination.

Syntax:

```
GET <server>/detector/layout/rotation?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

```
Info =
{
  "IfaceName" : "Spidr",
  "SW_version" : "19081915",
  "FW_version" : "18052510",
  "PixCount" : 262144,
  "RowLen" : 512,
  "NumberOfChips" : 4,
  "NumberOfRows" : 2,
  "MpxType" : 6,
  "Boards" : [ {
    "ChipboardId" : "41000039",
    "IpAddress" : "127.0.0.10",
    "PortNumber" : 8192,
    "Chips" : [ {
       “Id”: 0,
       “Name”: "W0002_H10”
     },{
       “Id”: 1,
       “Name”: “W0002_I10”
     },{
       “Id”: 2,
       “Name”: “W0002_J10”
     },{
       “Id”: 3,
       “Name”: “W0002_K10"
     } ]
  } ],
  "MaxPulseCount" : 2147483647,
  "MaxPulseHeight" : 1.0,
  "MaxPulsePeriod" : 34.35973836,
  "TimerMaxVal" : 34.35973836,
  "TimerMinVal" : 8.0E-9,
  "TimerStep" : 8.0E-9
}
```

Parameter	Value	Description
IfaceName	string	Interface name
SW_version	string	Readout Software version
FW_version	string	Readout Firmware version
PixCount	integer	Sum of number of pixels of all chips
RowLen	integer	Row length
NumberOfChips	integer	Number of chips
NumberOfRows	integer	Number of rows of whole chip layout
MpxType	integer	Medipix type
Boards	array	
ChipboardId	string	
IpAddress	string	
PortNumber	integer	
MaxPulseCount	integer	
MaxPulseHeight	float	
MaxPulsePeriod	float	
Chips	array	
Id	integer	Chip ID
Name	string	Chip name (from wafer and position)
TimerMaxVal	float	
TimerMinVal	float	
TimerStep	float	

Table 4.6: detector info parameters

4.5.5 Detector Health JSON structure

```
Health =
{
      "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
}
```

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

Table 4.7: Detector health parameters

4.5.6 Detector Layout JSON structure

```
"Chips": [ {
    "Orientation": "LtRBtT",
    "X": 1,
    "Y": 0
  }, {
    "Orientation": "RtLTtB",
    "X": 1,
    "Y": 1
  }, {
    "Orientation": "RtLTtB",
    "X": 0,
    "Y": 1
  }, {
    "Orientation": "LtRBtT",
    "X": 0,
    "Y": 0
  } ]
```

Parameter	Value	Description
Orientation	String, 8 possible values LtRBtT, RtLBtT, LtRTtB, RtLTtB, BtTLtR, TtBLtR, BtTRtL, TtBRtL	All 8 possible rotation and mirroring combinations. LtR: Left to Right BtT: Bottom to Top
X	integer	X position of the chip
Y	integer	Y position of the chip

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.

```
Config =
{
  "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" ]
}
```

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	
ChainMode	NONE LEADER FOLLOWER	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 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 (1-3/4-6)
TriggerOut	[1, 6]	Sets the HDMI channel numbers (1-3/4-6)
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	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

	AUTOTRIGSTART_TIMERSTOP CONTINUOUS	acq. is started by trigger from HW, stopped by HW timer acq. is started by software, stopped by software Note: this setting controls the shutter behavior. For a more detailed description, consult the trigger manual.
nTriggers	[0, max] integer	Number of triggers
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.

Table 4.9: Detector config parameters

4.5.8 DACs JSON structure

```
DACs = {
  "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	description
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

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	Info: combined measurement.
GET	`/measurement/image`	Returns image Data: image	Info: the image as specified as 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.

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

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
```

4.7 /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 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

```
dashboard =
{
  "Server" : {
    "Notifications" : [ ],
    "SoftwareVersion" : "2.0",
    "SoftwareTimestamp" : "2020/11/23 10:53"
  },
  "Measurement" : {
    "StartDateTime" : 1606126865448,
    "TimeLeft" : 0,
    "ElapsedTime" : 1.656,
    "FrameCount" : 100,
    "PixelEventRate" : 1455120,
    "Status" : "DA_IDLE"
  },
  "Detector" : {
    "DetectorType" : "Tpx3"
  }
}
```

Parameter	Values	description
Notifications	Array of notifications (TBD)	Notifications (Will be added later)
SoftwareVersion	String	Software Version
SoftwareTimestamp	String	Timestamp of the software build
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
PixelEventRate	Integer	Total number of observed pixel hits per second.
Status	Enum: [DA_IDLE, DA_PREPARING, DA_RECORDING, DA_STOPPING]	Indicating data acquisition state: : idle : 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 4.11: dashboard 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 header prepended.
The chunk header consists of:

- 4 bytes indicating the type: “TPX3”;
- 1 byte indicating the chip index;
- 1 byte reserved;
- 2 bytes containing the net size of the chunk.

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):

1. Pixel data with type 0xb, the maximum timestamp is 26.8435456 s;
2. TDC data with type 0x6, the maximum timestamp is 107.3741824 s;
3. Global time with type 0x4, the maximum is ~81 days;
4. Control indications 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. TDC data packet

63 - 56 bit	55 - 44 bit	43 - 9 bit	8 - 5 bit	4 – 0 bit
0x6	Trigger counter	Timestamp	Stamp	Reserved

Table 6.2. Pixel data packet

63 - 60 bit	59 - 44 bit	43 - 30 bit	29 - 20 bit	19 – 16 bit	15 – 0 bit
0xb	PixAddr	ToA	ToT	FToA	SPIDR time

6.2 Image File

In SERVAL, image files can be saved as .tiff format. It is 256 x 256 pixels and 32-bit range. It is possible to save .tiff file and raw data file 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

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 generates 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.