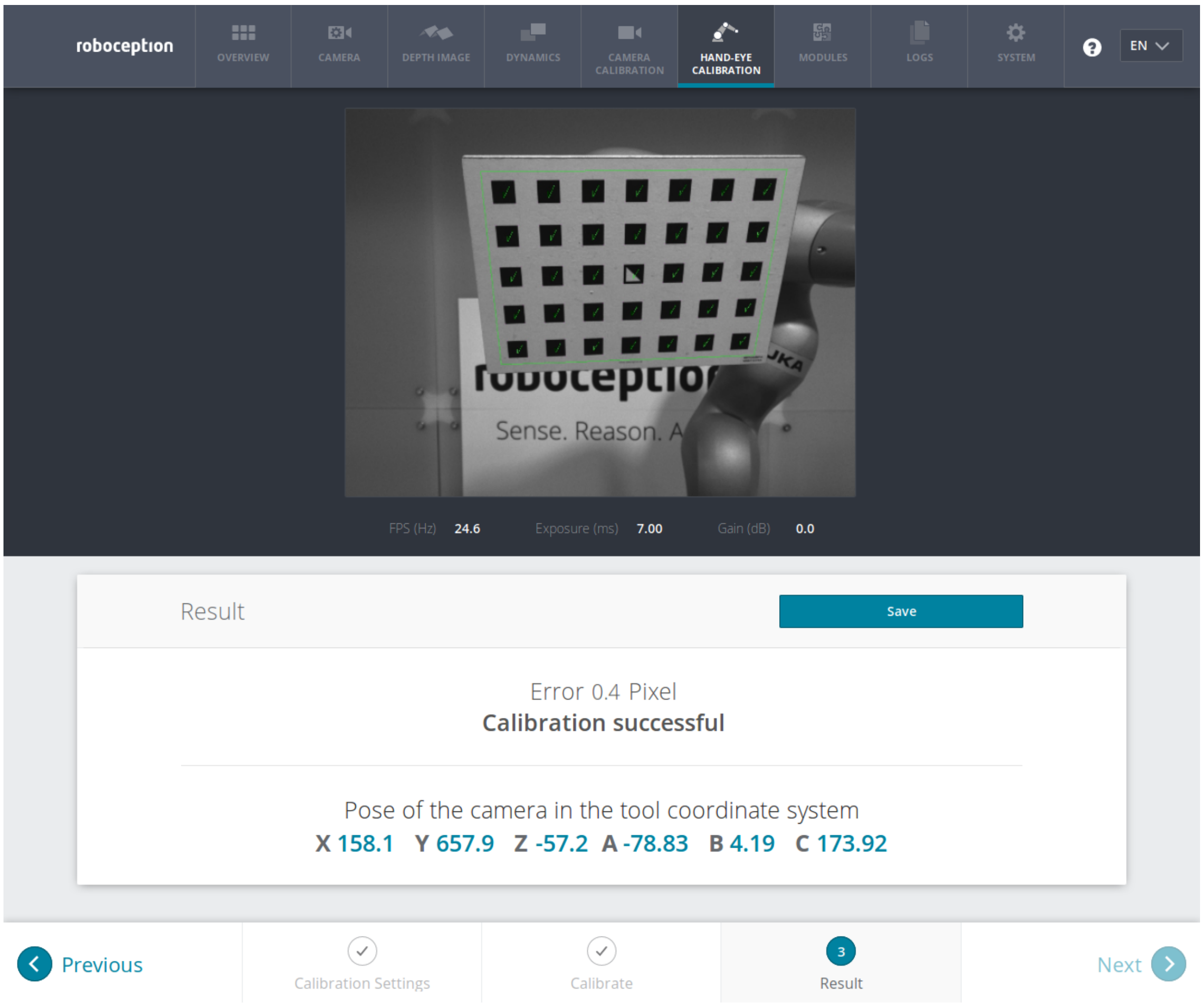


FPS (Hz) 24.6 Exposure (ms) 7.00 Gain (dB) 0.0

Error 0.4 Pixel
Calibration successful

Pose of the camera in the tool coordinate system
X 158.1 Y 657.9 Z -57.2 A -78.83 B 4.19 C 173.92

Fig. 6.7.9: Result of the hand-eye calibration process displayed in the Web GUI

6.7.4 Parameters

The hand-eye calibration component is called `rc_hand_eye_calibration` in the REST-API and is represented by the *Hand-Eye Calibration* tab in the *Web GUI* (Section 4.5). The user can change the calibration parameters there or use the *REST-API interface* (Section 8.2).

Parameter overview

This component offers the following run-time parameters.

Table 6.7.1: The `rc_hand_eye_calibration` component's run-time parameters

Name	Type	Min	Max	Default	Description
`grid_height`	float64	0.0	10.0	0.0	The height of the calibration pattern in meters
`grid_width`	float64	0.0	10.0	0.0	The width of the calibration pattern in meters
`robot_mounted`	bool	False	True	True	Whether the camera is mounted on the robot

This component reports no status values.

Description of run-time parameters

The parameter descriptions are given with the corresponding Web GUI names in brackets.

`grid_width` (***Grid Width (m)***) Width of the calibration grid in meters. The width should be measured with a very great accuracy, preferably with sub-millimeter accuracy.

`grid_height` (***Grid Height (m)***) Height of the calibration grid in meters. The height should be measured with a very great accuracy, preferably with sub-millimeter accuracy.

`robot_mounted` (***Sensor Mounting***) If set to 1, the *rc_visard* is mounted on the robot. If set to 0, the *rc_visard* is mounted statically and the calibration grid is mounted on the robot.

(***Pose***) For convenience, the user can choose in the Web GUI between calibration in *XYZABC* format or in *XYZ+quaternion* format (see *Pose formats*, Section 13.1). When calibrating using the REST-API, the calibration result will always be given in *XYZ+quaternion*.

6.7.5 Services

The REST-API service calls offered to programmatically conduct the hand-eye calibration and to store or restore this component's parameters are explained below.

`save_parameters` With this service call, the current parameter settings of the hand-eye calibration component are persisted to the *rc_visard*. That is, these values are applied even after reboot.

This service requires no arguments.

This service returns no response.

`reset_defaults` restores and applies the default values for this component's parameters ("factory reset"). Does not affect the calibration result itself or any of the `slots` saved during calibration. Only parameters such as the grid dimensions and the mount type will be reset.

> **Warning:** The user must be aware that calling this service causes the current parameter settings to be irrecoverably lost.

This service requires no arguments.

This service returns no response.

`set_pose` provides a robot pose as calibration pose to the hand-eye calibration routine.

This service requires the following arguments:

```
{
  "pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  },
  "slot": "int32"
}
```

This service returns the following response:

```
{
  "message": "string",
  "status": "int32",
  "success": "bool"
}
```

The `slot` argument is used to assign numbers to the different calibration poses. At each instant when `set_pose` is called, an image is recorded. This service call fails if the grid was undetectable in the current image.

Table 6.7.2: Return codes of the `set_pose` service call

status	success	Description
1	`true`	pose stored successfully
3	`true`	pose stored successfully; collected enough poses for calibration, i.e., ready to calibrate
4	`false`	calibration grid was not detected, e.g., not fully visible in camera image
8	`false`	no image data available
12	`false`	given orientation values are invalid

`reset_calibration` deletes all previously provided poses and corresponding images. The last saved calibration result is reloaded. This service might be used to (re-)start the hand-eye calibration from scratch.

This service requires no arguments.

This service returns the following response:

```
{
  "message": "string",
  "status": "int32",
  "success": "bool"
}
```

`calibrate` calculates and returns the hand-eye calibration transformation with the robot poses configured by the `set_pose` service.

Note: For calculating the hand-eye calibration transformation at least three robot calibration poses are required (see `set_pose` service). However, four calibration poses are recommended.

This service requires no arguments.

This service returns the following response:

```
{
  "error": "float64",
  "message": "string",
  "pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  },
  "robot_mounted": "bool",
  "status": "int32",
```

(continues on next page)

(continued from previous page)

```
  "success": "bool"
}
```

Table 6.7.3: Return codes of the `calibrate` service call

status	success	Description
0	`true`	calibration successful; returned resulting calibration pose
1	`false`	not enough poses to perform calibration
2	`false`	calibration result is invalid, please verify the input data
3	`false`	given calibration grid dimensions are not valid

`save_calibration` persistently saves the result of hand-eye calibration to the *rc_visard* and overwrites the existing one. The stored result can be retrieved any time by the `get_calibration` service.

This service requires no arguments.

This service returns the following response:

```
{
  "message": "string",
  "status": "int32",
  "success": "bool"
}
```

Table 6.7.4: Return codes of the `save_calibration` service call

status	success	Description
0	`true`	calibration saved successfully
1	`false`	could not save calibration file
2	`false`	calibration result is not available

`remove_calibration` removes the persistent hand-eye calibration on the *rc_visard*. After this call the `get_calibration` service reports again that no hand-eye calibration is available.

This service requires no arguments.

This service returns the following response:

```
{
  "message": "string",
  "status": "int32",
  "success": "bool"
}
```

Table 6.7.5: Return codes of the `get_calibration` service call

status	success	Description
0	`true`	removed persistent calibration, sensor reports as uncalibrated
1	`true`	no persistent calibration found, sensor reports as uncalibrated
2	`false`	could not remove persistent calibration

`get_calibration` returns the hand-eye calibration currently stored on the *rc_visard*.

This service requires no arguments.

This service returns the following response:

```
{
  "error": "float64",
  "message": "string",
```

(continues on next page)

(continued from previous page)

```
  "pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  },
  "robot_mounted": "bool",
  "status": "int32",
  "success": "bool"
}
```

Table 6.7.6: Return codes of the `get_calibration` service call

`status`	`success`	Description
0	`true`	returned valid calibration pose
2	`false`	calibration result is not available

7 Optional software components

The *rc_visard* offers optional software components that can be activated by purchasing a separate *license* (Section 9.6).

The *rc_visard*'s optional software consists of the following components:

- ***SLAM*** **(`rc_slam`, Section 7.1)** performs simultaneous localization and mapping for correcting accumulated poses. The *rc_visard*'s covered trajectory is offered via the *REST-API interface* (Section 8.2).
- ***IO and Projector Control*** **(`rc_iocontrol`, Section 7.2)** provides control over the general purpose inputs and outputs with special modes for controlling an external random dot projector.
- ***TagDetect*** **(`rc_april_tag_detect` and `rc_qr_code_detect`, Section 7.3)** allows the detection of AprilTags and QR codes, as well as the estimation of their poses.
- ***ItemPick and BoxPick*** **(`rc_itempick` and `rc_boxpick`, Section 7.4)** provides an out-of-the-box perception solution for robotic pick-and-place applications of unknown objects or boxes.
- ***SilhouetteMatch*** **(`rc_silhouettematch`, Section 7.5)** provides an object detection solution for objects placed on a plane.

7.1 SLAM

The SLAM component is part of the sensor dynamics component. It provides additional accuracy for the pose estimate of the stereo INS. When the *rc_visard* moves through the world, the pose estimate slowly accumulates errors over time. The SLAM component can correct these pose errors by recognizing previously visited places.

The acronym SLAM stands for Simultaneous Localization and Mapping. The SLAM component creates a map consisting of the image features as used in the visual odometry component. The map is later used to correct accumulated pose errors. This is most apparent in applications where, e.g., a robot returns to a previously visited place after covering a large distance (this is called a *loop closure*). In this case, the robot can re-detect image features that are already stored in its map and can use this information to correct the drift in the pose estimate that accumulated since the last visit.

When closing a loop, not only the current pose, but also the past pose estimates (the trajectory of the *rc_visard*), are corrected. Continuous trajectory correction leads to a more accurate map. On the other hand, the accuracy of the full trajectory is important when it is used to build an integrated world model, e.g., by projecting the 3D point clouds obtained (see *Computing depth images and point clouds*, Section 6.2.2) into a common coordinate frame. The full trajectory can be requested from the SLAM component for this purpose.

Note: The SLAM component is optionally available for the *rc_visard* and will run on board the sensor. If a SLAM license is stored on the *rc_visard*, then the SLAM component is shown as *Available* on the *Web GUI*'s *Overview* page and in the *License* section of the *System* page.

7.1.1 Usage

The SLAM component can be activated at any time, either via the rc_dynamics interface (see the documentation of the respective *Services*, Section 6.3.3) or from the *Dynamics* page of the *Web GUI*.

The pose estimate of the SLAM component will be initialized with the current estimate of the stereo INS - and thus the origin will be where the stereo INS was started.

Since the SLAM component builds on the motion estimates of the stereo INS component, the latter will automatically be started up if it is not yet running when SLAM is started.

When the SLAM component is running, the corrected pose estimates will be available via the datastreams *pose*, *pose_rt*, and *dynamics* of the rc_dynamics component.

The full trajectory is available through the service `get_trajectory`, see *Services* (Section 7.1.4) below for details.

To store the feature map on the *rc_visard*, the SLAM component provides the service `save_map`, which can be used only during runtime (state "RUNNING") or after stopping (state "HALTED"). A stored map can be loaded before startup using the service `load_map`, which is only applicable in state "IDLE" (use the `reset` service to go back to "IDLE" when SLAM is in state "HALTED").

Note that mistaken localization at (visually) similar places may happen more easily when initially localizing in a loaded map than when localizing during continuous operation. Choosing a starting point with a unique visual appearance avoids this problem.

The SLAM component will therefore assume that the *rc_visard* is started in the rough vincinity (a few meters) of the origin of the map. The origin of the map is where the Stereo INS module was started when the map was recorded.

7.1.2 Memory limitations

In contrast to the other software components running on the *rc_visard*, the SLAM component needs to accumulate data over time, e.g., motion measurements and image features. Further, the optimization of the trajectory requires substantial amounts of memory, particularly when closing large loops. Therefore the memory requirements of the SLAM component increase over time.

Given the memory limitations of the hardware, the SLAM component needs to reduce its own memory footprint when running continuously. When the available memory runs low, the SLAM component will fix parts of the trajectory, i.e. no further optimization will be done on these parts. A minimum of 10 minutes of the trajectory will be kept unfixed at all times.

When the available memory runs low despite the above measures, two options are available. The first option is that the SLAM component automatically goes to the HALTED state, where it stops processing, but the trajectory (up to the stopping time) is still available. This is the default behavior.

The second option is to keep running until the memory is exhausted. In that case, the SLAM component will be restarted. If the `autorecovery` parameter is set to true, the SLAM component will recover its previous position and resume mapping. Otherwise it will go to FATAL state, requiring to be restarted via the rc_dynamics interface (see *Services*, Section 6.3.3).

The operation time until the memory limit is reached is strongly dependent on the trajectory of the sensor.

Warning: Because of the memory limitations, it is not recommended to run SLAM at the same time as *Stereo matching* in full resolution, because the memory available to SLAM will be greatly reduced. In the worst case, a long running SLAM process may even be forcefully reset, when full-resolution stereo matching is turned on.

7.1.3 Parameters

The SLAM component is called `rc_slam` in the REST-API. The user can change the SLAM parameters using the *REST-API interface*.

Parameter overview

This component offers the following run-time parameters.

Table 7.1.1: The `rc_slam` component's run-time parameters

Name	Type	Min	Max	Default	Description
`autorecovery`	bool	False	True	True	In case of fatal errors recover corrected position and restart mapping
`halt_on_low_memory`	bool	False	True	True	When the memory runs low, go to halted state

This component reports the following status values.

Table 7.1.2: The `rc_slam` component's status values

Name	Description
`map_frames`	Number of frames that constitute the map
`state`	The current state of the rc_slam node
`trajectory_poses`	Number of poses in the estimated trajectory

The reported `state` can take one of the following values.

Table 7.1.3: Possible states of the `rc_slam` component

State name	Description
IDLE	The component is ready, but idle. No trajectory data is available.
WAITING_FOR_DATA	The component was started but is waiting for data from stereo INS or VO.
RUNNING	The component is running.
HALTED	The component is stopped. The trajectory data is still available. No new information is processed.
RESETTING	The component is being stopped and the internal data is being cleared.
RESTARTING	The component is being restarted.
FATAL	A fatal error has occured.

7.1.4 Services

The SLAM component offers the following services.

Note: Activation and deactivation of the SLAM component is done via the service interface of rc_dynamics (see *Services*, Section 6.3.3).

`reset` clears the internal state of the SLAM component. This service is to be used after stopping the SLAM component using the rc_dynamics interface (see the respective *Services*, Section 6.3.3). The SLAM component maintains the estimate of the full trajectory even when stopped. This service clears this estimate and frees the respective memory. The returned status is RESETTING.

This service requires no arguments.

This service returns the following response:

```
{
  "accepted": "bool",
  "current_state": "string"
}
```

`get_trajectory` With this service call the trajectory can be retrieved. The service arguments allow to select a subsection of the trajectory by defining a `start_time` and an `end_time`. Both are optional, i.e., they could be left empty or filled with zero values, which results in the subsection to include the trajectory from the very beginning, or to the very end, respectively, or both. If not empty or zero, they can be defined either as absolute timestamps or to be relative to the trajectory (`start_time_relative` and `end_time_relative` flags). If defined to be relative, the values' signs indicate to which point in time they relate to: Positive values define an offset to the start time of the trajectory; negative values are interpreted as an offset from the end time of the trajectory. The below diagram illustrates three examples for the relative parameterization.

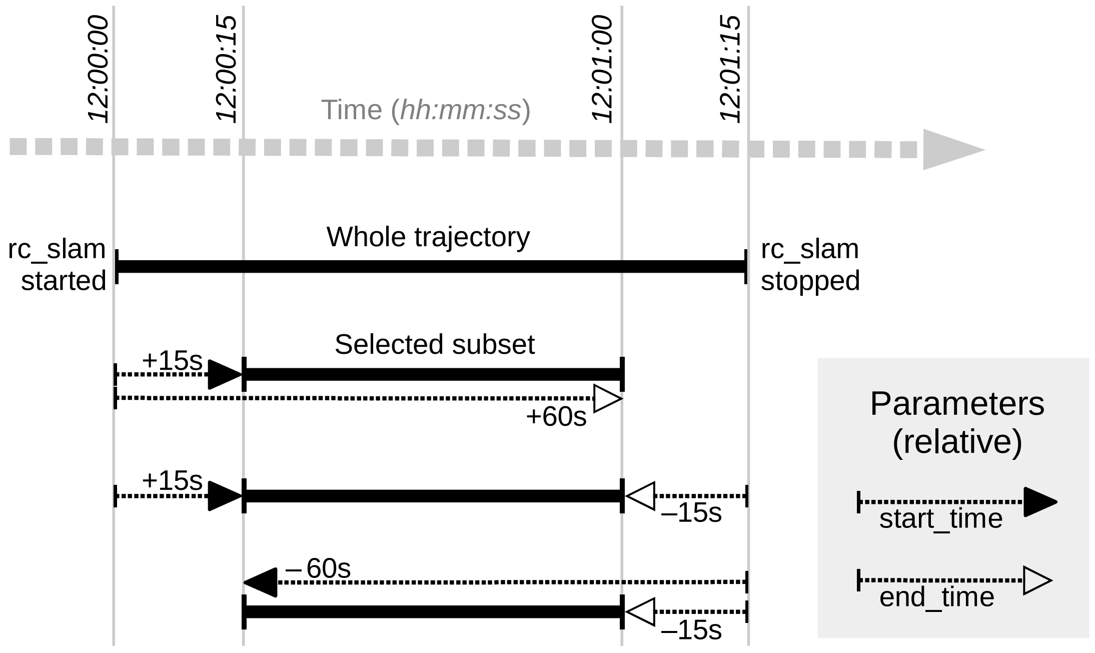


Fig. 7.1.1: Examples for combinations of relative start and end times for the `get_trajectory` service. All combinations shown select the same subset of the trajectory.

Note: A relative `start_time` of zero will select everything from the start of the trajectory, whereas a relative `end_time` of zero will select everything to the end of the trajectory. Absolute zero values effectively do the same, so one can set all values zero to get the full trajectory.

The field `return_code` in the answer contains a status value and message. The value will be zero or greater on success, where zero is the regular case and positive values indicate a special condition. Negative values signal an error.

The field `producer` indicates where the trajectory data comes from and is always `slam`.

The following table contains a list of common codes:

Code	Description
0	Success
-1	An invalid argument was provided (e.g., an invalid time range)
101	Trajectory is empty, because there is no data in the given time range
102	Trajectory is empty, because there is no data at all (e.g., when SLAM is IDLE)

This service requires the following arguments:

```
{
  "end_time": {
    "nsec": "int32",
    "sec": "int32"
  },
  "end_time_relative": "bool",
  "start_time": {
    "nsec": "int32",
    "sec": "int32"
  },
  "start_time_relative": "bool"
}
```

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "trajectory": {
    "name": "string",
    "parent": "string",
    "poses": [
      {
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "timestamp": {
          "nsec": "int32",
          "sec": "int32"
        }
      }
    ],
    "producer": "string",
    "timestamp": {
      "nsec": "int32",
      "sec": "int32"
    }
  }
}
```

`save_map` Stores the current state as a map to persistent memory. The map consists of a set of fixed map frames. It does not contain the full trajectory that has been covered.

Note: Only abstract feature positions and descriptions are stored in the map. No actual footage of the cameras is stored with the map, nor is it possible to reconstruct images or image parts from the stored information.

Warning: The map is lost on software updates or rollbacks

This service requires no arguments.

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`load_map` Loads a previously saved map. This is only applicable when the SLAM component is IDLE. It is not possible to load a map into a running system. A loaded map can be cleared with the `reset` service call.

This service requires no arguments.

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

remove_map Removes the stored map from the persistent memory.

This service requires no arguments.

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

7.2 IO and Projector Control

The IOControl component allows reading the status of the general purpose digital inputs and controlling the digital general purpose outputs (*Wiring*, Section 3.5). The outputs can be set to low or high, or configured to be high for the exposure time of every image. Additionally, the outputs can be configured to be high for the exposure time of every second image.

Note: The IOControl component is optional and requires the purchase of a separate *license* (Section 9.6).

7.2.1 Parameters

The IOControl component is called `rc_iocontrol` in the REST-API. The user can change the parameters via REST-API (*REST-API interface*, Section 8.2) or GigE Vision using the DigitalIOControl parameters `LineSelector` and `LineSource` (*Category: DigitalIOControl*, Section 8.1.1).

Parameter overview

This component offers the following run-time parameters.

Table 7.2.1: The `rc_iocontrol` component's run-time parameters

Name	Type	Min	Max	Default	Description
`out1_mode`	string	-	-	ExposureActive	Low, High, ExposureActive, ExposureAlternateActive
`out2_mode`	string	-	-	Low	Low, High, ExposureActive, ExposureAlternateActive

This component reports no status values.

Description of run-time parameters

out1_mode and out2_mode (*Out1* and *Out2*) The output modes for GPIO Out 1 and Out 2 can be set individually:

`Low` sets the ouput permanently to low. This is the factory default of Out 2.

`High` sets the output permanently to high.

`ExposureActive` sets the output to high for the exposure time of every image. This is the factory default of Out 1.

`ExposureAlternateActive` sets the output to high for the exposure time of every second image.

Note: The parameters can only be changed if the IOControl license is available on the *rc_visard*. Otherwise, the parameters will stay at their factory defaults, i.e. `out1_mode` = `ExposureActive` and `out2_mode` = `Low`.

Figure Fig. 7.2.1 shows which images are used for stereo matching and transmission via GigE Vision in `ExposureActive` mode with a user defined frame rate of 8 Hz.

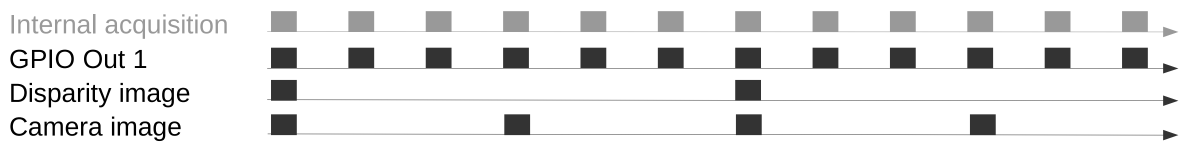


Fig. 7.2.1: Example of using the `ExposureActive` mode for GPIO Out 1 with a user defined frame rate setting of 8 Hz. The internal image acquisition is always 25 Hz. GPIO Out 1 is high for the exposure time of every image. A disparity image is computed for camera images that are send out via GigE Vision according to the user defined frame rate.

The mode `ExposureAlternateActive` is meant to be used when an external random dot projector is connected to the GPIO Out 1 of the *rc_visard*. A side effect of setting output 1 to `ExposureAlternateActive` is that the *stereo matching* (Section 6.2) component only uses images if output 1 is high, i.e. projector is on. The maximum framerate that is used for stereo matching is therefore halve of the frame rate configured by the user (see *FPS*, Section 6.1.3). All other components like *visual odometry* (Section 6.4), *TagDetect* (Section 7.3) and *ItemPick* (Section 7.4) use images where the output is low, i.e. projector is off. Figure Fig. 7.2.2 shows an example.

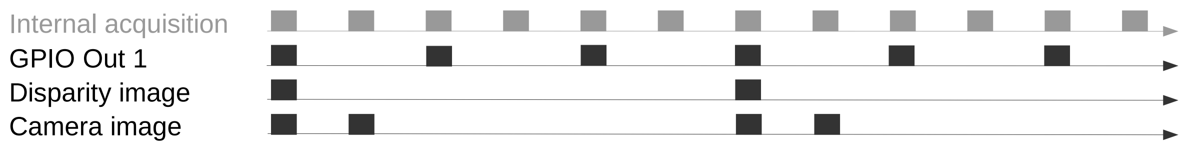


Fig. 7.2.2: Example of using the `ExposureAlternateActive` mode for GPIO Out 1 with a user defined frame rate setting of 8 Hz. The internal image acquisition is always 25 Hz. GPIO Out 1 is high for the exposure time of every second image. A disparity image is computed for images where Out 1 is high and that are send out via GigE Vision according to the user defined frame rate. In `ExposureAlternateActive` mode, images are always transmitted pairwise. One with Out 1 high for which a disparity image might be available and one with Out 1 low.

Note: In `ExposureAlternateActive` mode, an image with output high (i.e. with projection) is always 40 ms away from an image with output low (i.e. without projection), regardless of the user configured frame rate. This needs to be considered when synchronizing disparity and camera images without projection in this special mode.

The functionality can also be controlled by the digital IO control parameters of the GenICam interface (*Category: DigitalIOControl*, Section 8.1.1).

7.2.2 Services

The IOControl component offers the following services.

get_io_values This service call retrieves the current state of the general purpose inputs and outputs. The returned time stamp is the time of measurement. The call returns an error if the *rc_visard* does not have an IOControl license.

This service requires no arguments.

This service returns the following response:

```
{
  "in1": "bool",
  "in2": "bool",
  "out1": "bool",
  "out2": "bool",
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```

Possible return codes are shown below.

Table 7.2.2: Possible return codes of the `get_io_values` service call.

Code	Description
0	Success
-2	Internal error
-9	License for *iocontrol* is not available

save_parameters With this service call, the component's current parameter settings are persisted to the *rc_visard*. That is, these values are applied even after reboot.

This service requires no arguments.

This service returns no response.

reset_defaults Restores and applies the default values for this component's parameters ("factory reset").

This service requires no arguments.

This service returns no response.

Warning: The user must be aware that calling this service causes the current parameter settings for the IOControl component to be irrecoverably lost.

7.3 TagDetect

7.3.1 Introduction

The TagDetect components run on board the *rc_visard* and allow the detection of 2D bar codes and tags. Currently, there are TagDetect components for *QR codes* and *AprilTags*. The components furthermore compute the position and orientation of each tag in the 3D camera coordinate system, making it simple to manipulate a tag with a robot or to localize the camera with respect to a tag.

Note: The TagDetect components are optional and require a separate *license* (Section 9.6) to be purchased.

Tag detection is made up of three steps:

1. Tag reading on the 2D image pair (see *Tag reading*, Section 7.3.2).
2. Estimation of the pose of each tag (see *Pose estimation*, Section 7.3.3).
3. Re-identification of previously seen tags (see *Tag re-identification*, Section 7.3.4).

In the following, the two supported tag types are described, followed by a comparison.

QR code

Fig. 7.3.1: Sample QR code

QR codes are two-dimensional bar codes that contain arbitrary user-defined data. There is wide support for decoding of QR codes on commodity hardware such as smartphones. Also, many online and offline tools are available for the generation of such codes.

The "pixels" of a QR code are called *modules*. Appearance and resolution of QR codes change with the amount of data they contain. While the special patterns in the three corners are always 7 modules wide, the number of modules between them increases the more data is stored. The lowest-resolution QR code is of size 21x21 modules and can contain up to 152 bits.

Even though many QR code generation tools support generation of specially designed QR codes (e.g., containing a logo, having round corners, or having dots as modules), a reliable detection of these tags by the *rc_visard*'s TagDetect component is not guaranteed. The same holds for QR codes which contain characters that are not part of regular ASCII.

AprilTag

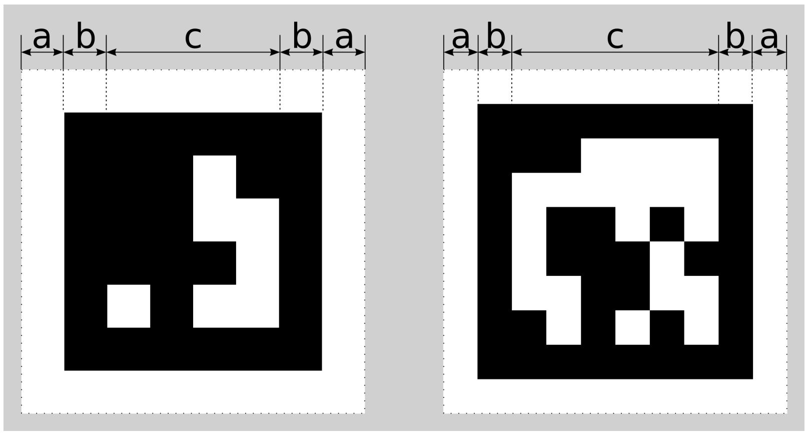


Fig. 7.3.2: A 16h5 tag (left) and a 36h11 tag (right). AprilTags consist of a mandatory white (a) and black (b) border and a variable amount of data bits (c).

AprilTags are similar to QR codes. However, they are specifically designed for robust identification at large distances. As for QR codes, we will call the tag pixels *modules*. Fig. 7.3.2 shows how AprilTags are structured. They are surrounded by a mandatory white and black border, each one module wide. In the center, they carry a variable amount of data modules. Other than QR codes, they do not contain any user-defined information but are identified by a predefined *family* and *ID*. The tags in Fig. 7.3.2 for example are of family 16h5 and 36h11 and have id 0 and 11, respectively. All supported families are shown in Table 7.3.1.

Table 7.3.1: AprilTag families

Family	Number of tag IDs	Recommended
16h5	30	-
25h7	242	-
25h9	35	o
36h10	2320	o
36h11	587	+

For each family, the number before the “h” states the number of data modules contained in the tag: While a 16h5 tag contains 16 (4x4) data modules ((c) in Fig. 7.3.2), a 36h11 tag contains 36 (6x6) modules. The number behind the “h” refers to the Hamming distance between two tags of the same family. The higher, the more robust is the detection, but the fewer individual tag IDs are available for the same number of data modules (see Table 7.3.1).

The advantage of fewer data modules (as for 16h5 compared to 36h11) is the lower resolution of the tag. Hence, each tag module is larger and the tag therefore can be detected from a larger distance. This, however, comes at a price: First, fewer data modules lead to fewer individual tag IDs. Second, and more importantly, detection robustness is significantly reduced due to a higher false positive rate; i.e, tags are mixed up or nonexistent tags are detected in random image texture or noise.

For these reasons we recommend using the 36h11 family and highly discourage the use of the 16h5 and 25h7 families. The latter families should only be used if a large detection distance really is necessary for an application. However, the maximum detection distance increases only by approximately 25% when using a 16h5 tag instead of a 36h11 tag.

Pre-generated AprilTags can be downloaded at the AprilTag project website (https://april.eecs.umich.edu/software/apriltag.html). There, each family consists of multiple PNGs containing single tags and one PDF containing each tag on a separate page. Each pixel in the PNGs corresponds to one AprilTag module. When printing the tags, special care must be taken to also include the white border around the tag that is contained in the PNGs as well as PDFs (see (a) in Fig. 7.3.2). Moreover, the tags should be scaled to the desired printing size without any interpolation, so that the sharp edges are preserved.

Comparison

Both QR codes and AprilTags have their up and down sides. While QR codes allow arbitrary user-defined data to be stored, AprilTags have a pre-defined and limited set of tags. On the other hand, AprilTags have a lower resolution and can therefore be detected at larger distances. Moreover, the continuous white to black edge around AprilTags allow for more precise pose estimation.

Note: If user-defined data is not required, AprilTags should be preferred over QR codes.

7.3.2 Tag reading

The first step in the tag detection pipeline is reading the tags on the 2D image pair. This step takes most of the processing time and its precision is crucial for the precision of the resulting tag pose. To control the speed of this step, the `quality` parameter can be set by the user. It results in a downscaling of the image pair before reading the tags. "H" (*High*) yields the largest maximum detection distance and highest precision, but also the highest processing time. "L" (*Low*) results in the smallest maximum detection distance and lowest precision, but processing requires less than half of the time. "M" (*Medium*) lies in between. Please note that this quality parameter has no relation to the quality parameter of *Stereo matching* (Section 6.2).

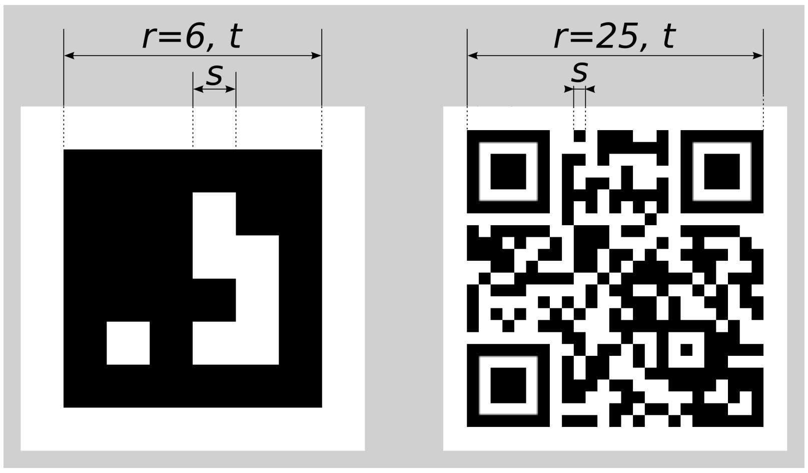


Fig. 7.3.3: Visualization of module size s, size of a tag in modules r, and size of a tag in meters t for AprilTags (left) and QR codes (right)

The maximum detection distance z at quality "H" can be approximated by using the following formulae,

$$z = \frac{fs}{p},$$

$$s = \frac{t}{r},$$

where f is the *focal length* (Section 6.1.2) in pixels and s is the size of a module in meters. s can easily be calculated by the latter formula, where t is the size of the tag in meters and r is the width of the code in modules (for AprilTags without the white border). Fig. 7.3.3 visualizes these variables. p denotes the number of image pixels per module required for detection. It is different for QR codes and AprilTags. Moreover, it varies with the tag's angle to the camera and illumination. Approximate values for robust detection are:

- AprilTag: $p = 5$ pixels/module
- QR code: $p = 6$ pixels/module

The following tables give sample maximum distances for different situations, assuming a focal length of 1075 pixels and the parameter `quality` to be set to "H".

Table 7.3.2: Maximum detection distance examples for AprilTags with a width of $t = 4$ cm

AprilTag family	Tag width	Maximum distance
36h11 (recommended)	8 modules	1.1 m
16h5	6 modules	1.4 m

Table 7.3.3: Maximum detection distance examples for QR codes with a width of $t = 8$ cm

Tag width	Maximum distance
29 modules	0.49 m
21 modules	0.70 m

7.3.3 Pose estimation

For each detected tag, the pose of this tag in the camera coordinate frame is estimated. A requirement for pose estimation is that a tag is fully visible in the left and right camera image. The coordinate frame of the tag is aligned as shown below.

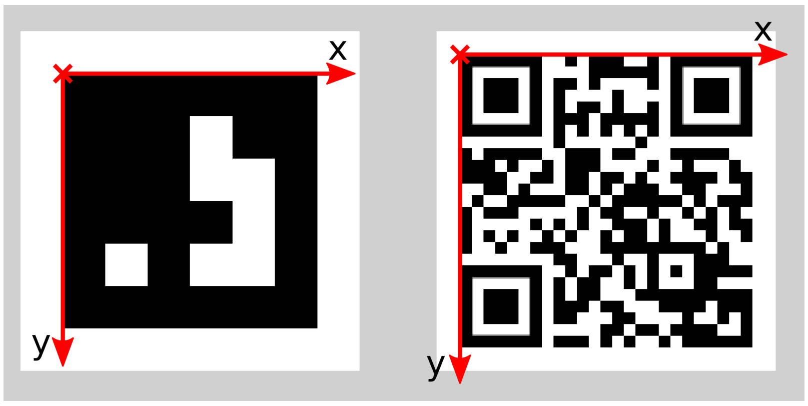


Fig. 7.3.4: Coordinate frames of AprilTags (left) and QR codes (right)

The z-axis is pointing “into” the tag. Please note that for AprilTags, although having the white border included in their definition, the coordinate system’s origin is placed exactly at the transition from the white to the black border. Since AprilTags do not have an obvious orientation, the origin is defined as the upper left corner in the orientation they are pre-generated in.

During pose estimation, the tag’s size is also estimated, while assuming the tag to be square. For QR codes, the size covers the full tag. For AprilTags, the size covers only the black part of the tag, hence ignoring the outermost white border.

The user can also specify the approximate size ($\pm 10\%$) of tags with a specific ID. All tags not matching this size contraint are automatically filtered out. This information is further used to resolve ambiguities in pose estimation that may arise if multiple tags with the same ID are visible in the left and right image and these tags are aligned in parallel to the image rows.

Note: For best pose estimation results one should make sure to accurately print the tag and to attach it to a rigid and as planar as possible surface. Any distortion of the tag or bump in the surface will degrade the estimated pose.

Warning: It is highly recommended to set the approximate size of a tag. Otherwise, if multiple tags with the same ID are visible in the left or right image, pose estimation may compute a wrong pose if these tags have the same orientation and are approximately aligned in parallel to the image rows. However, even if the approximate size is not given, the TagDetect components try to detect such situations and filter out affected tags.

Below tables give approximate precisions of the estimated poses of AprilTags and QR codes. We distinguish between lateral precision (i.e., in x and y direction) and precision in z direction. It is assumed that `quality` is set to "H" and that the *rc_visard*'s viewing direction is roughly parallel to the tag's normal. The size of a tag does not have a significant effect on the lateral or z precision; however, in general, larger tags improve precision. With respect to precision of the orientation especially around the x and y axes, larger tags clearly outperform smaller ones.

Table 7.3.4: Approximate pose precision for AprilTags

Distance	*rc_visard* 65 - lateral	*rc_visard* 65 - z	*rc_visard* 160 - lateral	*rc_visard* 160 - z
0.3 m	0.4 mm	0.9 mm	0.4 mm	0.8 mm
1.0 m	0.7 mm	3.3 mm	0.7 mm	3.3 mm

Table 7.3.5: Approximate pose precision for QR codes

Distance	*rc_visard* 65 - lateral	*rc_visard* 65 - z	*rc_visard* 160 - lateral	*rc_visard* 160 - z
0.3 m	0.6 mm	2.0 mm	0.6 mm	1.3 mm
1.0 m	2.6 mm	15 mm	2.6 mm	7.9 mm

7.3.4 Tag re-identification

Each tag has an ID; for AprilTags it is the *family* plus *tag ID*, for QR codes it is the contained data. However, these IDs are not unique, since the same tag may appear multiple times in a scene.

For distinction of these tags, the TagDetect components also assign each detected tag a unique identifier. To help the user identifying an identical tag over multiple detections, tag detection tries to re-identify tags; if successful, a tag is assigned the same unique identifier again. Tag re-identification compares the positions of the corners of the tags in a static coordinate frame to find identical tags. Tags are assumed identical if they did not or only slightly move in that static coordinate frame. For that static coordinate frame to be available, *dynamic-state estimation* (Section 6.3) must be switched on. If it is not, the sensor is assumed to be static; tag re-identification will then not work across sensor movements.

By setting the `max_corner_distance` threshold, the user can specify how much a tag is allowed move in the static coordinate frame between two detections to be considered identical. This parameter defines the maximum distance between the corners of two tags, which is shown in Fig. 7.3.5. The Euclidean distances of all four corresponding tag corners are computed in 3D. If none of these distances exceeds the threshold, the tags are considered identical.

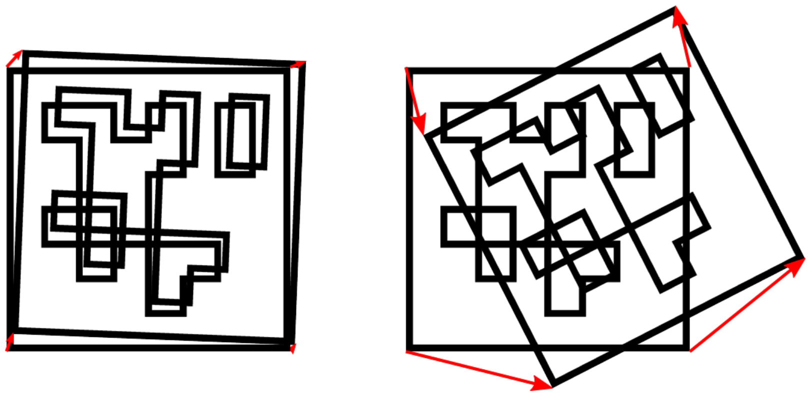


Fig. 7.3.5: Simplified visualization of tag re-identification. Euclidean distances between associated tag corners in 3D are compared (red arrows).

After a number of tag detection runs, previously detected tag instances will be discarded if they are not detected in the meantime. This can be configured by the parameter `forget_after_n_detections`.

7.3.5 Interfaces

There are two separate components for tag detection of the sensor, one for detecting AprilTags and one for QR codes, named `rc_april_tag_detect` and `rc_qr_code_detect`, respectively. Apart from the component names they share the same interface definition.

In addition to the REST-API, the TagDetect components provide tabs on the Web GUI, on which they can be tried out and configured manually.

Parameters and status values

In the following, the parameters and status values are listed based on the example of `rc_qr_code_detect`. They are the same for `rc_april_tag_detect`.

This component offers the following run-time parameters.

Table 7.3.6: The `rc_qr_code_detect` component's run-time parameters

Name	Type	Min	Max	Default	Description
`detect_inverted_tags`	bool	False	True	False	Detect tags with black and white exchanged
`forget_after_n_detections`	int32	1	1000	30	Number of detection runs after which to forget about a previous tag during tag re-identification
`max_corner_distance`	float64	0.001	0.01	0.005	Maximum distance of corresponding tag corners in meters during tag re-identification
`quality`	string	-	-	H	Quality of tag detection (H, M or L)
`use_cached_images`	bool	False	True	False	Use most recently received image pair instead of waiting for a new pair

This component reports the following status values.

Table 7.3.7: The `rc_qr_code_detect` component's status values

Name	Description
`state`	The current state of the node

The reported `state` can take one of the following values.

Table 7.3.8: Possible states of the TagDetect components

State name	Description
IDLE	The component is idle.
RUNNING	The component is running and ready for tag detection.
FATAL	A fatal error has occurred.

Services

The TagDetect components implement a state machine for starting and stopping. The actual tag detection can be triggered via `detect`.

start starts the component by transitioning from IDLE to RUNNING.

When running, the component receives images from the stereo camera and is ready to perform tag detections. To save computing resources on the sensor, the component should only be running when necessary.

This service requires no arguments.

This service returns the following response:

```
{
  "accepted": "bool",
  "current_state": "string"
}
```

stop stops the component by transitioning to IDLE.

This transition can be performed from state RUNNING and FATAL. All tag re-identification information is cleared during stopping.

This service requires no arguments.

This service returns the following response:

```
{
  "accepted": "bool",
  "current_state": "string"
}
```

restart restarts the component. If in RUNNING or FATAL, the component will be stopped and then started. If in IDLE, the component will be started.

This service requires no arguments.

This service returns the following response:

```
{
  "accepted": "bool",
  "current_state": "string"
}
```

detect triggers a tag detection. Depending on the use_cached_images parameter, the component will use the latest received image pair (if set to true) or wait for a new pair that is captured after the service call was triggered (if set to false, this is the default). Even if set to true, tag detection will never use one image pair twice.

It is recommended to call detect in state RUNNING only. It is also possible to be called in state IDLE, resulting in an auto-start and stop of the component. This, however, has some drawbacks: First, the call will take considerably longer; second, tag re-identification will not work. It is therefore highly recommended to manually start the component before calling detect.

This service requires the following arguments:

```
{
  "tags": [
    {
      "id": "string",
      "size": "float64"
    }
  ]
}
```

This service returns the following response:

```
{
  "return_code": {
```

(continues on next page)

(continued from previous page)

```
    "message": "string",
    "value": "int16"
  },
  "tags": [
    {
      "id": "string",
      "instance_id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "size": "float64",
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      }
    }
  ],
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```

Request: `tags` is the list of tag IDs that the TagDetect component should detect. For QR codes, the ID is the contained data. For AprilTags, it is *"<family>_<id>"*, so, e.g., for a tag of family 36h11 and ID 5, it is *"36h11_5"*. Naturally, the AprilTag component can only be triggered for AprilTags, and the QR code component only for QR codes.

The `tags` list can also be left empty. In that case, all detected tags will be returned. This feature should be used only during development and debugging of an application. Whenever possible, the concrete tag IDs should be listed, on the one hand avoiding some false positives, on the other hand speeding up tag detection by filtering tags not of interest.

For AprilTags, the user can not only specify concrete tags but also a complete family by setting the ID to "<family>", so, e.g., "36h11". All tags of this family will then be detected. It is further possible to specify multiple complete tag families or a combination of concrete tags and complete tag families; for instance, triggering for "36h11", "25h9_3", and "36h10" at the same time.

In addition to the ID, the approximate size ($\pm 10\%$) of a tag can be set with the `size` parameter. As described in *Pose estimation*, Section 7.3.3, this information helps to resolve ambiguities in pose estimation that may arise in certain situations.

Response: `timestamp` is set to the timestamp of the image pair the tag detection ran on.

`tags` contains all detected tags. `id` is the ID of the tag, similar to `id` in the request. `instance_id` is the random unique identifier of the tag assigned by tag re-identification. `pose` contains `position` and `orientation`. The orientation is in quaternion format. `pose_frame` is set to the coordinate frame above pose refers to. It will always be "camera". `size` will be set to the estimated tag size in meters; for AprilTags, the white border is not included.

`return_code` holds possible warnings or error codes in `value`, which are represented by a value greater than or less than 0, respectively. The respective error message can be found in `message`. The following table contains a list of common codes:

Code	Description
0	Success
-1	An invalid argument was provided
-4	A timeout occurred while waiting for the image pair
-9	The license is not valid
-101	Internal error
-102	There was a backwards jump of system time
-200	A fatal internal error occurred
101	A warning occurred during tag reading
102	A warning occurred during pose estimation
200	Multiple warnings occurred; see list in `message`
201	The component was not in state `RUNNING`

Tags might be omitted from the `detect` response due to several reasons, e.g., if a tag is visible in only one of the cameras or if pose estimation did not succeed. These filtered-out tags are noted in the log, which can be accessed as described in *Downloading log files* (Section 9.7).

A visualization of the latest detection is shown on the Web GUI tabs of the TagDetect components. Please note that this visualization will only be shown after calling the detection service at least once. On the Web GUI, one can also manually try the detection by clicking the *Detect* button.

Due to changes in system time on the sensor there might occur jumps of timestamps, forward as well as backward (see *Time synchronization*, Section 8.5). Forward jumps do not have an effect on the TagDetect component. Backward jumps, however, invalidate already received images. Therefore, in case a backwards time jump is detected, an error of value -102 will be issued on the next `detect` call, also to inform the user that the timestamps included in the response will jump back.

`save_parameters` With this service call, the TagDetect component's current parameter settings are persisted to the *rc_visard*. That is, these values are applied even after reboot.

`reset_defaults` Restores and applies the default values for this component's parameters ("factory reset") as given in the table above.

7.4 ItemPick and BoxPick

7.4.1 Introduction

The ItemPick and BoxPick components are optional on-board components of the *rc_visard*.

Note: The components are optional and require separate ItemPick or BoxPick *licenses* (Section 9.6) to be purchased.

The components provide out-of-the-box perception solutions for robotic pick-and-place applications. ItemPick targets the detection of flat surfaces of unknown objects for picking with a suction gripper. BoxPick detects rectangular surfaces and determines their position, orientation and size for grasping. The interface of both components is very similar. Therefore both components are described together in this chapter.

In addition, both components offer:

- a dedicated page on the *rc_visard* *Web GUI* (Section 4.5) for easy setup, configuration, testing, and application tuning
- the definition of regions of interest to select relevant volumes in the scene
- a load carrier detection functionality for bin-picking applications, to provide grasps for items inside a bin only
- the definition of compartments inside a load carrier to provide grasps for specific volumes of the bin only

- support for static and robot-mounted *rc_visard* devices and optional integration with the on-board *Hand-eye calibration* (Section 6.7) component, to provide grasps in the user-configured external reference frame
- a quality value associated to each suggested grasp and related to the flatness of the grasping surface
- sorting of grasps according to gravity and size so that items on top of a pile are grasped first.

Note: In this chapter, cluster and surface are used as synonyms and identify a set of points (or pixels) with defined geometrical properties.

Note: In this chapter, load carrier and bin are used as synonyms and identify a container with four walls, a floor and a rectangular rim.

7.4.2 Data types

Region of Interest

A region of interest defines a volume in space which is of interest for a specific user-application. The ItemPick and BoxPick components currently support regions of interest of the following types:

- `BOX`, with dimensions `box.x`, `box.y`, `box.z`.
- `SPHERE`, with radius `sphere.radius`.

The user can specify the region of interest `pose` in the `camera` or the `external` coordinate system (see *Hand-eye calibration*).

Both components can persistently store up to 10 different regions of interest, each one identified by a different `id`. The configuration of regions of interest is normally performed offline (e.g. on the ItemPick or BoxPick page of the *rc_visard* Web GUI), during the set up of the desired application.

Note: As opposed to the component parameters, the configured regions of interest are persistent even over firmware updates and rollbacks.

The region of interest can narrow the volume that is searched for a load carrier model, or select a volume which only contains items to be grasped.

Note: If the region of interest filter is not applied, the components process the whole scene visible to the camera.

Load Carrier

A load carrier (bin) is a container with four walls, a floor and a rectangular rim. It is defined by its `outer_dimensions` and `inner_dimensions`.

The load carrier detection algorithm is based on the detection of the load carrier rectangular rim. By default, the rectangular `rim_thickness` is computed from the outer and inner dimensions. As an alternative, its value can also be explicitly specified by the user.

Note: Typically, outer and inner dimensions of a load carrier are available in the specifications of the load carrier manufacturer.

The load carrier reference frame is defined such that its origin is at the center of the load carrier outer box and its z axis is perpendicular to the load carrier floor.

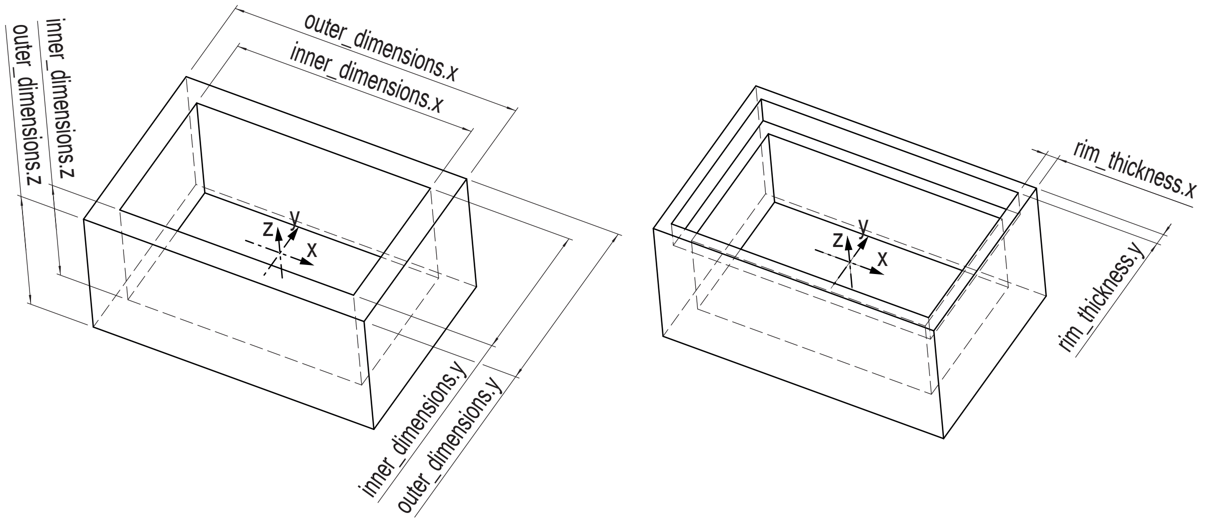


Fig. 7.4.1: Load carrier models and reference frame.

The user can optionally specify a prior for the load carrier `pose`. The detected load carrier pose is guaranteed to have the minimum rotation with respect to the load carrier prior pose. If no prior is specified, the algorithm searches for a load carrier whose floor is perpendicular to the estimated gravity vector.

The components can persistently store up to 10 different load carrier models, each one identified by a different `id`. The configuration of a load carrier model is normally performed offline (e.g. on the ItemPick or BoxPick page of the *rc_visard* Web GUI), during the set up the desired application.

Note: As opposed to the component parameters, the configured load carrier models are persistent even over firmware updates and rollbacks.

The modules enable the computation of grasps for a specific volume of the load carrier (`load_carrier_compartment`). The compartment is a box whose `pose` is defined with respect to the load carrier reference frame.

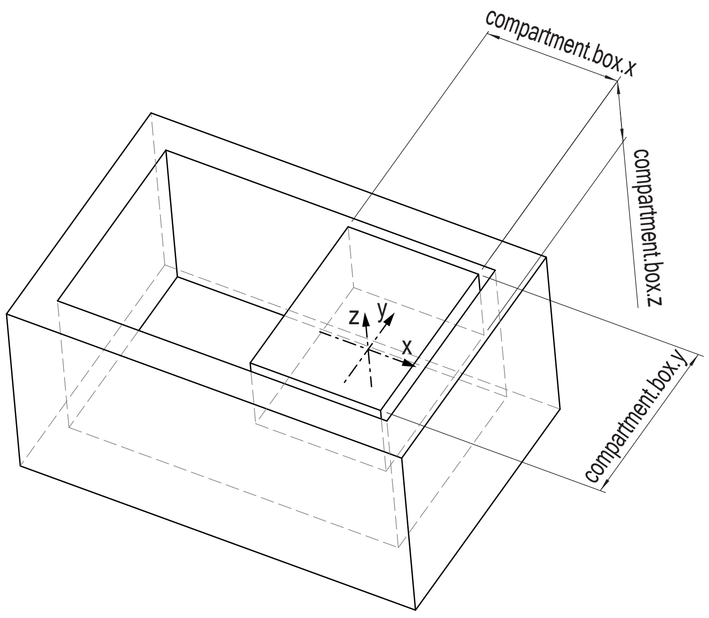


Fig. 7.4.2: Compartment inside a load carrier.

Suction Grasp

A grasp provided by the ItemPick and BoxPick components represents the recommended pose of the TCP (Tool Center Point) of the suction gripper. The grasp orientation is a right-handed coordinate system and is defined such that its z axis is normal to the surface pointing inside the object at the grasp position and its x axis is directed along the maximum elongation of the surface.

The computed grasp pose is the center of the biggest ellipse that can be inscribed in each surface.

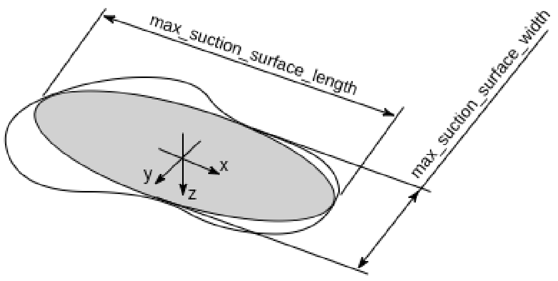


Fig. 7.4.3: Illustration of suction grasp with coordinate system and ellipse representing the maximum suction surface.

Each grasp includes the dimensions of the maximum suction surface available, modelled as an ellipse of axes `max_suction_surface_length` and `max_suction_surface_width`. The user is enabled to filter grasps by specifying the minimum suction surface required by the suction device in use.

In the BoxPick component, the grasp position corresponds to the center of the detected rectangle and the dimensions of the maximum suction surface available matches the estimated rectangle dimensions. Detected rectangles with missing data or occlusions by other objects for more than 15% of their surface do not get an associated grasp.

The grasp definition is complemented by a `uuid` (Universally Unique Identifier) and the `timestamp` of the oldest image that was used to compute the grasp.

Item model

The ItemPick and BoxPick components allow to specify a model for the items to be picked. Each item model includes minimum and maximum dimenstions of the expected items.

- The ItemPick component supports specifying the minimum and maximum sizes of unknown flexible and/or deformable items.
- The BoxPick component supports specifying the minimum and maximum length and width of several rectangles.

7.4.3 Interaction with other components

Internally, the ItemPick and BoxPick components depend on, and interact with other on-board components as listed below.

Note: All changes and configuration updates to these components will affect the performance of the ItemPick and BoxPick components.

Stereo camera and Stereo matching

The ItemPick and BoxPick components make internally use of the following data:

- Rectified images from the *Stereo camera* component (`rc_stereocamera`, Section 6.1);

- Disparity, error, and confidence images from the *Stereo matching* component (`rc_stereomatching`, Section 6.2).

Sensor dynamics

For each load carrier detection and grasp computation, the components estimate the gravity vector by subscribing to the IMU data stream from the *Sensor dynamics* component (`rc_dynamics`, Section 6.3).

> **Note:** The gravity vector is estimated from linear acceleration readings from the on-board IMU. For this reason, the ItemPick and BoxPick components require the *rc_visard* to remain still while the gravity vector is being estimated.

IO and Projector Control

In case the *rc_visard* is used in conjunction with an external random dot projector and the *IO and Projector Control* component (`rc_iocontrol`, Section 7.2), the output mode for the GPIO output in use should be se to `ExposureAlternateActive`, as explained in the *Description of run-time parameters* (Section 7.2.1) of the *IO and Projector Control* component.

No additional changes are required to use the ItemPick and BoxPick components in combination with a random dot projector.

Hand-eye calibration

In case the *rc_visard* has been calibrated to a robot, two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). All poses provided to and from the component are in the camera frame, and no prior knowledge about the pose of the *rc_visard* in the environment is required. This means that the configured regions of interest and load carriers move with the camera. It is the user's responsibility to update the configured poses if the camera frame moves (e.g. with a robot-mounted sensor).
2. **External frame** (`external`). All poses provided to and from the component are in the external frame, configured by the user during the hand-eye calibration process. The component relies on the on-board *Hand-eye calibration component* to retrieve the sensor mounting (static or robot mounted) and the hand-eye transformation. If the sensor mounting is static, no further information is needed. If the sensor is robot-mounted, the `robot_pose` is required to transform poses to and from the `external` frame.

> **Note:** If no hand-eye calibration is available, all `pose_frame` values should be set to `camera`.

All `pose_frame` values that are not `camera` or `external` are rejected.

7.4.4 Parameters and Status Values

The ItemPick and BoxPick components are called `rc_itempick` and `rc_boxpick` in the REST-API. The user can explore and configure the `rc_itempick` and `rc_boxpick` component's run-time parameters, e.g. for development and testing, using the *rc_visard* *Web GUI* (Section 4.5) or *Swagger UI* (Section 8.2.4).

This component offers the following run-time parameters.

Table 7.4.1: The `rc_itempick` and `rc_boxpick` components application parameters

Name	Type	Min	Max	Default	Description
`max_grasps`	int32	1	20	5	Maximum number of provided grasps

Table 7.4.2: The rc_itempick and rc_boxpick components load carrier detection parameters

Name	Type	Min	Max	Default	Description
`load_carrier_crop_distance`	float64	0.0	0.02	0.005	Safety margin in meters by which the load carrier inner dimensions are reduced to define the region of interest for grasp computation
`load_carrier_model_tolerance`	float64	0.003	0.025	0.008	Indicates how much the estimated load carrier dimensions are allowed to differ from the load carrier model dimensions in meters

Table 7.4.3: The rc_itempick and rc_boxpick components surface clustering parameters

Name	Type	Min	Max	Default	Description
cluster_max_dimension	float64	0.05	0.8	0.3	**Only for rc_itempick**. Diameter of the largest sphere enclosing each cluster in meters. Clusters larger than this value are filtered out before grasp computation.
cluster_max_curvature	float64	0.005	0.5	0.11	Maximum curvature allowed within one cluster. The smaller this value, the more clusters will be split apart.
clustering_patch_size	int32	3	10	4	**Only for rc_itempick**. Size in pixels of the square patches the depth map is subdivided into during the first clustering step
clustering_max_surface_rmse	float64	0.0005	0.01	0.004	Maximum root-mean-square error (RMSE) in meters of points belonging to a surface
clustering_discontinuity_factor	float64	0.5	5.0	1.0	Factor used to discriminate depth discontinuities within a patch. The smaller this value, the more clusters will be split apart.
item_model_tolerance	float64	0.0	0.05	0.0	**Only for rc_itempick**. This parameter is deprecated. Indicates how much the estimated item dimensions are allowed to differ from the item model dimensions in meters

This component reports the following status values.

Table 7.4.4: The rc_itempick and rc_boxpick components status values

Name	Description
state	The current state of the rc_itempick and rc_boxpick node
last_timestamp_processed	The timestamp of the last processed dataset
data_acquisition_time	Time in seconds required by the last active service to acquire images. Standard values are between 0.5 s and 0.6 s with High depth image quality.
load_carrier_detection_time	Processing time of the last load carrier detection in seconds
grasp_computation_time	Processing time of the last grasp computation in seconds

The reported state can take one of the following values.

Table 7.4.5: Possible states of the ItemPick and BoxPick components

State name	Description
IDLE	The component is idle.
RUNNING	The component is running and ready for load carrier detection and grasp computation.
FATAL	A fatal error has occurred.

7.4.5 Services

The user can explore and call the `rc_itempick` and `rc_boxpick` component's services, e.g. for development and testing, using *Swagger UI* (Section 8.2.4) or the *rc_visard Web GUI* (Section 4.5).

Each service component provides a `return_code`, which consists of a value plus an optional message.

A successful service returns with a `return_code` value of `0`. Negative `return_code` values indicate that the service failed. Positive `return_code` values indicate that the service succeeded with additional information. The smaller value is selected in case a service has multiple `return_code` values, but all messages are appended in the `return_code` message.

The following table contains a list of common codes:

Table 7.4.6: Return codes of the ItemPick and BoxPick services

Code	Description
0	Success
-1	An invalid argument was provided
-4	Data acquisition took longer than the maximum allowed time of 3.0 seconds
-301	More than one item model of type `UNKNOWN` provided to the `compute_grasps` service
-302	More than one load carrier provided to the `detect_load_carriers` service, but only one is supported
100	The requested load carriers were not detected in the scene
101	No valid surfaces or grasps were found in the scene
102	The detected load carrier is empty
200	The component is in `IDLE` state
300	A valid `robot_pose` was provided as argument but it is not required
400	No `item_models` were provided to the `compute_grasps` service request
500	The region of interest visualization images could not be generated during the call to `set_region_of_interest`
600	An existent persistent model was overwritten by the call to `set_load_carrier` or `set_region_of_interest`

The ItemPick and BoxPick components offer the following services.

`start` Starts the component. If the command is accepted, the component moves to state `RUNNING`. The `current_state` value in the service response may differ from `RUNNING` if the state transition is still in process when the service returns.

This service requires no arguments.

This service returns the following response:

```
{
  "accepted": "bool",
  "current_state": "string"
}
```

`stop` Stops the component. If the command is accepted, the component moves to state `IDLE`. The `current_state` value in the service response may differ from `IDLE` if the state transition is still in process when the service returns.

This service requires no arguments.

This service returns the following response:

```
{
  "accepted": "bool",
  "current_state": "string"
}
```

set_region_of_interest Persistently stores a region of interest on the *rc_visard*. All configured regions of interest are persistent over firmware updates and rollbacks.

See *Region of Interest* the definition of the region of interest type.

This service requires the following arguments:

```
{
  "region_of_interest": {
    "box": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "id": "string",
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "pose_frame": "string",
    "sphere": {
      "radius": "float64"
    },
    "type": "string"
  },
  "robot_pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  }
}
```

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

get_regions_of_interest Returns the configured regions of interest with the requested `region_of_interest_ids`. If no `region_of_interest_ids` are provided, all configured regions of interest are returned.

This service requires the following arguments:

```
{
  "region_of_interest_ids": [
    "string"
  ]
}
```

This service returns the following response:

```
{
  "regions_of_interest": [
    {
      "box": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "sphere": {
        "radius": "float64"
      },
      "type": "string"
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

delete_regions_of_interest Deletes the configured regions of interest with the requested `region_of_interest_ids`. All regions of interest to be deleted must be explicitly stated in `region_of_interest_ids`.

This service requires the following arguments:

```
{
  "region_of_interest_ids": [
    "string"
  ]
}
```

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`set_load_carrier` Persistently stores a load carrier on the *rc_visard*. All configured load carriers are persistent over firmware updates and rollbacks.

See *Load Carrier* for the definition of the load carrier type.

This service requires the following arguments:

```
{
  "load_carrier": {
    "id": "string",
    "inner_dimensions": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "outer_dimensions": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "pose_frame": "string",
    "rim_thickness": {
      "x": "float64",
      "y": "float64"
    }
  }
}
```

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`get_load_carriers` Returns the configured load carriers with the requested `load_carrier_ids`. If no `load_carrier_ids` are provided, all configured load carriers are returned.

This service requires the following arguments:

```
{
  "load_carrier_ids": [
    "string"
  ]
}
```

This service returns the following response:

```
{
  "load_carriers": [
    {
      "id": "string",
      "inner_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "outer_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "rim_thickness": {
        "x": "float64",
        "y": "float64"
      }
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`delete_load_carriers` Deletes the configured load carriers with the requested `load_carrier_ids`. All load carriers to be deleted must be explicitly stated in `load_carrier_ids`.

This service requires the following arguments:

```
{
  "load_carrier_ids": [
    "string"
  ]
}
```

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`detect_load_carriers` Triggers a load carrier detection. All images used by the node are guaranteed to be newer than the service trigger time.

This service requires the following arguments:

```
{
  "load_carrier_ids": [
    "string"
  ],
  "pose_frame": "string",
  "region_of_interest_id": "string",
  "robot_pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  }
}
```

This service returns the following response:

```
{
  "load_carriers": [
    {
      "id": "string",
      "inner_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "outer_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
```

(continues on next page)

(continued from previous page)

```
      "pose_frame": "string",
      "rim_thickness": {
        "x": "float64",
        "y": "float64"
      }
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```

Required arguments:

`pose_frame`: defines the output pose frame for the detected load carriers.

`load_carrier_ids`

`robot_pose`: only if working in `pose_frame="external"` and the *rc_visard* is robot-mounted.

Optional arguments:

`region_of_interest_id`: delimits the volume of space where to search for the load carrier. The processing time for load carrier detection increases with the size of the selected region of interest.

`detect_items` (only available for BoxPick) Triggers the detection of rectangles. Processing includes load carrier detection in the region of interest. The poses are given relative to the centers of the rectangles. The z-axis points towards the camera. Multiple rectangles can be specified with different dimension ranges. All images used by the node are guaranteed to be newer than the service trigger time.

If successful, the service returns a list of rectangles and (optionally) the detected load carriers.

This service requires the following arguments:

```
{
  "item_models": [
    {
      "rectangle": {
        "max_dimensions": {
          "x": "float64",
          "y": "float64"
        },
        "min_dimensions": {
          "x": "float64",
          "y": "float64"
        }
      },
      "type": "string"
    }
  ],
  "load_carrier_compartment": {
    "box": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "pose": {
      "orientation": {
```

(continues on next page)

(continued from previous page)

```
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    }
  },
  "load_carrier_id": "string",
  "pose_frame": "string",
  "region_of_interest_id": "string",
  "robot_pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  }
}
```

This service returns the following response:

```
{
  "items": [
    {
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "rectangle": {
        "x": "float64",
        "y": "float64"
      },
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      },
      "type": "string",
      "uuid": "string"
    }
  ],
```

(continues on next page)

(continued from previous page)

```
  "load_carriers": [
    {
      "id": "string",
      "inner_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "outer_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "rim_thickness": {
        "x": "float64",
        "y": "float64"
      }
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```

Required arguments:

`pose_frame`: defines the output pose frame for the detected rectangles.

`item_models`: defines a list of rectangles with minimum and maximum size, with the minimum dimensions strictly smaller than the maximum dimensions. At least one `item_model` is always required. The dimensions should be given as precise as possible to avoid misdetections.

`robot_pose`: only if working in `pose_frame="external"` and the *rc_visard* is robot-mounted.

Optional arguments:

`region_of_interest_id`: delimits the volume of space where to search for the load carrier or selects a volume which contains items to be grasped if no `load_carrier_id` is set. The processing time for load carrier detection and grasp computation increases with the size of the selected region of interest.

`load_carrier_id`: limits grasp computation to the content of the detected load carrier.

`load_carrier_compartment`: selects a compartment within the detected load carrier.

`compute_grasps` (for ItemPick) Triggers the computation of grasping poses for a suction device. All images used by the node are guaranteed to be newer than the service trigger time.

If successful, the service returns a sorted list of grasps and (optionally) the detected load carriers.

This service requires the following arguments:

```
{
  "item_models": [
    {
      "type": "string",
      "unknown": {
        "max_dimensions": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "min_dimensions": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      }
    }
  ],
  "load_carrier_compartment": {
    "box": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    }
  },
  "load_carrier_id": "string",
  "pose_frame": "string",
  "region_of_interest_id": "string",
  "robot_pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  },
  "suction_surface_length": "float64",
  "suction_surface_width": "float64"
}
```

This service returns the following response:

```
{
  "grasps": [
    {
      "item_uuid": "string",
      "max_suction_surface_length": "float64",
      "max_suction_surface_width": "float64",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "quality": "float64",
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      },
      "type": "string",
      "uuid": "string"
    }
  ],
  "load_carriers": [
    {
      "id": "string",
      "inner_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "outer_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "rim_thickness": {
        "x": "float64",
        "y": "float64"
      }
    }
```

(continues on next page)

(continued from previous page)

```
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```

Required arguments:

`pose_frame`: defines the output pose frame for the computed grasps.

`suction_surface_length`: length of the suction device grasping surface.

`suction_surface_width`: width of the suction device grasping surface.

`robot_pose`: only if working in `pose_frame="external"` and the *rc_visard* is robot-mounted.

Optional arguments:

`region_of_interest_id`: delimits the volume of space where to search for the load carrier or selects a volume which contains items to be grasped if no `load_carrier_id` is set. The processing time for load carrier detection and grasp computation increases with the size of the selected region of interest.

`load_carrier_id`: limits grasp computation to the content of the detected load carrier.

`load_carrier_compartment`: selects a compartment within the detected load carrier.

`item_models`: defines a list of unknown items with minimum and maximum dimensions, with the minimum dimensions strictly smaller than the maximum dimensions. Only one `item_model` of type `UNKNOWN` is currently supported.

`compute_grasps` (for BoxPick) Triggers the detection of rectangles and computation of a grasp pose for the detected rectangles. The poses are given relative to the centers of the rectangles. Multiple rectangles can be specified with different dimension ranges. All images used by the node are guaranteed to be newer than the service trigger time.

If successful, the service returns a list of rectangles, a list of computed grasps and (optionally) the detected load carriers. Each grasp includes the `uuid` of the corresponding rectangle and vice versa.

This service requires the following arguments:

```
{
  "item_models": [
    {
      "rectangle": {
        "max_dimensions": {
          "x": "float64",
          "y": "float64"
        },
        "min_dimensions": {
          "x": "float64",
          "y": "float64"
        }
      },
      "type": "string"
    }
  ],
  "load_carrier_compartment": {
    "box": {
      "x": "float64",
```

(continues on next page)

(continued from previous page)

```
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      }
    },
    "load_carrier_id": "string",
    "pose_frame": "string",
    "region_of_interest_id": "string",
    "robot_pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "suction_surface_length": "float64",
    "suction_surface_width": "float64"
}
```

This service returns the following response:

```
{
  "grasps": [
    {
      "item_uuid": "string",
      "max_suction_surface_length": "float64",
      "max_suction_surface_width": "float64",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "quality": "float64",
      "timestamp": {
```

(continues on next page)

(continued from previous page)

```
        "nsec": "int32",
        "sec": "int32"
      },
      "type": "string",
      "uuid": "string"
    }
  ],
  "items": [
    {
      "grasp_uuids": [
        "string"
      ],
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "rectangle": {
        "x": "float64",
        "y": "float64"
      },
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      },
      "type": "string",
      "uuid": "string"
    }
  ],
  "load_carriers": [
    {
      "id": "string",
      "inner_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "outer_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
```

(continues on next page)

(continued from previous page)

```
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "rim_thickness": {
        "x": "float64",
        "y": "float64"
      }
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```

Required arguments:

`pose_frame`: defines the output pose frame for the computed grasps and detected rectangles.

`item_models`: defines a list of rectangles with minimum and maximum size, with the minimum dimensions strictly smaller than the maximum dimensions. At least one `item_model` is always required. The dimensions should be given as precise as possible to avoid misdetections.

`suction_surface_length`: length of the suction device grasping surface.

`suction_surface_width`: width of the suction device grasping surface.

`robot_pose`: only if working in `pose_frame="external"` and the *rc_visard* is robot-mounted.

Optional arguments:

`region_of_interest_id`: delimits the volume of space where to search for the load carrier or selects a volume which contains items to be grasped if no `load_carrier_id` is set. The processing time for load carrier detection and grasp computation increases with the size of the selected region of interest.

`load_carrier_id`: limits grasp computation to the content of the detected load carrier.

`load_carrier_compartment`: selects a compartment within the detected load carrier.

`save_parameters` This service saves the currently set parameters persistently. Thereby, the same parameters will still apply after a reboot of the sensor. The node parameters are not persistent over firmware updates and rollbacks.

`reset_to_defaults` This service resets all parameters of the component to its default values, as listed in above table. The reset does not apply to regions of interest and load carriers.

7.5 SilhouetteMatch

7.5.1 Introduction

The SilhouetteMatch component is an optional on-board component of the *rc_visard*, which detects objects by matching a predefined silhouette (“template”) to edges in an image.

Note: This component is optional and requires a separate SilhouetteMatch *license* (Section 9.6) to be purchased.

For the SilhouetteMatch component to work, special object templates are required for each type of object to be detected. Roboception offers a template generation service on their website (https://roboception.com/en/template-request/), where the user can upload CAD files or recorded data of the objects and request object templates for the SilhouetteMatch component.

The object templates consist of significant edges of each object. These template edges are matched to the edges detected in the left and right camera images, considering the actual size of the objects and their distance from the *rc_visard*. The poses of the detected objects are returned and can be used for grasping, for example.

Suitable objects

The SilhouetteMatch component is intended for objects which have significant edges on a common plane that is parallel to the base plane on which the objects are placed. This applies to flat, nontransparent objects, such as routed, laser-cut or water-cut 2D parts and flat-machined parts. More complex parts can also be detected if there are significant edges on a common plane, e.g. a special pattern printed on a flat surface.

The SilhouetteMatch component works best for objects on a texture-free base plane. The color of the base plane should be chosen such that a clear contrast between the objects and the base plane appears in the intensity image.

Suitable scene

The scene must meet the following conditions to be suitable for the SilhouetteMatch component:

- The objects to be detected must be suitable for the SilhouetteMatch component as described above.
- Only objects belonging to one specific template are visible at a time (unmixed scenario). In case other objects are visible as well, a proper region of interest (ROI) must be set.
- All visible objects are lying on a common base plane, which has to be calibrated.
- The offset between the base plane normal and the line of sight of the *rc_visard* does not exceed 10 degrees.
- The objects are not partially or fully occluded.
- All visible objects are right side up (no flipped objects).
- The object edges to be matched are visible in both, left and right camera images.

7.5.2 Base-plane calibration

Before objects can be detected, a base-plane calibration must be performed. Thereby, the distance and angle of the plane on which the objects are placed is measured and stored persistently on the *rc_visard*.

Separating the detection of the base plane from the actual object detection renders scenarios possible in which the base plane is temporarily occluded. Moreover, it increases performance of the object detection for scenarios where the base plane is fixed for a certain time; thus, it is not necessary to continuously re-detect the base plane.

The base-plane calibration can be performed in three different ways, which will be explained in more detail further down:

- AprilTag based
- Stereo based
- Manual

The base-plane calibration is successful if the normal vector of the estimated base plane is at most 10 degrees offset to the line of sight of the *rc_visard*. If the base-plane calibration is successful, it will be stored persistently on the *rc_visard* until it is removed or a new base-plane calibration is performed.

In scenarios where the base plane is not accessible for calibration, a plane parallel to the base-plane can be calibrated. Then an `offset` parameter can be used to shift the estimated plane onto the actual base plane where the objects are placed. The `offset` parameter gives the distance in meters by which the estimated plane is shifted towards to *rc_visard*.

In the REST-API, a plane is defined by a `normal` and a `distance`. `normal` is a normalized 3-vector, specifying the normal of the plane. The normal points away from the camera. `distance` represents the distance of the plane from the camera along the normal. Normal and distance can also be interpreted as a, b, c, and d components of the plane equation, respectively:

$$ax + by + cz + d = 0$$

Note: To avoid privacy issues, the image of the persistently stored base-plane calibration will appear blurred after rebooting the *rc_visard*.

AprilTag based base-plane calibration

AprilTag detection (ref. *TagDetect*, Section 7.3) is used to find AprilTags in the scene and fit a plane through them. At least three AprilTags must be placed on the base plane so that they are visible in the left and right camera images. The tags should be placed such that they are spanning a triangle that is as large as possible. The larger the triangle, the more accurate is the resulting base-plane estimate. Use this method if the base plane is untextured and no external random dot projector is available. This calibration mode is available via the REST-API and the Web GUI.

Stereo based base-plane calibration

The 3D point cloud computed by SGM onboard the *rc_visard* is used to find the plane which is farthest away from the *rc_visard*. Therefore, the region of interest (ROI) for this method must be set such that only the relevant base plane is included. The base plane must not be completely occluded by objects. Use this method if the base plane is well textured or you can make use of a random dot projector to project texture on the base plane. This calibration mode is available via the REST-API and the Web GUI.

Manual base-plane calibration

The base plane can be set manually if its parameters are known, e.g. from previous calibrations. This calibration mode is only available via the REST-API and not the Web GUI.

7.5.3 Setting a region of interest

If objects are to be detected only in part of the camera's field of view, a region of interest (ROI) can be set accordingly. This ROI is defined as a rectangular part of the left camera image, and can be set via the REST-API or in the Web GUI. The Web GUI offers an easy-to-use selection tool. Up to 10 ROIs can be set and stored persistently on the *rc_visard*. Each ROI must have a unique name to address a specific ROI in the base-plane calibration or object detection process.

In the REST-API, a 2D ROI is defined by the following values:

- `id`: Unique name of the region of interest
- `offset_x`, `offset_y`: offset in pixels along the x-axis and y-axis from the top-left corner of the image, respectively
- `width`, `height`: width and height in pixels

7.5.4 Detection of objects

Objects can only be detected after a successful base-plane calibration. It must be ensured that the position and orientation of the base plane does not change before the detection of objects. Otherwise, the base-plane calibration must be renewed.

For triggering the object detection, in general, the following information must be provided to the SilhouetteMatch component:

- The template of the object to be detected in the scene.
- The coordinate frame in which the poses of the detected objects shall be returned (ref. *Hand-eye calibration*).

Optionally, further information can be given to the SilhouetteMatch component:

- An offset in case the objects are lying not on the base plane but on a plane parallel to it. The offset is the distance between both planes given in the direction towards the camera. If omitted, an offset of 0 is assumed.
- The region of interest in which the objects should be detected. If omitted, objects are matched in the whole image.
- The current robot pose in case the chosen coordinate frame for the poses is `external` and the *rc_visard* is mounted on the robot (possible only via the REST-API).

On the Web GUI the detection can be tested in the "Try Out" section of the SilhouetteMatch component's tab. The result is visualized as shown in figure Fig. 7.5.1.

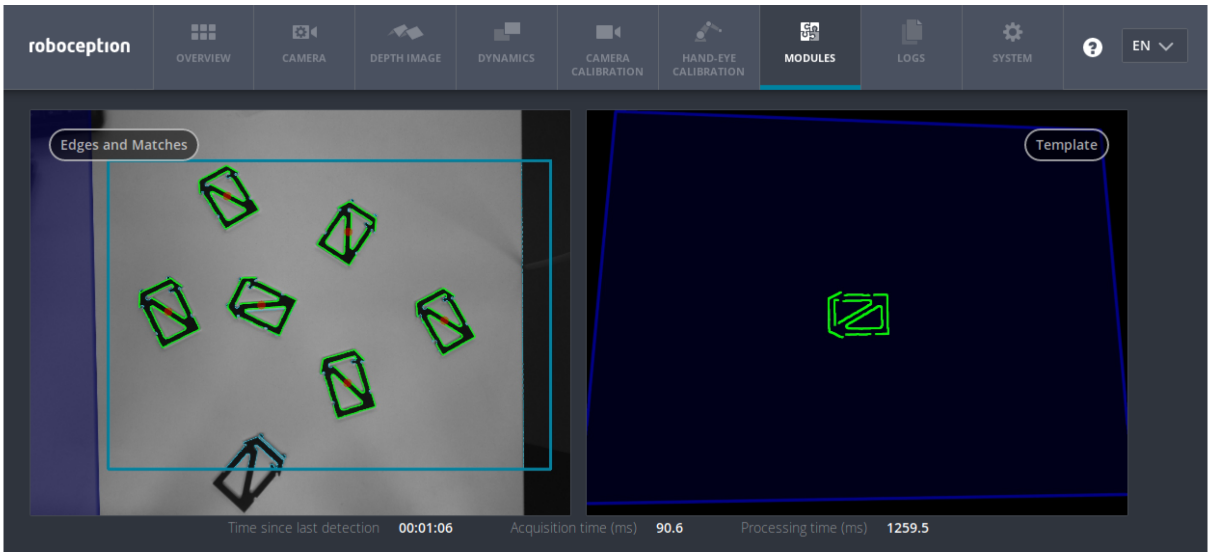


Fig. 7.5.1: Result image of the SilhouetteMatch component as shown in the Web GUI

The right image shows the calibrated base plane in blue and the template to be matched in green. The template is warped to the size and tilt matching objects on the calibrated base plane would have.

The left image shows the detection result. The shaded blue area on the left is the region of the left camera image which does not overlap with the right image, and in which no objects can be detected. The chosen region of interest is shown as bold petrol rectangle. The detected edges in the image are shown in light blue and the matches with the template are shown in green. The red circles are the origins of the detected objects as defined in the template. The poses of the object origins in the chosen coordinate frame are returned as results. In case the objects are rotationally symmetric, all returned poses will have the same orientation. For rotationally non-symmetric objects, the orientation of the detected objects is aligned with the normal of the base plane.

The detection results and runtimes are affected by several run-time parameters which are listed and explained further down. Improper parameters can lead to time-outs of the SilhouetteMatch component's detection process.

7.5.5 Interaction with other components

Internally, the SilhouetteMatch component depends on, and interacts with other on-board components as listed below.

Note: All changes and configuration updates to these components will affect the performance of the SilhouetteMatch component.

Stereo camera and stereo matching

The SilhouetteMatch component makes internally use of the rectified images from the *Stereo camera* component (`rc_stereocamera`, Section 6.1). Thus, the exposure time should be set properly to achieve the optimal performance of the component.

For base-plane calibration in stereo mode the disparity images from the *Stereo matching* component (`rc_stereomatching`, Section 6.2) are used. Apart from that, the stereo-matching component should not be run in parallel to the SilhouetteMatch component, because the detection runtime increases.

For best results it is recommended to enable *smoothing* (Section 6.2.4) for *Stereo matching*.

IO and Projector Control

In case the *rc_visard* is used in conjunction with an external random dot projector and the *IO and Projector Control* component (`rc_iocontrol`, Section 7.2), the projector should be used for the stereo-based base-plane calibration.

The projected pattern must not be visible in the left and right camera images during object detection as it interferes with the matching process. Therefore, it must either be switched off or operated in `ExposureAlternateActive` mode.

Hand-eye calibration

In case the *rc_visard* has been calibrated to a robot, the SilhouetteMatch component can automatically provide poses in the robot coordinate frame. For the SilhouetteMatch node's *Services*, the frame of the input and output poses and plane coordinates can be controlled with the `pose_frame` argument.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). All poses and plane coordinates provided to and by the component are in the camera frame.
2. **External frame** (`external`). All poses and plane coordinates provided to and by the component are in the external frame, configured by the user during the hand-eye calibration process. The component relies on the on-board *Hand-eye calibration component* to retrieve the sensor mounting (static or robot mounted) and the hand-eye transformation.

All `pose_frame` values that are not `camera` or `external` are rejected.

Note: If no hand-eye calibration is available, all `pose_frame` values should be set to `camera`.

Note: If the hand-eye calibration has changed after base-plane calibration, the base-plane calibration will be marked as invalid and must be renewed.

Depending on the value of `pose_frame`, it is necessary to additionally provide the current robot pose (`robot_pose`) to the SilhouetteMatch component:

- If `pose_frame` is set to `external`, providing the robot pose is obligatory.
- If `pose_frame` is set to `camera`, providing the robot pose is optional.

If the current robot pose is provided during calibration, it is stored persistently on the sensor. If the updated robot pose is later provided during `get_base_plane_calibration` or `detect_object` as well, the base-plane calibration will be transformed automatically to this new robot pose. This enables the user to change the robot pose (and thus sensor position) between base-plane calibration and object detection.

Note: Object detection can only be performed if the limit of 10 degrees angle offset between the base plane normal and the line of sight of the *rc_visard* is not exceeded.

7.5.6 Parameters and status values

The SilhouetteMatch software component is called `rc_silhouettematch` in the REST-API. The user can explore and configure the `rc_silhouettematch` component's run-time parameters, e.g. for development and testing, using the *rc_visard Web GUI* (Section 4.5) or *Swagger UI* (Section 8.2.4).

Parameter overview

This component offers the following run-time parameters.

Table 7.5.1: The `rc_silhouettematch` component's run-time parameters

Name	Type	Min	Max	Default	Description
`edge_sensitivity`	float64	0.1	1.0	0.6	sensitivity of the edge detector
`match_max_distance`	float64	0.0	10.0	2.5	maximum allowed distance in pixels between the template and the detected edges in the image
`match_percentile`	float64	0.7	1.0	0.85	percentage of template pixels that must be within the maximum distance to successfully match the template
`max_number_of_detected_objects`	int32	1	20	10	maximum number of detected objects
`quality`	string	-	-	High	H(igh), M(edium), or L(ow)

This component reports the following status values.

Table 7.5.2: The `rc_silhouettematch` component's status values

Name	Description
`data_acquisition_time`	Time in seconds required by the last active service to acquire images
`detect_service_time`	Processing time of the object dection, including data acquisition time
`calibrate_service_time`	Processing time of the base-plane calibration, including data acquisition time
`last_timestamp_processed`	The timestamp of the last processed dataset

Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's SilhouetteMatch Module tab. The name in the Web GUI is given in brackets behind the parameter name and the parameters are listed in the order they appear in the Web GUI:

`quality` **(*Quality*)** Object detection can be performed on images with different resolutions: high (1280 x 960), medium (640 x 480) and low (320 x 240). The lower the resolution, the lower the detection time, but the fewer details of the objects are visible.

max_number_of_detected_objects (***Maximum Object Number***) This parameter gives the maximum number of objects to detect in the scene. If more than the given number of objects can be detected in the scene, only the objects with the highest matching results are returned.

match_max_distance (***Maximum Matching Distance***) This parameter gives the maximum allowed pixel distance of an image edge pixel from the object edge pixel in the template to be still considered as matching. If the object is not perfectly represented in the template, it might not be detected when this parameter is low. High values, however, might lead to false detections in case of a cluttered scene or the presence of similar objects, and also increase runtime.

match_percentile (***Matching Percentile***) This parameter indicates how strict the matching process should be. The matching percentile is the ratio of template pixels that must be within the Maximum Matching Distance to successfully match the template. The higher this number, the more accurate the match must be to be considered as valid.

edge_sensitivity (***Edge Sensitivity***) This parameter influences how many edges are detected in the camera images. The higher this number, the more edges are found in the intensity image. That means, for lower numbers, only the most significant edges are considered for template matching. A large number of edges in the image might increase the detection time.

7.5.7 Services

The user can explore and call the `rc_silhouettematch` component's services, e.g. for development and testing, using *Swagger UI* (Section 8.2.4) or the *rc_visard Web GUI* (Section 4.5).

Each service component provides a `return_code`, which consists of a value plus an optional message.

A successful service returns with a `return_code` value of `0`. Negative `return_code` values indicate that the service failed. Positive `return_code` values indicate that the service succeeded with additional information.

Table 7.5.3: Return codes of the SilhouetteMatch component services

Code	Description
0	Success
-1	An invalid argument was provided
-3	An internal timeout occurred, e.g. during object detection
-4	Data acquisition took longer than the maximum allowed time of 3.0 seconds
-7	Data could not be read or written to persistent storage
-100	An internal error occurred
-101	Detection of the base plane failed
-102	The hand-eye calibration changed since the last base-plane calibration
-103	The maximum number of regions of interest has been reached
-104	Offset between the base plane normal and the line of sight of the *rc_visard* exceeds 10 degrees
101	An existing region of interest was overwritten
102	The provided robot pose was ignored
103	The base plane was not transformed to the current sensor pose, e.g. because no robot pose was provided during base-plane calibration

The SilhouetteMatch component offers the following services.

calibrate_base_plane Triggers the calibration of the base plane, see *Base-plane calibration*. A successful base-plane calibration is stored persistently on the *rc_visard* and returned by this service. The base-plane calibration is persistent over firmware updates and rollbacks.

All images used by the service are guaranteed to be newer than the service trigger time.

This service requires the following arguments:

```
{
  "offset": "float64",
  "plane": {
```

(continues on next page)

(continued from previous page)

```
    "distance": "float64",
    "normal": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  },
  "plane_estimation_method": "string",
  "pose_frame": "string",
  "region_of_interest_2d_id": "string",
  "robot_pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  }
}
```

This service returns the following response:

```
{
  "plane": {
    "distance": "float64",
    "normal": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "pose_frame": "string"
  },
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```

Required arguments:

`plane_estimation_method`: method to use for base-plane calibration. Valid values are `STEREO`, `APRILTAG`, `MANUAL`.

`pose_frame`: see *Hand-eye calibration*.

Potentially required arguments:

`plane` if `plane_estimation_method` is `MANUAL`: plane that will be set as base-plane calibration.

`robot_pose`: see *Hand-eye calibration*.

Optional arguments:

`offset`: offset in meters by which the estimated plane will be shifted towards the camera.

`get_base_plane_calibration` Returns the configured base-plane calibration.

This service requires the following arguments:

```
{
  "pose_frame": "string",
  "robot_pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  }
}
```

This service returns the following response:

```
{
  "plane": {
    "distance": "float64",
    "normal": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "pose_frame": "string"
  },
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

Required arguments:

`pose_frame`: see *Hand-eye calibration*.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration*.

`delete_base_plane_calibration` Deletes the configured base-plane calibration.

This service requires no arguments.

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`set_region_of_interest_2d` Persistently stores a 2D region of interest on the *rc_visard*. All configured 2D regions of interest are persistent over firmware updates and rollbacks.

See *Setting a region of interest* for the definition of the 2D region of interest type.

This service requires the following arguments:

```
{
  "region_of_interest_2d": {
    "height": "uint32",
    "id": "string",
    "offset_x": "uint32",
    "offset_y": "uint32",
    "width": "uint32"
  }
}
```

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`get_regions_of_interest_2d` Returns the configured 2D regions of interest with the requested `region_of_interest_2d_ids`. If no `region_of_interest_2d_ids` are provided, all configured 2D regions of interest are returned.

This service requires the following arguments:

```
{
  "region_of_interest_2d_ids": [
    "string"
  ]
}
```

This service returns the following response:

```
{
  "regions_of_interest": [
    {
      "height": "uint32",
      "id": "string",
      "offset_x": "uint32",
      "offset_y": "uint32",
      "width": "uint32"
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`delete_regions_of_interest_2d` Deletes the configured 2D regions of interest with the requested `region_of_interest_2d_ids`. All 2D regions of interest to be deleted must be explicitly specified in `region_of_interest_2d_ids`.

This service requires the following arguments:

```
{
  "region_of_interest_2d_ids": [
    "string"
  ]
}
```

This service returns the following response:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

`detect_object` Triggers an object detection and returns the pose of all found object instances. The maximum number of returned instances can be controlled with the `max_number_of_detected_objects` parameter.

All images used by the service are guaranteed to be newer than the service trigger time.

See *Detection of objects* for more details about the object detection.

This service requires the following arguments:

```
{
  "object_to_detect": {
    "object_id": "string",
    "region_of_interest_2d_id": "string"
  },
  "offset": "float64",
  "pose_frame": "string",
  "robot_pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  }
}
```

This service returns the following response:

```
{
  "instances": [
    {
      "id": "string",
      "object_id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      }
```

(continues on next page)

(continued from previous page)

```
    }
  ],
  "object_id": "string",
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```

Required arguments:

`object_id` in `object_to_detect`: ID of the template which should be detected.

`pose_frame`: see *Hand-eye calibration*.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration*.

Optional arguments:

`offset`: offset in meters by which the base-plane calibration will be shifted towards the camera.

`save_parameters` **(*Save*)** This service saves the currently set parameters persistently. Thereby, the same parameters will still apply after a reboot of the sensor. The node parameters are not persistent over firmware updates and rollbacks.

`reset_to_defaults` **(*Reset*)** This service resets all parameters of the component to its default values, as listed in above table. The reset does not apply to regions of interest and base-plane calibration.

7.5.8 Template Upload

For template upload, download and listing, special REST-API endpoints are provided. Up to 50 templates can be stored persistently on the *rc_visard*.

`GET /nodes/rc_silhouettematch/templates`

Get list of all rc_silhouettematch templates.

Template request

```
GET /api/v1/nodes/rc_silhouettematch/templates HTTP/1.1
Host: <rcvisard>
```

Template response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "id": "string"
  }
]
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns array of Template)*
- 404 Not Found – node not found

Referenced Data Models

- *Template* (Section 8.2.3)

GET /nodes/rc_silhouettematch/templates/{id}

Get an rc_silhouettematch template. If the requested content-type is application/octet-stream, the template is returned as file.

Template request

```
GET /api/v1/nodes/rc_silhouettematch/templates/<id> HTTP/1.1
Host: <rcvisard>
```

Template response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "string"
}
```

Parameters

- **id** (`string`) – id of the template *(required)*

Response Headers

- Content-Type – application/json application/octet-stream

Status Codes

- 200 OK – successful operation *(returns Template)*
- 404 Not Found – node or template not found

Referenced Data Models

- *Template* (Section 8.2.3)

PUT /nodes/rc_silhouettematch/templates/{id}

Create or update an rc_silhouettematch template.

Template request

```
PUT /api/v1/nodes/rc_silhouettematch/templates/<id> HTTP/1.1
Host: <rcvisard>
Accept: multipart/form-data application/json
```

Template response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "string"
}
```

Parameters

- **id** (`string`) – id of the template *(required)*

Form Parameters

- **file** – template file *(required)*

Request Headers

- Accept – multipart/form-data application/json

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns Template)*
- 404 Not Found – node or template not found
- 403 Forbidden – forbidden, e.g. because there is no valid license for this component.
- 413 Request Entity Too Large – Template too large
- 400 Bad Request – Template is not valid or max number of templates reached

Referenced Data Models

- *Template* (Section 8.2.3)

`DELETE /nodes/rc_silhouettematch/templates/{id}`

Remove an rc_silhouettematch template.

Template request

```
DELETE /api/v1/nodes/rc_silhouettematch/templates/<id> HTTP/1.1
Host: <rcvisard>
Accept: application/json
```

Parameters

- **id** (*string*) – id of the template *(required)*

Request Headers

- Accept – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation
- 404 Not Found – node or template not found
- 403 Forbidden – forbidden, e.g. because there is no valid license for this component.

8 Interfaces

Four interfaces are provided for configuring and obtaining data from the *rc_visard*:

1. *GigE Vision 2.0/GenICam* (Section 8.1)

   Images and camera related configuration.

2. *REST API* (Section 8.2)

   API to configure the *rc_visard*, query status information, request streams, etc.

3. *rc_dynamics streams* (Section 8.3)

   Real-time streams containing state estimates with poses, velocities, etc. are provided over the *rc_dynamics* interface. It sends *protobuf*-encoded messages via UDP.

4. *Ethernet KRL Interface (EKI)* (Section 8.4)

   API to configure the *rc_visard* and do service calls from KUKA KSS robots.

8.1 GigE Vision 2.0/GenICam image interface

Gigabit Ethernet for Machine Vision ("GigE Vision®" for short) is an industrial camera interface standard based on UDP/IP (see http://www.gigevision.com). The *rc_visard* is a GigE Vision® version 2.0 device and is hence compatible with all GigE Vision® 2.0 compliant frameworks and libraries.

GigE Vision® uses GenICam to describe the camera/device features. For more information about this *Generic Interface for Cameras* see http://www.genicam.org/.

Via this interface the *rc_visard* provides features such as

- discovery,
- IP configuration,
- configuration of camera related parameters,
- image grabbing, and
- time synchronization via IEEE 1588-2008 PrecisionTimeProtocol (PTPv2).

Note: The *rc_visard* supports jumbo frames of up to 9000 bytes. Setting an MTU of 9000 on your GigE Vision client side is recommended for best performance.

Note: Roboception provides tools and a C++ API with examples for discovery, configuration, and image streaming via the GigE Vision/GenICam interface. See http://www.roboception.com/download.

8.1.1 Important GenICam parameters

The following list gives an overview of the relevant GenICam features of the *rc_visard* that can be read and/or changed via the GenICam interface. In addition to the standard parameters, which are defined in the Standard Feature Naming Convention (SFNC, see http://www.emva.org/standards-technology/genicam/genicam-downloads/), *rc_visard* devices also offer custom parameters that account for special features of the *Stereo camera* (Section 6.1) and the *Stereo matching* (Section 6.2) component.

Important standard GenICam features

Category: ImageFormatControl

`ComponentSelector`

- type: Enumeration, one of `Intensity`, `IntensityCombined`, `Disparity`, `Confidence`, or `Error`
- default: -
- description: Allows the user to select one of the five image streams for configuration (see *Chunk data*, Section 8.1.1).

`ComponentIDValue` (read-only)

- type: Integer
- description: The ID of the image stream selected by the `ComponentSelector`.

`ComponentEnable`

- type: Boolean
- default: -
- description: If set to `true`, it enables the image stream selected by `ComponentSelector`; otherwise, it disables the stream. Using `ComponentSelector` and `ComponentEnable`, individual image streams can be switched on and off.

`Width` (read-only)

- type: Integer
- description: Image width in pixel of image stream that is currently selected by `ComponentSelector`.

`Height` (read-only)

- type: Integer
- description: Image height in pixel of image stream that is currently selected by `ComponentSelector`.

`WidthMax` (read-only)

- type: Integer
- description: Maximum width of an image, which is always 1280 pixels.

`HeightMax` (read-only)

- type: Integer
- description: Maximum height of an image in the streams. This is always 1920 pixels due to the stacked left and right images in the `IntensityCombined` stream (see *Chunk data*, Section 8.1.1).

`PixelFormat`

- type: Enumeration with some of `Mono8`, `YCbCr411_8` (color sensors only), `Coord3D_C16`, `Confidence8` and `Error8`
- description: Pixel format of the selected component. The enumeration only permits to choose the format among the possibly formats for the selected component. For a color sensor, `Mono8` or `YCbCr411_8` can be chosen for the `Intensity` and `IntensityCombined` component.

Category: AcquisitionControl

`AcquisitionFrameRate`

- type: Float, ranges from 1 Hz to 25 Hz
- default: 25 Hz
- description: Frame rate of the camera (*FPS*, Section 6.1.3).

`ExposureAuto`

- type: Enumeration, one of `Continuous` or `Off`
- default: `Continuous`
- description: Can be set to `Off` for manual exposure mode or to `Continuous` for auto exposure mode (*Exposure*, Section 6.1.3).

`ExposureTime`

- type: Float, ranges from 66 µs to 18000 µs
- default: 5000 µs
- description: The cameras' exposure time in microseconds for the manual exposure mode (*Manual*, Section 6.1.3).

Category: AnalogControl

`GainSelector` (read-only)

- type: Enumeration, is always `All`
- default: `All`
- description: The *rc_visard* currently supports only one overall gain setting.

`Gain`

- type: Float, ranges from 0 dB to 18 dB
- default: 0 dB
- description: The cameras' gain value in decibel that is used in manual exposure mode (*Gain*, Section 6.1.3).

`BalanceWhiteAuto` (color sensors only)

- type: Enumeration, one of `Continuous` or `Off`
- default: `Continuous`
- description: Can be set to `Off` for manual white balancing mode or to `Continuous` for auto white balancing. This feature is only available on color sensors (*wb_auto*, Section 6.1.3).

`BalanceRatioSelector` (color sensors only)

- type: Enumeration, one of `Red` or `Blue`
- default: `Red`
- description: Selects ratio to be modified by `BalanceRatio`. `Red` means red to green ratio and `Blue` means blue to green ratio. This feature is only available on color sensors.

`BalanceRatio` (color sensors only)

- type: Float, ranges from 0.125 to 8
- default: 1.2 if `Red` and 2.4 if `Blue` is selected in `BalanceRatioSelector`

- description: Weighting of red or blue to green color channel. This feature is only available on color sensors (*wb_ratio*, Section 6.1.3).

Category: DigitalIOControl

Note: If IOControl license is not available, then the outputs will be configured according to the factory defaults and cannot be changed. The inputs will always return the logic value false, regardless of the signals on the physical inputs.

`LineSelector`

- type: Enumeration, one of `Out1`, `Out2`, `In1` or `In2`
- default: `Out1`
- description: Selects the input or output line for getting the current status or setting the source.

`LineStatus` (read-only)

- type: Boolean
- description: Current status of the line selected by `LineSelector`.

`LineStatusAll` (read-only)

- type: Integer
- description: Current status of GPIO inputs and outputs represented in the lowest four bits.

Table 8.1.1: Meaning of bits of `LineStatusAll` field.

Bit	4	3	2	1
GPIO	In 2	In 1	Out 2	Out 1

`LineSource` (read-only if IOControl component is not licensed)

- type: Enumeration, one of `ExposureActive`, `ExposureAlternateActive`, `Low` or `High`
- default: `ExposureActive` for `Out1` and `Low` for `Out2`
- description: Mode for output line selected by `LineSelector` as described in the IOControl module (*out1_mode and out2_mode*, Section 7.2.1). See also parameter `AcquisitionAlternateFilter` for filtering images in `ExposureAlternateActive` mode.

Category: TransportLayerControl / PtpControl

`PtpEnable`

- type: Boolean
- default: `false`
- description: Switches PTP synchronization on and off.

Category: Scan3dControl

`Scan3dDistanceUnit` (read-only)

- type: Enumeration, is always `Pixel`
- description: Unit for the disparity measurements, which is always `Pixel`.

`Scan3dOutputMode` (read-only)

- type: Enumeration, is always `DisparityC`
- description: Mode for the depth measurements, which is always `DisparityC`.

`Scan3dFocalLength` (read-only)

- type: Float
- description: Focal length in pixel of image stream selected by `ComponentSelector`. In case of the component `Disparity`, `Confidence` and `Error`, the value also depends on the resolution that is implicitely selected by `DepthQuality`.

`Scan3dBaseline` (read-only)

- type: Float
- description: Baseline of the stereo camera in meter.

`Scan3dPrinciplePointU` (read-only)

- type: Float
- description: Horizontal location of the principle point in pixel of image stream selected by `ComponentSelector`. In case of the component `Disparity`, `Confidence` and `Error`, the value also depends on the resolution that is implicitely selected by `DepthQuality`.

`Scan3dPrinciplePointV` (read-only)

- type: Float
- description: Vertical location of the principle point in pixel of image stream selected by `ComponentSelector`. In case of the component `Disparity`, `Confidence` and `Error`, the value also depends on the resolution that is implicitely selected by `DepthQuality`.

`Scan3dCoordinateScale` (read-only)

- type: Float
- description: The scale factor that has to be multiplied with the disparity values in the disparity image stream to get the actual disparity measurements. This value is always 0.0625.

`Scan3dCoordinateOffset` (read-only)

- type: Float
- description: The offset that has to be added to the disparity values in the disparity image stream to get the actual disparity measurements. For the *rc_visard*, this value is always 0 and can therefore be disregarded.

`Scan3dInvalidDataFlag` (read-only)

- type: Boolean
- description: Is always `true`, which means that invalid data in the disparity image is marked by a specific value defined by the `Scan3dInvalidDataValue` parameter.

`Scan3dInvalidDataValue` (read-only)

- type: Float
- description: Is the value which stands for invalid disparity. This value is always 0, which means that disparity values of 0 correspond to invalid measurements. To distinguish between invalid disparity measurements and disparity measurements of 0 for objects which are infinitely far away, the *rc_visard* sets the disparity value for the latter to the smallest possible disparity value of 0.0625. This still corresponds to an object distance of several hundred meters.

Category: ChunkDataControl

`ChunkModeActive`

- type: Boolean
- default: False
- description: Enables chunk data that is delivered with every image.

Custom GenICam features of the *rc_visard*

Category: ImageFormatControl

`ExposureTimeAutoMax`

- type: Float, ranges from 66 µs to 18000 µs
- default: 7000 µs
- description: Maximal exposure time in auto exposure mode (*Auto*, Section 6.1.3).

`ExposureRegionOffsetX`

- type: Integer in the range of 0 to 1280
- default: 0
- description: Horizontal offset of *exposure region* (Section 6.1.3) in pixel.

`ExposureRegionOffsetY`

- type: Integer in the range of 0 to 960
- default: 0
- description: Vertical offset of *exposure region* (Section 6.1.3) in pixel.

`ExposureRegionWidth`

- type: Integer in the range of 0 to 1280
- default: 0
- description: Width of *exposure region* (Section 6.1.3) in pixel.

`ExposureRegionHeight`

- type: Integer in the range of 0 to 960
- default: 0
- description: Height of *exposure region* (Section 6.1.3) in pixel.

Category: AcquisitionControl

`AcquisitionAlternateFilter` (read-only if IOControl component is not licensed)

- type: Enumeration, one of `Off`, `OnlyHigh` or `OnlyLow`
- default: `Off`
- description: If this parameter is set to `OnlyHigh` (or `OnlyLow`) and the `LineSource` is set to `ExposureAlternateActive` for any output, then only camera images are delivered that are captured while the output is high, i.e. a potentially connected projector is on (or low, i.e. a potentially connected projector is off). This parameter is a simple means for only getting images without projected pattern. The minimal time difference between camera and disparity images will be about 40 ms in this case (see *IOControl*, Section 7.2.1).

`AcquisitionMultiPartMode`

- type: Enumeration, one of `SingleComponent` or `SynchronizedComponents`
- default: `SingleComponent`

- description: Only effective in MultiPart mode. If this parameter is set to `SingleComponent` the images are sent immediately as a single component per frame/buffer when they become available. This is the same behavior as when MultiPart is not supported by the client. If set to `SynchronizedComponents` all enabled components are time synchronized on the *rc_visard* and only sent (in one frame/buffer) when they are all available for that timestamp.

Category: Scan3dControl

`FocalLengthFactor` (read-only)

- type: Float
- description: The focal length scaled to an image width of 1 pixel. To get the focal length in pixels for a certain image, this value must be multiplied by the width of the received image. See also parameter `Scan3dFocalLength`.

`Baseline` (read-only)

- type: Float
- description: This parameter is deprecated. The parameter `Scan3dBaseline` should be used instead.

Category: DepthControl

`DepthAcquisitionMode`

- type: Enumeration, one of `SingleFrame`, `SingleFrameOut1` or `Continuous`
- default: `Continuous`
- description: In single frame mode, stereo matching is performed upon each call of `DepthAcquisitionTrigger`. The `SingleFrameOut1` mode can be used to control an external projector. It sets the line source of `Out1` to `ExposureAlternateActive` upon each trigger and resets it to `Low` as soon as the images for stereo matching are grabbed. However, the line source will only be changed if the IOControl license is available. In continuous mode, stereo matching is performed continuously.

`DepthAcquisitionTrigger`

- type: Command
- description: This command triggers stereo matching of the next available stereo image pair, if `DepthAcquisitionMode` is set to `SingleFrame` or `SingleFrameOut1`.

`DepthQuality`

- type: Enumeration, one of `Low`, `Medium`, `High`, or `Full` **(only with StereoPlus license)**
- default: `High`
- description: Quality of disparity images. Lower quality results in disparity images with lower resolution (*Quality*, Section 6.2.4).

`DepthStaticScene`

- type: Boolean
- default: `False`
- description: True for averaging 8 consecutive camera images for improving the stereo matching result. (*Static*, Section 6.2.4).

`DepthDispRange`

- type: Integer, ranges from 32 pixels to 512 pixels
- default: 256 pixels

- description: Maximum disparity value in pixels (*Disparity Range*, Section 6.2.4).

`DepthSmooth` (read-only if StereoPlus license is not available)

- type: Boolean
- default: `False`
- description: True for advanced smoothing of disparity values. (*Smoothing*, Section 6.2.4).

`DepthFill`

- type: Integer, ranges from 0 pixel to 4 pixels
- default: 3 pixels
- description: Value in pixels for *Fill-In* (Section 6.2.4).

`DepthSeg`

- type: Integer, ranges from 0 pixel to 4000 pixels
- default: 200 pixels
- description: Value in pixels for *Segmentation* (Section 6.2.4).

`DepthMedian`

- type: Integer, ranges from 1 pixel to 5 pixels
- default: 1 pixel
- description: Value in pixels for *Median* filter smoothing (Section 6.2.4).

`DepthMinConf`

- type: Float, ranges from 0.0 to 1.0
- default: 0.0
- description: Value for *Minimum Confidence* filtering (Section 6.2.4).

`DepthMinDepth`

- type: Float, ranges from 0.1 m to 100.0 m
- default: 0.1 m
- description: Value in meters for *Minimum Distance* filtering (Section 6.2.4).

`DepthMaxDepth`

- type: Float, ranges from 0.1m to 100.0 m
- default: 100.0 m
- description: Value in meters for *Maximum Distance* filtering (Section 6.2.4).

`DepthMaxDepthErr`

- type: Float, ranges from 0.01 m to 100.0 m
- default: 100.0 m
- description: Value in meters for *Maximum Depth Error* filtering (Section 6.2.4).

Chunk data

The *rc_visard* supports chunk parameters that are transmitted with every image. Chunk parameters all have the prefix `Chunk`. Their meaning equals their non-chunk counterparts, except that they belong to the corresponding image, e.g. `Scan3dFocalLength` depends on `ComponentSelector` and `DepthQuality` as both can change the image resolution. The parameter `ChunkScan3dFocalLength` that is delivered with an image fits to the resolution of the corresponding image.

Particularly useful chunk parameters are:

- `ChunkComponentSelector` selects for which component to extract the chunk data in MultiPart mode.
- `ChunkComponentID` and `ChunkComponentIDValue` provide the relation of the image to its component (e.g. camera image or disparity image) without guessing from the image format or size.
- `ChunkLineStatusAll` provides the status of all GPIOs at the time of image acquisition. See `LineStatusAll` above for a description of bits.
- `ChunkScan3d...` parameters are useful for 3D reconstruction as described in Section *Image stream conversions* (Section 8.1.3).
- `ChunkPartIndex` provides the index of the image part in this MultiPart block for the selected component (`ChunkComponentSelector`).

Chunk data is enabled by setting the GenICam parameter `ChunkModeActive` to `True`.

8.1.2 Provided image streams

The *rc_visard* provides the following five different image streams via the GenICam interface:

Component name	PixelFormat	Width×Height	Description
`Intensity`	`Mono8` (monochrome sensors) `YCbCr411_8` (color sensors)	1280×960	Left rectified camera image
`IntensityCombined`	`Mono8` (monochrome sensors) `YCbCr411_8` (color sensors)	1280×1920	Left rectified camera image stacked on right rectified camera image
`Disparity`	`Coord3D_C16`	1280×1920 640×480 320×240 214×160	Disparity image in desired resolution, i.e., `DepthQuality` of `Full`, `High`, `Medium` or `Low`
`Confidence`	`Confidence8`	same as `Disparity`	Confidence image
`Error`	`Error8` (custom: 0x81080001)	same as `Disparity`	Disparity error image

Each image comes with a buffer timestamp and the *PixelFormat* given in the above table. This PixelFormat should be used to distinguish between the different image types. Images belonging to the same acquisition timestamp can be found by comparing the GenICam buffer timestamps.

8.1.3 Image stream conversions

The disparity image contains 16 bit unsigned integer values. These values must be multiplied by the scale value given in the GenICam feature *Scan3dCoordinateScale* to get the disparity values d in pixels. To compute the 3D object coordinates from the disparity values, the focal length and the baseline as well as the principle point are required. These parameters are transmitted as GenICam features *Scan3dFocalLength*, *Scan3dBaseline*, *Scan3dPrincipalPointU* and *Scan3dPrincipalPointV*. The focal length and principal point depend on the image resolution of the selected component. Knowing these values, the pixel coordinates and the disparities can be transformed into 3D object coordinates in the *sensor coordinate frame* (Section 3.7) using the equations described in *Computing depth images and point clouds* (Section 6.2.2).

Assuming that d_{ik} is the 16 bit disparity value at column i and row k of a disparity image, the 3D reconstruction in meters can be written with the GenICam parameters as

$$\begin{aligned}
P_x &= (i - \text{Scan3dPrincipalPointU})\,\frac{\text{Scan3dBaseline}}{d_{ik} \cdot \text{Scan3dCoordinateScale}},\\
P_y &= (k - \text{Scan3dPrincipalPointV})\,\frac{\text{Scan3dBaseline}}{d_{ik} \cdot \text{Scan3dCoordinateScale}},\\
P_z &= \text{Scan3dFocalLength}\,\frac{\text{Scan3dBaseline}}{d_{ik} \cdot \text{Scan3dCoordinateScale}}.
\end{aligned}$$

The confidence image contains 8 bit unsigned integer values. These values have to be divided by 255 to get the confidence as value between 0 an 1.

The error image contains 8 bit unsigned integer values. The error e_{ik} must be multiplied by the scale value given in the GenICam feature *Scan3dCoordinateScale* to get the disparity-error values d_{eps} in pixels. According to the description in *Confidence and error images* (Section 6.2.3), the depth error z_{eps} in meters can be computed with GenICam parameters as

$$z_{eps} = \frac{e_{ik} \cdot \text{Scan3dCoordinateScale} \cdot \text{Scan3dFocalLength} \cdot \text{Scan3dBaseline}}{(d_{ik} \cdot \text{Scan3dCoordinateScale})^2}.$$

Note: It is preferable to enable chunk data with the parameter *ChunkModeActive* and to use the chunk parameters *ChunkScan3dCoordinateScale*, *ChunkScan3dFocalLength*, *ChunkScan3dBaseline*, *ChunkScan3dPrincipalPointU* and *ChunkScan3dPrincipalPointV* that are delivered with every image, because their values already fit to the image resolution of the corresponding image.

For more information about disparity, error, and confidence images, please refer to *Stereo matching* (Section 6.2).

8.2 REST-API interface

Besides the *GenICam interface* (Section 8.1), the *rc_visard* offers a comprehensive RESTful web interface (REST-API) which any HTTP client or library can access. Whereas most of the provided parameters, services, and functionalities can also be accessed via the user-friendly *Web GUI* (Section 4.5), the REST-API serves rather as a machine-to-machine interface to programmatically

- set and get run-time parameters of computation nodes, e.g., of cameras, disparity calculation, and visual odometry;
- do service calls, e.g., to start and stop individual computational nodes, or to use offered services such as the hand-eye calibration;
- configure data streams that provide *rc_visard*'s *dynamic state estimates* (Section 6.3.2) as described in the *rc_dynamics interface* (Section 8.3);
- read the current state of the system and individual computational nodes; and
- update the *rc_visard*'s firmware or license.

Note: In the *rc_visard*'s REST-API, a *node* is a computational component that bundles certain algorithmic functionality and offers a holistic interface (parameters, services, current status). Examples for such nodes are the stereo matching node or the visual odometry node.

8.2.1 General API structure

The general **entry point** to the *rc_visard*'s API is `http://<rcvisard>/api/`, where `<rcvisard>` is either the device's IP address or its host name as known by the respective DHCP server, as explained in *network configura-*

tion (Section 4.3). Accessing this entry point with a web browser lets the user explore and test the full API during run-time using the *Swagger UI* (Section 8.2.4).

For actual HTTP requests, the **current API version is appended** to the entry point of the API, i.e., `http://<rcvisard>/api/v1`. All data sent to and received by the REST-API follows the JavaScript Object Notation (JSON). The API is designed to let the user **create, retrieve, modify, and delete** so-called **resources** as listed in *Available resources and requests* (Section 8.2.2) using the HTTP requests below.

Request type	Description
GET	Access one or more resources and return the result as JSON.
PUT	Modify a resource and return the modified resource as JSON.
DELETE	Delete a resource.
POST	Upload file (e.g., license or firmware image).

Depending on the type and the specific request itself, **arguments** to HTTP requests can be transmitted as part of the **path** (*URI*) to the resource, as **query** string, as **form data**, or in the **body** of the request. The following examples use the command line tool *curl*, which is available for various operating systems. See https://curl.haxx.se.

- Get a node's current status; its name is encoded in the path (URI)

```
curl -X GET 'http://<rcvisard>/api/v1/nodes/rc_stereomatching'
```

- Get values of some of a node's parameters using a query string

```
curl -X GET 'http://<rcvisard>/api/v1/nodes/rc_stereomatching/parameters?name=minconf&
↪name=maxdepth'
```

- Configure a new datastream; the destination parameter is transmitted as form data

```
curl -X PUT --header 'Content-Type: application/x-www-form-urlencoded' -d 'destination=10.0.
↪1.14%3A30000' 'http://<rcvisard>/api/v1/datastreams/pose'
```

- Set a node's parameter as JSON-encoded text in the body of the request

```
curl -X PUT --header 'Content-Type: application/json' -d '[{"name": "mindepth", "value": 0.
↪1}]' 'http://<rcvisard>/api/v1/nodes/rc_stereomatching/parameters'
```

As for the responses to such requests, some common return codes for the *rc_visard*'s API are:

Status Code	Description
`200 OK`	The request was successful; the resource is returned as JSON.
`400 Bad Request`	A required attribute or argument of the API request is missing or invalid.
`404 Not Found`	A resource could not be accessed; e.g., an ID for a resource could not be found.
`403 Forbidden`	Access is (temporarily) forbidden; e.g., some parameters are locked while a GigE Vision application is connected.
`429 Too many requests`	Rate limited due to excessive request frequency.

The following listing shows a sample response to a successful request that accesses information about the `rc_stereomatching` node's `minconf` parameter:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 157

{
    "name": "minconf",
    "min": 0,
    "default": 0,
    "max": 1,
    "value": 0,
    "type": "float64",
    "description": "Minimum confidence"
}
```

Note: The actual behavior, allowed requests, and specific return codes depend heavily on the specific resource, context, and action. Please refer to the *rc_visard*'s *available resources* (Section 8.2.2) and to each *software component's* (Section 6) parameters and services.

8.2.2 Available resources and requests

The available REST-API resources are structured into the following parts:

- `/nodes`: Access the *rc_visard*'s *software components* (Section 6) with their run-time status, parameters, and offered services.
- `/datastreams`: Access and manage data streams of the *rc_visard*'s *The rc_dynamics interface* (Section 8.3).
- `/logs`: Access the log files on the *rc_visard*.
- `/system`: Access the system state and manage licenses as well as firmware updates.

Nodes, parameters, and services

Nodes represent the *rc_visard*'s *software components* (Section 6), each bundling a certain algorithmic functionality. All available REST-API nodes can be listed with their service calls and parameters using

```
curl -X GET http://<rcvisard>/api/v1/nodes
```

Information about a specific node (e.g., `rc_stereocamera`) can be retrieved using

```
curl -X GET http://<rcvisard>/api/v1/nodes/rc_stereocamera
```

Status: During run-time, each node offers information about its current status. This includes not only the current **processing status** of the component (e.g., `running` or `stale`), but most nodes also offer run-time statistics or read-only parameters, so-called **status values**. As an example, the `rc_stereocamera` values can be retrieved using

```
curl -X GET http://<rcvisard>/api/v1/nodes/rc_stereocamera/status
```

Note: The returned **status values** are specific to individual nodes and are documented in the respective *software component* (Section 6).

Note: The **status values** are only reported when the respective node is in the `running` state.

Parameters: Most nodes expose parameters via the *rc_visard*'s REST-API to allow their run-time behaviors to be changed according to application context or requirements. The REST-API permits to read and write a parameter's value, but also provides further information such as minimum, maximum, and default values.

As an example, the `rc_stereomatching` parameters can be retrieved using

```
curl -X GET http://<rcvisard>/api/v1/nodes/rc_stereomatching/parameters
```

Its `median` parameter could be set to 3 using

```
curl -X PUT --header 'Content-Type: application/json' -d '{ "value": 3 }' http://<rcvisard>/
↪api/v1/nodes/rc_stereomatching/parameters/median
```

Note: Run-time parameters are specific to individual nodes and are documented in the respective *software component* (Section 6).

Note: Most of the parameters that nodes offer via the REST-API can be explored and tested via the *rc_visard*'s user-friendly *Web GUI* (Section 4.5).

Note: Some parameters exposed via the *rc_visard*'s REST-API are also available from the *GigE Vision 2.0/GenICam image interface* (Section 8.1). Please note that setting those parameters via the REST-API is prohibited if a GenICam client is connected.

In addition, each node that offers run-time parameters also features services to save, i.e., persist, the current parameter setting, or to restore the default values for all of its parameters.

Services: Some nodes also offer services that can be called via REST-API, e.g., to save and restore parameters as discussed above, or to start and stop nodes. As an example, the services of pose estimation (see *Stereo INS*, Section 6.5), could be listed using

```
curl -X GET http://<rcvisard>/api/v1/nodes/rc_stereo_ins/services
```

A node's service is called by issuing a `PUT` request for the respective resource and providing the service-specific arguments (see the `"args"` field of the *Service data model*, Section 8.2.3). As an example, egomotion estimation can be switched on by:

```
curl -X PUT --header 'Content-Type: application/json' -d '{ "args": {}  }' http://<rcvisard>
↪/api/v1/nodes/rc_dynamics/services/start
```

Note: The services and corresponding argument data models are specific to individual nodes and are documented in the respective *software component* (Section 6).

The following list includes all REST-API requests regarding the node's status, parameters, and services calls:

`GET /nodes`

Get list of all available nodes.

Template request

```
GET /api/v1/nodes HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "name": "rc_stereocalib",
    "parameters": [
      "grid_width",
      "grid_height",
```

(continues on next page)

(continued from previous page)

```
      "snap"
    ],
    "services": [
      "save_parameters",
      "reset_defaults",
      "change_state"
    ],
    "status": "stale"
  },
  {
    "name": "rc_stereocamera",
    "parameters": [
      "fps",
      "exp_auto",
      "exp_value",
      "exp_max"
    ],
    "services": [
      "save_parameters",
      "reset_defaults"
    ],
    "status": "running"
  },
  {
    "name": "rc_hand_eye_calibration",
    "parameters": [
      "grid_width",
      "grid_height",
      "robot_mounted"
    ],
    "services": [
      "save_parameters",
      "reset_defaults",
      "set_pose",
      "reset",
      "save",
      "calibrate",
      "get_calibration"
    ],
    "status": "stale"
  },
  {
    "name": "rc_stereo_ins",
    "parameters": [],
    "services": [],
    "status": "stale"
  },
  {
    "name": "rc_stereomatching",
    "parameters": [
      "force_on",
      "quality",
      "disprange",
      "seg",
      "median",
      "fill",
      "minconf",
      "mindepth",
      "maxdepth",
      "maxdeptherr"
    ],
```

(continues on next page)

(continued from previous page)

```
    "services": [
      "save_parameters",
      "reset_defaults"
    ],
    "status": "running"
  },
  {
    "name": "rc_stereovisodo",
    "parameters": [
      "disprange",
      "nkey",
      "ncorner",
      "nfeature"
    ],
    "services": [
      "save_parameters",
      "reset_defaults"
    ],
    "status": "stale"
  }
]
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns array of NodeInfo)*

Referenced Data Models

- *NodeInfo* (Section 8.2.3)

GET /nodes/{node}

Get info on a single node.

Template request

```
GET /api/v1/nodes/<node> HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "name": "rc_stereocamera",
  "parameters": [
    "fps",
    "exp_auto",
    "exp_value",
    "exp_max"
  ],
  "services": [
    "save_parameters",
    "reset_defaults"
  ],
  "status": "running"
}
```

Parameters

- **node** (*string*) – name of the node *(required)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns NodeInfo)*
- 404 Not Found – node not found

Referenced Data Models

- *NodeInfo* (Section 8.2.3)

GET /nodes/{node}/parameters

Get parameters of a node.

Template request

```
GET /api/v1/nodes/<node>/parameters?name=<name> HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "default": 25,
    "description": "Frames per second in Hz",
    "max": 25,
    "min": 1,
    "name": "fps",
    "type": "float64",
    "value": 25
  },
  {
    "default": true,
    "description": "Switching between auto and manual exposure",
    "max": true,
    "min": false,
    "name": "exp_auto",
    "type": "bool",
    "value": true
  },
  {
    "default": 0.007,
    "description": "Maximum exposure time in s if exp_auto is true",
    "max": 0.018,
    "min": 6.6e-05,
    "name": "exp_max",
    "type": "float64",
    "value": 0.007
  }
]
```

Parameters

- **node** (*string*) – name of the node *(required)*

Query Parameters

- **name** (*string*) – limit result to parameters with name *(optional)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns array of Parameter)*
- 404 Not Found – node not found

Referenced Data Models

- *Parameter* (Section 8.2.3)

PUT /nodes/{node}/parameters

Update multiple parameters.

Template request

```
PUT /api/v1/nodes/<node>/parameters HTTP/1.1
Host: <rcvisard>
Accept: application/json

[
  {
    "name": "string",
    "value": {}
  }
]
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "default": 25,
    "description": "Frames per second in Hz",
    "max": 25,
    "min": 1,
    "name": "fps",
    "type": "float64",
    "value": 10
  },
  {
    "default": true,
    "description": "Switching between auto and manual exposure",
    "max": true,
    "min": false,
    "name": "exp_auto",
    "type": "bool",
    "value": false
  },
  {
    "default": 0.005,
    "description": "Manual exposure time in s if exp_auto is false",
    "max": 0.018,
    "min": 6.6e-05,
    "name": "exp_value",
    "type": "float64",
    "value": 0.005
  }
]
```

Parameters

- **node** (*string*) – name of the node *(required)*

Request JSON Array of Objects

- **parameters** (*Parameter*) – array of parameters *(required)*

Request Headers

- Accept – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns array of Parameter)*
- 404 Not Found – node not found
- 403 Forbidden – Parameter update forbidden, e.g. because they are locked by a running GigE Vision application or there is no valid license for this component.
- 400 Bad Request – invalid parameter value

Referenced Data Models

- *Parameter* (Section 8.2.3)

GET /nodes/{node}/parameters/{param}

Get a specific parameter of a node.

Template request

```
GET /api/v1/nodes/<node>/parameters/<param> HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "default": "H",
  "description": "Quality, i.e. H, M or L",
  "max": "",
  "min": "",
  "name": "quality",
  "type": "string",
  "value": "H"
}
```

Parameters

- **node** (*string*) – name of the node *(required)*
- **param** (*string*) – name of the parameter *(required)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns Parameter)*
- 404 Not Found – node or parameter not found

Referenced Data Models

- *Parameter* (Section 8.2.3)

PUT /nodes/{node}/parameters/{param}

Update a specific parameter of a node.

Template request

```
PUT /api/v1/nodes/<node>/parameters/<param> HTTP/1.1
Host: <rcvisard>
Accept: application/json

{
  "name": "string",
  "value": {}
}
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "default": "H",
  "description": "Quality, i.e. H, M or L",
  "max": "",
  "min": "",
  "name": "quality",
  "type": "string",
  "value": "M"
}
```

Parameters

- **node** (*string*) – name of the node *(required)*
- **param** (*string*) – name of the parameter *(required)*

Request JSON Object

- **parameter** (*Parameter*) – parameter to be updated as JSON object *(required)*

Request Headers

- Accept – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns Parameter)*
- 404 Not Found – node or parameter not found
- 403 Forbidden – Parameter update forbidden, e.g. because they are locked by a running GigE Vision application or there is no valid license for this component.
- 400 Bad Request – invalid parameter value

Referenced Data Models

- *Parameter* (Section 8.2.3)

GET /nodes/{node}/services

Get descriptions of all services a node offers.

Template request

```
GET /api/v1/nodes/<node>/services HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "args": {},
    "description": "Restarts the component.",
    "name": "restart",
    "response": {
      "accepted": "bool",
      "current_state": "string"
    }
  },
  {
    "args": {},
    "description": "Starts the component.",
    "name": "start",
    "response": {
      "accepted": "bool",
      "current_state": "string"
    }
  },
  {
    "args": {},
    "description": "Stops the component.",
    "name": "stop",
    "response": {
      "accepted": "bool",
      "current_state": "string"
    }
  }
]
```

Parameters

- **node** (`string`) – name of the node *(required)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns array of Service)*
- 404 Not Found – node not found

Referenced Data Models

- *Service* (Section 8.2.3)

GET /nodes/{node}/services/{service}

Get description of a node's specific service.

Template request

```
GET /api/v1/nodes/<node>/services/<service> HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "args": {
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "slot": "int32"
  },
  "description": "Save a pose (grid or gripper) for later calibration.",
  "name": "set_pose",
  "response": {
    "message": "string",
    "status": "int32",
    "success": "bool"
  }
}
```

Parameters

- **node** (*string*) – name of the node *(required)*
- **service** (*string*) – name of the service *(required)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns Service)*
- 404 Not Found – node or service not found

Referenced Data Models

- *Service* (Section 8.2.3)

PUT /nodes/{node}/services/{service}

Call a service of a node. The required args and resulting response depend on the specific node and service.

Template request

```
PUT /api/v1/nodes/<node>/services/<service> HTTP/1.1
Host: <rcvisard>
Accept: application/json

{
  "args": {}
}
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "name": "set_pose",
  "response": {
    "message": "Grid detected, pose stored.",
    "status": 1,
    "success": true
  }
}
```

Parameters

- **node** (*string*) – name of the node *(required)*
- **service** (*string*) – name of the service *(required)*

Request JSON Object

- **service args** (*Service*) – example args *(required)*

Request Headers

- Accept – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns Service)*
- 404 Not Found – node or service not found
- 403 Forbidden – Service call forbidden, e.g. because there is no valid license for this component.

Referenced Data Models

- *Service* (Section 8.2.3)

GET /nodes/{node}/status

Get status of a node.

Template request

```
GET /api/v1/nodes/<node>/status HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "running",
  "timestamp": 1503075030.2335997,
  "values": {
    "baseline": "0.0650542",
    "color": "0",
    "exp": "0.00426667",
    "focal": "0.844893",
    "fps": "25.1352",
    "gain": "12.0412",
```

(continues on next page)

(continued from previous page)

```
    "height": "960",
    "temp_left": "39.6",
    "temp_right": "38.2",
    "time": "0.00406513",
    "width": "1280"
  }
}
```

Parameters

- **node** (*string*) – name of the node *(required)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns NodeStatus)*
- 404 Not Found – node not found

Referenced Data Models

- *NodeStatus* (Section 8.2.3)

Datastreams

The following resources and requests allow access to and configuration of the *The rc_dynamics interface* data streams (Section 8.3). These REST-API requests offer

- showing available and currently running data streams, e.g.,

```
curl -X GET http://<rcvisard>/api/v1/datastreams
```

- starting a data stream to a destination, e.g.,

```
curl -X PUT --header 'Content-Type: application/x-www-form-urlencoded' -d 'destination=
↪<target-ip>:<target-port>' http://<rcvisard>/api/v1/datastreams/pose
```

- and stopping data streams, e.g.,

```
curl -X DELETE http://<rcvisard>/api/v1/datastreams/pose?destination=<target-ip>:<target-
↪port>
```

The following list includes all REST-API requests associated with data streams:

GET /datastreams

Get list of available data streams.

Template request

```
GET /api/v1/datastreams HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "description": "Pose of left camera at VisualOdometry rate (~10Hz)",
```

(continues on next page)

(continued from previous page)

```
    "destinations": [
      "192.168.1.13:30000"
    ],
    "name": "pose",
    "protobuf": "Frame",
    "protocol": "UDP"
  },
  {
    "description": "Pose of left camera (RealTime 200Hz)",
    "destinations": [
      "192.168.1.100:20000",
      "192.168.1.42:45000"
    ],
    "name": "pose_rt",
    "protobuf": "Frame",
    "protocol": "UDP"
  },
  {
    "description": "Raw IMU (InertialMeasurementUnit) values (RealTime 200Hz)",
    "destinations": [],
    "name": "imu",
    "protobuf": "Imu",
    "protocol": "UDP"
  },
  {
    "description": "Dynamics of sensor (pose, velocity, acceleration) (RealTime 200Hz)",
    "destinations": [
      "192.168.1.100:20001"
    ],
    "name": "dynamics",
    "protobuf": "Dynamics",
    "protocol": "UDP"
  }
]
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns array of Stream)*

Referenced Data Models

- *Stream* (Section 8.2.3)

GET /datastreams/{stream}

Get datastream configuration.

Template request

```
GET /api/v1/datastreams/<stream> HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "description": "Pose of left camera at VisualOdometry rate (~10Hz)",
```

(continues on next page)

(continued from previous page)

```
  "destinations": [
    "192.168.1.13:30000"
  ],
  "name": "pose",
  "protobuf": "Frame",
  "protocol": "UDP"
}
```

Parameters

- **stream** (*string*) – name of the stream *(required)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns Stream)*
- 404 Not Found – datastream not found

Referenced Data Models

- *Stream* (Section 8.2.3)

PUT /datastreams/{stream}

Update a datastream configuration.

Template request

```
PUT /api/v1/datastreams/<stream> HTTP/1.1
Host: <rcvisard>
Accept: application/x-www-form-urlencoded
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "description": "Pose of left camera at VisualOdometry rate (~10Hz)",
  "destinations": [
    "192.168.1.13:30000",
    "192.168.1.25:40000"
  ],
  "name": "pose",
  "protobuf": "Frame",
  "protocol": "UDP"
}
```

Parameters

- **stream** (*string*) – name of the stream *(required)*

Form Parameters

- **destination** – destination (“IP:port”) to add *(required)*

Request Headers

- Accept – application/x-www-form-urlencoded

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns Stream)*
- 404 Not Found – datastream not found

Referenced Data Models

- *Stream* (Section 8.2.3)

DELETE /datastreams/{stream}

Delete a destination from the datastream configuration.

Template request

```
DELETE /api/v1/datastreams/<stream>?destination=<destination> HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "description": "Pose of left camera at VisualOdometry rate (~10Hz)",
  "destinations": [],
  "name": "pose",
  "protobuf": "Frame",
  "protocol": "UDP"
}
```

Parameters

- **stream** (*string*) – name of the stream *(required)*

Query Parameters

- **destination** (*string*) – destination IP:port to delete, if not specified all destinations are deleted *(optional)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns Stream)*
- 404 Not Found – datastream not found

Referenced Data Models

- *Stream* (Section 8.2.3)

System and logs

The following resources and requests expose the *rc_visard*'s system-level API. They enable

- access to log files (system-wide or component-specific)
- access to information about the device and run-time statistics such as date, MAC address, clock-time synchronization status, and available resources;
- management of installed software licenses; and
- the *rc_visard* to be updated with a new firmware image.

GET /logs

Get list of available log files.

Template request

```
GET /api/v1/logs HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "date": 1503060035.0625782,
    "name": "rcsense-api.log",
    "size": 730
  },
  {
    "date": 1503060035.741574,
    "name": "stereo.log",
    "size": 39024
  },
  {
    "date": 1503060044.0475223,
    "name": "camera.log",
    "size": 1091
  },
  {
    "date": 1503060035.2115774,
    "name": "dynamics.log"
  }
]
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns array of LogInfo)*

Referenced Data Models

- *LogInfo* (Section 8.2.3)

GET /logs/{log}

Get a log file. Content type of response depends on parameter 'format'.

Template request

```
GET /api/v1/logs/<log>?format=<format>&limit=<limit> HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "date": 1503060035.2115774,
  "log": [
    {
```

(continues on next page)

(continued from previous page)

```
      "component": "rc_stereo_ins",
      "level": "INFO",
      "message": "Running rc_stereo_ins version 2.4.0",
      "timestamp": 1503060034.083
    },
    {
      "component": "rc_stereo_ins",
      "level": "INFO",
      "message": "Starting up communication interfaces",
      "timestamp": 1503060034.085
    },
    {
      "component": "rc_stereo_ins",
      "level": "INFO",
      "message": "Autostart disabled",
      "timestamp": 1503060034.098
    },
    {
      "component": "rc_stereo_ins",
      "level": "INFO",
      "message": "Initializing realtime communication",
      "timestamp": 1503060034.209
    },
    {
      "component": "rc_stereo_ins",
      "level": "INFO",
      "message": "Startet state machine in state IDLE",
      "timestamp": 1503060034.383
    },
    {
      "component": "rc_stereovisodo",
      "level": "INFO",
      "message": "Init stereovisodo ...",
      "timestamp": 1503060034.814
    },
    {
      "component": "rc_stereovisodo",
      "level": "INFO",
      "message": "rc_stereovisodo: Using standard VO",
      "timestamp": 1503060034.913
    },
    {
      "component": "rc_stereovisodo",
      "level": "INFO",
      "message": "rc_stereovisodo: Playback mode: false",
      "timestamp": 1503060035.132
    },
    {
      "component": "rc_stereovisodo",
      "level": "INFO",
      "message": "rc_stereovisodo: Ready",
      "timestamp": 1503060035.212
    }
  ],
  "name": "dynamics.log",
  "size": 695
}
```

Parameters

- **log** (*string*) – name of the log file *(required)*

Query Parameters

- **format** (*string*) – return log as JSON or raw (one of `json`, `raw`; default: `json`) *(optional)*
- **limit** (*integer*) – limit to last x lines in JSON format (default: `100`) *(optional)*

Response Headers

- Content-Type – text/plain application/json

Status Codes

- 200 OK – successful operation *(returns Log)*
- 404 Not Found – log not found

Referenced Data Models

- *Log* (Section 8.2.3)

GET /system

Get system information on sensor.

Template request

```
GET /api/v1/system HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "firmware": {
    "active_image": {
      "image_version": "rc_visard_v1.1.0"
    },
    "fallback_booted": true,
    "inactive_image": {
      "image_version": "rc_visard_v1.0.0"
    },
    "next_boot_image": "active_image"
  },
  "hostname": "rc-visard-02873515",
  "link_speed": 1000,
  "mac": "00:14:2D:2B:D8:AB",
  "ntp_status": {
    "accuracy": "48 ms",
    "synchronized": true
  },
  "ptp_status": {
    "master_ip": "",
    "offset": 0,
    "offset_dev": 0,
    "offset_mean": 0,
    "state": "off"
  },
  "ready": true,
  "serial": "02873515",
  "time": 1504080462.641875,
  "uptime": 65457.42
}
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns SysInfo)*

Referenced Data Models

- *SysInfo* (Section 8.2.3)

GET /system/license

Get information about licenses installed on sensor.

Template request

```
GET /api/v1/system/license HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "components": {
    "calibration": true,
    "fusion": true,
    "hand_eye_calibration": true,
    "rectification": true,
    "self_calibration": true,
    "slam": false,
    "stereo": true,
    "svo": true
  },
  "valid": true
}
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns LicenseInfo)*

Referenced Data Models

- *LicenseInfo* (Section 8.2.3)

POST /system/license

Update license on sensor with a license file.

Template request

```
POST /api/v1/system/license HTTP/1.1
Host: <rcvisard>
Accept: multipart/form-data
```

Form Parameters

- `file` – license file *(required)*

Request Headers

- Accept – multipart/form-data

Status Codes

- 200 OK – successful operation
- 400 Bad Request – not a valid license

PUT /system/reboot

Reboot the sensor.

Template request

```
PUT /api/v1/system/reboot HTTP/1.1
Host: <rcvisard>
```

Status Codes

- 200 OK – successful operation

GET /system/rollback

Get information about currently active and inactive firmware/system images on sensor.

Template request

```
GET /api/v1/system/rollback HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "active_image": {
    "image_version": "rc_visard_v1.1.0"
  },
  "fallback_booted": false,
  "inactive_image": {
    "image_version": "rc_visard_v1.0.0"
  },
  "next_boot_image": "active_image"
}
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns FirmwareInfo)*

Referenced Data Models

- *FirmwareInfo* (Section 8.2.3)

PUT /system/rollback

Rollback to previous firmware version (inactive system image).

Template request

```
PUT /api/v1/system/rollback HTTP/1.1
Host: <rcvisard>
```

Status Codes

- 200 OK – successful operation
- 500 Internal Server Error – internal error

- 400 Bad Request – already set to use inactive partition on next boot

GET /system/update

Get information about currently active and inactive firmware/system images on sensor.

Template request

```
GET /api/v1/system/update HTTP/1.1
Host: <rcvisard>
```

Sample response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "active_image": {
    "image_version": "rc_visard_v1.1.0"
  },
  "fallback_booted": false,
  "inactive_image": {
    "image_version": "rc_visard_v1.0.0"
  },
  "next_boot_image": "active_image"
}
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns FirmwareInfo)*

Referenced Data Models

- *FirmwareInfo* (Section 8.2.3)

POST /system/update

Update firmware/system image with a mender artifact. Reboot is required afterwards in order to activate updated firmware version.

Template request

```
POST /api/v1/system/update HTTP/1.1
Host: <rcvisard>
Accept: multipart/form-data
```

Form Parameters

- **file** – mender artifact file *(required)*

Request Headers

- Accept – multipart/form-data

Status Codes

- 200 OK – successful operation
- 400 Bad Request – client error, e.g. no valid mender artifact

8.2.3 Data type definitions

The REST-API defines the following data models, which are used to access or modify *the available resources* (Section 8.2.2) either as required attributes/parameters of the requests or as return types.

FirmwareInfo: Information about currently active and inactive firmware images, and what image is/will be booted.

An object of type FirmwareInfo has the following properties:

- **active_image** (*ImageInfo*) - see description of *ImageInfo*
- **fallback_booted** (boolean) - true if desired image could not be booted and fallback boot to the previous image occured
- **inactive_image** (*ImageInfo*) - see description of *ImageInfo*
- **next_boot_image** (string) - firmware image that will be booted next time (one of `active_image`, `inactive_image`)

Template object

```
{
  "active_image": {
    "image_version": "string"
  },
  "fallback_booted": false,
  "inactive_image": {
    "image_version": "string"
  },
  "next_boot_image": "string"
}
```

FirmwareInfo objects are nested in *SysInfo*, and are used in the following requests:

- *GET /system/rollback*
- *GET /system/update*

ImageInfo: Information about specific firmware image.

An object of type ImageInfo has the following properties:

- **image_version** (string) - image version

Template object

```
{
  "image_version": "string"
}
```

ImageInfo objects are nested in *FirmwareInfo*.

LicenseComponents: List of the licensing status of the individual software components. The respective flag is true if the component is unlocked with the currently applied software license.

An object of type LicenseComponents has the following properties:

- **calibration** (boolean) - camera calibration component
- **fusion** (boolean) - stereo ins/fusion components
- **hand_eye_calibration** (boolean) - hand-eye calibration component
- **rectification** (boolean) - image rectification component
- **self_calibration** (boolean) - camera self-calibration component
- **slam** (boolean) - SLAM component
- **stereo** (boolean) - stereo matching component

- **svo** (boolean) - visual odometry component

Template object

```
{
  "calibration": false,
  "fusion": false,
  "hand_eye_calibration": false,
  "rectification": false,
  "self_calibration": false,
  "slam": false,
  "stereo": false,
  "svo": false
}
```

LicenseComponents objects are nested in *LicenseInfo*.

LicenseInfo: Information about the currently applied software license on the sensor.

An object of type LicenseInfo has the following properties:

- **components** (*LicenseComponents*) - see description of *LicenseComponents*
- **valid** (boolean) - indicates whether the license is valid or not

Template object

```
{
  "components": {
    "calibration": false,
    "fusion": false,
    "hand_eye_calibration": false,
    "rectification": false,
    "self_calibration": false,
    "slam": false,
    "stereo": false,
    "svo": false
  },
  "valid": false
}
```

LicenseInfo objects are used in the following requests:

- *GET /system/license*

Log: Content of a specific log file represented in JSON format.

An object of type Log has the following properties:

- **date** (float) - UNIX time when log was last modified
- **log** (array of *LogEntry*) - the actual log entries
- **name** (string) - mame of log file
- **size** (integer) - size of log file in bytes

Template object

```
{
  "date": 0,
  "log": [
    {
      "component": "string",
      "level": "string",
      "message": "string",
      "timestamp": 0
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "component": "string",
      "level": "string",
      "message": "string",
      "timestamp": 0
    }
  ],
  "name": "string",
  "size": 0
}
```

Log objects are used in the following requests:

- *GET /logs/{log}*

LogEntry: Representation of a single log entry in a log file.

An object of type LogEntry has the following properties:

- **component** (string) - component name that created this entry
- **level** (string) - log level (one of `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`)
- **message** (string) - actual log message
- **timestamp** (float) - Unix time of log entry

Template object

```
{
  "component": "string",
  "level": "string",
  "message": "string",
  "timestamp": 0
}
```

LogEntry objects are nested in *Log*.

LogInfo: Information about a specific log file.

An object of type LogInfo has the following properties:

- **date** (float) - UNIX time when log was last modified
- **name** (string) - name of log file
- **size** (integer) - size of log file in bytes

Template object

```
{
  "date": 0,
  "name": "string",
  "size": 0
}
```

LogInfo objects are used in the following requests:

- *GET /logs*

NodeInfo: Description of a computational node running on sensor.

An object of type NodeInfo has the following properties:

- **name** (string) - name of the node
- **parameters** (array of string) - list of the node's run-time parameters

- **services** (array of string) - list of the services this node offers
- **status** (string) - status of the node (one of `unknown`, `down`, `stale`, `running`)

Template object

```
{
  "name": "string",
  "parameters": [
    "string",
    "string"
  ],
  "services": [
    "string",
    "string"
  ],
  "status": "string"
}
```

NodeInfo objects are used in the following requests:

- *GET /nodes*
- *GET /nodes/{node}*

NodeStatus: Detailed current status of the node including run-time statistics.

An object of type NodeStatus has the following properties:

- **status** (string) - status of the node (one of `unknown`, `down`, `stale`, `running`)
- **timestamp** (float) - Unix time when values were last updated
- **values** (object) - dictionary with current status/statistics of the node

Template object

```
{
  "status": "string",
  "timestamp": 0,
  "values": {}
}
```

NodeStatus objects are used in the following requests:

- *GET /nodes/{node}/status*

NtpStatus: Status of the NTP time sync.

An object of type NtpStatus has the following properties:

- **accuracy** (string) - time sync accuracy reported by NTP
- **synchronized** (boolean) - synchronized with NTP server

Template object

```
{
  "accuracy": "string",
  "synchronized": false
}
```

NtpStatus objects are nested in *SysInfo*.

Parameter: Representation of a node's run-time parameter. The parameter's 'value' type (and hence the types of the 'min', 'max' and 'default' fields) can be inferred from the 'type' field and might be one of the built-in primitive data types.

An object of type Parameter has the following properties:

- **default** (type not defined) - the parameter's default value
- **description** (string) - description of the parameter
- **max** (type not defined) - maximum value this parameter can be assigned to
- **min** (type not defined) - minimum value this parameter can be assigned to
- **name** (string) - name of the parameter
- **type** (string) - the parameter's primitive type represented as string (one of `bool`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `float32`, `float64`, `string`)
- **value** (type not defined) - the parameter's current value

Template object

```
{
  "default": {},
  "description": "string",
  "max": {},
  "min": {},
  "name": "string",
  "type": "string",
  "value": {}
}
```

Parameter objects are used in the following requests:

- *GET /nodes/{node}/parameters*
- *PUT /nodes/{node}/parameters*
- *GET /nodes/{node}/parameters/{param}*
- *PUT /nodes/{node}/parameters/{param}*

PtpStatus: Status of the IEEE1588 (PTP) time sync.

An object of type PtpStatus has the following properties:

- **master_ip** (string) - IP of the master clock
- **offset** (float) - time offset in seconds to the master
- **offset_dev** (float) - standard deviation of time offset in seconds to the master
- **offset_mean** (float) - mean time offset in seconds to the master
- **state** (string) - state of PTP (one of `off`, `unknown`, `INITIALIZING`, `FAULTY`, `DISABLED`, `LISTENING`, `PASSIVE`, `UNCALIBRATED`, `SLAVE`)

Template object

```
{
  "master_ip": "string",
  "offset": 0,
  "offset_dev": 0,
  "offset_mean": 0,
  "state": "string"
}
```

PtpStatus objects are nested in *SysInfo*.

Service: Representation of a service that a node offers.

An object of type Service has the following properties:

- **args** (*ServiceArgs*) - see description of *ServiceArgs*
- **description** (string) - short description of this service

- **name** (string) - name of the service
- **response** (*ServiceResponse*) - see description of *ServiceResponse*

Template object

```
{
  "args": {},
  "description": "string",
  "name": "string",
  "response": {}
}
```

Service objects are used in the following requests:

- *GET /nodes/{node}/services*
- *GET /nodes/{node}/services/{service}*
- *PUT /nodes/{node}/services/{service}*

ServiceArgs: Arguments required to call a service with. The general representation of these arguments is a (nested) dictionary. The specific content of this dictionary depends on the respective node and service call.

ServiceArgs objects are nested in *Service*.

ServiceResponse: The response returned by the service call. The general representation of this response is a (nested) dictionary. The specific content of this dictionary depends on the respective node and service call.

ServiceResponse objects are nested in *Service*.

Stream: Represention of a data stream offered by the rc_dynamics interface.

An object of type Stream has the following properties:

- **destinations** (array of *StreamDestination*) - list of destinations this data is currently streamed to
- **name** (string) - the data stream's name specifying which rc_dynamics data is streamed
- **type** (*StreamType*) - see description of *StreamType*

Template object

```
{
  "destinations": [
    "string",
    "string"
  ],
  "name": "string",
  "type": {
    "protobuf": "string",
    "protocol": "string"
  }
}
```

Stream objects are used in the following requests:

- *GET /datastreams*
- *GET /datastreams/{stream}*
- *PUT /datastreams/{stream}*
- *DELETE /datastreams/{stream}*

StreamDestination: A destination of an rc_dynamics data stream represented as string such as 'IP:port'

An object of type StreamDestination is of primitive type string.

StreamDestination objects are nested in *Stream*.

StreamType: Description of a data stream's protocol.

An object of type StreamType has the following properties:

- **protobuf** (string) - type of data-serialization, i.e. name of protobuf message definition
- **protocol** (string) - network protocol of the stream [UDP]

Template object

```
{
  "protobuf": "string",
  "protocol": "string"
}
```

StreamType objects are nested in *Stream*.

SysInfo: System information about the sensor.

An object of type SysInfo has the following properties:

- **firmware** (*FirmwareInfo*) - see description of *FirmwareInfo*
- **hostname** (string) - Hostname
- **link_speed** (integer) - Ethernet link speed in Mbps
- **mac** (string) - MAC address
- **ntp_status** (*NtpStatus*) - see description of *NtpStatus*
- **ptp_status** (*PtpStatus*) - see description of *PtpStatus*
- **ready** (boolean) - system is fully booted and ready
- **serial** (string) - sensor serial number
- **time** (float) - system time as Unix timestamp
- **uptime** (float) - system uptime in seconds

Template object

```
{
  "firmware": {
    "active_image": {
      "image_version": "string"
    },
    "fallback_booted": false,
    "inactive_image": {
      "image_version": "string"
    },
    "next_boot_image": "string"
  },
  "hostname": "string",
  "link_speed": 0,
  "mac": "string",
  "ntp_status": {
    "accuracy": "string",
    "synchronized": false
  },
  "ptp_status": {
    "master_ip": "string",
    "offset": 0,
    "offset_dev": 0,
    "offset_mean": 0,
    "state": "string"
  },
  "ready": false,
```

(continues on next page)

(continued from previous page)

```
  "serial": "string",
  "time": 0,
  "uptime": 0
}
```

SysInfo objects are used in the following requests:

- *GET /system*

Template: rc_silhouettematch template

An object of type Template has the following properties:

- **id** (string) - Unique identifier of the template

Template object

```
{
  "id": "string"
}
```

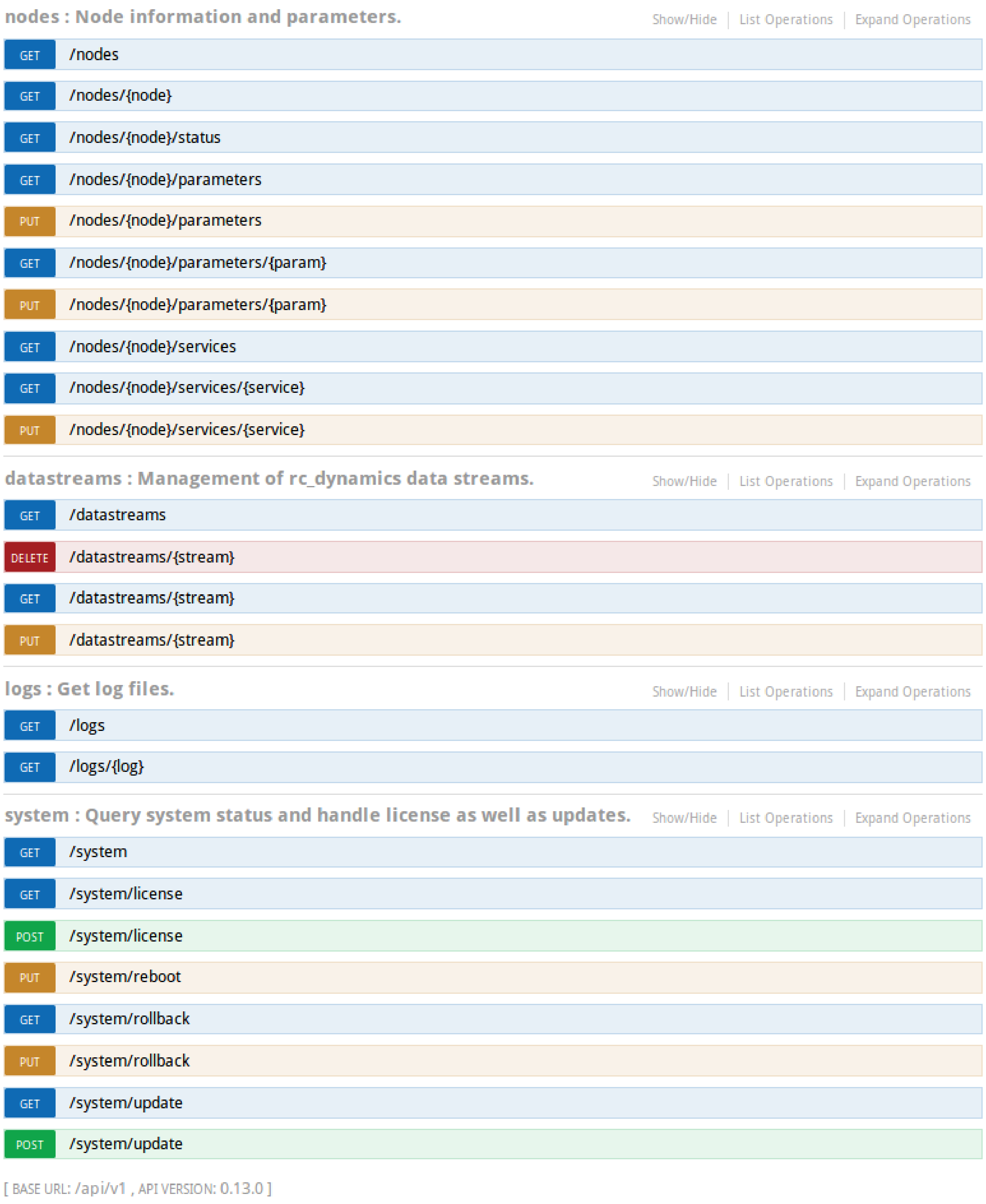
Template objects are used in the following requests:

- *GET /nodes/rc_silhouettematch/templates*
- *GET /nodes/rc_silhouettematch/templates/{id}*
- *PUT /nodes/rc_silhouettematch/templates/{id}*

8.2.4 Swagger UI

The *rc_visard*'s Swagger UI allows developers to easily visualize and interact with the REST-API, e.g., for development and testing. Accessing `http://<rcvisard>/api/` or `http://<rcvisard>/api/swagger` (the former will automatically be redirected to the latter) opens a visualization of the *rc_visard*'s general API structure including all *available resources and requests* (Section 8.2.2) and offers a simple user interface for exploring all of its features.

Note: Users must be aware that, although the *rc_visard*'s Swagger UI is designed to explore and test the REST-API, it is a fully functional interface. That is, any issued requests are actually processed and particularly `PUT`, `POST`, and `DELETE` requests might change the overall status and/or behavior of the device.



nodes : Node information and parameters. Show/Hide | List Operations | Expand Operations

GET /nodes
GET /nodes/{node}
GET /nodes/{node}/status
GET /nodes/{node}/parameters
PUT /nodes/{node}/parameters
GET /nodes/{node}/parameters/{param}
PUT /nodes/{node}/parameters/{param}
GET /nodes/{node}/services
GET /nodes/{node}/services/{service}
PUT /nodes/{node}/services/{service}

datastreams : Management of rc_dynamics data streams. Show/Hide | List Operations | Expand Operations

GET /datastreams
DELETE /datastreams/{stream}
GET /datastreams/{stream}
PUT /datastreams/{stream}

logs : Get log files. Show/Hide | List Operations | Expand Operations

GET /logs
GET /logs/{log}

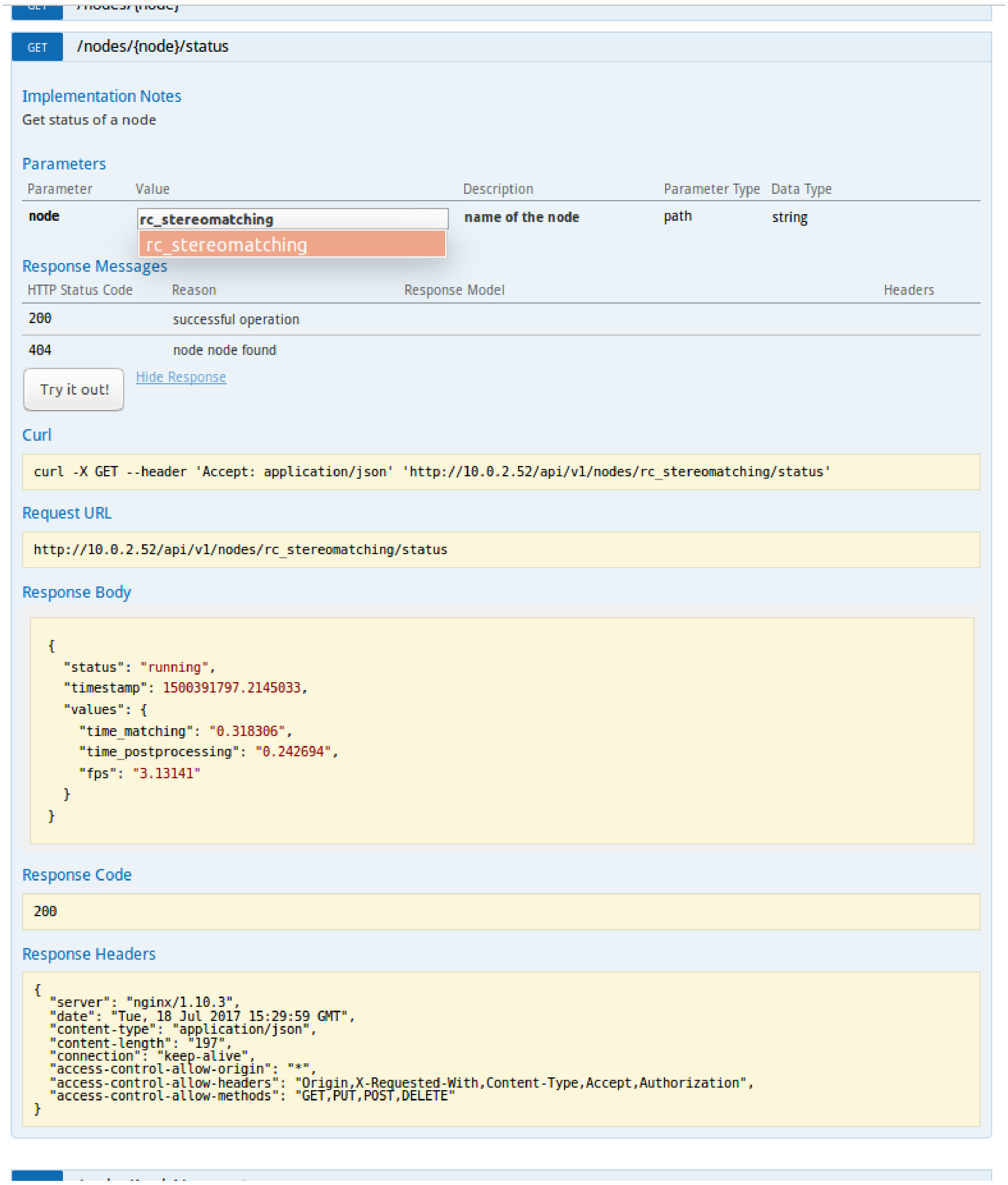
system : Query system status and handle license as well as updates. Show/Hide | List Operations | Expand Operations

GET /system
GET /system/license
POST /system/license
PUT /system/reboot
GET /system/rollback
PUT /system/rollback
GET /system/update
POST /system/update

[BASE URL: /api/v1 , API VERSION: 0.13.0]

Fig. 8.2.1: Initial view of the *rc_visard*'s Swagger UI with its resources and requests grouped into `nodes`, `datastreams`, `logs`, and `system`

Using this interface, available resources and requests can be explored by clicking on them to uncollapse or recollapse them. The following figure shows an example of how to get a node's current status by filling in the necessary parameter (`node` name) and clicking the *Try it out!* button. This action results in the Swagger UI showing, amongst others, the actual `curl` command that was executed when issuing the request as well as the response body showing the current status of the requested node in a JSON-formatted string.



GET /nodes/{node}/status

Implementation Notes

Get status of a node

Parameters

Parameter	Value	Description	Parameter Type	Data Type
node	rc_stereomatching	**name of the node**	path	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	successful operation		
404	node node found		

Try it out! Hide Response

Curl

```
curl -X GET --header 'Accept: application/json' 'http://10.0.2.52/api/v1/nodes/rc_stereomatching/status'
```

Request URL

```
http://10.0.2.52/api/v1/nodes/rc_stereomatching/status
```

Response Body

```
{
  "status": "running",
  "timestamp": 1500391797.2145033,
  "values": {
    "time_matching": "0.318306",
    "time_postprocessing": "0.242694",
    "fps": "3.13141"
  }
}
```

Response Code

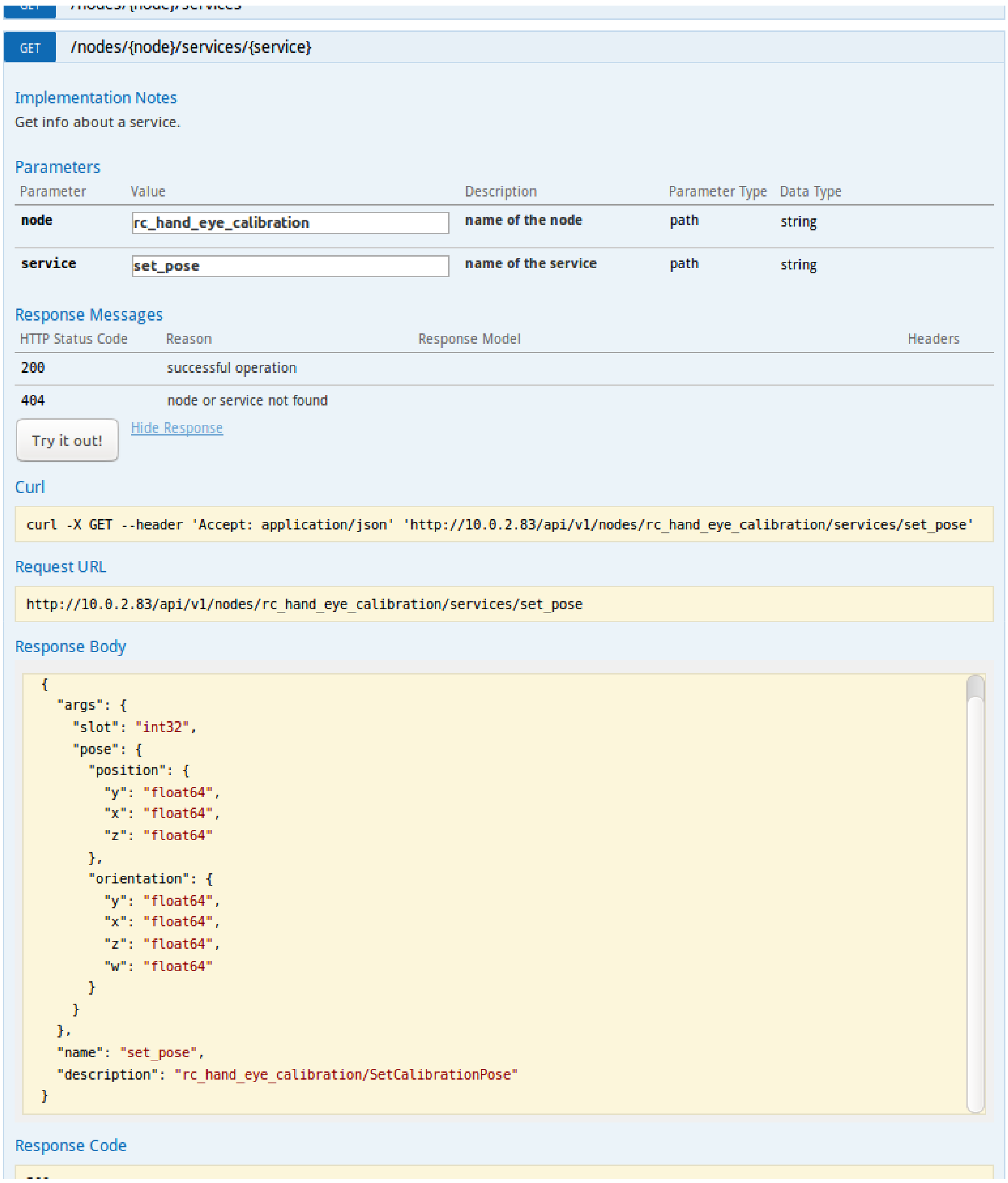
```
200
```

Response Headers

```
{
  "server": "nginx/1.10.3",
  "date": "Tue, 18 Jul 2017 15:29:59 GMT",
  "content-type": "application/json",
  "content-length": "197",
  "connection": "keep-alive",
  "access-control-allow-origin": "*",
  "access-control-allow-headers": "Origin,X-Requested-With,Content-Type,Accept,Authorization",
  "access-control-allow-methods": "GET,PUT,POST,DELETE"
}
```

Fig. 8.2.2: Result of requesting the `rc_stereomatching` node's status

Some actions, such as setting parameters or calling services, require more complex parameters to an HTTP request. The Swagger UI allows developers to explore the attributes required for these actions during run-time, as shown in the next example. In the figure below, the attributes required for the the `rc_hand_eye_calibration` node's `set_pose` service are explored by performing a `GET` request on this resource. The response features a full description of the service offered, including all required arguments with their names and types as a JSON-formatted string.



Get info about a service.

Parameter	Value	Description	Parameter Type	Data Type
node	rc_hand_eye_calibration	name of the node	path	string
service	set_pose	name of the service	path	string

HTTP Status Code	Reason	Response Model	Headers
200	successful operation		
404	node or service not found		

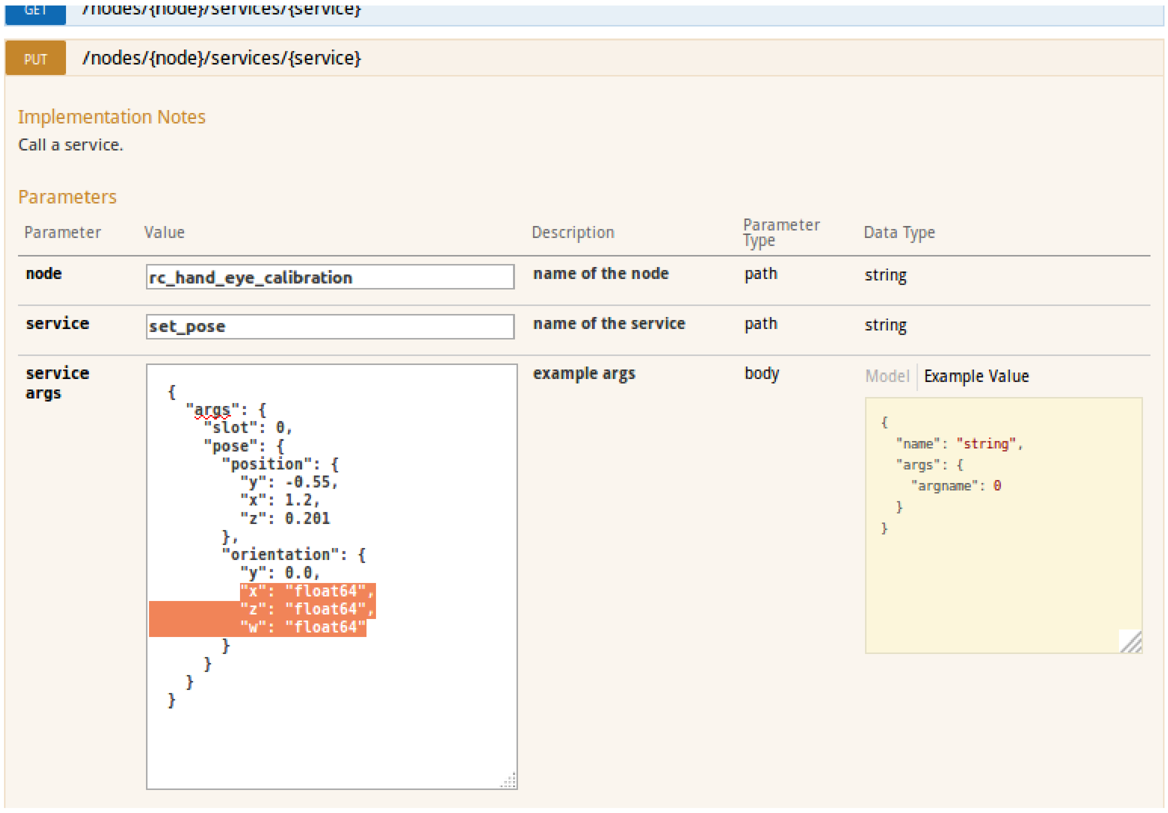
```
curl -X GET --header 'Accept: application/json' 'http://10.0.2.83/api/v1/nodes/rc_hand_eye_calibration/services/set_pose'
```

```
http://10.0.2.83/api/v1/nodes/rc_hand_eye_calibration/services/set_pose
```

```
{
  "args": {
    "slot": "int32",
    "pose": {
      "position": {
        "y": "float64",
        "x": "float64",
        "z": "float64"
      },
      "orientation": {
        "y": "float64",
        "x": "float64",
        "z": "float64",
        "w": "float64"
      }
    }
  },
  "name": "set_pose",
  "description": "rc_hand_eye_calibration/SetCalibrationPose"
}
```

Fig. 8.2.3: The result of the GET request on the `set_pose` service shows the required arguments for this service call.

Users can easily use this preformatted JSON string as a template for the service arguments to actually call the service:


GET /nodes/{node}/services/{service}

PUT /nodes/{node}/services/{service}

Implementation Notes

Call a service.

Parameters

Parameter	Value	Description	Parameter Type	Data Type
node	rc_hand_eye_calibration	name of the node	path	string
service	set_pose	name of the service	path	string
service args	(see below)	example args	body	Model / Example Value

```
{
  "args": {
    "slot": 0,
    "pose": {
      "position": {
        "y": -0.55,
        "x": 1.2,
        "z": 0.201
      },
      "orientation": {
        "y": 0.0,
        "x": "float64",
        "z": "float64",
        "w": "float64"
      }
    }
  }
}
```

```
{
  "name": "string",
  "args": {
    "argname": 0
  }
}
```

Fig. 8.2.4: Filling in the arguments of the `set_pose` service request

8.3 The rc_dynamics interface

The rc_dynamics interface offers continuous, real-time data-stream access to *rc_visard*'s several *dynamic state estimates* (Section 6.3.2) as continuous, real-time data streams. It allows state estimates of all offered types to be configured to be streamed to any host in the network. The *Data-stream protocol* (Section 8.3.3) used is agnostic vis-à-vis operating system and programming language.

8.3.1 Starting/stopping dynamic-state estimation

The *rc_visard*'s dynamic-state estimates are only available if the respective component, i.e., the *sensor dynamics component* (Section 6.3), is turned on. This can be done either in the Web GUI - a respective switch is offered in the *Dynamics* tab - or via the REST-API by using the component's service calls. A sample curl request to start dynamic-state estimation would look like:

```
curl -X PUT --header 'Content-Type: application/json' -d '{}' 'http://<rcvisard>/api/v1/nodes/rc_
↪dynamics/services/start'
```

Note: To save computational resources, it is recommended to stop dynamic-state estimation when not needed any longer.

8.3.2 Configuring data streams

Availabe data streams, i.e., dynamic-state estimates, can be listed and configured by the *rc_visard*'s *REST-API* (Section 8.2.2), e.g., a list of all available data streams can be requested with *`GET /datastreams`*. For a detailed description of the following data streams, please refer to *Available state estimates* (Section 6.3.2).

Table 8.3.1: Available data streams via the rc_dynamics interface

Name	Protocol	Protobuf	Description
`dynamics`	UDP	*Dynamics*	Dynamics of sensor (pose, velocity, acceleration) from INS or SLAM (best effort depending on availability) at realtime frequency (IMU rate)
`dynamics_ins`	UDP	*Dynamics*	Dynamics of sensor (pose, velocity, acceleration) from stereo INS at realtime frequency (IMU rate)
`pose`	UDP	*Frame*	Pose of left camera from INS or SLAM (best effort depending on availability) at maximum camera frequency (fps)
`pose_rt`	UDP	*Frame*	Pose of left camera from INS or SLAM (best effort depending on availability) at realtime frequency (IMU rate)
`pose_ins`	UDP	*Frame*	Pose of left camera from stereo INS at maximum camera frequency (fps)
`pose_rt_ins`	UDP	*Frame*	Pose of left camera from stereo INS at realtime frequency (IMU rate)
`imu`	UDP	*Imu*	Raw IMU (Inertial Measurement Unit) values at realtime frequency (IMU rate)

The general procedure for working with the rc_dynamics interface is the following:

1. **Request a data stream via REST-API.** The following sample `curl` command issues a *`PUT /datastreams/{stream}`* request to initiate a stream of type `pose_rt` from the *rc_visard* to client host `10.0.1.14` at port `30000`:

   ```
   curl -X PUT --header 'Content-Type: application/x-www-form-urlencoded' --header
   ↪'Accept: application/json' -d 'destination=10.0.1.14:30000' 'http://<rcvisard>/api/v1/
   ↪datastreams/pose_rt'
   ```

2. **Receive and deserialize data.** With a successful request, the stream is initiated and data of the specified stream type is continuously sent to the client host. According to the *Data-stream protocol* (Section 8.3.3), the client needs to receive, deserialize and process the data.

3. **Stop a requested data stream via REST-API.** The following sample `curl` command issues a *`DELETE /datastreams/{stream}`* request to delete, i.e., stop, the previously requested stream of type `pose_rt` with destination `10.0.1.14:30000`:

   ```
   curl -X DELETE --header 'Accept: application/json' 'http://<rcvisard>/api/v1/
   ↪datastreams/pose_rt?destination=10.0.1.14:30000'
   ```

   To remove all destinations for a stream, simply omit the destination parameter.

Warning: Data streams can not be deleted automatically, i.e., the *rc_visard* keeps streaming data even if the client-side is disconnected or has stopped consuming the sent datagrams. A maximum of 10 destinations per stream are allowed. It is therefore strongly recommended to stop data streams via the REST-API when they are or no longer used.

8.3.3 Data-stream protocol

Once a data stream is established, data is continuously sent to the specified client host and port (`destination`) via the following protocol:

Network protocol: The only currently supported network protocol is *UDP*, i.e., data is sent as UDP datagrams.

Data serialization: The data being sent is serialized via Google protocol buffers. The following message type definitions are used.

- The *camera-pose streams* and *real-time camera-pose streams* (Section 6.3.2) are serialized using the `Frame` message type:

```
message Frame
{
  optional PoseStamped pose  = 1;
  optional string parent     = 2; // Name of the parent frame
  optional string name       = 3; // Name of the frame
  optional string producer   = 4; // Name of the producer of this data
}
```

The producer field can take the values ins, slam, rt_ins, and rt_slam, indicating whether the data was computed by SLAM or Stereo INS, and is real-time (rt) or not.

- The *real-time dynamics stream* (Section 6.3.2) is serialized using the Dynamics message type:

```
message Dynamics
{
  optional Time timestamp                      = 1; // Time when the data was␣
↪captured
  optional Pose pose                           = 2;
  optional string pose_frame                   = 3; // Name of the frame that␣
↪the pose is given in
  optional Vector3d linear_velocity            = 4; // Linear velocity in m/s
  optional string linear_velocity_frame        = 5; // Name of the frame that␣
↪the linear_velocity is given in
  optional Vector3d angular_velocity           = 6; // Angular velocity in rad/s
  optional string angular_velocity_frame       = 7; // Name of the frame that␣
↪the angular_velocity is given in
  optional Vector3d linear_acceleration        = 8; // Gravity compensated␣
↪linear acceleration in m/s²
  optional string linear_acceleration_frame    = 9; // Name of the frame that␣
↪the acceleration is given in
  repeated double covariance                   = 10 [packed=true]; // Row-major␣
↪representation of the 15x15 covariance matrix
  optional Frame cam2imu_transform             = 11; // pose of the left camera␣
↪wrt. the IMU frame
  optional bool possible_jump                  = 12; // True if there possibly␣
↪was a jump in the pose estimation
  optional string producer                     = 13; // Name of the producer of␣
↪this data
}
```

The producer field can take the values rt_ins and rt_slam, indicating whether the data was computed by SLAM or Stereo INS.

- The *IMU stream* (Section 6.3.2) is serialized using the Imu message type:

```
message Imu
{
  optional Time timestamp                      = 1; // Time when the data was␣
↪captured
  optional Vector3d linear_acceleration        = 2; // Linear acceleration in m/
↪s² measured by the IMU
  optional Vector3d angular_velocity           = 3; // Angular velocity in rad/
↪s measured by the IMU
}
```

- The nested types PoseStamped, Pose, Time, Quaternion, and Vector3D are defined as follows:

```
message PoseStamped
{
  optional Time timestamp  = 1; // Time when the data was captured
  optional Pose pose       = 2;
}
```

```
message Pose
{
  optional Vector3d position      = 1; // Position in meters
  optional Quaternion orientation = 2; // Orientation as unit quaternion
  repeated double covariance      = 3 [packed=true]; // Row-major␣
↪representation of the 6x6 covariance matrix (x, y, z, rotation about X axis,␣
↪rotation about Y axis, rotation about Z axis)
}
```

```
message Time
{
  /// \brief Seconds
  optional int64 sec = 1;

  /// \brief Nanoseconds
  optional int32 nsec = 2;
}
```

```
message Quaternion
{
  optional double x = 2;
  optional double y = 3;
  optional double z = 4;
  optional double w = 5;
}
```

```
message Vector3d
{
  optional double x = 1;
  optional double y = 2;
  optional double z = 3;
}
```

8.4 KUKA Ethernet KRL Interface

The *rc_visard* provides an Ethernet KRL Interface (EKI Bridge), which allows communicating with the *rc_visard* from KUKA KRL via KUKA.EthernetKRL XML.

Note: The component is optional and requires a separate Roboception's EKIBridge *license* (Section 9.6) to be purchased.

Note: The KUKA.EthernetKRL add-on software package version 2.2 or newer must be activated on the robot controller to use this component.

The EKI Bridge can be used to programmatically

- do service calls, e.g. to start and stop individual computational nodes, or to use offered services such as the hand-eye calibration or the computation of grasp poses;
- set and get run-time parameters of computation nodes, e.g. of the camera, or disparity calculation.

8.4.1 Ethernet connection configuration

The EKI Bridge listens on port 7000 for EKI XML messages and transparently bridges the *rc_visard*'s *REST-API* (Section 8.2). The received EKI messages are transformed to JSON and forwarded to the *rc_visard*'s REST-API. The response from the REST-API is transformed back to EKI XML.

The EKI Bridge gives access to run-time parameters and offered services of all computational nodes described in *Software components* (Section 6) and *Optional software components* (Section 7).

The Ethernet connection to the *rc_visard* on the robot controller is configured using XML configuration files. The EKI XML configuration files of all nodes running on the *rc_visard* are available for download at:

https://doc.rc-visard.com/latest/en/eki.html#eki-xml-configuration-files

Each node offering run-time parameters has an XML configuration file for setting and getting its parameters. These are named following the scheme `<node_name>-parameters.xml`. Each node's service has its own XML configuration file. These are named following the scheme `<node_name>-<service_name>.xml`.

All elements in the XML files are preset, except for the IP of the *rc_visard* in the network.

These files must be stored in the directory `C:\KRC\ROBOTER\Config\User\Common\EthernetKRL` of the robot controller and they are read in when a connection is initialized.

As an example, an Ethernet connection to configure the `rc_stereomatching` parameters is established with the following KRL code.

```
DECL EKI_Status RET
RET = EKI_INIT("rc_stereomatching-parameters")
RET = EKI_Open("rc_stereomatching-parameters")

; ----------- Desired operation -----------

RET = EKI_Close("rc_stereomatching-parameters")
```

Note: The EKI Bridge automatically terminates the connection to the client if the received XML telegram is invalid.

8.4.2 Generic XML structure

For data transmission, the EKI Bridge uses `<req>` as root XML element (short for request).

The root tag always includes the following elements.

- `<node>`. This includes a child XML element used by the EKI Bridge to identify the target node. The node name is already included in the XML configuration file.
- `<end_of_request>`. End of request flag that triggers the request.

The following listing shows the generic XML structure for data transmission.

```
<SEND>
  <XML>
    <ELEMENT Tag="req/node/<node_name>" Type="STRING"/>
    <ELEMENT Tag="req/end_of_request" Type="BOOL"/>
  </XML>
</SEND>
```

For data reception, the EKI Bridge uses `<res>` as root XML element (short for response). The root tag always includes a `<return_code>` child element.

```
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

Note: By default the XML configuration files uses 998 as flag to notify KRL that the response data record has being received. If this value is already in use, it should be changed in the corresponding XML configuration file.

Return code

The `<return_code>` element consists of a `value` and a `message` attribute.

As for all other components, a successful request returns with a `res/return_code/@value` of 0. Negative values indicate that the request failed. The error message is contained in `res/return_code/@message`. Positive values indicate that the request succeeded with additional information, contained in `res/return_code/@message` as well.

The following codes can be issued by the EKI Bridge component.

Table 8.4.1: Return codes of the EKI Bridge component

Code	Description
0	Success
-1	Parsing error in the conversion from XML to JSON
-2	Internal error
-9	Missing or invalid license for EKI Bridge component
-11	Connection error from the REST-API

Note: The EKI Bridge can also return return code values specific to individual nodes. They are documented in the respective *software component* (Section 6).

Note: Due to limitations in KRL, the maximum length of a string returned by the EKI Bridge is 512 characters. All messages larger than this value are truncated.

8.4.3 Services

For the nodes' services, the XML schema is generated from the service's arguments and response in JavaScript Object Notation (JSON) described in *Software components* (Section 6) and *Optional software components* (Section 7). The conversion is done transparently, except for the conversion rules described below.

Conversions of poses:

A pose is a JSON object that includes `position` and `orientation` keys.

```
{
  "pose": {
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64",
    },
    "orientation": {
      "x": "float64",
      "y": "float64",
      "z": "float64",
      "w": "float64",
    }
  }
}
```

This JSON object is converted to a KRL `FRAME` in the XML message.

```
<pose X="..." Y="..." Z="..." A="..." B="..." C="..."></pose>
```

Positions are converted from meters to millimeters and orientations are converted from quaternions to KUKA ABC (in degrees).

Note: No other unit conversions are included in the EKI Bridge. All dimensions and 3D coordinates that don't belong to a pose are expected and returned in meters.

Arrays:

Arrays are identified by adding the child element `<le>` (short for list element) to the list name. As an example, the JSON object

```
{
  "rectangles": [
    {
      "x": "float64",
      "y": "float64"
    }
  ]
}
```

is converted to the XML fragment

```
<rectangles>
  <le>
    <x>...</x>
    <y>...</y>
  </le>
</rectangles>
```

Use of XML attributes:

All JSON keys whose values is a primitive data type and don't belong to an array are stored in attributes. As an example, the JSON object

```
{
  "item": {
    "uuid": "string",
    "confidence": "float64",
    "rectangle": {
      "x": "float64",
      "y": "float64"
    }
  }
}
```

is converted to the XML fragment

```
<item uuid="..." confidence="...">
  <rectangle x="..." y="...">
  </rectangle>
</item>
```

Request XML structure

The `<SEND>` element in the XML configuration file for a generic service follows the specification below.

```
<SEND>
  <XML>
```

(continues on next page)

(continued from previous page)

```
    <ELEMENT Tag="req/node/<node_name>" Type="STRING"/>
    <ELEMENT Tag="req/service/<service_name>" Type="STRING"/>
    <ELEMENT Tag="req/args/<argX>" Type="<argX_type>"/>
    <ELEMENT Tag="req/end_of_request" Type="BOOL"/>
  </XML>
</SEND>
```

The `<service>` element includes a child XML element that is used by the EKI Bridge to identify the target service from the XML telegram. The service name is already included in the configuration file.

The `<args>` element includes the service arguments and should be configured with `EKI_Set<Type>` KRL instructions.

As an example, the `<SEND>` element of the `rc_itempick`'s `get_load_carriers` service (see *ItemPick and BoxPick*, Section 7.4) is:

```
<SEND>
  <XML>
    <ELEMENT Tag="req/node/rc_itempick" Type="STRING"/>
    <ELEMENT Tag="req/service/get_load_carriers" Type="STRING"/>
    <ELEMENT Tag="req/args/load_carrier_ids/le" Type="STRING"/>
    <ELEMENT Tag="req/end_of_request" Type="BOOL"/>
  </XML>
</SEND>
```

The `<end_of_request>` element allows to have arrays in the request. For configuring an array, the request is split into as many packages as the size of the array. The last telegram contains all tags, including the `<end_of_request>` flag, while all other telegrams contain one array element each.

As an example, for requesting two load carrier models to the `rc_itempick`'s `get_load_carriers` service, the user needs to send two XML messages. The first XML telegram is:

```
<req>
  <args>
    <load_carrier_ids>
      <le>load_carrier1</le>
    </load_carrier_ids>
  </args>
</req>
```

This telegram can be sent from KRL with the `EKI_Send` command, by specifying the list element as path:

```
DECL EKI_STATUS RET
RET = EKI_SetString("rc_itempick-get_load_carriers", "req/args/load_carrier_ids/le",
↪"load_carrier1")
RET = EKI_Send("rc_itempick-get_load_carriers", "req/args/load_carrier_ids/le")
```

The second telegram includes all tags and triggers the request to the `rc_itempick` node:

```
<req>
  <node>
    <rc_itempick></rc_itempick>
  </node>
  <service>
    <get_load_carriers></get_load_carriers>
  </service>
  <args>
    <load_carrier_ids>
      <le>load_carrier2</le>
    </load_carrier_ids>
  </args>
```

(continues on next page)

(continued from previous page)

```
  <end_of_request></end_of_request>
</req>
```

This telegram can be sent from KRL by specifying `req` as path for `EKI_Send`:

```
DECL EKI_STATUS RET
RET = EKI_SetString("rc_itempick-get_load_carriers", "req/args/load_carrier_ids/le",
↪"load_carrier2")
RET = EKI_Send("rc_itempick-get_load_carriers", "req")
```

Response XML structure

The `<RECEIVE>` element in the XML configuration file for a generic service follows the specification below:

```
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/<resX>" Type="<resX_type>"/>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

As an example, the `<RECEIVE>` element of the `rc_april_tag_detect`'s `detect` service (see *TagDetect*, Section 7.3) is:

```
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/timestamp/@sec" Type="INT"/>
    <ELEMENT Tag="res/timestamp/@nsec" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/tags/le/pose_frame" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/timestamp/@sec" Type="INT"/>
    <ELEMENT Tag="res/tags/le/timestamp/@nsec" Type="INT"/>
    <ELEMENT Tag="res/tags/le/pose/@X" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@Y" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@Z" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@A" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@B" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@C" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/instance_id" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/id" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/size" Type="REAL"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

For arrays, the response includes multiple instances of the same XML element. Each element is written into a separate buffer within EKI and can be read from the buffer with KRL instructions. The number of instances can be requested with `EKI_CheckBuffer` and each instance can then be read by calling `EKI_Get<Type>`.

As an example, the tag poses received after a call to the `rc_april_tag_detect`'s `detect` service can be read in KRL using the following code:

```
DECL EKI_STATUS RET
DECL INT i
DECL INT num_instances
DECL FRAME poses[32]
```

(continues on next page)

(continued from previous page)

```
DECL FRAME pose = {X 0.0, Y 0.0, Z 0.0, A 0.0, B 0.0, C 0.0}

RET = EKI_CheckBuffer("rc_april_tag_detect-detect", "res/tags/le/pose")
num_instances = RET.Buff
for i=1 to num_instances
  RET = EKI_GetFrame("rc_april_tag_detect-detect", "res/tags/le/pose", pose)
  poses[i] = pose
endfor
RET = EKI_ClearBuffer("rc_april_tag_detect-detect", "res")
```

Note: Before each request from EKI to the *rc_visard*, all buffers should be cleared in order to store only the current response in the EKI buffers.

8.4.4 Parameters

All nodes' parameters can be set and queried from the EKI Bridge. The XML configuration file for a generic node follows the specification below:

```
<SEND>
  <XML>
    <ELEMENT Tag="req/node/<node_name>" Type="STRING"/>
    <ELEMENT Tag="req/parameters/<parameter_x>/@value" Type="INT"/>
    <ELEMENT Tag="req/parameters/<parameter_y>/@value" Type="STRING"/>
    <ELEMENT Tag="req/end_of_request" Type="BOOL"/>
  </XML>
</SEND>
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/parameters/<parameter_x>/@value" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_x>/@default" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_x>/@min" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_x>/@max" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@value" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@default" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@min" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@max" Type="REAL"/>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

The request is interpreted as a *get* request if all parameter's `value` attributes are empty. If any `value` attribute is non-empty, it is interpreted as *set* request of the non-empty parameters.

As an example, the current value of all parameters of `rc_stereomatching` can be queried using with the XML telegram:

```
<req>
  <node>
    <rc_stereomatching></rc_stereomatching>
  </node>
  <parameters></parameters>
  <end_of_request></end_of_request>
</req>
```

This XML telegram can be sent out with Ethernet KRL using:

```
DECL EKI_STATUS RET
RET = EKI_Send("rc_stereomatching-parameters", "req")
```

The response from the EKI Bridge contains all parameters:

```
<res>
  <parameters>
    <acquisition_mode default="Continuous" max="" min="" value="Continuous"/>
    <quality default="High" max="" min="" value="High"/>
    <static_scene default="0" max="1" min="0" value="0"/>
    <disprange default="256" max="512" min="32" value="256"/>
    <seg default="200" max="4000" min="0" value="200"/>
    <smooth default="1" max="1" min="0" value="1"/>
    <median default="1" max="5" min="1" value="1"/>
    <fill default="3" max="4" min="0" value="3"/>
    <minconf default="0.5" max="1.0" min="0.5" value="0.5"/>
    <mindepth default="0.1" max="100.0" min="0.1" value="0.1"/>
    <maxdepth default="100.0" max="100.0" min="0.1" value="100.0"/>
    <maxdeptherr default="100.0" max="100.0" min="0.01" value="100.0"/>
  </parameters>
  <return_code message="" value="0"/>
</res>
```

The `quality` parameter of `rc_stereomatching` can be set to `Low` by the XML telegram:

```
<req>
    <node>
      <rc_stereomatching></rc_stereomatching>
    </node>
    <parameters>
      <quality value="L"></quality>
    </parameters>
    <end_of_request></end_of_request>
</req>
```

This XML telegram can be sent out with Ethernet KRL using:

```
DECL EKI_STATUS RET
RET = EKI_SetString("rc_stereomatching-parameters", "req/parameters/quality/@value", "L
↪")
RET = EKI_Send("rc_stereomatching-parameters", "req")
```

In this case, only the applied value of `quality` is returned by the EKI Bridge:

```
<res>
  <parameters>
    <quality default="High" max="" min="" value="Low"/>
  </parameters>
  <return_code message="" value="0"/>
</res>
```

8.5 Time synchronization

The *rc_visard* provides timestamps with all images and messages. To compare these with the time on the application host, the time needs to be properly synchronized. This can be done either via the Networt Time Protocol (NTP), which is the default, or the Precision Time Protocol (PTP).

Note: The *rc_visard* does not have a backup battery for its real time clock and hence does not retain time across power cycles. The system time starts in the year 2000 at power up and is then automatically set via NTP if a server can be found.

The current system time as well as NTP and PTP status can be queried via *REST API* (Section 8.2) and seen on the *Web GUI*'s (Section 4.5) *System* tab.

Note: Depending on the reachability of NTP servers or PTP masters it might take up to several minutes until the time is synchronized.

8.5.1 NTP

The Network Time Protocol (NTP) is a TCP/IP protocol for synchronizing time over a network. A client periodically requests the current time from a server, and uses it to set and correct its own clock.

By default the *rc_visard* tries to reach NTP servers from the NTP Pool Project, which will work if the *rc_visard* has access to the internet.

If the *rc_visard* is configured for *DHCP* (Section 4.3.1) (which is the default setting), it will also request NTP servers from the DHCP server and try to use those.

8.5.2 PTP

The Precision Time Protocol (PTP, also known as IEEE1588) is a protocol which offers more precise and robust clock synchronization than with NTP.

The *rc_visard* can be configured to act as a PTP slave via the standard *GigE Vision 2.0/GenICam interface* (Section 8.1) using the `GevIEEE1588` parameter.

At least one PTP master providing time has to be running in the network. On Linux the respective command for starting a PTP master on ethernet port `eth0` is, e.g., `sudo ptpd --masteronly --foreground -i eth0`.

While the *rc_visard* is synchronized with a PTP master (*rc_visard* in PTP status SLAVE), the NTP synchronization is paused.

9 Maintenance

Warning: The customer does not need to open the *rc_visard*'s housing to perform maintenance. Unauthorized opening will void the warranty.

9.1 Lens cleaning

Glass lenses with antireflective coating are used to reduce glare. Please take special care when cleaning the lenses. To clean them, use a soft lens-cleaning brush to remove dust or dirt particles. Then use a clean microfiber cloth that is designed to clean lenses, and gently wipe the lens using a circular motion to avoid scratches that may compromise the sensor's performance. For stubborn dirt, high purity isopropanol or a lens cleaning solution formulated for coated lenses (such as the Uvex Clear family of products) may be used.

9.2 Camera calibration

The cameras are calibrated during production. Under normal operation conditions, the calibration will be valid for the life time of the sensor. High impact, such as occurring when dropping the *rc_visard*, can change the camera's parameters slightly. In this case, calibration can be verified and recalibration undertaken via the Web GUI (see *Camera calibration*, Section 6.6).

9.3 Updating the firmware

Information about the current firmware image version can be found on the *Web GUI*'s (Section 4.5) *System* tab in the *System information* row. It can also be accessed via the *rc_visard*'s *REST-API interface* (Section 8.2) using the `GET /system` request. Users can use either the Web GUI or the REST-API to update the firmware.

Warning: After a firmware update, all of the software components' configured parameters will be reset to their defaults. Please make sure these settings are persisted on the application-side or client PC (e.g., using the *REST-API interface*, Section 8.2) to request all parameters and store them prior to executing the update.

The following settings are excluded from this and will be persisted across a firmware update:

- the *rc_visard*'s network configuration including an optional static IP address and the user-specifed device name,
- the latest result of the *Hand-eye calibration* (Section 6.7), i.e., recalibrating the *rc_visard* w.r.t. a robot is not required, unless mounting has changed, and
- the latest result of the *Camera calibration* (Section 6.6), i.e., recalibration of the *rc_visard*'s stereo cameras is not required.

Step 1: Download the newest firmware version. Firmware updates will be supplied from of a Mender artifact file identified by its `.mender` suffix.

If a new firmware update is available for your *rc_visard* device, the respective file can be downloaded to a local computer from http://www.roboception.com/download.

Step 2: Upload the update file. To update with the *rc_visard*'s REST-API, users may refer to the `POST /system/update` request.

To update the firmware via the Web GUI, locate the *Software Update* row on the *System* tab and press the *Upload Update* button (see Fig. 9.3.1). Select the desired update image file (file extension `.mender`) from the local file system and open it to start the update.

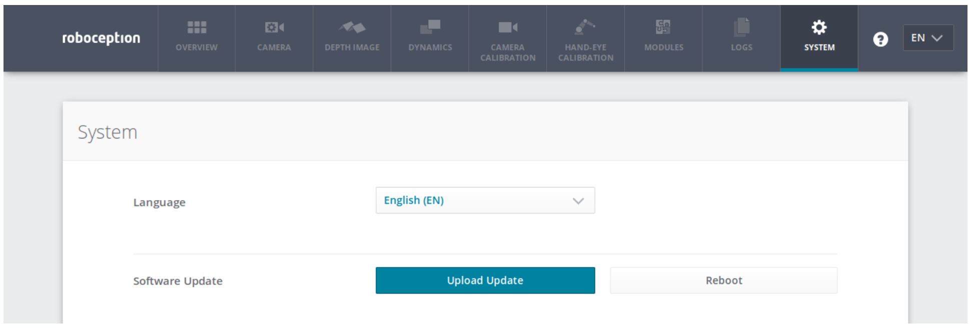


Fig. 9.3.1: Web GUI *System* tab

Note: Depending on the network architecture and configuration the upload may take several minutes. During the update via the Web GUI, a progress bar indicates the progress of the upload as shown in Fig. 9.3.2.

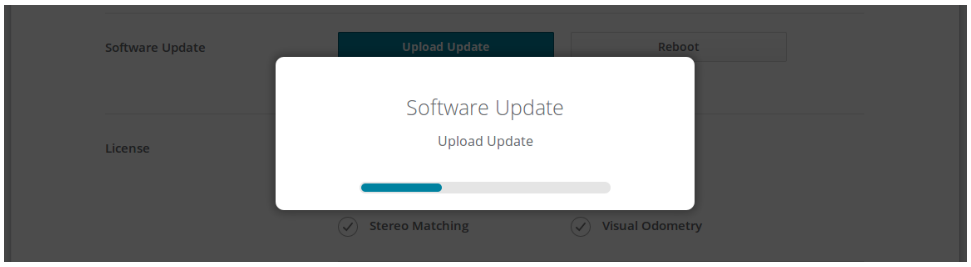
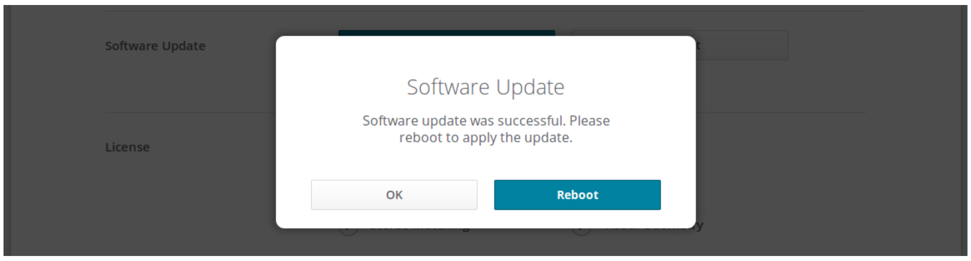


Fig. 9.3.2: Software update progress bar

Note: Depending on the web browser, the update progress status shown in Fig. 9.3.2 may indicate the completion of the update too early. Please wait until the context window shown in Fig. 9.3.3 opens. Expect an overall update time of at least five minutes.



Software update was successful. Please reboot to apply the update.

Fig. 9.3.3: Software update rebooting screen

Warning: Do not close the web browser tab which contains the Web GUI or press the renew button on this tab, because it will abort the update procedure. In that case, repeat the update procedure from the beginning.

Step 3: Reboot the *rc_visard*. To apply a firmware update to the *rc_visard* device, a reboot is required after having uploaded the new image version.

Note: The new image version is uploaded to the inactive partition of the *rc_visard*. Only after rebooting will the inactive partition be activated, and the active partition will become inactive. If the updated firmware image cannot be loaded, this partition of the *rc_visard* remains inactive and the previously installed firmware version from the active partition will be used automatically.

As for the REST-API, the reboot can be performed by the `PUT /system/reboot` request.

After having uploaded the new firmware via the Web GUI, a context window is opened as shown in Fig. 9.3.3 offering to reboot the device immediately or to postpone it. To reboot the *rc_visard* at a later time, use the *Reboot* button on the Web GUI's *System* tab.

Step 4: Confirm the firmware update. After rebooting the *rc_visard*, please check the firmware image version number of the currently active image to make sure that the updated image was successfully loaded. You can do so either via the Web GUI's *System* tab or via the REST-API's `GET /system/update` request.

Please contact Roboception in case the firmware update could not be applied successfully.

9.4 Restoring the previous firmware version

After a successful firmware update, the previous firmware image is stored on the inactive partition of the *rc_visard* and can be restored in case needed. This procedure is called a *rollback*.

Note: Using the latest firmware as provided by Roboception is strongly recommended. Hence, rollback functionality should only be used in case of serious issues with the updated firmware version.

Rollback functionality is only accessible via the *rc_visard*'s *REST-API interface* (Section 8.2) using the `PUT /system/rollback` request. It can be issued using any HTTP-compatible client or using a web browser as described in *Swagger UI* (Section 8.2.4). Like the update process, the rollback requires a subsequent device reboot to activate the restored firmware version.

Warning: Like during a firmware update, all software components' parameters will be reset to their defaults. Please make sure these settings are persisted on the application-side or client PC (e.g., using the *REST-API interface*, Section 8.2) prior to executing the rollback.

9.5 Rebooting the *rc_visard*

An *rc_visard* reboot is necessary after updating the firmware or performing a software rollback. It can be issued either programmatically, via the *rc_visard*'s *REST-API interface* (Section 8.2) using the `PUT /system/reboot` request, or manually on the *Web GUI*'s (Section 4.5) *System* tab. The reboot is finished when the LED turns green again.

9.6 Updating the software license

Licenses that are purchased from Roboception for enabling additional features can be installed via the *Web GUI*'s (Section 4.5) *System* panel. The *rc_visard* has to be rebooted to apply the licenses.

9.7 Downloading log files

During operation, the *rc_visard* logs important information, warnings, and errors into files. If the *rc_visard* exhibits unexpected or erroneous behavior, the log files can be used to trace its origin. Log messages can be viewed and filtered using the *Web GUI*'s (Section 4.5) *Logs* tab. If contacting the support (*Contact*, Section 12), the log files are very useful for tracking possible problems. To download them as a .tar.gz file, click on *Download all logs* on the Web GUI's *Logs* tab.

Besides the Web GUI, the logs are also accessible via the *rc_visard*'s *REST-API interface* (Section 8.2) using the `GET /logs` and `GET /logs/{log}` requests.

10 Accessories

10.1 Connectivity kit

Roboception offers an optional connectivity kit to aid customers with setting up the *rc_visard*. It consists of a:

- network cable with straight M12 plug to straight RJ45 connector in either 2 m or 5 m length;
- power adapter cable with straight M12 socket to DC barrel connector in 30 cm length;
- 24 V, 30 W desktop power supply.

Connecting the *rc_visard* to residential or office grid power requires a power supply that meets EN 55011 Class B emission standards. The E2CFS 30W 24V by EGSTON System Electronics Eggenburg GmbH (http://www.egston.com) contained in the connectivity kit is certified accordingly. However, it does not meet immunity standards for industrial environments under EN 61000-6-2.

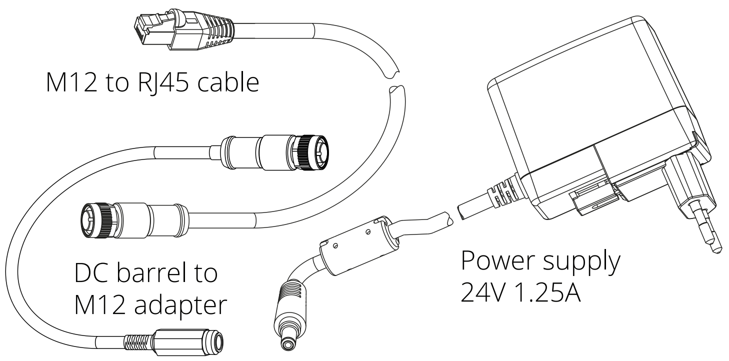


Fig. 10.1.1: The optional connectivity kit's components

10.2 Wiring

Cables are by default not provided with the *rc_visard*. It is the customer's responsibility to obtain appropriate parts. The following sections provide an overview of suggested components.

10.2.1 Ethernet connections

The *rc_visard* provides an industrial 8-pin A-coded M12 socket connector for Ethernet connectivity. Various cabling solutions can be obtained directly from third party vendors.

CAT5 (1 Gbps) M12 plug to RJ45

- Straight M12 plug to straight RJ45 connector, 10 m length: Phoenix Contact NBC-MS/ 10,0-94B/R4AC SCO, Art.-Nr.: 1407417

- Straight M12 plug to straight RJ45 connector, 10 m length: MURR Electronics Art.-Nr.: 7700-48521-S4W1000
- Angled M12 plug to straight RJ45 connector, 10 m length: MURR Electronics Art.-Nr.: 7700-48551-S4W1000

10.2.2 Power connections

An 8-pin A-coded M12 plug connector is provided for power and GPIO connectivity. Various cabling solutions can be obtained from third party vendors. A selection of M12 to open ended cables is provided below. Customers are required to provide power and GPIO connections to the cables according to the pinouts described in *Wiring* (Section 3.5). The *rc_visard*'s housing must be connected to ground.

Sensor/Actor cable M12 socket to open end

- Straight M12 socket connector to open end, shielded, 10m length: Phoenix Contact SAC-8P-10,0-PUR/M12FS SH, Art.Nr.: 1522891
- Angled M12 socket connector to open end, shielded 10m length: Phoenix Contact SAC-8P-10,0-PUR/M12FR SH, Art.Nr.: 1522943

Sensor/Actor M12 socket for field termination

- Phoenix Contact SACC-M12FS-8CON-PG9-M, Art.Nr.:1513347
- TE Connectivity T4110011081-000 (metal housing)
- TE Connectivity T4110001081-000 (plastic housing)

10.2.3 Power supplies

The *rc_visard* is classified as an EN-55011 Class A industrial device. For connecting the sensor to residential grid power, a power supply under EN 55011/55022 Class B has to be used.

It is the customer's responsibility to obtain and install a suitable power supply satisfying EN 61000-6-2 for permanent installation in industrial environments. One example that satisfies both EN 61000-6-2 and EN 55011/55022 Class B is the DIN-Rail mounted PULS MiniLine ML60.241 24V/DC 2.5 A by PULS GmbH (http://www.pulspower.com). A certified electrician must perform installation.

Only one *rc_visard* shall be connected to a power supply at any time, and the total length of cables must be less than 30 m.

10.3 Spare parts

No user-serviceable spare parts are currently available for *rc_visard* devices.

11 Troubleshooting

11.1 LED colors

During the boot process, the LED will change color several times to indicate stages in the boot process:

Table 11.1.1: LED color codes

LED color	Boot stage
white	power supply OK
yellow	normal boot process in progress
purple	
blue	
green	boot complete, *rc_visard* ready

The LED will signal some warning or error states to support the user during troubleshooting.

Table 11.1.2: LED color trouble codes

LED color	Warning or error state
off	no power to the sensor
brief red flash every 5 seconds	no network connectivity
red while sensor appears to function normally	high-temperature warning (case has exceeded 60 °C)
red while case is below 60 °C	Some process has terminated and failed to restart.

11.2 Hardware issues

LED does not illuminate

The *rc_visard* does not start up.

- Ensure that cables are connected and secured properly.
- Ensure that adequate DC voltage (18 V to 30 V) with correct polarity is applied to the power connector at the pins labeled as **Power** and **Ground** as described in the device's *pin assignment specification* (Section 3.5.1). Connecting the sensor to voltage outside of the specified range, to alternating current, with reversed polarity, or to a supply with voltage spikes will lead to permanent hardware damage.

LED turns red while the sensor appears to function normally

This may indicate a high housing temperature. The sensor might be mounted in a position that obstructs free airflow around the cooling fins.

- Clean cooling fins and housing.
- Ensure a minimum of 10 cm free space in all directions around cooling fins to provide adequate convective cooling.
- Ensure that ambient temperature is within specified range.

The sensor may slow down processing when cooling is insufficient or the ambient temperature exceeds the specified range.

Reliability issues and/or mechanical damage

This may be an indication of ambient conditions (vibration, shock, resonance, and temperature) being outside of specified range. Please refer to the *specification of environmental conditions* (Section 3.3.1).

- Operating the *rc_visard* outside of specified ambient conditions might lead to damage and will void the warranty.

Electrical shock when touching the sensor

This indicates an electrical fault in sensor, cabling, or power supply or adjacent system.

- Immediately turn off power to the system, disconnect cables, and have a qualified electrician check the setup.
- Ensure that the sensor housing is properly grounded; check for large ground loops.

11.3 Connectivity issues

LED briefly flashes red every 5 seconds

If the LED briefly flashes red every 5 seconds, then the *rc_visard* is not able to detect a network link.

- Check that the network cable is properly connected to the *rc_visard* and the network.
- If no problem is visible, then replace the Ethernet cable.

A GigE Vision client or rcdiscover-gui cannot detect the camera

- Check whether the *rc_visard*'s LED flashes briefly every 5 seconds (check the cable if it does).
- Ensure that the *rc_visard* is connected to the same subnet (the discovery mechanism uses broadcasts that will not work across different subnets).

The Web GUI is inaccessible

- Ensure that the *rc_visard* is turned on and connected to the same subnet as the host computer.
- Check whether the *rc_visard*'s LED flashes briefly every 5 seconds (check the cable if it does).
- Check whether `rcdiscover-gui` detects the sensor. If it reports the *rc_visard* as unreachable, then the *rc_visard*'s *network configuration* (Section 4.3) is wrong.
- If the *rc_visard* is reported as reachable, try double clicking the entry to open the Web GUI in a browser.
- If this does not work, try entering the *rc_visard*'s reported IP address directly in the browser as target address.

Too many Web GUIs are open at the same time

The Web GUI consumes the *rc_visard*'s processing resources to compress images to be transmitted and for statistical output that is regularly polled by the browser. Leaving several instances of the Web GUI open on the same or different computers can significantly diminish the *rc_visard*'s performance. The Web GUI is meant for configuration and validation, not to permanently monitor the *rc_visard*.

11.4 Camera-image issues

The camera image is too bright

- If the *rc_visard* is in manual exposure mode, decrease the exposure time (see *Parameters*, Section 6.1.3), or
- switch to auto-exposure mode (see *Parameters*, Section 6.1.3).

The camera image is too dark

- If the *rc_visard* is in manual exposure mode, increase the exposure time (see *Parameters*, Section 6.1.3), or
- switch to auto-exposure mode (see *Parameters*, Section 6.1.3).

The camera image is too noisy

Large gain factors cause high-amplitude image noise. To decrease the image noise,

- use an additional light source to increase the scene's light intensity, or
- choose a greater maximal auto-exposure time (see *Parameters*, Section 6.1.3).

The camera image is out of focus

- Check whether the object is too close to the lens and increase the distance between the object and the lens if it is.
- Check whether the lenses are dirty and clean them if they are (see *Lens cleaning*, Section 9.1).
- If none of the above applies, a severe hardware problem might exist. Please *contact support* (Section 12).

The camera image is blurred

Fast motions in combination with long exposure times can cause blur. To reduce motion blur,

- decrease the motion speed of the *rc_visard*,
- decrease the motion speed of objects in the field of view of the *rc_visard*, or
- decrease the exposure time of the cameras (see *Parameters*, Section 6.1.3).

The camera image is fuzzy

- Check whether the lenses are dirty and clean them if so (see *Lens cleaning*, Section 9.1).
- If none of the above applies, a severe hardware problem might exist. Please *contact support* (Section 12).

The camera image frame rate is too low

- Increase the image frame rate as described in *Parameters* (Section 6.1.3).
- The maximal frame rate of the cameras is 25 Hz.

11.5 Depth/Disparity, error, and confidence image issues

All these guidelines also apply to error and confidence images, because they correspond directly to the disparity image.

The disparity image is too sparse or empty

- Check whether the camera images are well exposed and sharp. Follow the instructions in *Camera-image issues* (Section 11.4) if applicable.
- Check whether the scene has enough texture (see *Stereo matching*, Section 6.2) and install an external pattern projector if required.
- Increase the *Disparity Range* and decrease the *Minimum Distance* (Section 6.2.4).
- Increase the *Maximum Distance* (Section 6.2.4).
- Check whether the object is too close to the cameras. Consider the different depth ranges of the *rc_visard* variants as specified in the device's *technical specification* (Section 3.2.2).
- Decrease the *Minimum Confidence* (Section 6.2.4).
- Increase the *Maximum Depth Error* (Section 6.2.4).
- Choose a lesser *Disparity Image Quality* (Section 6.2.4). Coarser resolution disparity images are generally less sparse.
- Check the cameras' calibration and recalibrate if required (see *Camera calibration*, Section 6.6).

The disparity images' frame rate is too low

- Check and increase the frame rate of the camera images (see *Parameters*, Section 6.1.3). The frame rate of the disparity image cannot be greater than the frame rate of the camera images.
- Choose a lesser *Disparity Image Quality* (Section 6.2.4). High-resolution disparity images are only available at about 3 Hz. Full 25 Hz can only be achieved for low-resolution disparity images as described in the *technical specifications* (Section 3.2.1).
- Decrease the *Disparity Range* and increase the *Minimum Distance* (Section 6.2.4) as much as possible for the application.
- Decrease the *Median filtering value* (Section 6.2.4).

The disparity image does not show close objects

- Check whether the object is too close to the cameras. Consider the depth ranges of the *rc_visard* variants as described in the *technical specifications* (Section 3.2.2).
- Increase the *Disparity Range* (Section 6.2.4).
- Decrease the *Minimum Distance* (Section 6.2.4).

The disparity image does not show distant objects

- Increase the *Maximum Distance* (Section 6.2.4).
- Increase the *Maximum Depth Error* (Section 6.2.4).
- Decrease the *Minimum Confidence* (Section 6.2.4).

The disparity image is too noisy

- Increase the *Segmentation value* (Section 6.2.4).
- Increase the *Fill-In value* (Section 6.2.4).
- Increase the *Median filtering value* (Section 6.2.4).

The disparity values or the resulting depth values are too inaccurate

- Decrease the distance between the *rc_visard* and the scene. Depth-measurement error grows quadratically with the distance from the cameras.
- Check whether the scene contains repetitive patterns and remove them if it does. They could cause wrong disparity measurements.
- Check whether the chosen *rc_visard* variant is correct for the application. Particularly consider the different depth ranges as described in the *technical specifications* (Section 3.2.2).

The disparity image is too smooth

- Decrease the *Median filtering value* (Section 6.2.4).
- Decrease the *Fill-In value* (Section 6.2.4).

The disparity image does not show small structures

- Decrease the *Segmentation value* (Section 6.2.4).
- Decrease the *Fill-In value* (Section 6.2.4).

11.6 Dynamics issues

State estimates are unavailable

- Check in the Web GUI that pose estimation has been switched on (see *Parameters*, Section 6.4.1).
- Check in the Web GUI that the update rate is about 200 Hz.
- Check the *Logs* in the Web GUI for errors.

The state estimates are too noisy

- Adapt the parameters for visual odometry as described in *Parameters* (Section 6.4.1).
- Check whether the *camera pose stream* has enough accuracy.

Pose estimation has jumps

- Has the SLAM component been turned on? SLAM can cause jumps when reducing errors due to a loop closure.
- Adapt the parameters for visual odometry as described in *Parameters* (Section 6.4.1).

Pose frequency is too low

- Use the real-time pose stream with a 200 Hz update rate. See *Stereo INS* (Section 6.5).

Delay/Latency of pose is too great

- Use the real-time pose stream. See *Stereo INS* (Section 6.5).

11.7 GigE Vision/GenICam issues

No images

- Check that the components are enabled. See `ComponentSelector` and `ComponentEnable` in *Important GenICam parameters* (Section 8.1.1).

12 Contact

12.1 Support

For support issues, please see http://www.roboception.com/support or contact support@roboception.de.

12.2 Downloads

Software SDKs, etc. can be downloaded from http://www.roboception.com/download.

12.3 Address

Roboception GmbH
Kaflerstrasse 2
81241 Munich
Germany

Web: http://www.roboception.com
Email: info@roboception.de
Phone: +49 89 889 50 79-0

13 Appendix

13.1 Pose formats

13.1.1 XYZABC format

The XYZABC format is used to express a pose by 6 values. XYZ is the position in millimeters. ABC are Euler angles in degrees. The convention used for Euler angles is ZYX, i.e., A rotates around the Z axis, B rotates around the Y axis, and C rotates around the X axis. The elements of the rotation matrix can be computed by using

$$\begin{aligned}
r_{11} &= \cos B \cos A,\\
r_{12} &= \sin C \sin B \cos A - \cos C \sin A,\\
r_{13} &= \cos C \sin B \cos A + \sin C \sin A,\\
r_{21} &= \cos B \sin A,\\
r_{22} &= \sin C \sin B \sin A + \cos C \cos A,\\
r_{23} &= \cos C \sin B \sin A - \sin C \cos A,\\
r_{31} &= -\sin B,\\
r_{32} &= \sin C \cos B, \text{ and}\\
r_{33} &= \cos C \cos B.
\end{aligned}$$

Note: The trigonometric functions sin and cos are assumed to accept values in degrees. The argument needs to be multiplied by the factor $\frac{\pi}{180}$ if they expect their values in radians.

Using these values, the rotation matrix R and translation vector T are defined as

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}, \qquad T = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}.$$

The transformation can be applied to a point P by

$$P' = RP + T.$$

13.1.2 XYZ+quaternion format

The XYZ+quaternion format is used to express a pose by a position and a unit quaternion. XYZ is the position in meters. The quaternion is a vector of length 1 that defines a rotation by four values, i.e., $q = \begin{pmatrix} a & b & c & w \end{pmatrix}^T$ with $||q|| = 1$. The corresponding rotation matrix and translation vector are defined by

$$R = 2 \begin{pmatrix} \frac{1}{2} - b^2 - c^2 & ab - cw & ac + bw \\ ab + cw & \frac{1}{2} - a^2 - c^2 & bc - aw \\ ac - bw & bc + aw & \frac{1}{2} - a^2 - b^2 \end{pmatrix}, \qquad T = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}.$$

The transformation can be applied to a point P by

$$P' = RP + T.$$

Note: In XYZ+quaternion format, the pose is defined in meters, whereas in the XYZABC format, the pose is defined in millimeters.

HTTP Routing Table

Index