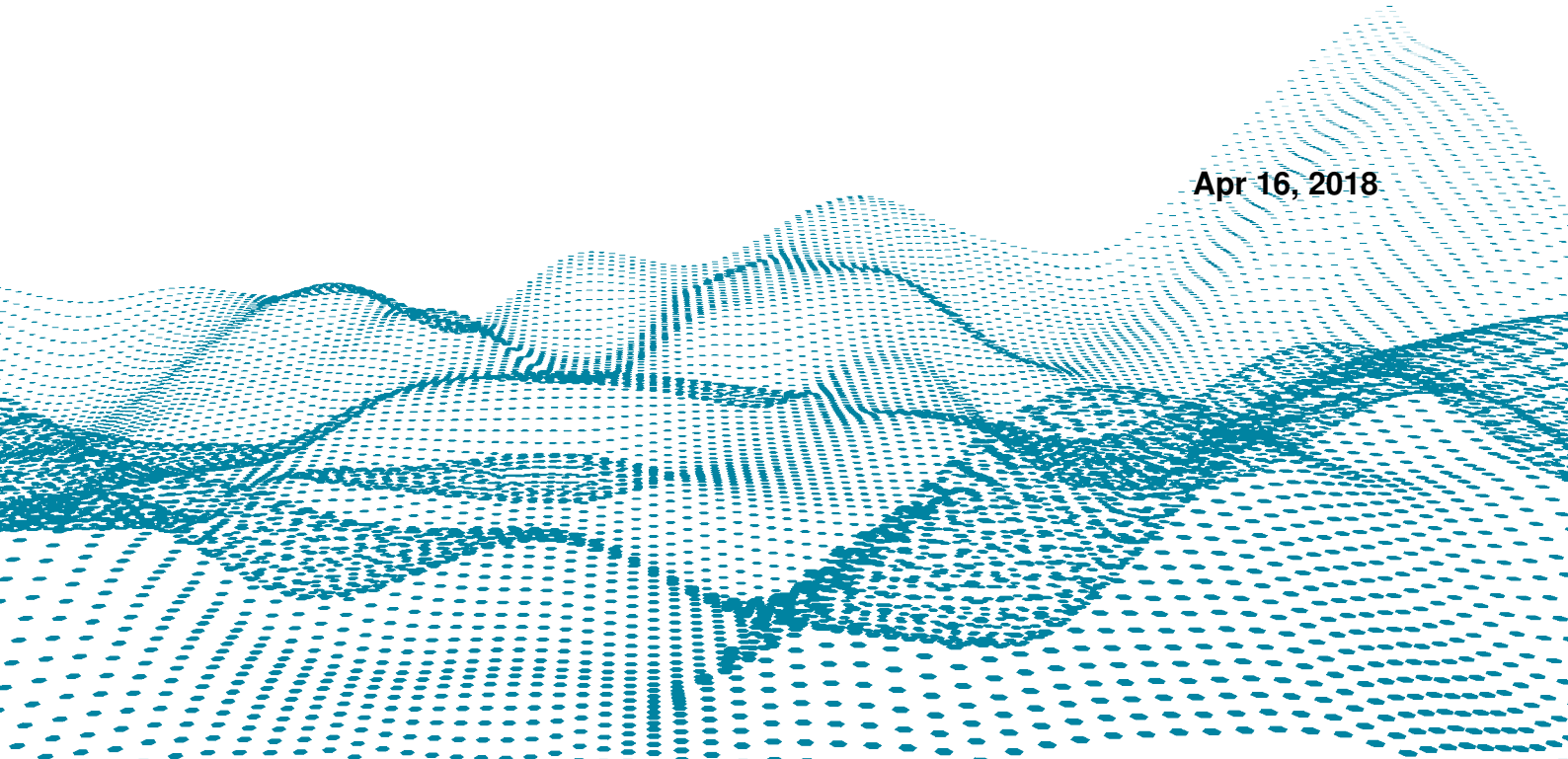
# rc_visard

***Release 1.2.1***

**Roboception GmbH**

**Apr 16, 2018**



# Contents

# 1 Introduction

## Revisions

This product may be modified without notice, when necessary, due to product improvements, modifications, or changes in specifications. If such modification is made, the manual will also be revised; see revision information.

**Revision** 1.2.1 Apr 16, 2018

**Copyright**



## Indications in the manual

To prevent damage to the equipment and ensure the user's safety, this manual indicates each precaution related to safety with *Warning*. Supplementary information is provided as a *Note*.

**Warning:** Warnings in this manual indicate procedures and actions that must be observed to avoid danger of injury to the operator/user, or damage to the equipment. Software-related warnings indicate procedures that must be observed to avoid malfunctions or unexpected behavior of the software.

**Note:** Notes are used in this manual to indicate supplementary relevant information.

## 1.1 Overview

The 3D sensor *rc_visard* provides real-time camera images and disparity images, which are also used to compute depth images and 3D point clouds. Additionally, it provides confidence and error images as quality measures for each image acquisition. The sensor provides self-localization based on image and inertial data. A mobile navigation solution can be established with the optional on-board SLAM module. The *rc_visard* is an IP54-protected sensor that offers an intuitive web and a standardized GenICam interface, making it compatible with all major image processing libraries. The *rc_visard* is offered with two different stereo baselines: The *rc_visard* 65 is optimally suited for mounting on robotic manipulators, whereas the *rc_visard* 160 can be employed as a navigation or as externally-fixed sensor. The *rc_visard*'s intuitive calibration, configuration, and use enable 3D vision for everyone.

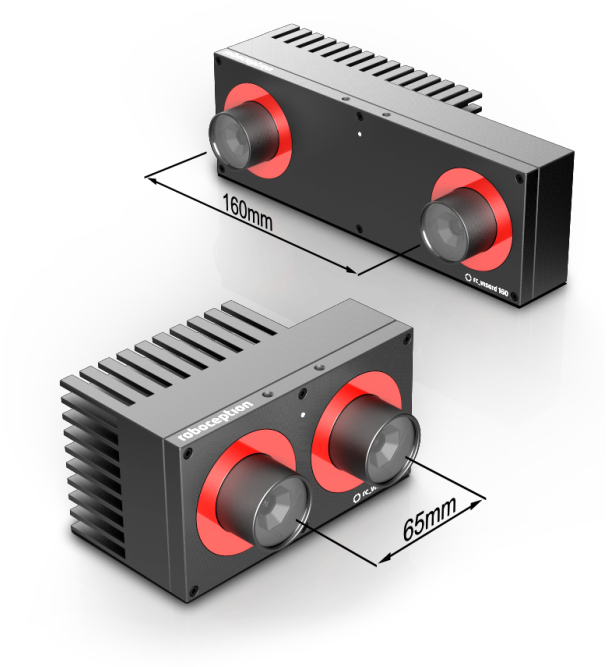



Fig. 1.1: *rc_visard* 65 and *rc_visard* 160

The terms "sensor," "*rc_visard* 65," and "*rc_visard* 160" used throughout the manual all refer to the Roboception *rc_visard* family of self-registering cameras. Installation and control for all sensors are exactly the same, and all use the same mounting base.

**Note:** Unless specified, the information provided in this manual applies to both the *rc_visard* 65 and *rc_visard* 160 versions of the Roboception sensor.

**Note:** This manual uses the metric system and mostly uses the units meter and millimeter. Unless otherwise specified, all dimensions in technical drawings are in millimeters.

## 1.2 Warranty

Any changes or modifications not expressly approved by Roboception could void the user's warranty and guarantee rights.

**Warning:** The *rc_visard* sensor utilizes complex hardware and software technology that may not always function as intended. The purchaser must design its application to ensure that any failure or the *rc_visard* sensor does not cause personal injury, property damage, or other losses.

**Warning:** Do not attempt to take apart, open, service, or modify the *rc_visard*. Doing so could present the risk of electric shock or other hazard. Any evidence of any attempt to open and/or modify the device, including any peeling, puncturing, or removal of any of the labels, will void the Limited Warranty.

**Warning:** CAUTION: to comply with the European CE requirement, all cables used to connect this device must be shielded and grounded. Operation with incorrect cables may result in interference with other devices or undesired effects of the product.

**Note:** This product may not be treated as household waste. By ensuring this product is disposed of correctly, you will help to protect the environment. For more detailed information about the recycling of this product, please contact your local authority, your household waste disposal service provider, or the product's supplier.

## 1.3 Applicable standards

### 1.3.1 Interfaces

The *rc_visard* supports the following interface standards:

GEN<i>CAM

The Generic Interface for Cameras standard is the basis for plug & play handling of cameras and devices.

GiGE VISION

GigE Vision® is an interface standard for transmitting high-speed video and related control data over Ethernet networks.

### 1.3.2 Approvals

The *rc_visard* has received the following approvals:

CE

EC Declaration of Conformity

TÜV SÜD C US

certification by TÜV Süd

### 1.3.3 Standards

The *rc_visard* has been tested to be in compliance with the following standards:

- AS/NZS CISPR32 : 2015 Information technology equipment, Radio disturbance characteristics, Limits and methods of measurement
- CISPR 32 : 2015 Electromagnetic compatibility of multimedia equipment - Emission requirements
- GB 9254 : 2008 This standard is out of the accreditation scope. Information technology equipment, Radio disturbance characteristics, Limits and methods of measurement
- EN 55032 : 2015 Electromagnetic compatibility of multimedia equipment - Emission requirements
- EN 55024 : 2010 +A1:2015 Information technology equipment, Immunity characteristics, Limits and methods of measurement
- CISPR 24 : 2015 +A1:2015 International special committee on radio interference, Information technology equipment-Immunity characteristics-Limits and methods of measurement
- EN 61000-6-2 : 2005 Electromagnetic compatibility (EMC) Part 6-2:Generic standards - Immunity for industrial environments
- EN 61000-6-3 : 2007+A1:2011 Electromagnetic compatibility (EMC) - Part 6-3: Generic standards - Emission standard for residential, commercial and light-industrial environments

## 1.4 Glossary

**DHCP** The Dynamic Host Configuration Protocol (DHCP) is used to automatically assign an *IP* address to a network device. Some DHCP servers only accept known devices. In this case, an administrator needs to configure the DHCP server with the fixed *MAC address* of a device.

**DNS**

**mDNS** The Domain Name Server (DNS) manages the host names and *IP* addresses of all network devices. It is responsible for resolving the host name into the IP address for communication with a device. A DNS can be configured to get this information automatically when a device appears on a network or manually by an administrator. In contrast, *multicast DNS* (mDNS) works without a central server by querying all devices on a local network each time a host name needs to be resolved. mDNS is available by default on Linux and Mac operating systems and is used when '.local' is appended to a host name.

**GenICam** GenICam is a generic standard interface for cameras. It serves as a unified interface around other standards such as *GigE Vision*, Camera Link, USB, etc. See http://genicam.org for more information.

**GigE** Gigabit Ethernet (GigE) is a networking technology for transmitting data at one gigabit per second.

**GigE Vision** GigE Vision® is a standard for configuring cameras and transmitting images over a *GigE* network link. See http://gigevision.com for more information.

**IMU** An Inertial Measurement Unit (IMU) consists of three accelerometers and three gyroscopes that measure the linear accelerations and the turn rates in all three dimensions.

**INS** An Inertial Navigation System (INS) is a 3D measurement system which uses inertial measurements (accelerations and turn rates) to compute position and orientation information. We refer to our combination of stereo vision and inertial navigation as stereo INS.

**IP**

**IP address** The Internet Protocol (IP) is a standard for sending data between devices in a computer network. Every device requires an IP address, which must be unique in the network. The IP address can be configured by *DHCP*, *Link Local*, or manually.

**Link Local** Link Local is a technology where network devices associate themselves with an *IP address* and check if it is unique in the local network. Link Local can be used if *DHCP* is unavailable and manual IP configuration is not or cannot be done. Link Local is especially useful for connecting a network device directly to a host computer. By default, Windows 10 reverts automatically to Link Local if DHCP is unavailable. Under Linux, Link Local must be enabled manually in the network manager.

**MAC address** The Media Access Control (MAC) address is a unique, persistent address for networking devices. It is also known as the hardware address of a device. In contrast to the *IP address*, the MAC address is (normally) permanently given to a device and does not change.

**NTP** The Network Time Protocol (NTP) is a TCP/IP protocol for synchronizing time over a network. Basically a client requests the current time from a server, and uses it to set its own clock.

**PTP** The Precision Time Protocol (PTP, also known as IEEE1588) is a protocol which enables more precise and robust clock synchronization than with NTP.

**SDK** A Software Development Kit (SDK) is a collection of software development tools or a collection of software modules.

**SGM** SGM stands for Semi-Global Matching and is a state-of-the-art stereo matching algorithm which offers brief run times and a great accuracy, especially at object borders, fine structures, and in weakly textured areas.

**SLAM** SLAM stands for Simultaneous Localization and Mapping and describes the process of creating a map of an unknown environment and simultaneously updating the sensor pose within the map.

**UDP** The User Datagram Protocol (UDP) is the minimal message-oriented transport layer of the Internet Protocol (*IP*) family. It uses a simple connectionless transmission model with a minimum of protocol mechanism such as integrity verification (via checksum). The *rc_visard* uses UDP for publishing its *estimated dynamical states* (Section 6.3.2) via the *rc_dynamics interface* (Section 8.3). To receive this data, applications may

use datagram sockets to bind to the endpoint of the data transmission consisting in a combination of an *IP address* and a service port number such as `192.168.0.100:49500`, which is typically referred to as a *destination* of an rc_dynamics data stream in this documentation.

**URI**

**URL** A Uniform Resource Identifier (URI) is a string of characters identifying resources of the *rc_visard*'s REST-API. An example of such a URI is `/nodes/rc_stereocamera/parameters/fps`, which points to the `fps` run-time parameter of the stereo camera component.

A Uniform Resource Locator (URL) additionally specifies the full network location and protocol, i.e., an exemplary URL to locate the above resource would be `https://<rcvisard>/api/v1/nodes/rc_stereocamera/parameters/fps` where `<rcvisard>` refers to the *rc_visard*'s *IP address*.

**XYZ+quaternion** Format to represent a pose. See *XYZ+quaternion format* (Section 13.1.2) for its definition.

**XYZABC format** Format to represent a pose. See *XYZABC format* (Section 13.1.1) for its definition.

# 2 Safety

**Warning:** The operator must have read and understood all of the instructions in this manual before handling the *rc_visard* sensor.

**Note:** The term "operator" refers to anyone responsible for any of the following tasks performed in conjunction with *rc_visard*:

- Installation
- Maintenance
- Inspection
- Calibration
- Programming
- Decommissioning

This manual explains the *rc_visard*'s various components and general operations regarding the product's whole life-cycle, from installation through operation to decommissioning.

The drawings and photos in this documentation are representative examples; differences may exist between them and the delivered product.

## 2.1 General warnings

**Note:** Any use of the *rc_visard* in noncompliance with these warnings is inappropriate and may cause injury or damage as well as void the warranty.

**Warning:**

- The *rc_visard* needs to be properly mounted before use.
- All cable sets need to be secured to the *rc_visard* and the mount.
- Cords must be at most 30 m long.
- An appropriate DC power source must supply power to the *rc_visard*.
- Each *rc_visard* must be connected to a separate power supply.
- The *rc_visard*'s housing must be grounded.
- The *rc_visard*'s and any related equipment's safety guidelines must always be satisfied.
- The *rc_visard* does not fall under the purview of the machinery, low voltage, or medical directives.

#### Risk assessment and final application:

The *rc_visard* may be used on a robot. Robot, *rc_visard*, and any other equipment used in the final application must be evaluated with a risk assessment. The system integrator's duty is to ensure respect for all local safety measures and regulations. Depending on the application, there may be risks that need additional protection/safety measures.

## 2.2 Intended use

The *rc_visard* is intended for data acquisition (e.g., images, disparity images, and egomotion) in stationary and mobile robotic applications. The *rc_visard* is intended for installation on a robot, automated machinery, mobile platform, or stationary equipment. It can also be used for data acquisition in other applications.

**Warning:** The *rc_visard* is **NOT** intended for safety critical applications.

The GigE Vision® industry standard used by the *rc_visard* does not support authentication and encryption. All data from and to the sensor is transmitted without authentication and encryption and could be monitored or manipulated by a third party. It is the operator's responsibility to connect the *rc_visard* only to a secured internal network.

**Warning:** The *rc_visard* must be connected to secured internal networks.

The *rc_visard* may be used only within the scope of its technical specification. Any other use of the sensor is deemed unintended use. Roboception will not be liable for any damages resulting from any improper or unintended use.

**Warning:** Always comply with local and/or national laws, regulations and directives on automation safety and general machine safety.

# 3 Hardware specification

**Note:** The following hardware specifications are provided here as a general reference; differences with the product might exist.

## 3.1 Scope of delivery

Standard delivery for an *rc_visard* includes the *rc_visard* sensor and a quickstart guide only. The full manual is available in digital form and is always installed on the sensor, accessible through the *Web GUI* (Section 4.5), and available at http://www.roboception.com/documentation.

**Note:** The following items are not included in the delivery unless otherwise specified:
- Couplings, adapters, mounts
- Power supply unit, cabling, and fuses
- Network cabling

Please refer to *Accessories* (Section 10) for suggested third-party cable vendors.

A connectivity kit can be purchased for the *rc_visard*. It contains an M12 to RJ45 network cable, 24 V power supply, and a DC plug to M12 power adapter. Please refer to *Accessories* (Section 10) for details.

**Note:** The connectivity kit is intended only for initial setup, not for permanent installation in industrial environment.

The following picture shows the important parts of the *rc_visard* which are referenced later in the documentation.

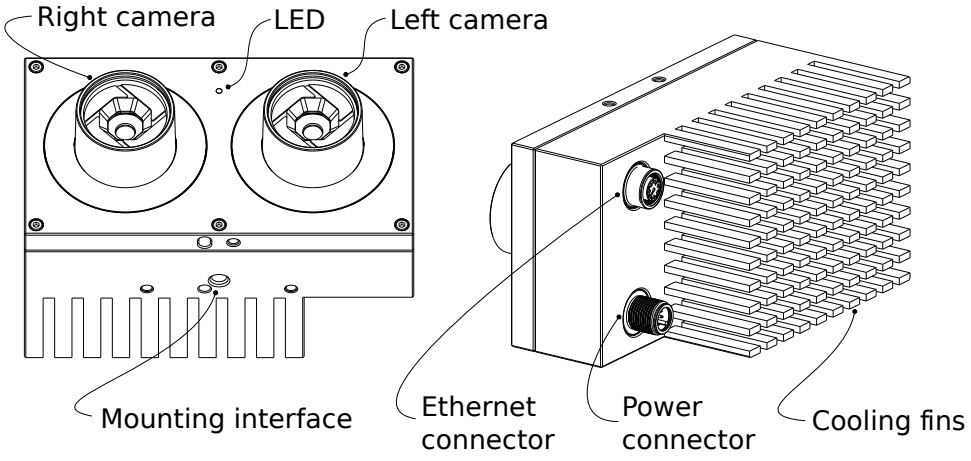



Fig. 3.1: Parts description

## 3.2 Technical specification

The common technical specifications for both *rc_visard* variants are given in Table 3.1.

Table 3.1: Common technical specifications for both *rc_visard* models

| | ***rc_visard* 65 / *rc_visard* 160** |
|---|---|
| Image resolution | 1280 x 960 pixel, color or monochrome |
| Field of view | Horizontal: 61°, Vertical: 48° |
| IR Cutoff | 650 nm |
| Depth image | 640 x 480 pixel (high) @ 3 Hz<br>320 x 240 pixel (medium) @ 15 Hz<br>214 x 160 pixel (low) @ 25 Hz |
| Egomotion | 200 Hz, low latency |
| Computing unit | Nvidia Tegra K1 |
| Power supply | 18 V to 30 V |
| Cooling | Passive |

The *rc_visard* 65 and *rc_visard* 160 differ in their baselines, which affects depth range and resolution as well as the sensors' size and weight.

Table 3.2: Differing technical specifications for the *rc_visard* variants

| | ***rc_visard* 65** | ***rc_visard* 160** |
|---|---|---|
| Baseline | 65 mm | 160 mm |
| Depth range | 0.2 m to infinity | 0.5 m to infinity |
| Depth resolution | 0.5 mm @ 0.2 m<br>15 mm @ 1.0 m | 1.5 mm @ 0.5 m<br>6 mm @ 1.0 m<br>23 mm @ 2.0 m<br>50 mm @ 3.0 m |
| Size (W x H x L) | 135 mm x 75 mm x 96 mm | 230 mm x 75 mm x 84 mm |
| Mass | 0.68 kg | 0.84 kg |

The *rc_visard* can be equipped with on-board software modules such as SLAM for additional features. These software modules can be ordered and require a license update.

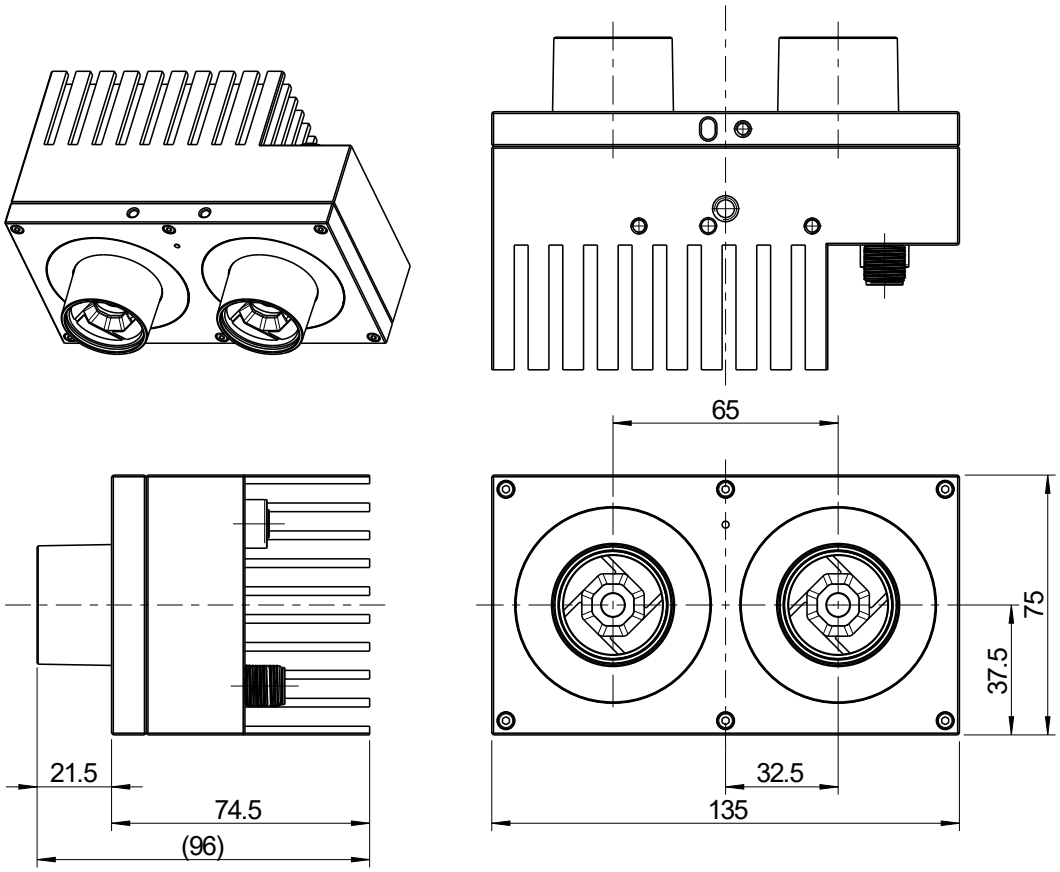



Fig. 3.2: Overall dimensions of the *rc_visard* 65

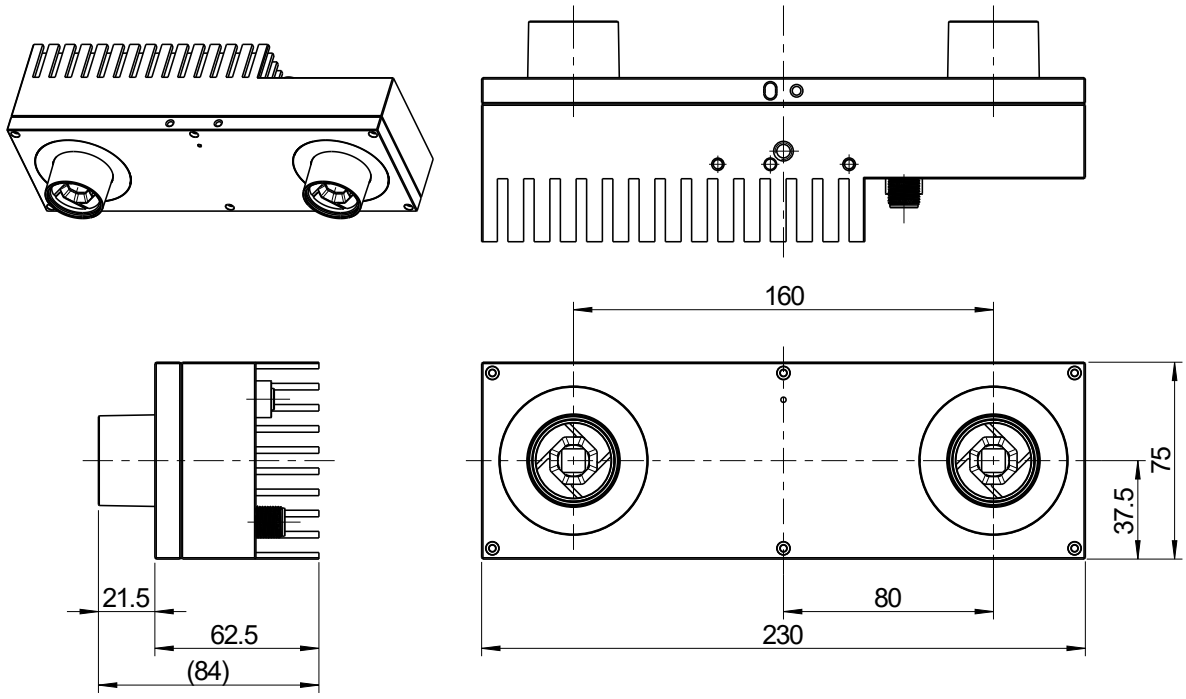



Fig. 3.3: Overall dimensions of the *rc_visard* 160

CAD models of the *rc_visard* can be downloaded from http://www.roboception.com/download. The CAD models are provided as-is, with no guarantee of correctness. When a material property of aluminium is assigned (density of $2.76 \frac{g}{cm^3}$), the mass properties of the CAD model are within 5% of the product with respect to weight and center of mass, and within 10% with respect to moment of inertia.

## 3.3 Environmental and operating conditions

The *rc_visard* is designed for industrial applications. Always respect the storage, transport, and operating environmental conditions outlined in Table 3.3.

Table 3.3: Environmental conditions

| | ***rc_visard* 65 / *rc_visard* 160** |
|---|---|
| Storage/Transport temperature | -25 °C to 70 °C |
| Operating temperature | 0 °C to 50 °C |
| Relative humidity (non condensing) | 20 % to 80 % |
| Vibration | 5 g |
| Shock | 50 g |
| Protection class | IP54 |
| Others | • Free from corrosive liquids or gases<br>• Free from explosive liquids or gases<br>• Free from powerful electromagnetic interference |

The *rc_visard* is designed for an operating (surrounding environment) temperature of 0 °C to 50 °C and relies on convective (passive) cooling. Unobstructed airflow, especially around the cooling fins, needs to be ensured during use. The *rc_visard* should only be mounted using the provided mechanical mounting interface, and each part of the housing must remain uncovered. A free space of at least 10 cm extending in all directions from the housing, and sufficient air exchange with the environment is required to ensure adequate cooling. Cooling fins must be free of dirt and other contamination.

The housing temperature depends on the processing load, sensor orientation, and surrounding environmental temperatures. When the sensor's exposed housing surfaces exceed 60°C, the LED at the front will turn from green to red.

**Warning:** For hand-guided applications, a heat-insulated handle should be attached to the sensor to reduce the risk of burn injuries due to skin exposure to surface temperatures exceeding 60°C.

## 3.4 Power-supply specifications

The *rc_visard* needs to be supplied by a DC voltage source. The *rc_visard*'s standard package doesn't include a DC power supply. The power supply contained in the connectivity kit may be used for initial setup. For permanent installation, it is the customer's responsibility to provide suitable DC power. The sensor is qualified as industrial equipment Class A under EN55011. As such, each *rc_visard* must be connected to a separate power supply. Connection to domestic grid power is only allowed through a power supply certified as EN55011 Class B.

Table 3.4: Absolute maximum ratings for power supply

| | ***Min*** | ***Nominal*** | ***Max*** |
|---|---|---|---|
| Supply voltage | 18.0 V | 24 V | 30.0 V |
| Max power consumption | | | 25 W |
| Overcurrent protection | Supply must be fuse-protected to a maximum of 2 A | | |
| EMC compliance | Industrial equipment under EN55011 Class A | | |

**Warning:** Exceeding maximum power rating values may lead to damage of the *rc_visard*, power supply, and connected equipment.

**Warning:** A separate power supply must power each *rc_visard*.

**Warning:** Connection to domestic grid power is allowed through a power supply certified as EN55011 Class B only.

## 3.5 Wiring

Cables are not provided with the *rc_visard* standard package. It is the customer's responsibility to obtain the proper cabling. *Accessories* (Section 10) provides an overview of suggested components.

**Warning:** Proper cable management is mandatory. Cabling must always be secured to the *rc_visard* mount with a strain-relief clamp so that no forces due to cable movements are exerted on the *rc_visard*'s M12 connectors. Enough slack needs to be provided to allow for full range of movement of the *rc_visard* without straining the cable. The cable's minimum bend radius needs to be observed.

The *rc_visard* provides an industrial 8-pin A-coded M12 socket connector for Ethernet connectivity and an 8-pin A-coded M12 plug connector for power and GPIO connectivity. Both connectors are located at the back. Their locations (distance from centerlines) are identical for the *rc_visard* 65 and *rc_visard* 160. The location of both connectors on the *rc_visard* 65 is shown as an example in Fig. 3.4.

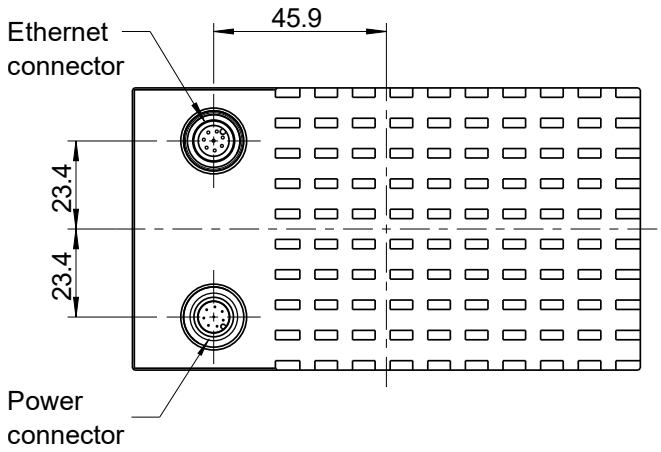



Fig. 3.4: Locations of the electrical connections for the *rc_visard* 65, with Ethernet on top and power on the bottom

Connectors are rotated so that standard 90° angled connectors will exit horizontally, away from the camera (away from the cooling fins).

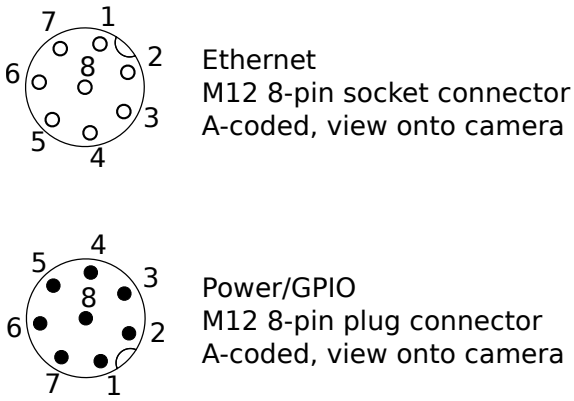



Fig. 3.5: Pin positions for power and Ethernet connector

Pin assignments for the Ethernet connector are given in Fig. 3.6.

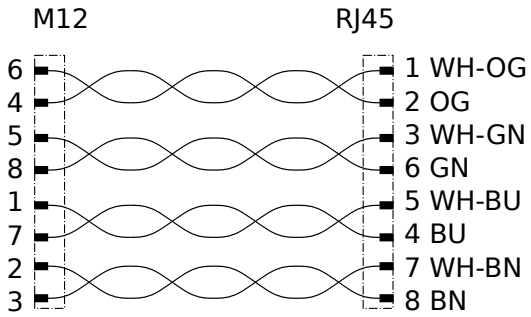



Fig. 3.6: Pin assignments for M12 to Ethernet cabling

Pin assignments for the power connector are given in Table 3.5.

Table 3.5: Pin assignments for the power connector

| **Pin** | ***Assignment*** |
|---|---|
| 1 | GPIO In 2 |
| 2 | Power |
| 3 | GPIO In 1 |
| 4 | GPIO Gnd |
| 5 | GPIO Vcc |
| 6 | GPIO Out 1 (image exposure) |
| 7 | Gnd |
| 8 | GPIO Out 2 |

GPIOs are decoupled by photocoupler. *GPIO In 1* and *2* as well as *GPIO Out 2* are reserved for future functionality and currently provide no user-accessible functionality. These pins should be left floating.

**Warning:** It is especially important that during the boot phase *GPIO In 1* is left floating or remains low. The *rc_visard* will not boot if the pin is high during boot time.

*GPIO Out 1* by default provides an exposure sync signal with a logic high level for the duration of the exposure.

GPIO circuitry and specifications are shown in Fig. 3.7. The maximum rated voltage for *GPIO In* and *GPIO Vcc* is 30 V.

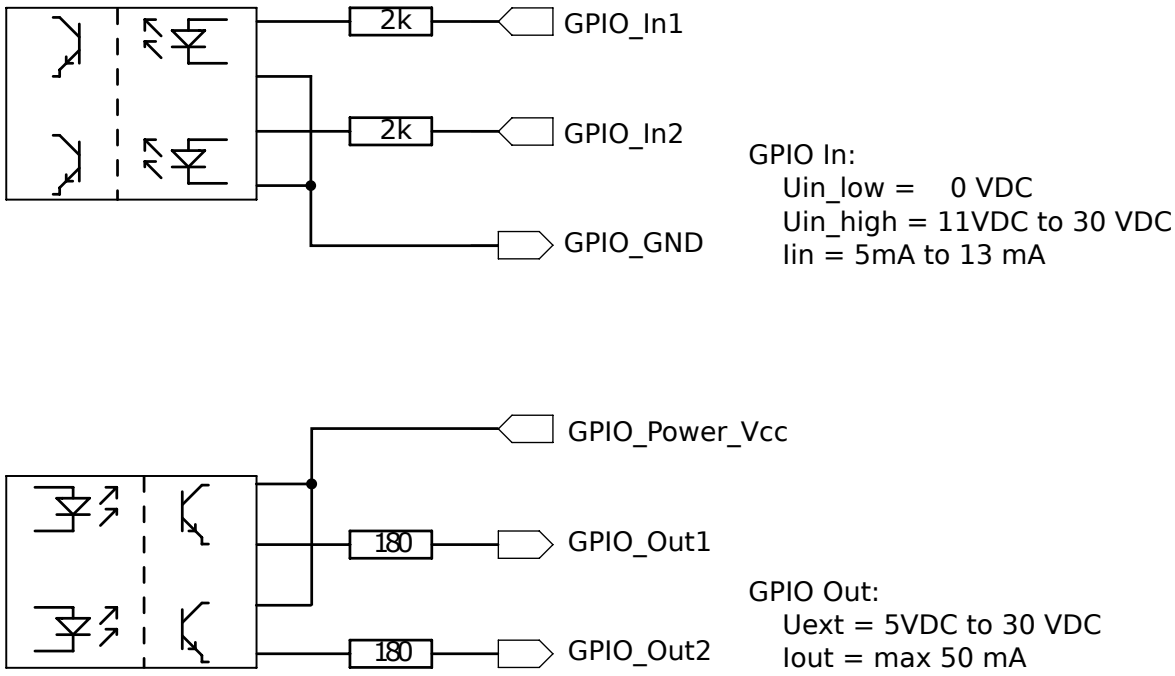



Fig. 3.7: GPIO circuitry and specifications – do not connect signals higher than 30 V

**Warning:** Do not connect signals with voltages higher than 30 V to the *rc_visard*.

## 3.6 Mechanical interface

The *rc_visard* 65 and *rc_visard* 160 offer identical mounting-point setups at the bottom.

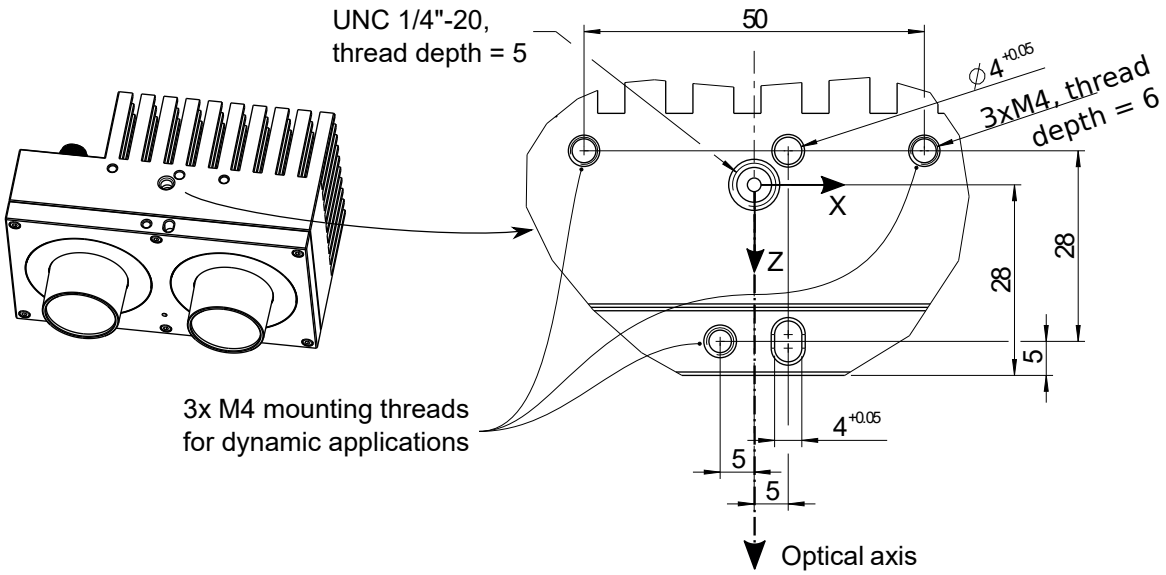



Fig. 3.8: Mounting-point for connecting the *rc_visard* to robots or other mountings

For troubleshooting and static applications, the sensor may be mounted using the standardized tripod thread (UNC 1/4”-20) indicated at the coordinate-frame origin. For dynamic applications such as mounting on a robotic arm, the sensor must be mounted with three M4 (metric standard) 8.8 machine screws tightened to 2.5 Nm and secured with a medium-strength threadlocking adhesive such as Loctite 243. Maximum thread depth is 6 mm. The two 4 mm diameter holes may be used for positioning pins (ISO 2338 4 m6) to ensure precise repositioning of the sensor.

**Warning:** For dynamic applications, the *rc_visard* must be mounted with three M4 8.8 machine screws tightened to 2.5 Nm torque and secured with threadlocking adhesive. Do not use high-strength bolts. The engaged thread depth must be at least 5 mm.

## 3.7 Coordinate frames

The *rc_visard*’s coordinate-frame origin is defined as the exit pupil of the left camera lens. This frame is called sensor coordinate frame or camera coordinate frame. An approximate location for the *rc_visard* 65 is shown in the next image.

The mounting-point frame for both *rc_visard* devices is defined to be at the bottom, centered in the tripod thread, with orientation identical to that of the sensor’s coordinate frame. Fig. 3.9 shows approximate offsets.

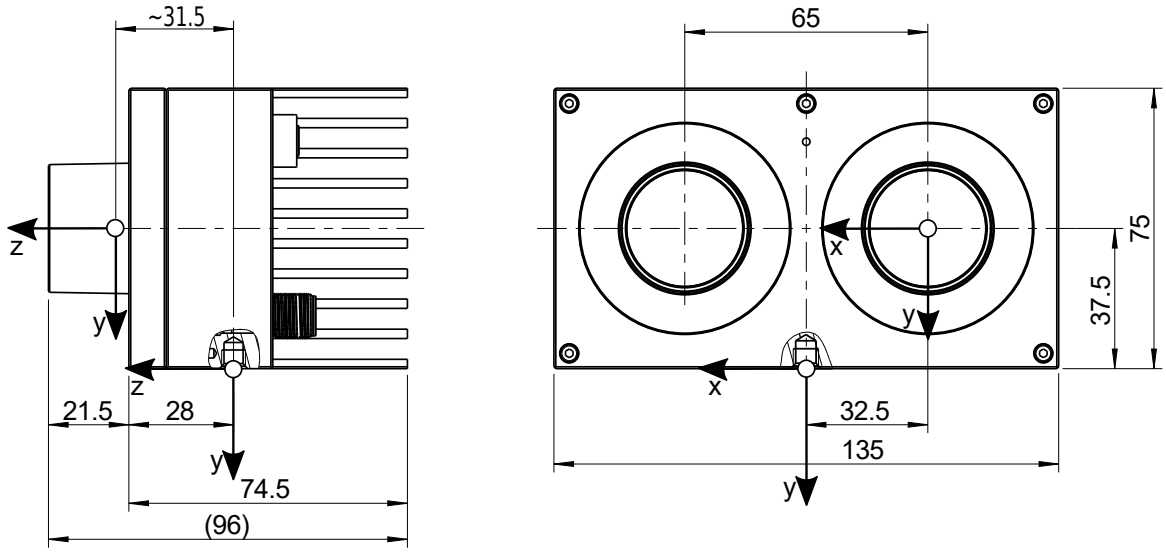



Fig. 3.9: Approximate location of sensor/camera coordinate frame (inside left lens) and mounting-point frame (at tripod thread) for the *rc_visard* 65

Approximate locations of sensor/camera coordinate frame and mounting-point frame for the *rc_visard* 160 are shown in Fig. 3.10.

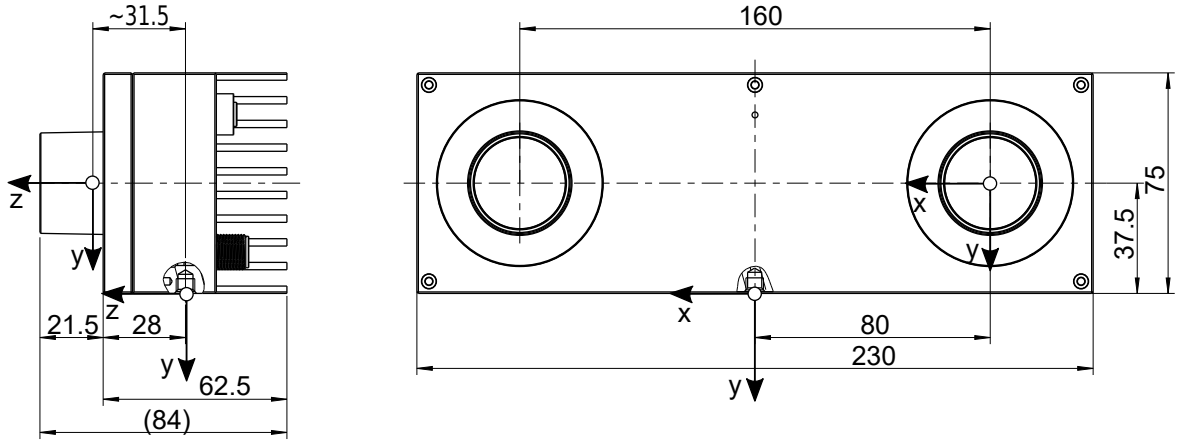



Fig. 3.10: Approximate locations of sensor/camera coordinate frame (inside left lens) and mounting-point frame (at tripod thread) for the *rc_visard* 160

**Note:** The correct offset between the sensor/camera frame and a robot coordinate frame can be calibrated through the *hand-eye-calibration procedure* (Section 6.7).

# 4 Installation

**Warning:** The instructions on *Safety* (Section 2) related to the *rc_visard* must be read and understood prior to installation.

## 4.1 Installation and configuration

The *rc_visard* offers a Gigabit Ethernet interface for connecting the device to a computer network. All communications to and from the device are performed via this interface. The *rc_visard* has an on-board computing resource that requires booting time after powering up the device.

## 4.2 Power up

**Note:** Always fully connect and tighten the M12 power connector on the *rc_visard before* turning on the power supply.

After connecting the *rc_visard* to the power, the LED on the front of the device should immediately illuminate. During the device's boot process, the LED will change color and will eventually turn green. This signals that all processes are up and running.

If the network is not plugged in or the network is not properly configured, then the LED will flash red every 5 seconds. In this case, the device's network configuration should be verified. See *LED colors* (Section 11.1) for more information on the LED color codes.

## 4.3 Network configuration

The *rc_visard* requires an Internet Protocol (*IP*) address for communication with other network devices. The IP address must be unique in the local network, and can be set automatically or manually.

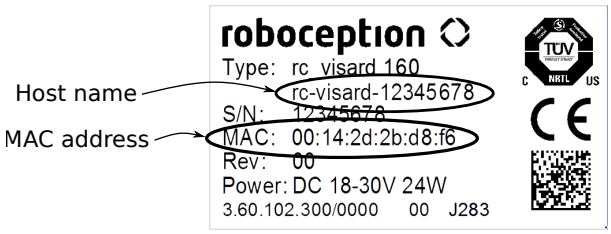



Fig. 4.1: Label on the *rc_visard*

### 4.3.1 Automatic configuration (factory default)

The Dynamic Host Configuration Protocol (*DHCP*) is preferred for setting an IP address. If DHCP is active on the *rc_visard*, which is the factory default, then the device tries to contact a DHCP server at startup and every time the network cable is plugged in. If a DHCP server is available on the network, then the IP address is automatically configured.

In some networks, the DHCP server is configured so that it only accepts known devices. In this case, the Media Access Control address (*MAC address*), which is printed on the sensor label, needs to be configured in the DHCP server. At the same time, the sensor's host name can also be set in the Domain Name Server (*DNS*). The host name is defined as `rc-visard-<serial number>`, which is also printed on the sensor. Both MAC address and host name should be sent to the network administrator for configuration.

If the *rc_visard* cannot contact a DHCP server for about 15 seconds after startup or after plugging in the network cable, it will try to assign itself a unique IP address. This process is called *Link Local*. This option is especially useful for connecting the *rc_visard* directly to a computer. The computer must be configured for Link Local as well. Link Local might already be configured as a standard fallback option, as it is under Windows 10. Other operating systems such as Linux require Link Local to be explicitly configured in their network managers.

### 4.3.2 Manual configuration

Specifying a persistent IP address manually might be useful in come cases. This is done via the sensor's standard *GigE Vision*® 2.0 interface, and requires a configuration tool to be installed on the host computer. We recommend using the IpConfigTool that is part of the Baumer GAPI SDK. The *SDK* can be downloaded free of charge for Windows and Linux from http://www.baumer.com.

After the configuration tool starts, it scans for all available GigE Vision® sensors on the network. All *rc_visard* devices can be uniquely identified by their serial number and MAC address, which are both printed on the device. If the device cannot be found, it can also be connected directly to the computer for configuration (see *Automatic configuration (factory default)*, Section 4.3.1).

**Warning:** The IP address must be unique and within the local network's range of valid addresses. Furthermore, the subnet mask must match the local network; otherwise, the *rc_visard* may become inaccessible. This can be avoided by using automatic configuration as explained in *Automatic configuration (factory default)* (Section 4.3.1).

## 4.4 Discovery of *rc_visard* devices

Devices that are powered up and connected to the local network or directly to a computer (see *Network configuration*, Section 4.3) can be found using the standard GigE Vision® discovery mechanism. Roboception offers the open-source tool `rcdiscover-gui`, which can be downloaded free of charge from http://www.roboception.com/download for Windows and Linux. The tool's Windows version consists of a single executable for Windows 7 and Windows 10, which can be executed without installation. For Linux an installation package is available for Ubuntu 14.04 and 16.04. At startup, all available *rc_visard* devices are listed with their names, serial numbers, current IP addresses, and unique MAC addresses. The discovery tool finds all devices reachable by global broadcasts. Misconfigured devices that are located in different subnets than the computer may also be listed. An icon in the discovery tool indicates whether devices are actually reachable via a web browser.

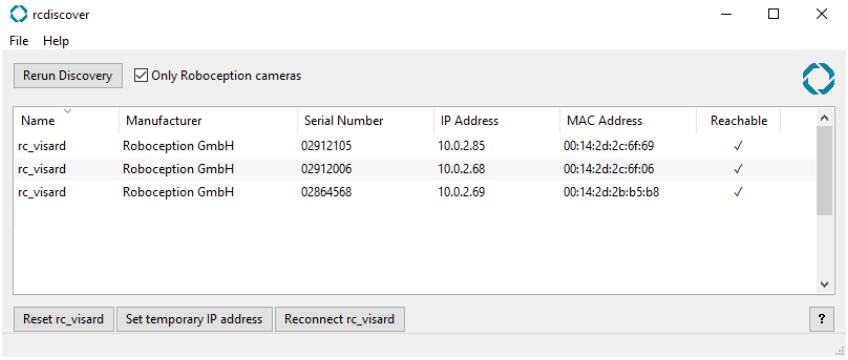



Fig. 4.2: `rcdiscover-gui` tool for finding connected *rc_visard* devices

After successful discovery, a double click on the device row opens the *Web GUI* (Section 4.5) of the device in the operating system's default web browser. Mozilla Firefox is recommended as web browser.

### 4.4.1 Resetting configuration

A misconfigured device can be reset by using the *Reset rc_visard* button in the discovery tool. The reset mechanism is only available for two minutes after device startup. Thus, the *rc_visard* may require rebooting before being able to reset the device.

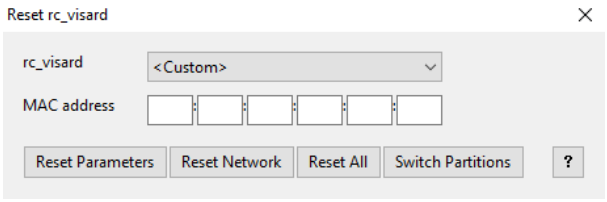



Fig. 4.3: Reset dialog of the `rcdiscover-gui` tool

If the discovery tool still successfully detects the the misconfigured *rc_visard*, then the latter can be selected from the *rc-visard* drop-down menu. Otherwise, the *rc_visard*'s MAC address, which is printed on the device label, can be entered manually into the designated fields.

One of four options can be chosen after entering the MAC address:

- *Reset Parameters*: Reset all *rc_visard* parameters, such as frame rate, that are configurable via *Web GUI* (Section 4.5).
- *Reset Network*: Reset network settings and user-defined name.
- *Reset All*: Reset the *rc_visard* parameters as well as network settings and user-defined name.
- *Switch Partitions*: Allows a rollback to be performed as described in *Restoring the previous firmware version* (Section 9.4).

A white status LED followed by a device reboot indicates a successful reset. If no reaction is noticeable, the two minutes time slot may have elapsed, requiring another reboot.

**Note:** The reset mechanism is only available for the first two minutes after startup.

## 4.5 Web GUI

The *rc_visard*'s Web GUI can be used to test, calibrate, and configure on-board processing. It can be accessed from any web browser, such as Firefox, Google Chrome, or Microsoft Edge, via the sensor's IP address. The easiest way to access the Web GUI is to simply double click on the desired device using the `rcdiscover-gui` tool as explained in *Discovery of rc_visard devices* (Section 4.4).

Alternatively, some network environments automatically configure the unique host name of the *rc_visard* in their Domain Name Server (*DNS*). In this case, the Web GUI can also be accessed directly using the *URL* `http://rc-visard-<serial-number>` by replacing `<serial-number>` with the serial number printed on the device label.

For Linux and Mac operating systems, this even works without DNS via the multicast Domain Name System (*mDNS*), which is automatically used if `.local` is appended to the host name. Thus, the URL simply becomes `http://rc-visard-<serial-number>.local`.

The Web GUI's overview page gives the most important information about on-board processing.

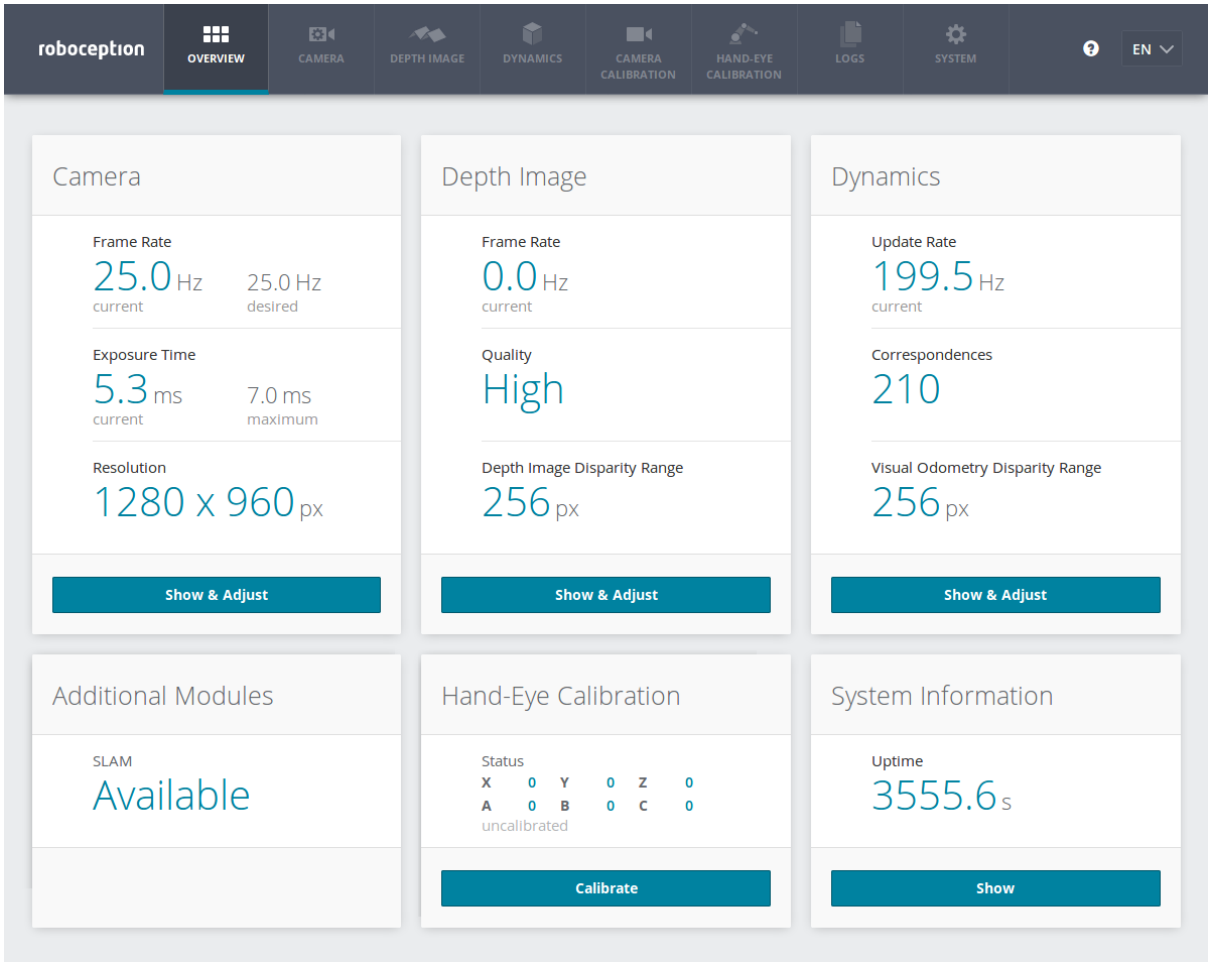



Fig. 4.4: Overview page of the *rc_visard*'s Web GUI

The page's top row permits access to the individual *rc_visard* modules.

- The *Camera* module shows a live stream of the device's left and right rectified images. The frame rate can be reduced to save bandwidth when streaming to a GigE Vision® client. Furthermore, exposure can be set manually or automatically. See *Parameters* (Section 6.1.3) for more information.
- The *Depth Image* module shows a live stream of the left rectified, depth, and confidence images. The page contains various settings for depth-image computation and filtering. See *Parameters* (Section 6.2.4) for

more information.

- The *Dynamics* module shows the location and movement of image features that are used to compute the *rc_visard*'s egomotion. Settings include the number of corners and features that should be used. See *Parameters* (Section 6.4.1) for more information.
- The *Camera Calibration* module permits the camera to be checked for proper calibration. In rare cases when the camera is no longer sufficiently calibrated, calibration also can be performed using this module. See *Camera calibration* (Section 6.6) for more information.
- The *Hand-Eye-Calibration* module allows the computation of the static transformation between the *rc_visard* and a coordinate system known in the robot system. This can be the flange coordinate system of a robotic arm if the *rc_visard* is attached to the flange. Alternatively, the *rc_visard* may be mounted statically in the robot environment and calibrated to any other static frame known in the robot system. See *Hand-eye calibration* (Section 6.7) for more information.
- The *Logs* module permits access to the log files on the *rc_visard*. The log files are typically checked if errors are suspected.
- The *System* module permits the firmware or the license file to be updated and provides some general information about the device.

Changed parameters are not persistent and will be lost when restarting the *rc_visard* unless they are saved by pressing the *Save* button before leaving the corresponding page.

Further information on all parameters in the Web GUI can be obtained by pressing the *Info* button next to each parameter.

# 5 The *rc_visard* in a nutshell

The *rc_visard* is a self-registering 3D camera. It provides rectified camera, disparity, confidence, and error images, which enable the viewed scene's depth values along with their uncertainties to be computed. Furthermore, the motion of visual features in the images is combined with acceleration and turn-rate measurements at a high rate, which enables the sensor to provide real-time estimates of its current pose, velocity, and acceleration.

## 5.1 Stereo vision

The *rc_visard* is based on *stereo vision* using the *SGM* (*Semi-Global Matching*) method. In stereo vision, 3D information about a scene can be extracted by comparing two images taken from different viewpoints. The main idea behind using a camera pair for measuring depth is the fact that object points appear at different positions in the two camera images depending on their distance from the camera pair. Very distant object points appear at approximately the same position in both images, whereas very close object points occupy different positions in the left and right camera image. The object points' displacement in the two images is called *disparity*. The larger the disparity, the closer the object is to the camera. The principle is illustrated in Fig. 5.1.

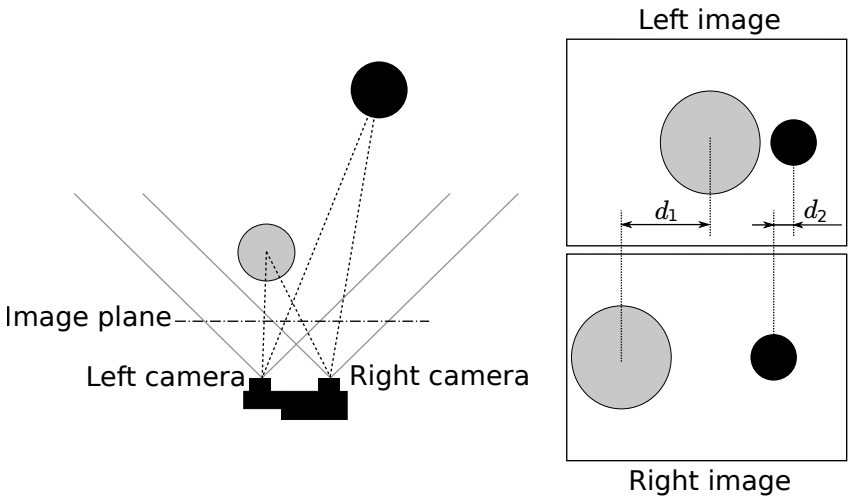



Fig. 5.1: Sketch of the stereo-vision principle: The more distant object (black) exhibits a smaller disparity $d_2$ than that of the close object (gray), $d_1$.

Stereo vision is a form of passive sensing, meaning that it emits neither light nor other signals to measure distances, but uses only light that the environment emits or reflects. The *rc_visard* can thus work indoors and outdoors and multiple *rc_visard* devices can work together without interferences.

To compute the 3D information, the stereo matching algorithm must be able to find corresponding object points in the left and right camera images. For this, the algorithm requires texture, meaning changes in image intensity values due to patterns or the objects' surface structure, in the images. Stereo matching is not possible for completely untextured regions, such as a flat white wall without any visible surface structure. The *SGM* stereo matching method used provides the best trade-off between runtime and accuracy, even for fine structures.

For stereo matching, the position and orientation of the left and right cameras relative to each other has to be known with very high accuracy. This is achieved by calibration. The *rc_visard*'s cameras are pre-calibrated during production. However, if the *rc_visard* has been decalibrated, during transport for example, then the user has to recalibrate the stereo camera.

The following *rc_visard* software components are required to compute 3D information:

- *Stereo camera*: This component is responsible for capturing synchronized stereo image pairs and transforming them into images approaching those taken by an ideal stereo camera (rectification) (Section 6.1).
- *Stereo matching*: This component computes disparities for the rectified stereo camera pair using *SGM* (Section 6.2).
- *Camera calibration*: This component enables the user to recalibrate the *rc_visard*'s stereo camera (Section 6.6).

## 5.2 Sensor dynamics

In addition to providing 3D information about the scene, the *rc_visard* can also estimate its *egomotion* or *dynamic state* in real time. This comprises its current pose, i.e., its position and orientation relative to a reference coordinate system or reference frame, as well as its velocity and acceleration. Measurements from stereo visual odometry (SVO) and the integrated Inertial Measurement Unit (*IMU*) are fused to compute this information. This combination is called a Visual Inertial Navigation System (V*INS*).

Visual odometry observes the motion of characteristic points in the camera images to estimate the camera motion. Object points are projected on different pixels in the camera image depending on the camera's viewing position. Each point's 3D coordinates can also be computed using stereo matching between the point positions in the left and right camera images. Thus, for two different viewing positions A and B, two sets of corresponding 3D points are computed. Assuming a static environment, the motion that transforms one set of points into the other is the camera's motion. The principle is illustrated for a simplified 2D case in Fig. 5.2.

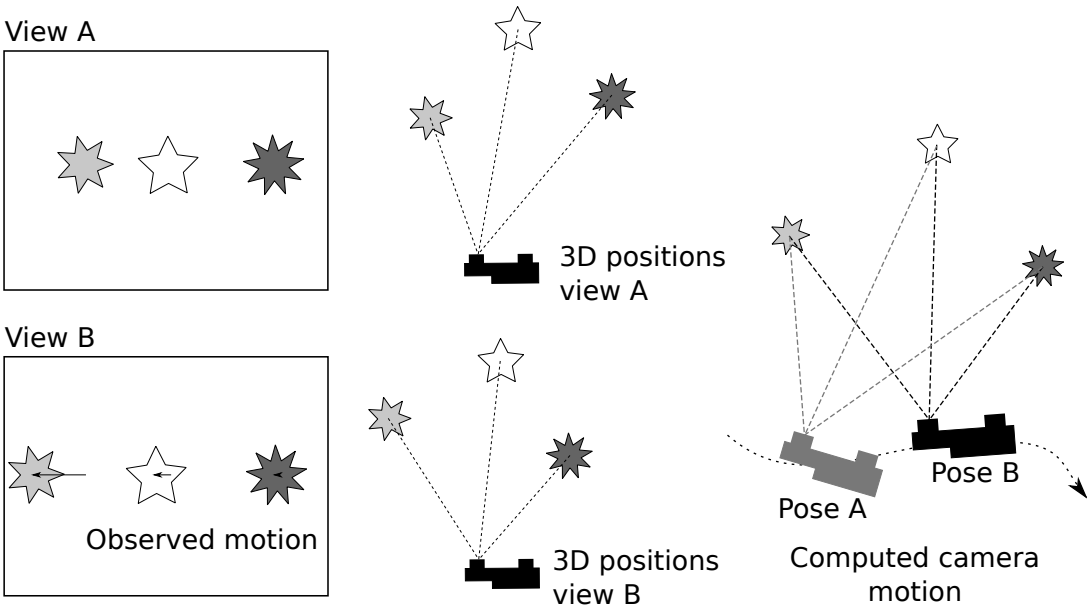



Fig. 5.2: Simplified sketch of the stereo visual odometry principle for 2D motions: Camera motion is computed from the observed motion of characteristic image points.

Since visual odometry relies on image-data quality, motion estimates deteriorate when the images are blurred or are poorly illuminated. Furthermore, visual odometry's frame rate is too low for control applications. That's why the *rc_visard* has an integrated Inertial Measurement Unit (IMU), which measures the accelerations and angular velocities that occur when the *rc_visard* moves. It also measures acceleration due to gravity, which

gives global orientation in the vertical direction. Further, IMU measurements have a high rate of 200 Hz. The *rc_visard*'s linear velocity, position, and orientation can be computed by integrating the IMU measurements. However, the integration results suffer from increasing drift over time. The *rc_visard* thus fuses accurate, but low-frequency and sometimes volatile visual odometry measurements with reliable high-rate IMU measurements to provide an accurate, robust, high-frequency estimate of the *rc_visard*'s current position, orientation, velocity, and acceleration, which can be used in a control loop.

In addition to the stereo camera component and the calibration component, pose-estimate computations require the following *rc_visard* software components:

- *Sensor dynamics*: This component handles starting, stopping, and streaming of the estimates for the individual components (Section 6.3).
  - *Visual odometry*: This component computes a motion estimate from the camera images (Section 6.4).
  - *Stereo INS*: This component fuses the motion estimates from visual odometry with the measurements from the integrated IMU to provide real-time pose estimates at a high frequency (Section 6.5).
  - *SLAM*: This component is optionally available for the *rc_visard* and creates an internal map of the environment, which is used to correct pose errors (Section 7.1).

## 5.3 Calibration relative to a robot

The *rc_visard* is designed for industrial environments including those featuring robotic applications in which the *rc_visard* is either mounted on a robot or statically in a robot work cell. To use the *rc_visard*'s output, the robot must know where the sensor is located in the robot coordinate frame. To compute the *rc_visard*'s location in the robot coordinate frame, the sensor offers the so-called *Hand-eye calibration software component* (Section 6.7). The calibration routine can be executed either programmatically via the *REST-API interface* or manually via the *Web GUI* (Section 4.5).

# 6 Software components

The *rc_visard* comes with several on-board software components, which provide camera images, 3D information, and dynamics state estimates, and allow calibration to be performed. Each software component corresponds to a *node* in the *REST-API interface* (Section 8.2). Fig. 6.1 gives an overview of the relationships between the different software components and the data they provide via *rc_visard*'s various *interfaces* (Section 8).

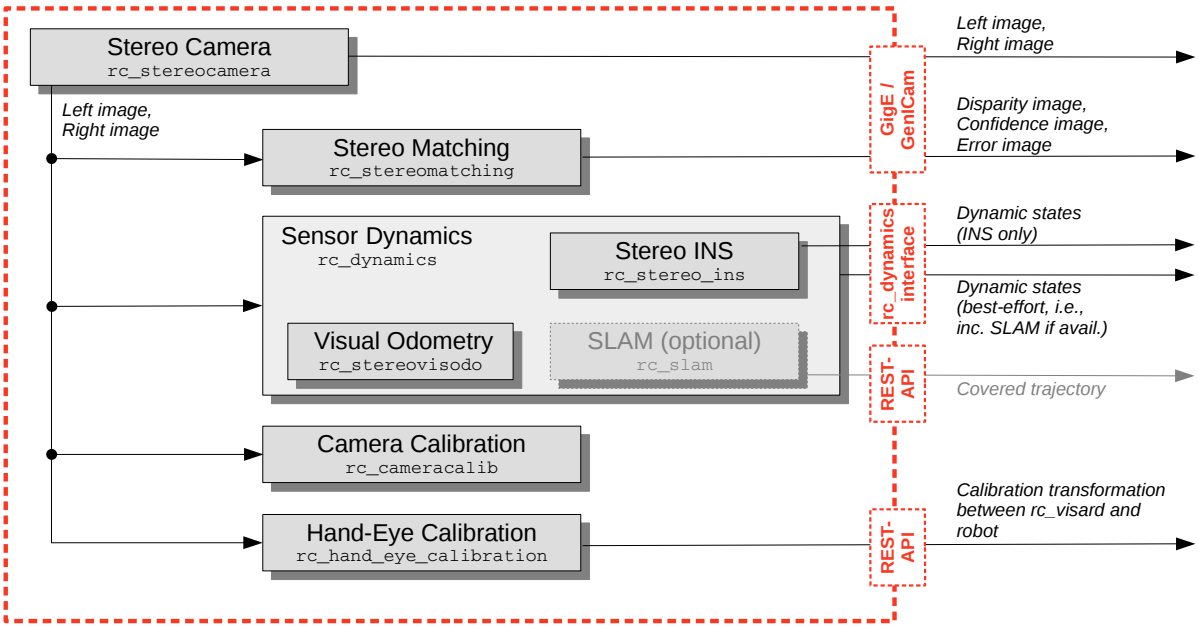



Fig. 6.1: Flowchart of the software components with their node names and the most important outputs

**Note:** Components marked as *optional* extend the *rc_visard*'s features. Customers can extend the license to purchase additional components.

The *rc_visard*'s on-board software consists of the following components:

- ***Stereo camera*** **(`rc_stereocamera`, Section 6.1)** acquires stereo image pairs and performs planar rectification for using the stereo camera as a measurement device. Images are provided both for further internal processing by other components and for external use as *GenICam image streams*.
- ***Stereo matching*** **(`rc_stereomatching`, Section 6.2)** uses the rectified stereo image pair to compute 3D depth information such as disparity, error, and confidence images. These are provided as GenICam streams, too.
- ***Sensor dynamics*** **(`rc_dynamics`, Section 6.3.)** provides estimates of *rc_visard*'s dynamic state such as its pose, velocity, and acceleration. These states are transmitted as continuous data streams via the *rc_dynamics interface*. For this purpose, the dynamics component manages and fuses data from the following individual subcomponents:

  - *Visual odometry* (**rc_stereovisodo, Section 6.4**) estimates the motion of the *rc_visard* device based on the motion of characteristic visual features in the left camera images.
  - *Stereo INS* (**rc_stereo_ins, Section 6.5**) combines visual odometry measurements with readings from the on-board Inertial Measurement Unit (IMU) to provide accurate and high-frequency state estimates in real time.

- *Camera calibration* (**rc_cameracalib, Section 6.6**) automatically checks and performs the self-calibration of the *rc_visard*'s stereo camera in case it has been decalibrated. It furthermore enables the user to check and perform recalibration manually via the *WEB GUI* (Section 4.5).
- *Hand-eye calibration* (**rc_hand_eye_calibration, Section 6.7**) enables the user to calibrate the *rc_visard* with respect to a robot, either via the Web GUI or the REST-API.

## 6.1 Stereo camera

The stereo camera component contains functionality for acquiring stereo image pairs and performing planar rectification needed to use the stereo camera as a measurement device.

### 6.1.1 Image acquisition

Acquiring stereo image pairs is the first step toward stereo vision. Since both cameras are equipped with global shutters and their chips are hardware-synchronized, all pixels of both camera images are always exposed at the exactly same time. *GPIO out 1* (Section 3.5) signals the respective exposure time. Additionally, the time in the middle of the image exposure is attached to the images as a timestamp. This timestamp becomes important for dynamic applications in which the *rc_visard* or the scene moves.

Exposure time can be set manually to a fixed value. This is useful in an environment where lighting is controlled so that it is always at the same intensity. The camera is set to auto exposure by default. In this mode, the *rc_visard* chooses the exposure time automatically, up to a user defined maximum. The permitted maximum is meant to limit the motion blur that occurs when taking images while the *rc_visard* or the scene is moving. The maximum exposure time thus depends on the application. If the maximum exposure time is reached, the auto-exposure algorithm uses the gain to increase image brightness. However, larger gain factors also amplify image noise. Thus, the maximum exposure time trades motion blur off against image noise under weak-light conditions.

### 6.1.2 Planar rectification

Camera parameters such as focal length, lens distortion, and the relationship of the cameras to each other must be exactly known to use the stereo camera as a measuring instrument. The parameters are determined by calibration (see *Camera calibration*, Section 6.6). The *rc_visard* is already calibrated at production time and normally requires no recalibration. The camera parameters describe with great precision all of the stereo-camera system's geometric properties, but the resulting model is complex and difficult to use.

Rectification is the process of remapping the images according to an ideal stereo-camera model. Lens distortion is removed and the images are aligned so that an object point is always projected onto the same image row in both images. The cameras' optical axes become exactly parallel. This means that points at infinite distance are projected onto the same image column in both images. The closer an object point is, the larger is the difference between its image columns in the right and left images. This difference is called disparity.

Mathematically, the object point $P = (P_x, P_y, P_z)$ is projected onto image point $p_l = (p_{lx}, p_{ly}, 1)$ in the left rectified image and onto $p_r = (p_{rx}, p_{ry}, 1)$ in the right rectified image by

$$A = \begin{pmatrix} f & 0 & \frac{w}{2} \\ 0 & f & \frac{h}{2} \\ 0 & 0 & 1 \end{pmatrix}, \qquad T_s = \begin{pmatrix} t \\ 0 \\ 0 \end{pmatrix},$$

$$s_1 p_l = AP,$$
$$s_2 p_r = A(P - T_s).$$

The focal length $f$ is the distance between the common image plane and the optical centers of the left and right cameras. It is measured in pixels. The baseline $t$ is the distance between the optical centers of the two cameras. The image width $w$ and height $h$ are measured in pixels, too. $s_1$ and $s_2$ are scale factors ensuring that the third coordinates of the image points $p_l$ and $p_r$ are equal to 1.

The *rc_visard* provides the time-stamped, rectified left and right images over the GenICam interface (see *Provided image streams*, Section 8.1.2). Live streams of the images are provided with reduced quality in the *Web GUI* (Section 4.5).

**Note:** The *rc_visard* reports a focal length factor via its various interfaces. It relates to the image width for supporting different image resolutions. The focal length $f$ in pixels can be easily obtained by multiplying the focal length factor by the image width in pixels.

### 6.1.3 Parameters

The stereo-camera software component is called `rc_stereocamera` and is represented by the *Camera* tab in the *Web GUI* (Section 4.5). The user can change the camera parameters there, or directly via the REST-API (*REST-API interface*, Section 8.2) or GigE Vision (*GigE Vision 2.0/GenICam image interface*, Section 8.1).

**Note:** Camera parameters cannot be changed via the Web GUI or REST-API if *rc_visard* is used via GigE Vision.

#### Parameter overview

This component offers the following run-time parameters.

Table 6.1: The `rc_stereocamera` component's run-time parameters

| Name | Type | Min | Max | Default | Description |
|---|---|---|---|---|---|
| `exp_auto` | bool | False | True | True | Switching between auto and manual exposure |
| `exp_height` | int32 | 0 | 959 | 0 | Height of auto exposure region. 0 for whole image. |
| `exp_max` | float64 | 6.6e-05 | 0.018 | 0.007 | Maximum exposure time in seconds if exp_auto is true |
| `exp_offset_x` | int32 | 0 | 1279 | 0 | First column of auto exposure region |
| `exp_offset_y` | int32 | 0 | 959 | 0 | First row of auto exposure region |
| `exp_value` | float64 | 6.6e-05 | 0.018 | 0.005 | Manual exposure time in seconds if exp_auto is false |
| `exp_width` | int32 | 0 | 1279 | 0 | Width of auto exposure region. 0 for whole image. |
| `fps` | float64 | 1.0 | 25.0 | 25.0 | Frames per second in Hertz |
| `gain_value` | float64 | 0.0 | 18.0 | 0.0 | Manual gain value in decibel if exp_auto is false |
| `wb_auto` | bool | False | True | True | Switching white balance on and off (only for color camera) |
| `wb_ratio_blue` | float64 | 0.125 | 8.0 | 2.4 | Blue to green balance ratio if wb_auto is false (only for color camera) |
| `wb_ratio_red` | float64 | 0.125 | 8.0 | 1.2 | Red to green balance ratio if wb_auto is false (only for color camera) |

This component reports the following status values.

Table 6.2: The `rc_stereocamera` component's status values

| Name | Description |
|---|---|
| `baseline` | Stereo baseline $t$ in meters |
| `color` | 0 for monochrome cameras, 1 for color cameras |
| `exp` | Actual exposure time in seconds. This value is shown below the image preview in the Web GUI as *Exposure (ms)*. |
| `focal` | Focal length factor normalized to an image width of 1 |
| `fps` | Actual frame rate of the camera images in Hertz. This value is shown in the Web GUI below the image preview as *FPS (Hz)*. |
| `gain` | Actual gain factor in decibel. This value is shown in the Web GUI below the image preview as *Gain (dB)*. |
| `height` | Height of the camera image in pixels |
| `temp_left` | Temperature of the left camera sensor in degrees Celsius |
| `temp_right` | Temperature of the right camera sensor in degrees Celsius |
| `time` | Processing time for image grabbing in seconds |
| `width` | Width of the camera image in pixels |

### Description of run-time parameters

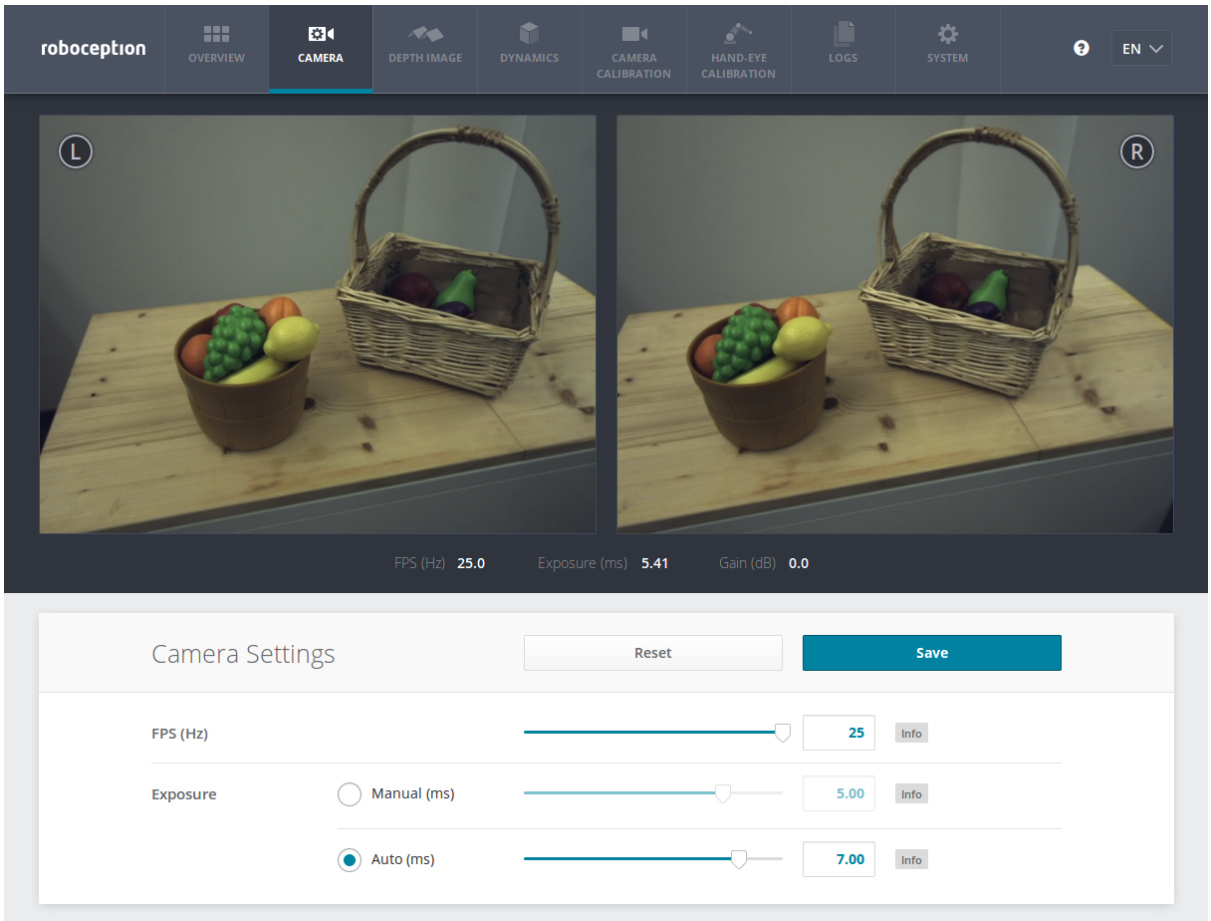



Fig. 6.2: The Web GUI's *Camera* tab

**`fps` (*FPS*)** This value is the cameras' frame rate (fps, frames per second), which determines the upper frequency at which depth images can be computed. This is also the frequency at which the *rc_visard* delivers images via GigE Vision. Reducing this frequency also reduces the network bandwidth required to transmit the images.

**exp_auto (*Exposure*)** This value can be set to 1 for auto-exposure mode, or to 0 for manual exposure mode. In manual exposure mode, the chosen exposure time is kept, even if the images are overexposed or underexposed. In auto-exposure mode, the exposure time and gain factor is chosen automatically to correctly expose the image. The last automatically determined exposure and gain values are set into `exp_value` and `gain_value` when switching auto-exposure off.

**exp_offset_x, exp_offset_y, exp_width, exp_height** These values define a rectangular region in the left rectified image for limiting the area used by auto exposure. The exposure time and gain factor of both images are chosen to optimally expose the defined region. This can lead to over- or underexposure of image parts outside the defined region. The region is visualized in the Web GUI by a rectangle in the left rectified image. If either the width or height is 0, then the whole left and right images are considered by the auto exposure function. This is the default.

**exp_value (*Manual*)** This value is the exposure time in manual exposure mode in seconds. This exposure time is kept constant even if the images are underexposed. In the Web GUI, this exposure time can be entered in milliseconds for convenience.

**gain_value** This value is the gain factor in decibel that can be set in manual exposure mode. Higher gain factors reduce the required exposure time but introduce noise. The value can only be set through the REST-API and GenICam, but not in the Web GUI.

**exp_max (*Auto*)** This value is the maximal exposure time in auto-exposure mode in seconds. In the Web GUI, this exposure time can be conveniently entered in milliseconds. The actual exposure time is adjusted automatically so that the images are exposed correctly. If the maximum exposure time is reached, but the images are still underexposed, the *rc_visard* stepwise increases the gain to increase the images' brightness. Limiting the exposure time is useful for avoiding or reducing motion blur during fast movements. However, higher gain introduces noise into the image. The best trade-off depends on the application.

**wb_auto** This value can be set to 1 for automatic white balancing or 0 for manually setting the ratio between the colors using `wb_ratio_red` and `wb_ratio_blue`. The last automatically determined ratios are set into `wb_ratio_red` and `wb_ratio_blue` when switching automatic white balancing off. White balancing is without function for monochrome cameras. The value can only be set through the REST-API and GenICam, but not in the Web GUI.

**wb_ratio_red and wb_ratio_blue** These values are used to set red to green and blue to green ratios for manual white balance. White balancing is without function for monochrome cameras. The values can only be set through the REST-API and GenICam, but not in the Web GUI.

These parameters are also available over the GenICam interface with slightly different names and partly with different units or data types (see *GigE Vision 2.0/GenICam image interface*, Section 8.1).

### 6.1.4 Services

The stereo camera component offers the following services for persisting and restoring parameter settings.

**save_parameters (*Save*)** With this service call, the stereo camera component's current parameter settings will be made persistent 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 (*Reset*)** Restores and applies the default values for this component's parameters ("factory reset").

> **Warning:** The user must be aware that by calling this service, the current parameter settings for the camera component are irrecoverably lost.

This service requires no arguments.

This service returns no response.

## 6.2 Stereo matching

The stereo matching component uses the rectified stereo-image pair and computes disparity, error, and confidence images.

### 6.2.1 Computing disparity images

After rectification, the left and right images have the nice property that an object point is projected onto the same pixel row in both images. That point's pixel column in the right image is always lower than or equal to the same point's pixel column in the left image. The term disparity signifies the difference between the pixel columns in the right and left images and expresses the depth or distance of the object point from the *rc_visard*. The disparity image stores the disparity values of all pixels in the left camera image.

The larger the disparity, the closer the object point. A disparity of 0 means that the projections of the object point are in the same image column and the object point is at infinite distance. Often, there are pixels for which disparity cannot be determined. This is the case for occlusions that appear on the left sides of objects, because these areas are not seen from the right camera. Furthermore, disparity cannot be determined for textureless areas. Pixels for which the disparity cannot be determined are marked as invalid with the special disparity value of 0. To distinguish between invalid disparity measurements and disparity measurements of 0 for objects that are infinitely far away, the disparity value for the latter is set to the smallest possible disparity value above 0.

To compute disparity values, the stereo matching algorithm has to find corresponding object points in the left and right camera images. These are points that represent the same object point in the scene. For stereo matching, the *rc_visard* uses *SGM* (*Semi-Global Matching*), which offers brief run times and a great accuracy, especially at object borders, fine structures, and in weakly textured areas.

A key requirement for any stereo matching method is the presence of texture in the image, i.e., image-intensity changes due to patterns or surface structure within the scene. In completely untextured regions such as a flat white wall without any structure, disparity values can either not be computed or the results are erroneous or have low confidence (see *Confidence and error images*, Section 6.2.3). The texture in the scene should not be an artificial, repetitive pattern, since those structures may lead to ambiguities and hence to wrong disparity measurements.

If the *rc_visard* has to work in untextured environments, then a static artificial texture can be projected onto the scene using an external pattern projector. This pattern should be random-like and not contain repetitive structures.

### 6.2.2 Computing depth images and point clouds

The following equations show how to compute an object point's actual 3D coordinates $P_x, P_y, P_z$ in the *sensor coordinate frame* (Section 3.7) from the disparity image's pixel coordinates $p_x, p_y$ and the disparity value $d$ in pixels:

$$\begin{aligned} P_x &= \frac{p_x \cdot t}{d} \\ P_y &= \frac{p_y \cdot t}{d} \\ P_z &= \frac{f \cdot t}{d}, \end{aligned} \tag{6.1}$$

where $f$ is the focal length after rectification in pixels and $t$ is the stereo baseline in meters, which was determined during calibration. These values are also transferred over the GenICam interface (see *Custom GenICam features of the rc_visard*, Section 8.1.1).

**Note:** The *rc_visard* reports a focal length factor via its various interfaces. It relates to the image width for supporting different image resolutions. The focal length $f$ in pixels can be easily obtained by multiplying the focal length factor by the image width in pixels.

Please note that equations (6.1) assume that the coordinate frame is centered in the middle of the image. The following figure shows the definition of the image coordinate frame.

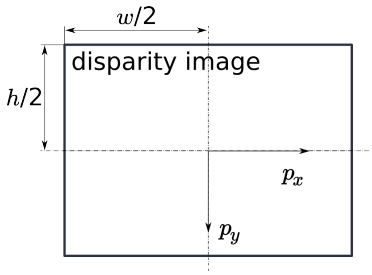



Fig. 6.3: The image coordinate frame's origin is defined to be at the image's center – $w$ is the image width and $h$ is the image height.

The same equations, but with the corresponding GenICam parameters are given in *Image stream conversions* :(Section 8.1.3).

The set of all object points computed from the disparity image gives the point cloud, which can be used for 3D modeling applications. The disparity image is converted into a depth image by replacing the disparity value in each pixel with the value of $P_z$.

**Note:** Roboception provides software and examples for receiving disparity images from the *rc_visard* via GigE Vision and computing depth images and point clouds. See http://www.roboception.com/download.

### 6.2.3 Confidence and error images

For each disparity image, the *rc_visard* provides an error image and a confidence image, which give uncertainty measures for each disparity value. These images have the same resolution and the same frame rate as the disparity image. The error image contains the disparity error $d_{eps}$ in pixels corresponding to the disparity value at the same image coordinates in the disparity image. The confidence image contains the corresponding confidence value $c$ between 0 and 1. The confidence is defined as the probability of the true disparity value being within the interval of three times the error around the measured disparity $d$, i.e., $[d - 3d_{eps}, d + 3d_{eps}]$. Thus, the disparity image with error and confidence values can be used in applications requiring probabilistic inference. The confidence and error values corresponding to an invalid disparity measurement will be 0.

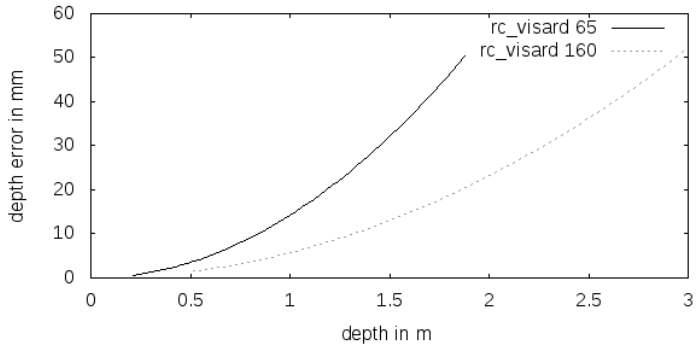
The disparity error $d_{eps}$ (in pixels) can be converted to a depth error $z_{eps}$ (in meters) using the focal length $f$ (in pixels), the baseline $t$ (in meters), and the disparity value $d$ (in pixels) of the same pixel in the disparity image:

$$z_{eps} = \frac{d_{eps} \cdot f \cdot t}{d^2}. \tag{6.2}$$

Combining equations (6.1) and (6.2) allows the depth error to be related to the depth:

$$z_{eps} = \frac{d_{eps} \cdot {P_z}^2}{f \cdot t}.$$

With the focal length and baselines of both *rc_visard* models and the typical disparity error $d_{eps}$ of 0.5 pixels, the depth error can be visualized as shown below.



The *rc_visard* provides time-stamped disparity, error, and confidence images over the GenICam interface (see *Provided image streams*, Section 8.1.2). Live streams of the images are provided with reduced quality in the *Web GUI* (Section 4.5).

### 6.2.4 Parameters

The stereo matching component is called `rc_stereomatching` in the REST-API and it is represented by the *Depth Image* tab in the *Web GUI* (Section 4.5). The user can change the stereo matching parameters there, or use the REST-API (*REST-API interface*, Section 8.2) or GigE Vision (*GigE Vision 2.0/GenICam image interface*, Section 8.1).

#### Parameter overview

This component offers the following run-time parameters.

Table 6.3: The `rc_stereomatching` component's run-time parameters

| Name | Type | Min | Max | Default | Description |
|---|---|---|---|---|---|
| `disprange` | int32 | 32 | 512 | 256 | Disparity range in pixels |
| `fill` | int32 | 0 | 4 | 3 | Disparity tolerance for hole filling in pixels |
| `force_on` | bool | False | True | False | Force processing even if there is no listener |
| `maxdepth` | float64 | 0.1 | 100.0 | 100.0 | Maximum depth in meters |
| `maxdeptherr` | float64 | 0.01 | 100.0 | 100.0 | Maximum depth error in meters |
| `median` | int32 | 1 | 5 | 1 | Window size for median filtering in pixels |
| `minconf` | float64 | 0.5 | 1.0 | 0.5 | Minimum confidence |
| `mindepth` | float64 | 0.1 | 100.0 | 0.1 | Minimum depth in meters |
| `quality` | string | - | - | H | S(taticHigh), H(igh), M(edium), or L(ow). |
| `seg` | int32 | 0 | 4000 | 200 | Minimum size of valid disparity segments in pixels |

This component reports the following status values.

Table 6.4: The `rc_stereomatching` component's status values

| Name | Description |
|---|---|
| `fps` | Actual frame rate of the disparity, error, and confidence images. This value is shown in the Web GUI below the image preview as *FPS (Hz)*. |
| `time_matching` | Time in seconds for performing stereo matching using *SGM* on the GPU |
| `time_postprocessing` | Time in seconds for postprocessing the matching result on the CPU |

Since SGM stereo matching and post processing run in parallel, the overall processing time for this component is the maximum of `time_matching` and `time_postprocessing`. This time is shown in the Web GUI below the image preview as *Processing Time (s)*.

#### Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's *Depth Image* 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:

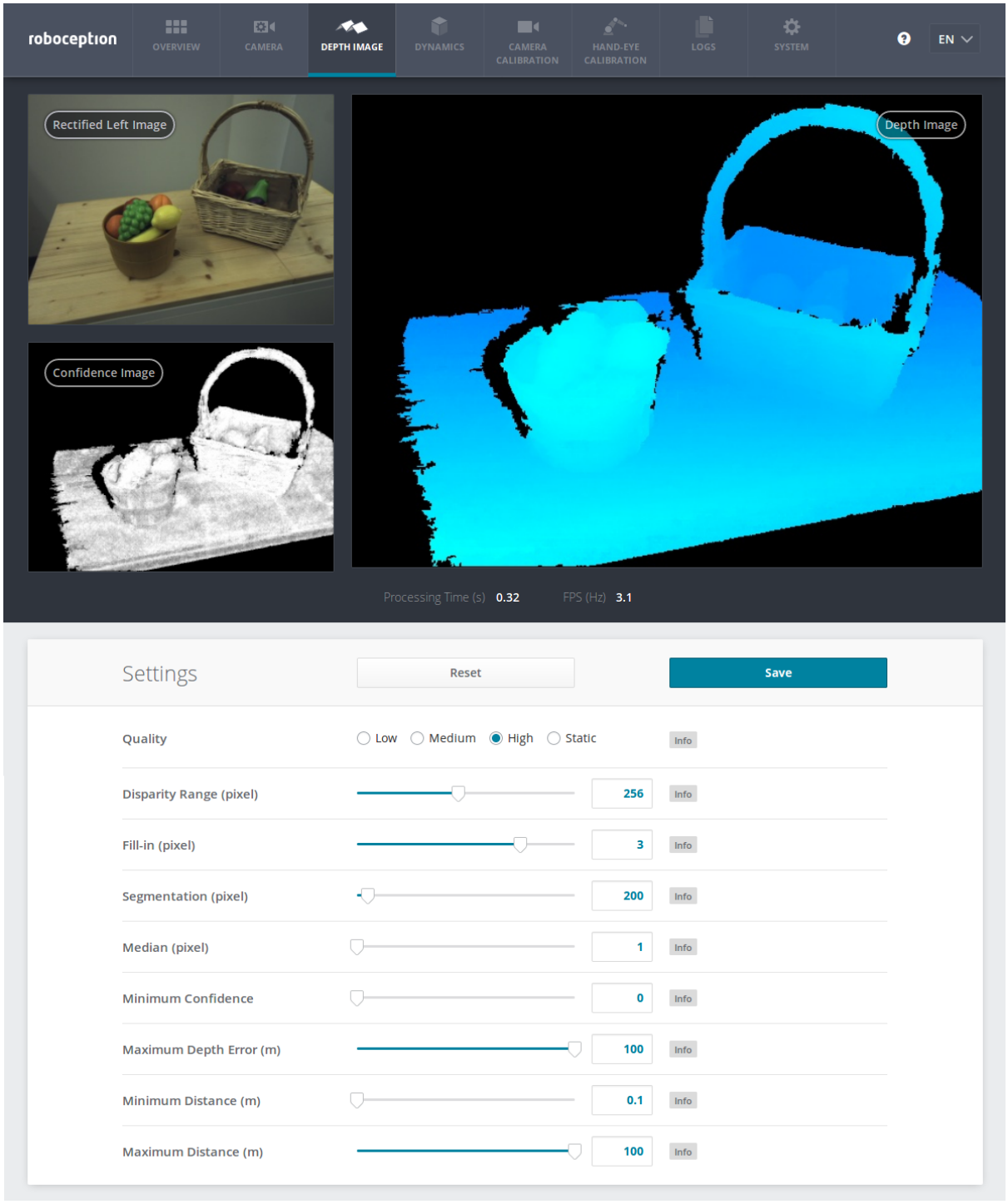



Fig. 6.4: The Web GUI's *Depth Image* tab

**`quality` (*Quality*)** Disparity images can be computed in three different resolutions: high (640 x 480), medium (320 x 240) and low (214 x 160). The lower the resolution, the higher the frame rate of the disparity image. A 25 Hz frame rate can be achieved only at the lowest resolution. Additionally, *static* can be chosen, which means high-resolution processing, limited to a maximal frame rate of 3 Hz and accumulation of all intermediate input images. The accumulation reduces noise, but is only suitable if the scene does not change. Please note that the frame rate of the disparity, confidence, and error images will always be less than or equal to the camera frame rate.

**`disprange` (*Disparity Range*)** The disparity range always start at 0 and goes up to the maximum disparity value a pixel in the disparity image can have. Increasing the disparity range results in a smaller minimum distance

that can be measured, because larger disparity values mean smaller distances. The disparity range is given in pixels and can be set to a value between 32 pixels and 512 pixels. Since a larger disparity range also means a larger search area for the matching pixel in the right rectified image, the processing time increases with a larger disparity range and the frame rate decreases. The disparity range's value is related to the high-resolution disparity image with 640 x 480 pixels and does not have to be scaled when a lower resolution is chosen. Thus, the chosen disparity range gives the same minimum distance for every image-quality option.

**`fill`** (***Fill-in***) This option is used to fill holes in the disparity image by interpolating a plane. Only holes smaller than the segmentation size (see below) are selected for interpolation. The fill-in value is the maximum allowed disparity deviation of any of the hole's border pixels from the interpolation plane. Only if all of its border pixels deviate less than the fill-in value from the plane, a hole will be filled. Larger fill-in values decrease the number of holes, but the interpolated values can have larger errors. The confidence for the interpolated pixels is set to a low value of 0.5. Their error is set to the mean deviation of the hole border pixels from the interpolation plane. A value of 0 effectively switches hole filling off.

**`seg`** (***Segmentation***) The segmentation parameter is used to set the minimum number of pixels that a connected disparity region in the disparity image must fill. Isolated regions that are smaller are set to invalid in the disparity image. This is useful for removing erroneous disparities. However, larger values may also remove real objects.

**`median`** (***Median***) This value gives the window side length in pixels for the median filter, which smoothes the disparity image. Larger values lead to oversmoothing and cost more processing time. A window size of 1 effectively turns this filter off.

**`minconf`** (***Minimum Confidence***) The minimum confidence can be set to filter potentially false disparity measurements. All pixels with less confidence than the chosen value are set to invalid in the disparity image.

**`maxdeptherr`** (***Maximum Depth Error***) The maximum depth error is used to filter measurements that are too inaccurate. All pixels with a larger depth error than the chosen value are set to invalid in the disparity image. The maximum depth error is given in meters. The depth error generally grows quadratically with an object's distance from the sensor (see *Confidence and error images*, Section 6.2.3).

**`mindepth`** (***Minimum Distance***) The minimum distance is the smallest distance from the sensor at which measurements should be possible. Larger values implicitly reduce the disparity range, which also reduces the computation time. The minimum distance is given in meters.

**`maxdepth`** (***Maximum Distance***) The maximum distance is the largest distance from the sensor at which measurements should be possible. Pixels with larger distance values are set to invalid in the disparity image. Setting this value to its maximum permits values up to infinity. The maximum distance is given in meters.

The same parameters are also available over the GenICam interface with slightly different names and partly with different data types (see *GigE Vision 2.0/GenICam image interface*, Section 8.1).

### 6.2.5 Services

The stereo matching component offers the following services for persisting and restoring parameter settings.

**`save_parameters`** (***Save***) With this service call, the stereo matching 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`** (***Reset***) Restores and applies the default values for this component's parameters ("factory reset").

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

This service requires no arguments.

This service returns no response.

## 6.3 Sensor dynamics

The dynamics component provides estimates of the sensor state. These include pose, linear velocity, linear acceleration, and rotational rates. The component handles starting and stopping, and streaming of the estimates for individual subcomponents:

- ***Visual odometry*** **(`rc_stereovisodo`)** estimates the camera's motion from the motion of characteristic image points in the left camera images (Section 6.4).
- ***Stereo INS*** **(`rc_stereo_ins`)** combines visual odometry measurements with readings from an inertial measurement unit (*IMU*) to provide accurate, high-frequency state estimates in real time (Section 6.5).
- ***SLAM*** **(`rc_slam`)** performs simultaneous localization and mapping (*SLAM*) for correcting accumulated poses (Section 7.1).

### 6.3.1 Coordinate frames for state estimation

The world coordinate frame for state estimation is defined as follows: The coordinate frame's z-axis points upward and is aligned with the gravity vector. The x-axis is orthogonal to the z-axis and points in the *rc_visard*'s viewing direction at the time when the pose estimation starts. The world frame's origin is located at the origin of the *rc_visard*'s IMU coordinate frame at the instant when state estimation is switched on.

If pose estimation is switched on when the *rc_visard*'s viewing direction parallels the gravity vector (with a tolerance range of 10 degrees), then the world coordinate frame's y-axis is aligned either with the IMU's positive or negative x-axis. In this orientation, the initial alignment of the world coordinate frame is no longer continuous. Thus, special care has to be taken when pose estimation has to be started at such an orientation.

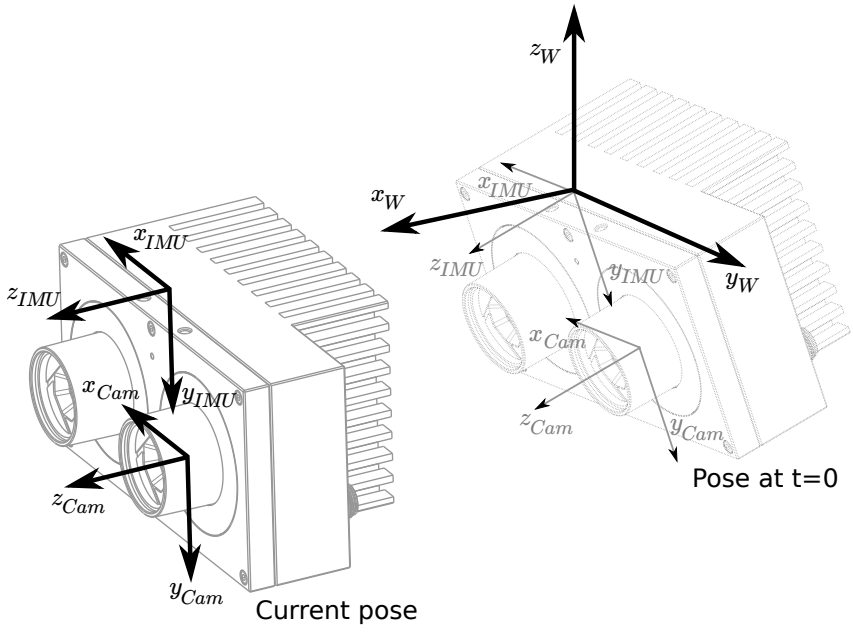



Fig. 6.5: Coordinate frames for state estimation. The IMU coordinate frame is inside the *rc_visard*'s housing. The *camera coordinate frame* (Section 3.7) is in the focal point of the left camera.

The transformation between the IMU coordinate frame and the camera/sensor frame is also estimated and provided in the *real-time dynamics stream* over the rc_dynamics interface (see *Interfaces*, Section 8).

**Warning:** The stereo INS component self-calibrates the IMU during its initialization. It is therefore required that the *rc_visard* is not moving and sufficient texture is visible during startup of the stereo INS component.

### 6.3.2 Available state estimates

The *rc_visard* provides seven different kinds of timestamped state-estimate data streams via the rc_dynamics interface (see *The rc_dynamics interface*, Section 8.3):

| Name | Frequency | Source | Description |
|---|---|---|---|
| *pose* | 25 Hz | best effort | Pose of camera frame, slightly delayed but most accurate |
| *pose_ins* | 25 Hz | *Stereo INS* | Pose of camera frame, slightly delayed but most accurate |
| *pose_rt* | 200 Hz | best effort | Pose of camera frame |
| *pose_rt_ins* | 200 Hz | *Stereo INS* | Pose of camera frame |
| *dynamics* | 200 Hz | best effort | Pose, velocity and acceleration in IMU frame |
| *dynamics_ins* | 200 Hz | *Stereo INS* | Pose, velocity and acceleration in IMU frame |
| *imu* | 200 Hz | *Stereo INS* | Raw IMU data |

*Best effort* here means that if *SLAM* is running, then it contains the loop-closure corrected estimates and is equivalent to the stream from *Stereo INS* when SLAM is not running.

#### Camera-pose streams (`pose` and `pose_ins`)

The *camera-pose streams* called `pose` and `pose_ins` are provided at 25 Hz with timestamps that correspond to image timestamps. The former stream is the best-effort estimate, combining `rc_slam` and `rc_stereo_ins` if the *SLAM* component is running. If SLAM is not running, then both data streams are equivalent. Pose values are given in world coordinates, and also refer to the *rc_visard*'s camera frame origin (see *Coordinate frames for state estimation*, Section 6.3.1). They are the most accurate estimates, taking all available *rc_visard* information into consideration. They can be used in modeling applications, where camera images, depth images, or point clouds have to be aligned highly accurately with each other. To ensure the greatest possible accuracy, these pose values are delayed until a corresponding visual odometry measurement is available.

#### Real-time camera-pose streams (`pose_rt` and `pose_rt_ins`)

Two *real-time pose streams* called `pose_rt` and `pose_rt_ins` are provided at the IMU rate of 200 Hz. The former stream is the best-effort estimate, combining `rc_slam` and `rc_stereo_ins` when the SLAM component is running. If SLAM is not running, then both data streams are equivalent. They consist of the pose estimates of the *rc_visard*'s camera frame origin (see *Coordinate frames for state estimation*, Section 6.3.1) in world coordinates. The values given in these streams correspond to the values in the *real-time dynamics streams*, but give the pose of the sensor/camera coordinate frame instead of that of the IMU coordinate frame.

#### Real-time dynamics streams (`dynamics` and `dynamics_ins`)

Two *real-time dynamics streams* called `dynamics` and `dynamics_ins` are provided at the IMU rate of 200 Hz. The former stream is the best-effort estimate, combining `rc_slam` and `rc_stereo_ins` when the SLAM component is running. If SLAM is not running, then both data streams are equivalent. The estimates can be used for real-time control of a robot. Since the values are provided in real time and visual odometry computation requires some processing time, the latest visual odometry estimate may not be included. Therefore, these estimates are in general slightly less accurate than those in the non-real-time *camera-pose streams* (see above), but are the best estimates available at this instant. The provided dynamics streams contain the *rc_visard*'s

- translation $\mathbf{p} = (x, y, z)^T$ in $m$,
- rotation $\mathbf{q} = (q_x, q_y, q_z, q_w)^T$ as unit quaternion,

- linear velocities $\mathbf{v} = (v_x, v_y, v_z)^T$ in $\frac{m}{s}$,
- angular velocities $\omega = (\omega_x, \omega_y, \omega_z)^T$ in $\frac{rad}{s}$,
- gravity-compensated linear accelerations $\mathbf{a} = (a_x, a_y, a_z)^T$ in $\frac{m}{s^2}$, and
- transformation from camera to IMU coordinate frame as pose with frame name and parent frame name.

For each component, the stream also provides the name of the coordinate frame in which the values are given. Translation, rotation, and linear velocities are given in the world frame; angular velocities and accelerations are given in the IMU frame (see *Coordinate frames for state estimation*, Section 6.3.1). All values refer to the IMU frame's origin. That means, for example, that linear velocity is the velocity of the IMU frame's origin in the world frame.

Lastly, the stream contains a `possible_jump` flag, which is set to *true* whenever the optional SLAM component (see *SLAM*, Section 7.1) corrects the state estimation after finding a loop closure. The state estimate can jump in this case, which should be considered when the values are used in a control loop. If SLAM is not running, the jump flag can be ignored and will stay *false*.

#### IMU data stream (`imu`)

The *IMU data stream* called `imu` is provided at the IMU rate of 200 Hz. It consists of the acceleration in x, y, z directions plus the angular velocities around these three axis. The values are calibrated but not bias- and gravity-compensated, and are given in the IMU frame. The transformation between IMU and sensor frame is provided in the *real-time dynamics stream*.

### 6.3.3 Services

The sensor dynamics component offers the following services for starting dynamics/motion estimation. All services return a numerical code of the entered state. The meaning of the returned state codes and names are given in Table 6.5.

Table 6.5: Possible states of the sensor dynamics component

| State name | Description |
|---|---|
| IDLE | The component is ready, but idle |
| WAITING_FOR_INS | Waiting for stereo INS to start up |
| WAITING_FOR_INS_AND_SLAM | Waiting for stereo INS and SLAM to start up |
| RUNNING | The stereo INS component is running (SLAM is not running) |
| WAITING_FOR_SLAM | Waiting for SLAM to start up (stereo INS is running) |
| RUNNING_WITH_SLAM | Both stereo INS and SLAM are running |
| FATAL | A fatal error has occured (either in stereo INS or SLAM) |

**`start`** Starts the stereo INS component. Transitions from state `IDLE` through `WAITING_FOR_INS` to `RUNNING`.

This service requires no arguments.

This service returns the following response:

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

**`start_slam`** Starts the SLAM and – if not yet started – the stereo INS component. From state `IDLE`: Transitions through `WAITING_FOR_INS_AND_SLAM` and `WAITING_FOR_SLAM` to `RUNNING_WITH_SLAM`. From state `RUNNING`: Transitions through `WAITING_FOR_SLAM` to `RUNNING_WITH_SLAM`.

This service requires no arguments.

This service returns the following response:

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

**stop** Stops the stereo INS and – if running – the SLAM components. The trajectory estimate of the SLAM component will still be available. Transitions from state RUNNING or RUNNING_WITH_SLAM to IDLE.

This service requires no arguments.

This service returns the following response:

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

**stop_slam** Stops the SLAM component. Stereo INS will continue to run. The trajectory estimate of the SLAM component will still be available. Transitions from state RUNNING_WITH_SLAM to RUNNING.

This service requires no arguments.

This service returns the following response:

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

**restart** Restarts to stereo INS. Equivalent to successive stop and start.

From state RUNNING or RUNNING_WITH_SLAM: Transitions through states IDLE and WAITING_FOR_INS to RUNNING.

This service requires no arguments.

This service returns the following response:

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

**restart_slam** Restarts to SLAM mode. Equivalent to successive stop and start_slam.

From state RUNNING or RUNNING_WITH_SLAM: Transitions through states IDLE, WAITING_FOR_INS_AND_SLAM, WAITING_FOR_SLAM to RUNNING_WITH_SLAM.

This service requires no arguments.

This service returns the following response:

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

The following diagram shows the main states and transitions. Intermediate states and the fatal error state are omitted for conceptual clarity.

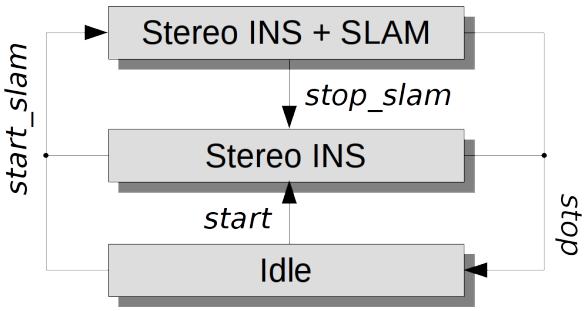



Fig. 6.6: Simplified state and transition diagram

These services shall respond quickly. Therefore, for services that cause a state transition the value of the returned `current_state` in general is the first new (intermediate) state that was transitioned to, not the final state. E.g., for the `start` command the returned `current_state` will be WAITING_FOR_INS, not state RUNNING. If the transition does not take place within 0.1 seconds, the current state is returned. See Table 6.5 for the meaning of the returned state codes.

**Note:** The state `FATAL` can only be left by calling `stop`, which performs a transition to the state `IDLE`. The services `restart` and `restart_slam` internally use `stop` and will also work as expected. `start` and `start_slam` only work if the state is `IDLE`, and do nothing if the state is `FATAL`.

**Note:** The dynamics components can also be started and stopped on the *Dynamics* page of the *Web GUI*.

**`get_cam2imu_transform`** returns the transformation from camera to IMU coordinate frame. This is equivalent to the `cam2imu_transform` in the *Dynamics message* (Section 8.3.3).

This service requires no arguments.

This service returns the following response:


```
{
  "name": "string",
  "parent": "string",
  "pose": {
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "timestamp": {
      "nsec": "int32",
      "sec": "int32"
    }
  }
}
```

## 6.4 Visual odometry

Visual odometry is part of the sensor dynamics component. It is used to estimate the camera's motion from the motion of characteristic image points (so-called image features) in left camera images. Image features are computed from image corners, which are image regions with high intensity gradients. Image features are used to look for matches between subsequent images to find correspondences. Their 3D coordinates are computed by stereo matching (independent from the disparity image). The camera's motion is computed from a set of corresponding 3D points between two images. To increase the robustness of visual odometry, correspondences are not only computed to the previous camera image but to a certain number of previous images, which are called *keyframes*. The best result is then chosen.

The visual-odometry frame rate is independent of the user setting in the stereo camera component. It is internally limited to 12 Hz but can be lower, depending on the number of features and keyframes. To ensure good pose-estimation quality, the frame rate should not drop significantly under 10 Hz.

The visual odometry component's measurements are not directly accessible on the *rc_visard*. Instead, they are internally fused with measurements from the integrated inertial measurement unit to increase robustness and frame rate and reduce latency. The result of the sensor data fusion is provided in the form of different streams (see *Stereo INS*, Section 6.5).

### 6.4.1 Parameters

The visual odometry software component is called `rc_stereovisodo` and it is represented by the *Dynamics* tab in the *Web GUI* (Section 4.5). The user can change the visual odometry parameters there, or use the REST-API (*REST-API interface*, Section 8.2).

#### Parameter overview

This component offers the following run-time parameters.

Table 6.6: The `rc_stereovisodo` component's run-time parameters

| Name | Type | Min | Max | Default | Description |
|---|---|---|---|---|---|
| `disprange` | int32 | 32 | 512 | 256 | Disparity range in pixels |
| `ncorner` | int32 | 50 | 4000 | 500 | Number of corners |
| `nfeature` | int32 | 50 | 4000 | 300 | Number of features |
| `nkey` | int32 | 1 | 4 | 4 | Number of keyframes |

This component reports the following status values.

Table 6.7: The `rc_stereovisodo` component's status values

| Name | Description |
|---|---|
| `corner` | Number of detected corners. This value is shown as *Corners* below the image preview in the Web GUI. |
| `correspondences` | Number of correspondences. This value is shown as *Correspondences* below the image preview in the Web GUI. |
| `feature` | Number of features. This value is shown as *Features* below the image preview in the Web GUI. |
| `fps` | Frame rate of the visual odometry in Hertz. This value is shown below the image preview as *Visual Odometry FPS (Hz)* in the Web GUI. |
| `time_frame` | Processing time in seconds to compute corners and features for each frame |
| `time_vo` | Processing time in seconds to compute the motion |

### Description of run-time parameters

Run-time parameters influence the number of features used to compute visual odometry. More features increase the visual odometry's robustness at the expense of more run time, which can reduce the frame rate. Although the resulting state estimate will always have a high frequency due to fusion with IMU measurements, high visual-odometry frame rates are nevertheless desirable, since these measurements are much more accurate than IMU measurements alone. A visual-odometry rate of at least 10 Hz should thus be aimed for. The visual-odometry frame rate is provided as a status parameter and is shown below the camera image on the *Web GUI*'s *Dynamics* page.

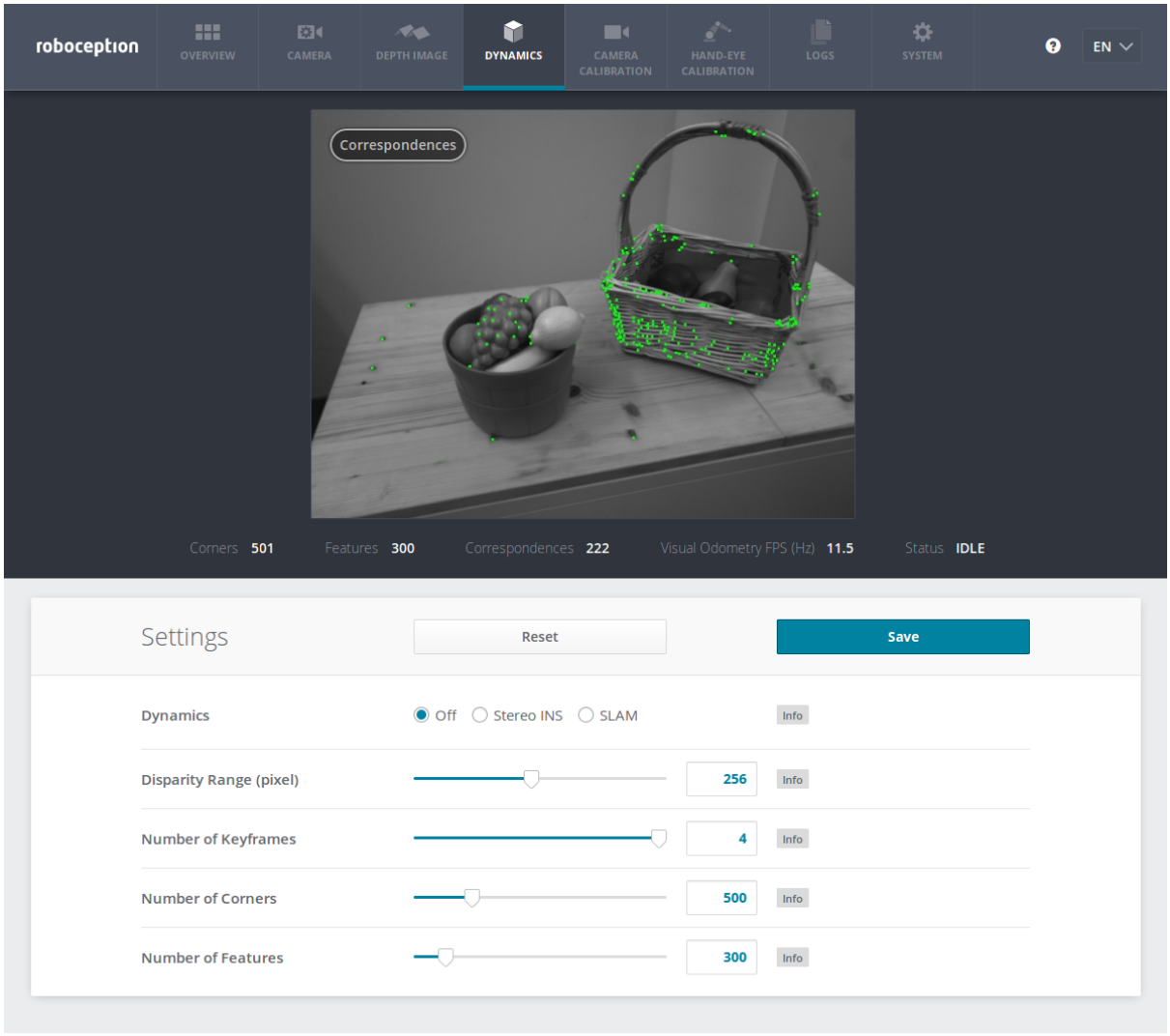



Fig. 6.7: The Web GUI's *Dynamics* tab

The camera image shown on this page depicts image features as small green dots. The bold green dots are the features in the current image for which correspondences could be found in a previous keyframe. Green lines depict the motion of these features relative to the previous keyframe. This visualization should help to find a good set of parameters for visual odometry. The number of correspondences is reported as a status parameter and is shown below the camera image on the *Web GUI*'s *Dynamics* page. For robust visual-odometry measurements, the parameters should be adjusted so that the resulting number of correspondences in the target environment is around at least 50 when the sensor is moving. The correspondence count will be larger when the *rc_visard* is static, and the number will change when the *rc_visard* moves through the environment. Short failures of the visual odometry are tolerated due to the fusion with IMU measurements. Longer failures should be avoided because they lead to large pose uncertainties and can lead to errors in the state estimation.

Each run-time parameter is represented by a row on the Web GUI's *Dynamics* tab. The name of the row is given

in brackets behind the parameter name, and the parameters are listed in the order they appear in the Web GUI:

**`start` (*Dynamics*)** This starts the sensor dynamics estimation components (see *Services*, Section 6.3.3).

**`disprange` (*Disparity Range*)** The disparity range gives the maximum disparity value for each image feature related to the resolution of the high-quality disparity image (640 x 480 pixels). The disparity range determines the minimum working distance of the visual odometry. When the disparity range is narrow, only more distant features are considered in the visual-odometry estimation. When choosing a broader disparity range, close features can also be used. Broader disparity ranges increase processing time, which can reduce the visual odometry's frame rate.

**`nkey` (*Number of Keyframes*)** More keyframes can increase the robustness and accuracy of the visual odometry, but they also increase processing time and can decrease the visual-odometry frame rate.

**`ncorner` (*Number of Corners*)** This value gives the approximate number of corners that will be detected in the left image. Larger numbers make visual odometry more robust and accurate but can lead to lower frame rates of the visual odometry.

**`nfeature` (*Number of Features*)** This value is the maximum number of features that will be derived from the corners. It is useful to detect more corners and select the best subset as features. Larger numbers make visual odometry more robust and accurate but can lead to lower visual-odometry frame rates. Fewer features might be computed, depending on the scene and movement. The actual number of features is reported below the camera image on the *Web GUI*'s *Dynamics* page.

**Note:** Increasing the number of keyframes, corners, or features will also increase robustness but will require more computation time and may reduce the frame rate, depending on other components active on the *rc_visard*. The visual-odometry frame rate should be at least 10 Hz.

### 6.4.2 Services

The visual odometry component offers the following services for persisting and restoring parameter settings. The names of the corresponding Web GUI buttons are added in brackets:

**`save_parameters` (*Save*)** With this service call, the current parameter settings of the visual odometry 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` (*Reset*)** Restores and applies the default values for this component's parameters ("factory reset").

**Warning:** The user must be aware that calling this service causes irrecoverable loss of the visual odometry component's current parameter settings.

This service requires no arguments.

This service returns no response.

This component offers no start or stop services itself, because the *dynamics component* (Section 6.3) starts and stops it.

## 6.5 Stereo INS

The stereo-vision-aided Inertial Navigation System (*INS*) component is part of the sensor dynamics component. It combines visual-odometry measurements with inertial measurement unit (*IMU*) data and provides robust, low

latency, real-time state estimates at a high rate. The IMU consists of three accelerometers and three gyroscopes, which measure accelerations and turn rates in all three dimensions. By fusing IMU and visual-odometry measurements, the state estimate has the same frequency as the IMU (200 Hz) and is very robust even under challenging lighting conditions and for fast motions.

**Note:** To achieve high-quality pose estimates, it must be ensured that sufficient texture is visible during runtime of the stereo INS component. In case no texture is visible for a longer period of time, the stereo INS component will stop instead of providing highly erroneous data.

### 6.5.1 Self-Calibration

During startup of the stereo INS component, it will self-calibrate the IMU using the visual-odometry measurements. For the self-calibration to succeed, it is required that

- the *rc_visard* is not moving and
- sufficient texture is visible

during startup of the stereo INS component. Failure to meet these requirements will most likely result in a constant drift of the pose estimates.

### 6.5.2 Parameters

The stereo INS component's node name is `rc_stereo_ins`.

This component has no run-time parameters.

This component reports the following status values.

Table 6.8: The `rc_stereo_ins` component's status values

| Name | Description |
|---|---|
| `freq` | Frequency of the stereo INS process in Hertz. This value is shown as *Update Rate* in the Web GUI *Overview* tab in the *Dynamics* area |
| `state` | String representing the internal state |

## 6.6 Camera calibration

To use the stereo camera as measuring instrument, camera parameters such as focal length, lens distortion, and the relationship of the cameras to each other must be exactly known. The parameters are determined by calibration and used for image rectification (see *Planar rectification*, Section 6.1.2), which is the basis for all other image processing modules. The *rc_visard* is calibrated at production time. Nevertheless, checking calibration and recalibration might be necessary if the *rc_visard* was exposed to strong mechanical impact. The camera calibration component is responsible for checking calibration and recalibrating.

### 6.6.1 Self-calibration

The camera calibration component automatically runs in self-calibration mode at a low frequency in the background. In this mode, the *rc_visard* observes the alignment of image rows of both rectified images. A mechanical impact, such as one caused by dropping the *rc_visard*, might result in a misalignment. If a significant misalignment is detected, then it is automatically corrected. After each reboot and after each correction, the current self-calibration offset is reported in the camera component's log file (see *Downloading log files*, Section 9.7) as:

> *"rc_stereocalib: Current self-calibration offset is 0.00, update counter is 0"*

The update counter is incremented after each automatic correction. It is reset to 0 after manual recalibration of the *rc_visard*.

Under normal conditions, such as the absence of mechanical impact on the *rc_visard*, self-calibration should never occur. Self-calibration allows the *rc_visard* to work normally even after misalignment is detected, since it is automatically corrected. Nevertheless, checking camera calibration manually is recommended if the update counter is not 0.

### 6.6.2 Calibration process

Manual calibration can be done through the Web GUI's *Camera Calibration* tab. This tab provides a wizard to guide the user through the calibration process.

**Note:** Camera calibration is normally unnecessary since the *rc_visard* is calibrated at production time. Therefore, calibration is only required after strong mechanical impacts, such as occur when dropping the *rc_visard*.

#### Step 1: Calibration settings

The quality of camera calibration heavily depends on the quality of the calibration grid. Calibration grids for the *rc_visard* can be obtained from Roboception.

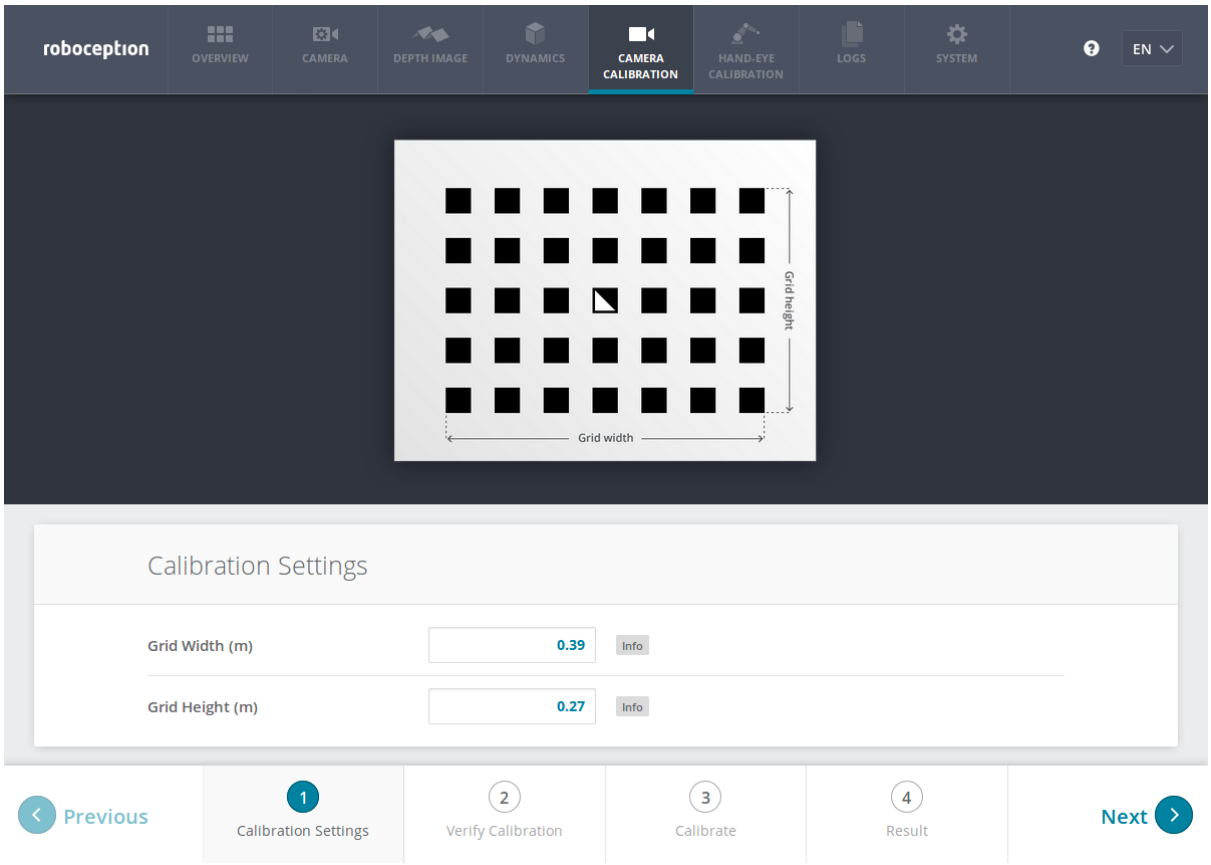



Fig. 6.8: Calibration settings

The *Camera calibration* component has to be selected in the *Web GUI* (Section 4.5) to verify or perform camera calibration. In the first step, the width and height of the grid must be specified as shown in the screenshot above. The *Next* button proceeds to the next step.

### Step 2: Verify calibration

In the second step, the current calibration can be verified. To perform the verification, the grid must be held such that it is simultaneously visible in both cameras. Make sure that all black squares of the grid are completely visible and not occluded. A green check mark overlays each correctly detected square. The correct detection of the grid is only possible if all of the black squares are detected. After the grid is detected, the calibration error is automatically computed, and the result is displayed on the screen.

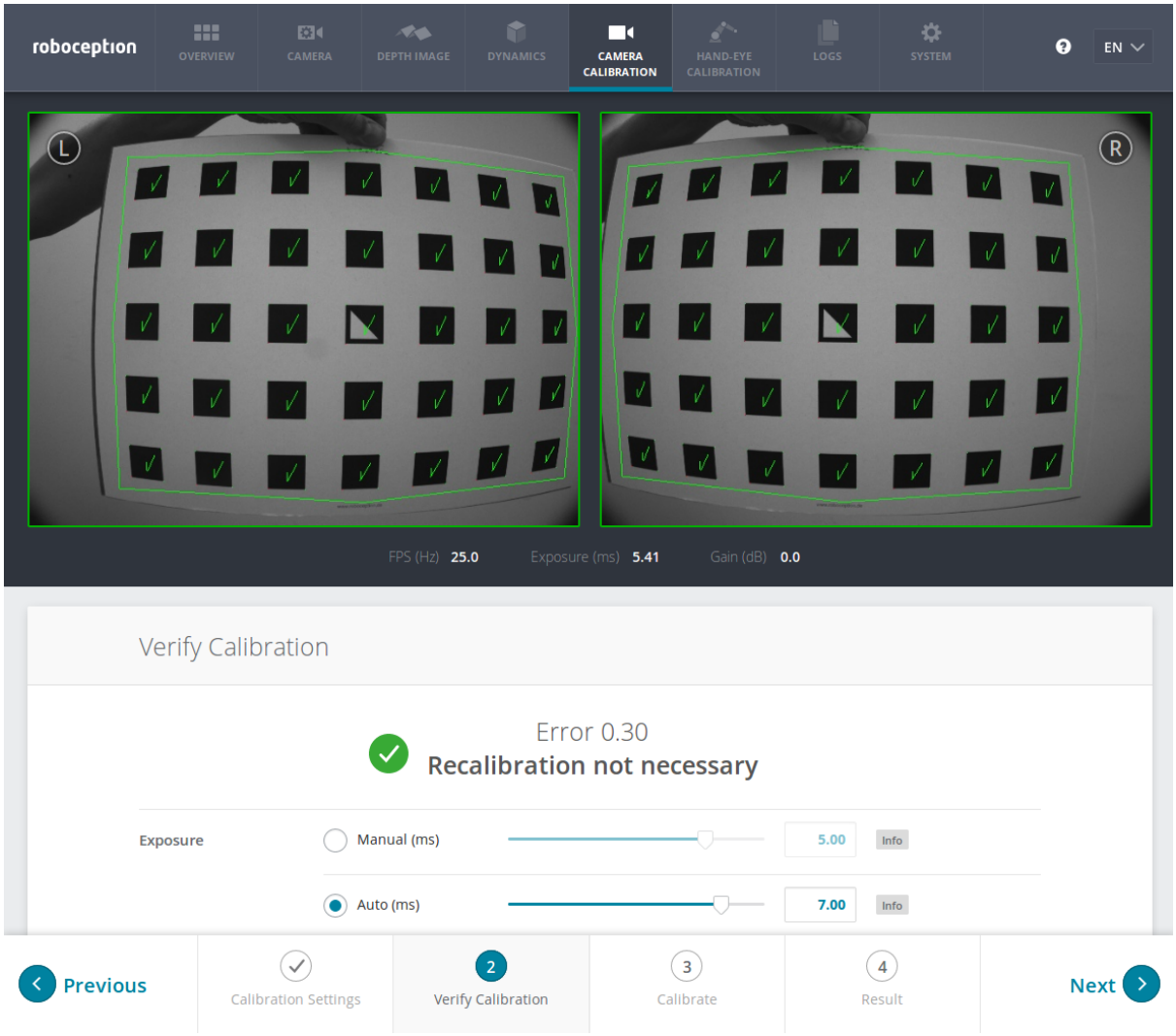



Fig. 6.9: Verification of calibration

Some of the squares not being detected, or being detected only briefly might indicate a low-quality or damaged calibration grid, or bad lighting conditions.

**Note:** To compute a meaningful calibration error, the grid should be held as closely as possible to the cameras. If the grid only covers a small section of the camera images, the calibration error will always be less than when the grid covers the full image.

The typical calibration error is around 0.3 pixels. If the error is less than 0.4 to 0.5 pixels, then the calibration procedure can be skipped. If the calibration error is greater, the calibration procedure should be performed to guarantee full sensor performance. The button *Next* starts the procedure.

**Warning:** A large error during verification can be due to miscalibrated cameras, an inaccurate calibration grid, or wrong grid width or height. Please make sure that the grid is accurate and the entered grid width and height are correct. Otherwise, manual calibration will actually decalibrate the cameras!

### Step 3: Performing calibration

The camera's exposure time should be set appropriately before starting the calibration. To achieve good calibration results, the images should be well-exposed and image noise should be avoided. Thus, the maximum auto-exposure time should be great enough to achieve a very small gain factor, ideally 0.0 dB. The gain factor is displayed below the camera images as shown in Fig. 6.10.

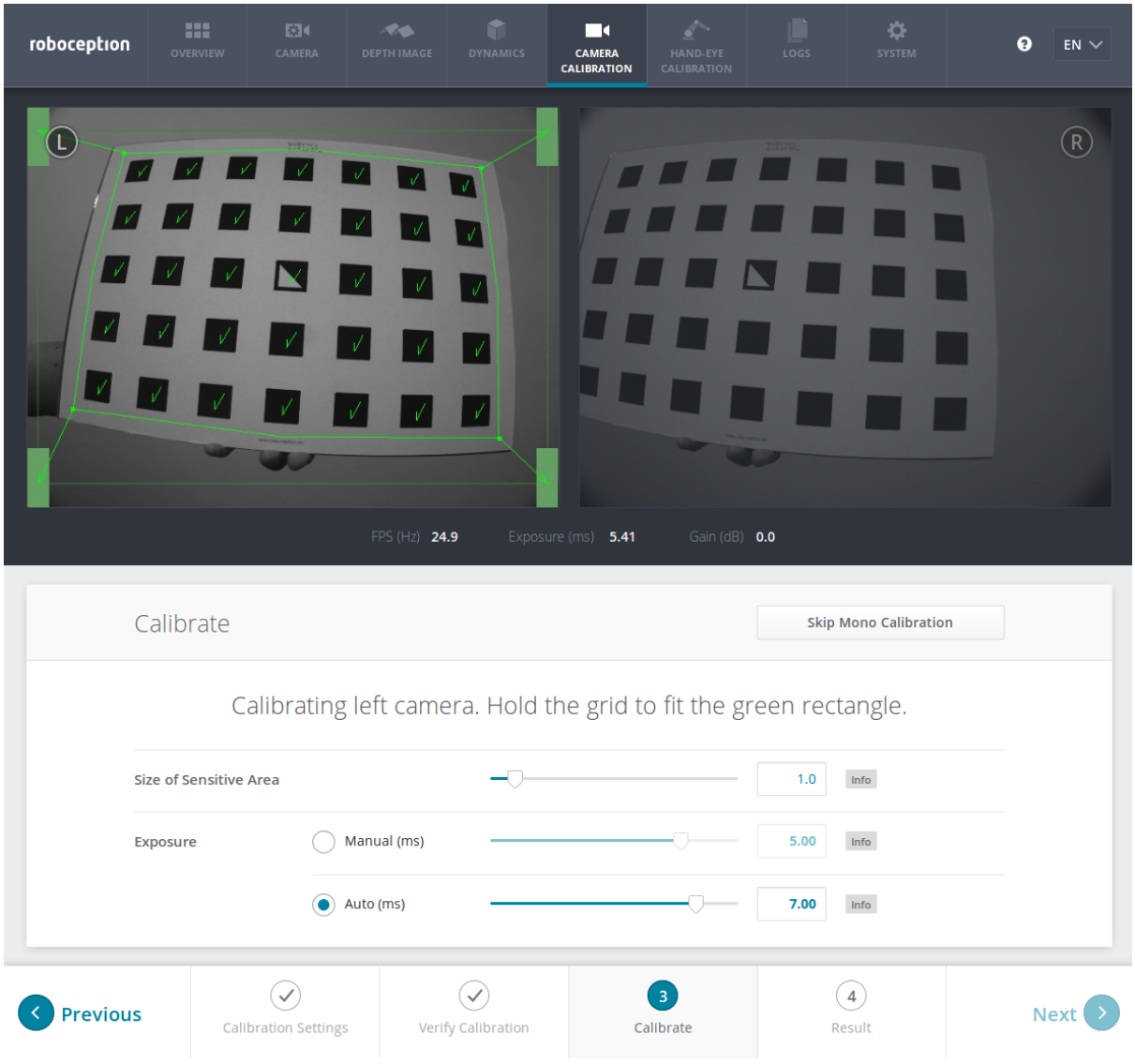



Fig. 6.10: Starting the calibration procedure

For calibration, the grid has to be held in certain poses. The arrows from the grid corners to the green areas indicate that all grid corners should be placed inside the green areas. The green areas are called sensitive areas. The *Size of Sensitive Area* slider can control their size to ease calibration as shown in the screen shot in Fig. 6.10. However, please be aware that increasing their size too much may result in slightly less calibration accuracy.

Holding the grid upside down is a common mistake made during calibration. Spotting this in this case is easy because the green lines from the grid corners into the green areas will cross each other as shown in Fig. 6.11.

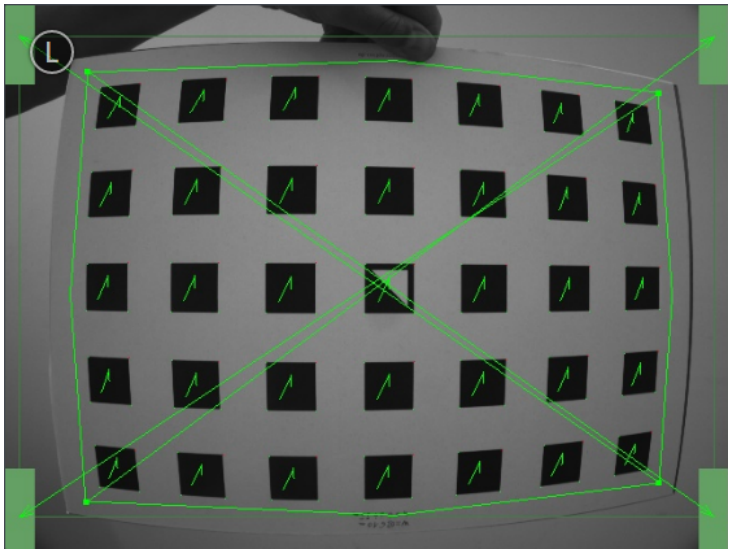


Fig. 6.11: Wrongly holding the grid upside down leads to crossed green lines.

**Note:** Calibration might appear cumbersome as it involves holding the grid in certain predefined poses. However, only this can ensure an unbiased, high-quality calibration result.

### Monocalibration

Full calibration consists of calibrating each camera individually and then performing a stereo calibration to determine the relationship between them. In most cases, the intrinsic calibration of each camera does not get corrupted. For this reason, *Skip Monocalibration* in the *Calibrate* tab should be clicked to skip monocalibration during the first recalibration. Continue with the guidelines given in *Stereo calibration*. If stereo calibration yields an unsatisfactory calibration error, then calibration should be repeated without skipping monocalibration.

The monocalibration process involves five poses for each camera as shown in Fig. 6.12.

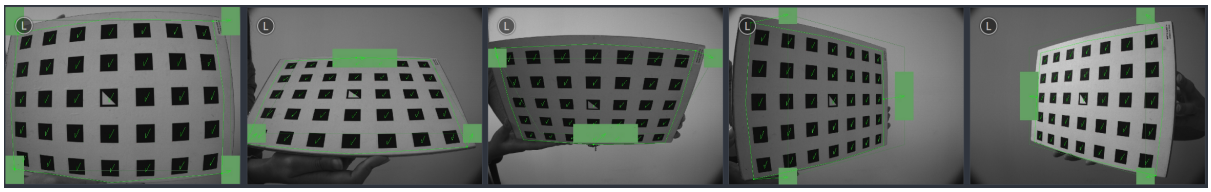


Fig. 6.12: Poses required for monocamera calibration

After the corners or sides of the grid are placed on top of the sensitive areas, the process automatically shows the next pose required. When the process is finished for the left camera, the same procedure is repeated for the right one.

### Stereo calibration

After monocalibration is completed or has been skipped, the stereo calibration process is started. During stereo calibration, both cameras are calibrated to each other to find their relative rotation and translation.

First, the grid should be held closer than 40 cm from the sensor. It must be fully visible in both images and the cameras should look perpendicularly onto the grid. A green outline that stays in the image indicates the images' acceptance.

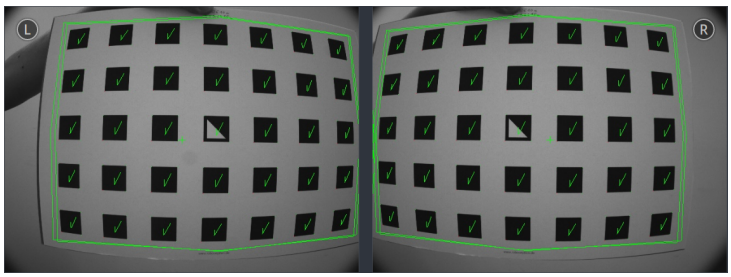


Fig. 6.13: Holding the grid closer than 40 cm during stereo calibration

Next, the grid should be held at least 1 m from the cameras. The small cross in the middle of the images should be inside of the grid and the cameras must look perpendicularly onto the grid. A green outline that stays in the image indicates the images' acceptance.

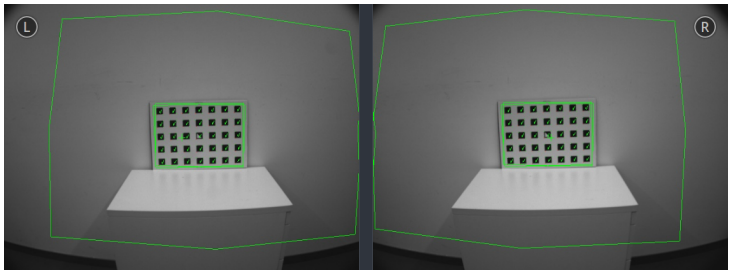


Fig. 6.14: Holding the grid farther away than 1 m during stereo calibration

**Note:** If the check marks on the calibration grid all vanish, then either the camera does not look perpendicularly onto the grid, the green cross in the middle of the images is not inside the grid, or the grid is too far away from the camera.

#### Step 4: Storing the calibration result

Clicking the *Compute Calibration* button finishes the process and displays the final result. The presented result is the mean reprojection error of all calibration points. It is given in pixels and typically has a value around 0.3.

**Note:** The given result is the minimum error left after calibration. The real error is definitely not less than this, but could in theory be larger. This is true for every camera-calibration algorithm and the reason why we enforce holding the grid in very specific poses. Doing so ensures that the real calibration error cannot significantly exceed the reported error.

Pressing *Save Calibration* applies the calibration and saves it to the sensor.

**Warning:** If a hand-eye calibration was stored on the *rc_visard* before camera calibration, the hand-eye calibration values could have become invalid. Please repeat the hand-eye calibration procedure.

### 6.6.3 Parameters

The component is called `rc_stereocalib` in the REST-API.

**Note:** The camera calibration component's available parameters and status values are for internal use only and may change in the future without further notice. Calibration should only be performed through the Web GUI as described above.

### 6.6.4 Services

**Note:** The camera calibration component's available service calls are for internal use only and may change in the future without further notice. Calibration should only be performed through the Web GUI as described above.

## 6.7 Hand-eye calibration

For applications, in which the *rc_visard* is integrated into one or more robot systems, it needs to be calibrated w.r.t. some robot reference frames. For this purpose, the *rc_visard* is shipped with an on-board calibration routine called the *hand-eye calibration* component.

**Note:** The implemented calibration routine is completely agnostic about the user-defined robot frame to which the *rc_visard* is calibrated. It might be a robot's end-effector (e.g., flange or tool center point) or any point on the robot structure. The method's only requirement is that the pose (i.e., translation and rotation) of this robot frame w.r.t. a user-defined external reference frame (e.g., world or robot mounting point) is exactly observable by the robot controller and can be reported to the calibration component.

The *Calibration routine* (Section 6.7.3) itself is an easy-to-use three-step procedure using a calibration grid. Calibration grids for the *rc_visard* can be obtained from Roboception.

### 6.7.1 Calibration interfaces

The following two interfaces are offered to conduct hand-eye calibration:

1. All services and parameters of this component required to conduct the hand-eye calibration **programmatically** are exposed by the *rc_visard*'s *REST-API interface* (Section 8.2). The respective node name of this component is `rc_hand_eye_calibration` and the respective service calls are documented *Services* (Section 6.7.5).

   **Note:** The described approach requires a network connection between the *rc_visard* and the robot controller to pass robot poses from the controller to the sensor's calibration component.

2. For use cases where robot poses cannot be passed programmatically to the *rc_visard*'s hand-eye calibration component, the *Web GUI*'s *Hand-Eye Calibration* tab (Section 4.5) offers a guided process to conduct the calibration routine **manually**.

   **Note:** During the process, the described approach requires the user to manually enter into the Web GUI robot poses, which need to be accessed from the respective robot-teaching device or handheld.

### 6.7.2 Sensor mounting

As illustrated in Fig. 6.15 and Fig. 6.16, two different use cases w.r.t. to the mounting of the *rc_visard* generally have to be considered:

1. The *rc_visard* is **mounted on the robot**, i.e., it is mechanically connected at its *mounting points* (Section 3.6) to a robot link (e.g., at its flange or a flange-mounted tool), and hence moves with the robot.
2. The *rc_visard* is not mounted on the robot but is fixed to a table or other place in the robot's vicinity and remains at a **static** position w.r.t. the robot.

While the general *Calibration routine* (Section 6.7.3) is very similar in both use cases, the calibration process's output, i.e., the resulting calibration transform, will be semantically different, and the fixture of the calibration grid will also differ.

**Calibration with a robot-mounted sensor** When calibrating a robot-mounted *rc_visard* with the robot, the calibration grid has to be secured in a static position w.r.t. the robot, e.g., on a table or some other fixed-base coordinate system as sketched in Fig. 6.15.

**Warning:** It is extremely important that the calibration grid does not move during step 2 of the *Calibration routine* (Section 6.7.3). Securely fixing its position to prevent unintended movements such as those caused by vibrations, moving cables, or the like is therefore strongly recommended.

The result of the calibration (step 3 of the *Calibration routine*, Section 6.7.3) is a pose $\mathbf{T}_{\text{camera}}^{\text{robot}}$ describing the (previoulsy unknown) relative positional and rotational transformation between the *rc_visard*'s *camera* frame and the user-selected *robot* frame such that

$$\mathbf{p}_{\text{robot}} = \mathbf{R}\left(\mathbf{T}_{\text{camera}}^{\text{robot}}\right) \cdot \mathbf{p}_{\text{camera}} + \mathbf{t}\left(\mathbf{T}_{\text{camera}}^{\text{robot}}\right) \ , \tag{6.3}$$

where $\mathbf{p}_{\text{robot}} = (x, y, z)^T$ is a 3D-point with its coordinates expressed in the *robot* frame, $\mathbf{p}_{\text{camera}}$ is the same point represented in the *camera* coordinate frame, and $\mathbf{R}(\mathbf{T})$ as well as $\mathbf{t}(\mathbf{T})$ are the corresponding $3 \times 3$ rotation matrix and $3 \times 1$ translation vector to a pose $\mathbf{T}$, respectively.

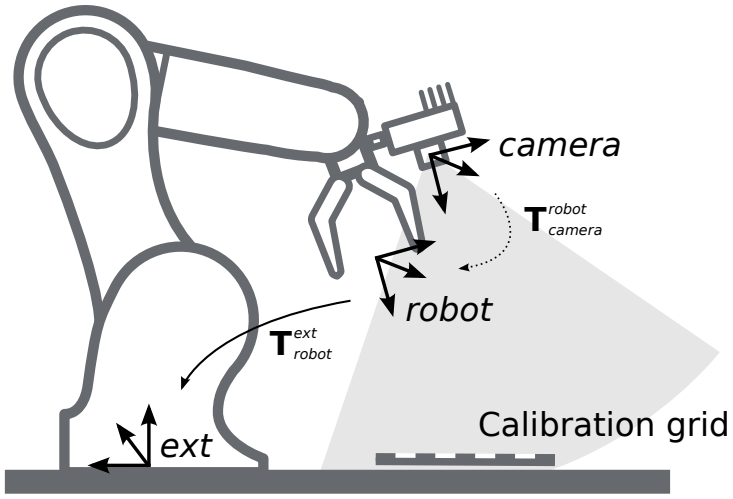



Fig. 6.15: Important frames and transformations for calibrating a robot-mounted *rc_visard*: The sensor is mounted with a fixed relative position to a user-defined *robot* frame (e.g., flange or TCP). It is important that the pose $\mathbf{T}_{\text{robot}}^{\text{ext}}$ of this *robot* frame w.r.t. a user-defined external reference frame *ext* is observable during the calibration routine. The result of the calibration process is the desired calibration transformation $\mathbf{T}_{\text{camera}}^{\text{robot}}$, i.e., the pose of the *camera* frame within the user-defined *robot* frame.

**Calibration with a statically-mounted sensor** In use cases where the *rc_visard* is positioned statically w.r.t. the robot, the calibration grid needs to be mounted to the robot as shown for example in Fig. 6.16 and Fig. 6.17.

**Note:** The hand-eye calibration component is completely agnostic about the exact mounting and positioning of the calibration grid w.r.t. the user-defined *robot* frame. That is, the relative positioning of the calibration grid to that frame neither needs to be known, nor it is relevant for the calibration routine, as shown in Fig. 6.17.

**Warning:** It is extremely important that the calibration grid is attached securely to the robot such that it does not change its relative position w.r.t. the user-defined *robot* frame during step 2 of the *Calibration routine* (Section 6.7.3).

Securely preventing unintended position changes such as those caused by vibrations, for example by mounting the calibration grid itself on a wooden support (suggested thickness min. 1 cm), which can then be screwed to the robot structure, e.g., its flange or tool, is therefore strongly recommended.

In this use case, the result of the calibration (step 3 of the *Calibration routine*, Section 6.7.3) is the pose $\mathbf{T}^{\text{ext}}_{\text{camera}}$ describing the (previoulsy unknown) relative positional and rotational transformation between the *rc_visard*'s *camera* frame and the user-selected external reference frame *ext* such that

$$\mathbf{p}_{\text{ext}} = \mathbf{R}\left(\mathbf{T}^{\text{ext}}_{\text{camera}}\right) \cdot \mathbf{p}_{\text{camera}} + \mathbf{t}\left(\mathbf{T}^{\text{ext}}_{\text{camera}}\right) \ , \tag{6.4}$$

where $\mathbf{p}_{\text{ext}} = (x, y, z)^T$ is a 3D point with its coordinates expressed in the external reference frame *ext*, $\mathbf{p}_{\text{camera}}$ is the same point represented in the *camera* coordinate frame, and $\mathbf{R}(\mathbf{T})$ as well as $\mathbf{t}(\mathbf{T})$ are the corresponding $3 \times 3$ rotation matrix and $3 \times 1$ translation vector to a pose $\mathbf{T}$, respectively.

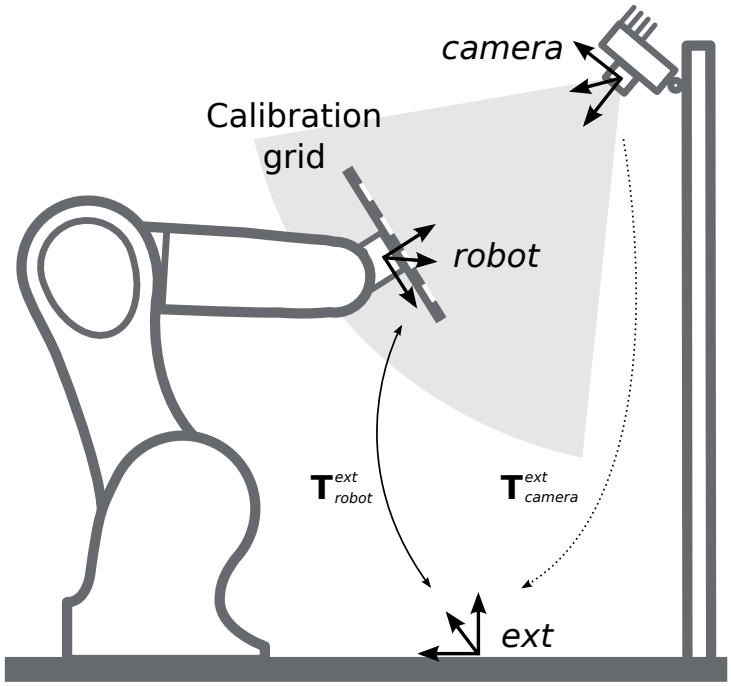



Fig. 6.16: Important frames and transformations for calibrating a statically mounted *rc_visard*: The sensor is mounted with a fixed position relative to a user-defined external reference frame *ext* (e.g., the world coordinate frame or the robot's mounting point). It is important that the pose $\mathbf{T}^{\text{ext}}_{\text{robot}}$ of the user-defined *robot* frame w.r.t. this frame is observable during the calibration routine. The result of the calibration process is the desired calibration transformation $\mathbf{T}^{\text{ext}}_{\text{camera}}$, i.e., the pose of the *camera* frame in the user-defined external reference frame *ext*.

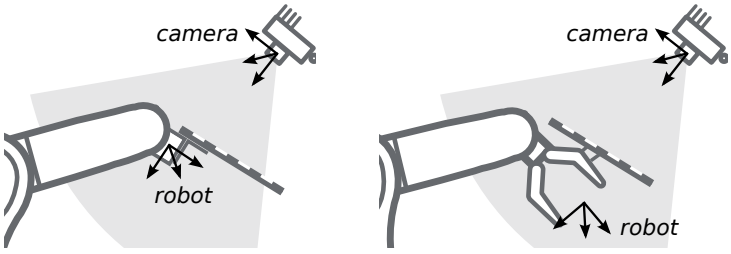



Fig. 6.17: Alternate mounting options for attaching the calibration grid to the robot

### 6.7.3 Calibration routine

The general hand-eye calibration routine consists of three steps, which are illustrated in Fig. 6.18. These three steps are also represented in the *Web GUI*'s guided hand-eye-calibration process (Section 4.5).

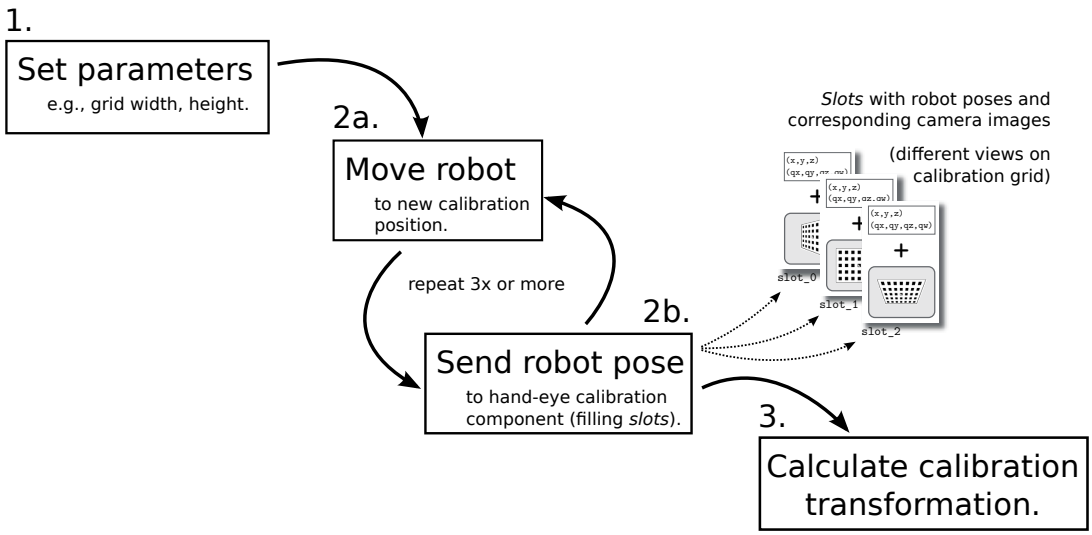



Fig. 6.18: Illustration of the three different steps involved in the hand-eye calibration routine

### Step 1: Setting parameters

Before starting the actual calibration routine, the grid size and sensor-mounting parameters have to be set to the component. As for the REST-API, the respective parameters are listed in *Parameters* (Section 6.7.4).

**Web GUI example:** The Web GUI offers an interface for entering these parameters during the first step of the calibration routine as shown in Fig. 6.19. In addition to grid size and sensor mounting, the Web GUI also offers a *Pose* setting to be defined by the user. It specifies the format used for reporting the robot poses in the upcoming step 2 of the calibration process, either as *XYZABC* for positions and Euler angles, or *XYZ+quaternion* for positions plus quaternions for representing rotations. See *Pose formats* (Section 13.1) for the exact definitions.

> **Note:** The *Pose* parameter is added to the Web GUI as a convenience option only. For reporting poses programmatically via REST-API, the *XYZ+quaternion* format is mandatory.

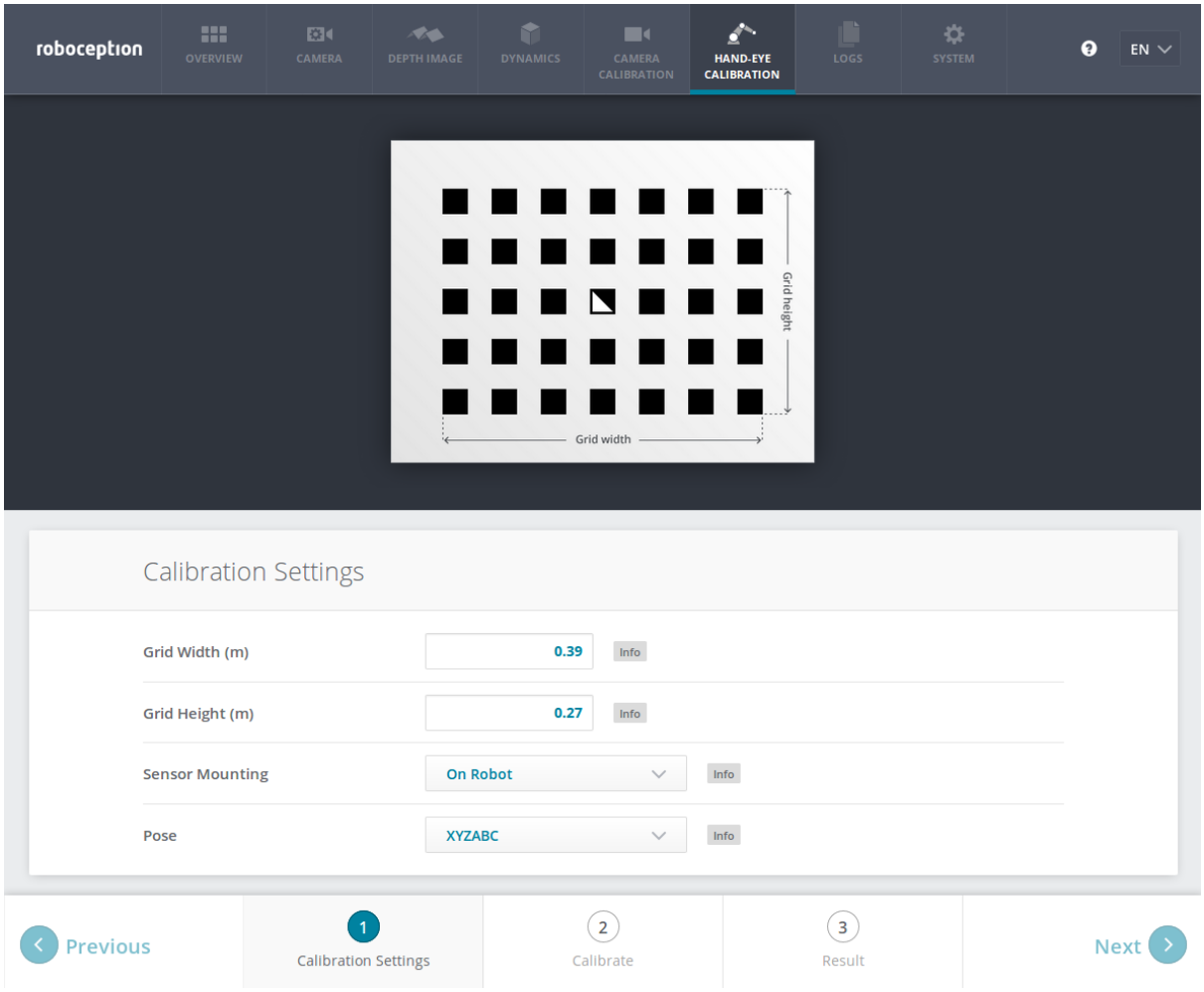



Fig. 6.19: Defining hand-eye calibration settings via the *rc_visard*'s Web GUI

### Step 2: Selecting and reporting robot calibration positions

In this step (2a.), the user defines several calibration positions for the robot to approach. These positions must each ensure that the calibration grid is completely visible in *rc_visard*'s left camera image. Furthermore, the robot positions need to be selected properly to achieve a variety of different perspectives for the *rc_visard* to perceive the calibration grid. Fig. 6.20 shows a schematic recommendation of four different view points.

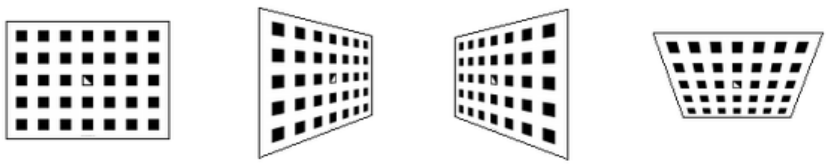


Fig. 6.20: Recommended views on the calibration grid during the calibration procedure

**Warning:** Calibration quality, i.e., the accuracy of the calculated calibration result, depends on the calibration-grid views provided. The more diverse the perspectives are, the better is the calibration. Choosing very similar views, i.e., varying the robot positions only slightly between different repetitions of step 2a., may lead to inaccurate estimation of the desired calibration transformation.

After the robot reaches each calibration position, the corresponding pose $\mathbf{T}^{\text{ext}}_{\text{robot}}$ of the user-defined *robot* frame in the user-defined external reference frame *ext* needs to be reported to the hand-eye calibration component (2b.). For this purpose, the component offers different *slots* to store the reported poses and the *rc_visard*'s corresponding left camera images. All filled slots will then be used to calculate the desired calibration transformation between the *rc_visard*'s *camera* frame and either the user-defined *robot* frame (robot-mounted sensor) or the user-defined external reference frame *ext* (static sensor).

**Note:** To successfully calculate the hand-eye calibration transformation, at least three different robot calibration poses need to be reported and stored in slots. However, to prevent errors induced by possible inaccurate measurements, at least **four calibration poses are recommended**.

To transmit the poses programmatically, the component's REST-API offers the `set_pose` service call (see *Services*, Section 6.7.5).

**Web GUI example:** After completing the calibration settings in step 1 and clicking *Next*, the Web GUI offers four different slots (*First View*, *Second View*, etc.) for the user to fill manually with robot poses. At the very top, a live stream from the camera is shown indicating whether the calibration grid is currently detected or not. Next to each slot, a figure suggests a respective dedicated viewpoint on the grid. For each slot, the robot must be operated to achieve the suggested view.

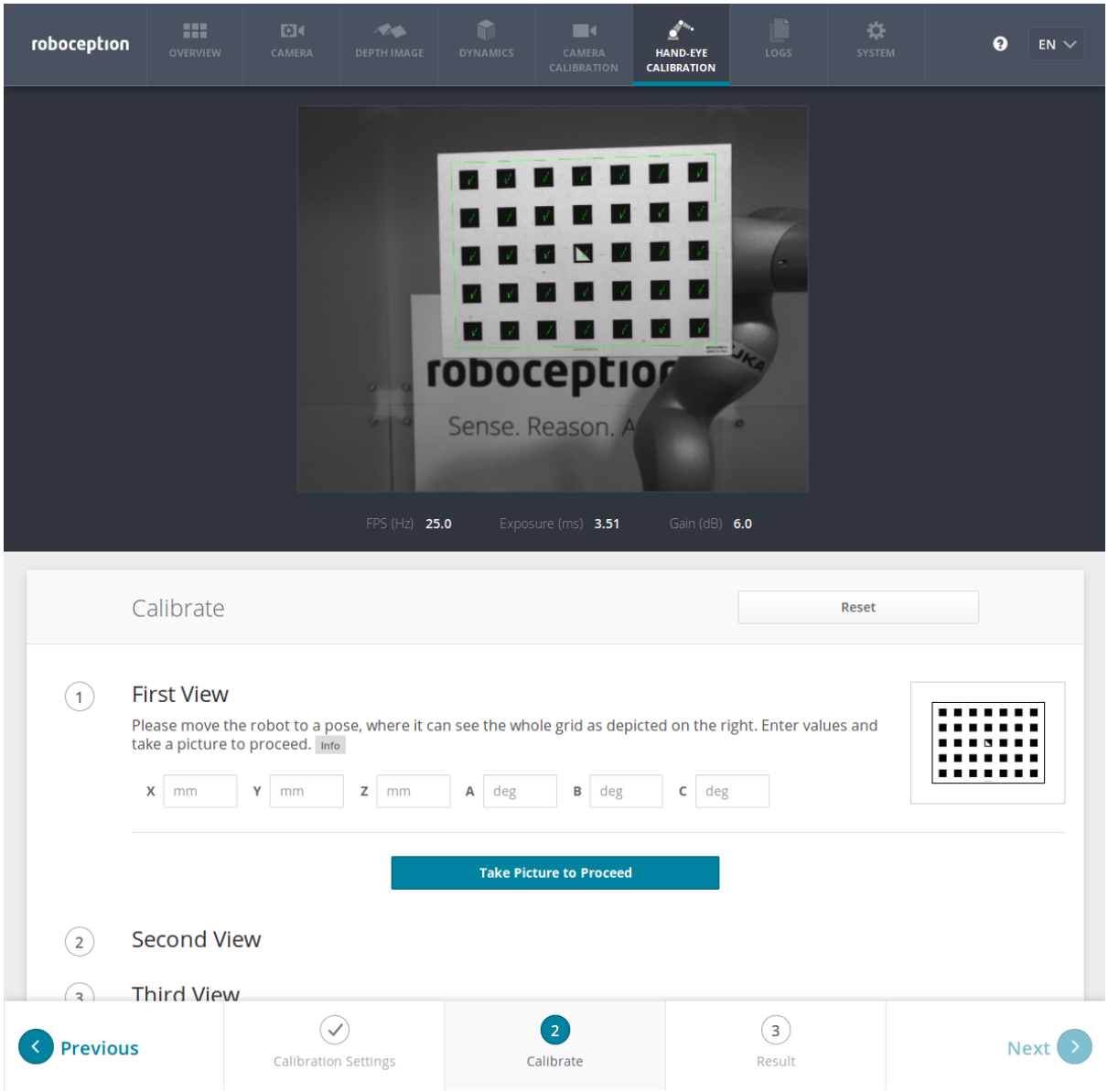



Fig. 6.21: First sample image in the hand-eye calibration process for a statically mounted *rc_visard*

Once the suggested view is achieved, the user-defined *robot* frame's pose needs to be entered manually into the respective text fields, and the corresponding camera image is captured using the *Take Picture to Proceed* button.

**Note:** The user's acquisition of robot pose data depends on the robot model and manufacturer – it might be read from a teaching or handheld device, which is shipped with the robot.

**Warning:** Please be careful to correctly and accurately enter the values; even small variations or typos may lead to calibration-process failure.

This procedure is repeated four times in total. Complying to the suggestions to observe the grid from above, left, front, and right, as sketched in Fig. 6.20, in this example the following corresponding camera images have been sent to the hand-eye calibration component with their associated robot pose:

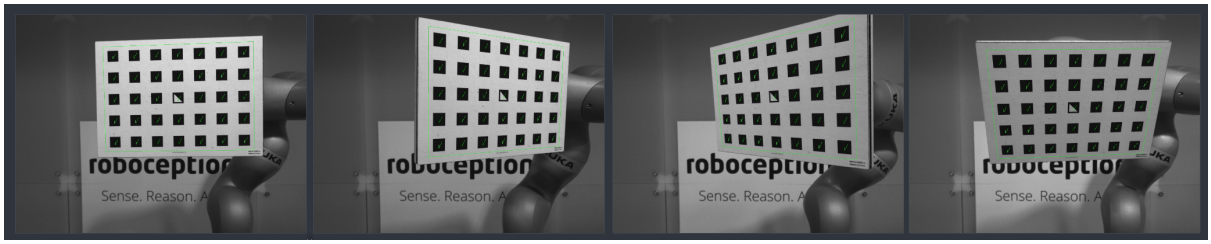


Fig. 6.22: Recorded camera images as input for the calibration procedure

### Step 3: Calculating and saving the calibration transformation

The final step in the hand-eye calibration routine consists of issuing the desired calibration transformation to be computed from the collected poses and camera images. The REST-API offers this functionality via the `calibrate` service call (see *Services*, Section 6.7.5). Depending on the way the *rc_visard* is mounted, this service computes and returns the transformation (i.e., the pose) between the *camera* frame and either the user-defined *robot* frame (robot-mounted sensor) or the user-defined external reference frame *ext* (statically mounted sensor); see *Sensor mounting* (Section 6.7.2).

To enable users to judge the quality of the resulting calibration transformation, the component also reports a calibration error. This value is measured in pixels and denotes the root mean square of the *reprojection error* averaged over all calibration slots and all corners of the calibration grid. However for a more intuitive understanding, this measurement might be normalized by utilizing *rc_visard*'s focal length $f$ in pixels:

$$E = \frac{E_{\text{camera}}}{f} \, .$$

**Note:** The *rc_visard* reports a focal length factor via its various interfaces. It relates to the image width for supporting different image resolutions. The focal length $f$ in pixels can be easily obtained by multiplying the focal length factor by the image width in pixels.

The value $E$ can now be interpreted as an object-related error in meters in the 3D-world. Given that the distance between the calibration grid and the *rc_visard* is one meter, the *average* accuracy associated with transforming the grid's coordinates from the *camera* frame to the target frame is $1 \cdot E$ m; assuming a distance of 0.5 meters, it measures $0.5 \cdot E$ m, etc.

**Web GUI example:** The Web GUI automatically triggers computation of the calibration result immediately after taking the last of the four pictures. The user just needs to click the *Next* button to proceed to the result. In this example with a statically mounted *rc_visard*, the resulting output is the pose of the sensor's left camera in the world coordinate system of the robot – represented in the *pose* format as specified in step 1 of the calibration routine.

The reported error of $E_{\text{camera}} = 0.4$ pixels in Fig. 6.23 transforms into a calibration accuracy of $E = \frac{E_{\text{camera}}}{f} \approx \frac{0.4}{1081.46} \approx 0.00036$, which is 0.36 mm at 1 meter distance – a submillimeter accuracy for this calibration run.

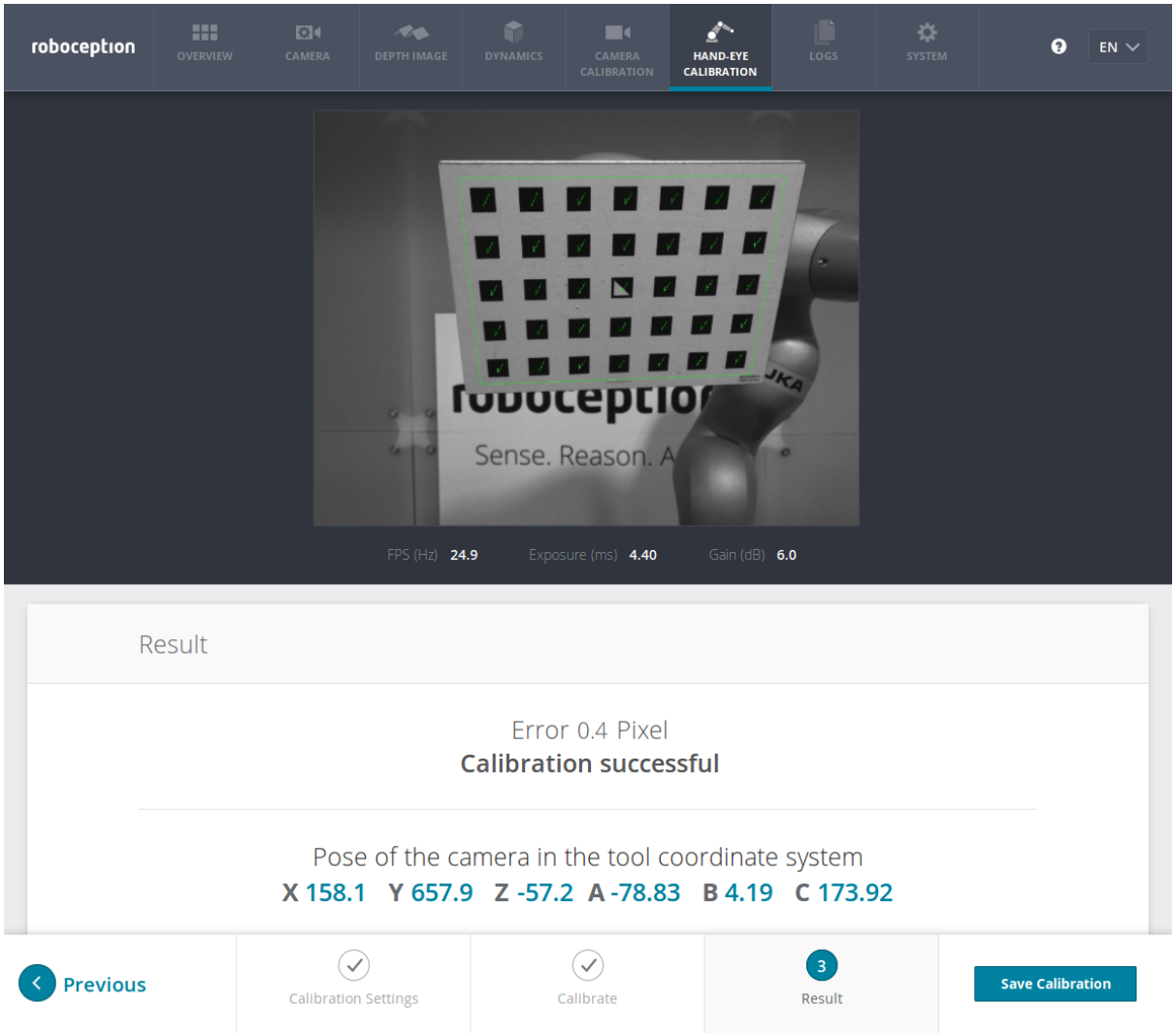



Fig. 6.23: Result of the hand-eye calibration process displayed in the Web GUI

### 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 6.9: 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 6.10: 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",
```

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

Table 6.11: 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 |
| 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 6.12: 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 6.13: 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",
  "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 6.14: 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).
- ***TagDetect*** **(`rc_april_tag_detect` and `rc_qr_code_detect`, Section 7.2)** allows the detection of April-Tags and QR codes, as well as the estimation of their poses.
- ***ItemPick*** **(`rc_itempick`, Section 7.3)** provides an out-of-the-box perception solution for robotic pick-and-place applications with suction grippers.

## 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.

### 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.

### 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: 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 7.2: The `rc_slam` component's status values

| Name | Description |
|---|---|
| `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 7.3: 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 trajectory start time; negative values are interpreted as an offset from the trajectory end time. The below diagram illustrates three examples for the relative parameterization.

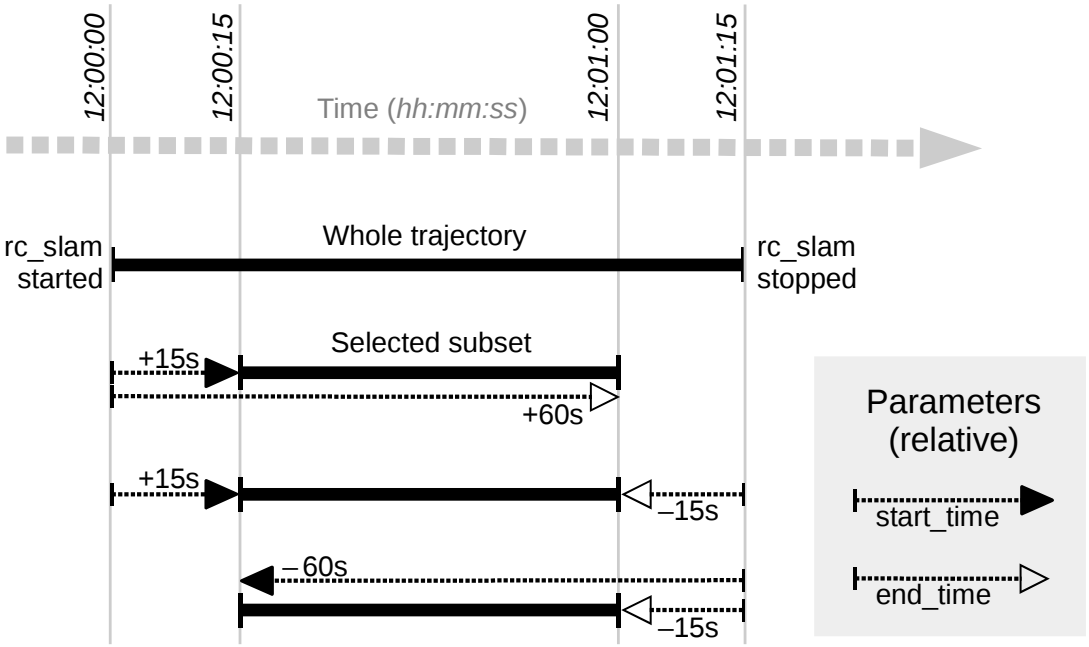



Fig. 7.1: Examples for combinations of relative start and end times for the `get_trajectory` service. All combinations shown select the same subset of the trajectory.

**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.

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:

```
{
  "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"
    }
  }
}
```


## 7.2 TagDetect

### 7.2.1 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 7.2.2).
2. Estimation of the pose of each tag (see *Pose estimation*, Section 7.2.3).
3. Re-identification of previously seen tags (see *Tag re-identification*, Section 7.2.4).

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

#### QR code

Fig. 7.2: 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

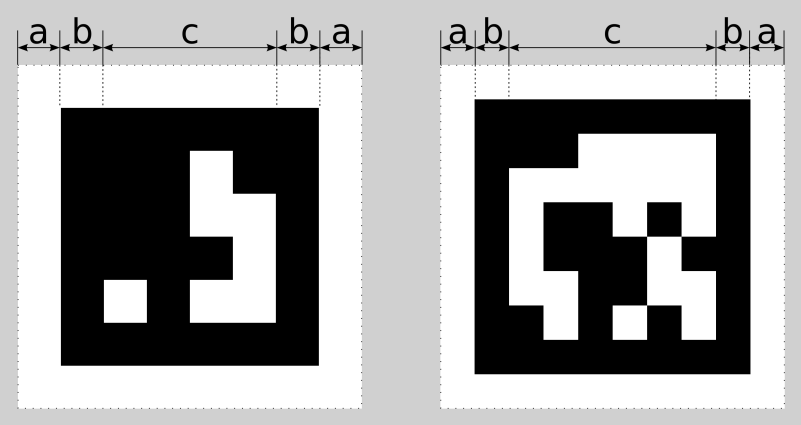



Fig. 7.3: A 16h5 tag (left) and a 36h11 tag (right). AprilTags consist of a mandatory white (a) and black (b) border and a variable amount of data bits (c).

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 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 for example are of family 16h5 and 36h11 and have id 0 and 11, respectively. All supported families are shown in Table 7.4.

Table 7.4: 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), 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 7.4).

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). 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.

### 7.2.2 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).

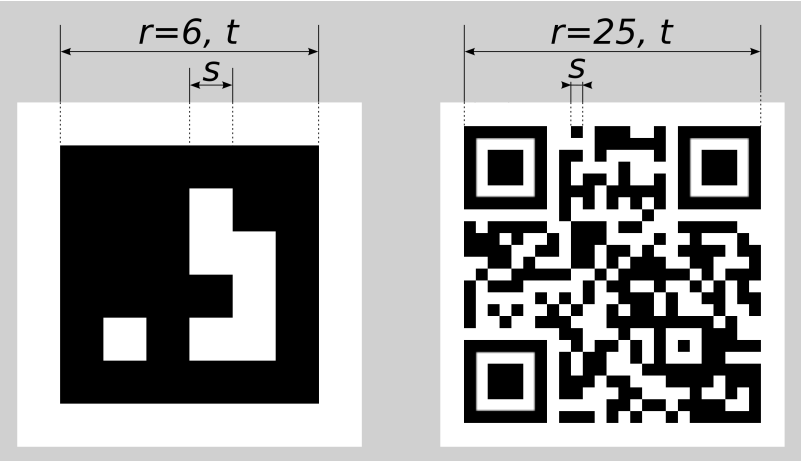



Fig. 7.4: Visualization of module size $s$, size of a tag in modules $r$, and size of a tag in meters $t$ for AprilTags (left) and QR codes (right)

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. 7.4 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 7.5: 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 7.6: 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 |

### 7.2.3 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.

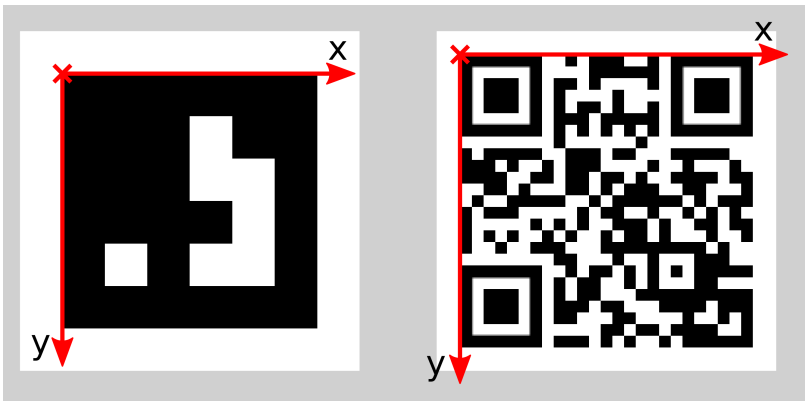



Fig. 7.5: Coordinate frames of AprilTags (left) and QR codes (right)

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.

**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:** 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. The TagDetect components try to detect such situations and filter out affected tags. In addition, the user is advised to check the estimated tag size for consistency; if pose estimation fails, the estimated tag size will be different from the printed tag size.

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 7.7: 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 7.8: 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 |

### 7.2.4 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. 7.6. 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.

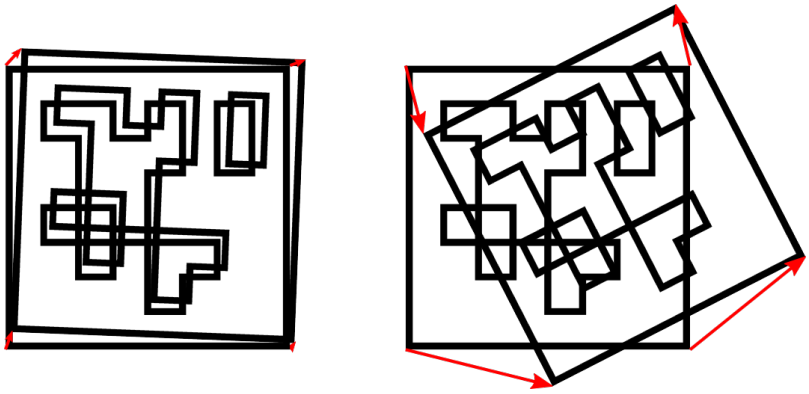


Fig. 7.6: Simplified visualization of tag re-identification. Euclidean distances between associated tag corners in 3D are compared (red arrows).

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`.

### 7.2.5 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.

#### Parameters and status values

This component offers the following run-time parameters.

Table 7.9: The `rc_qr_code_detect` component's run-time parameters

| Name | Type | Min | Max | Default | Description |
|---|---|---|---|---|---|
| `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 7.10: 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 7.11: 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"
    }
  ]
}
```

This service returns the following response:

```
{
  "return_code": {
    "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.

**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).

Due to changes in system time on the sensor there might occur jumps of timestamps, forward as well as backward (see *Time synchronization*, Section 8.4). 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.

## 7.3 ItemPick

The ItemPick component is an optional component that runs on board the *rc_visard*. It provides an out-of-the-box perception solution for robotic pick-and-place applications with suction grippers. ItemPick computes possible grasp poses for the suction device. Each suggested grasp includes a quality value related to the surface available for grasping. Visualization streams are available for parameter configuration and application tuning.

ItemPick is also ready to be used for bin-picking applications. It offers a load carrier detection solution to provide grasps for items inside the load carrier.

The ItemPick component works with both static and robot-mounted *rc_visard* devices and can optionally be combined with the on-board *Hand-eye calibration* (Section 6.7) component to provide grasp poses in the user-configured external reference frame.

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

# 8 Interfaces

Three 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.

## 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

**`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 *Provided image streams*, Section 8.1.2).

**`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`, `WidthMax` (read-only)**

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

**`Height`, `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 *Provided image streams*, Section 8.1.2).

**`PixelFormat`**

- type: Enumeration, `Mono8` or `YCbCr411_8` (color sensors only)
- description: Pixel format of the left and right rectified images that are returned in the components `Intensity` and `IntensityCombined`. The `YCbCr411_8` format is only available for color cameras.

**`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).

**`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).

**`GevIEEE1588`**

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

**`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`.

**`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. However, 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, we set 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.

### Custom GenICam features of the *rc_visard*

**`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).

**`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.

**`Baseline` (read-only)**

- type: Float
- description: The distance in meters between the left and the right camera.

**`DepthQuality`**

- type: Enumeration, one of `Low`, `Medium`, `High`, or `StaticHigh`
- default: `High`
- description: Quality of the disparity images. Lower quality results in disparity images with lower resolution (*Quality*, 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).

**`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).

### 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)<br>`YCbCr411_8` (color sensors) | 1280×960 | Left rectified camera image |
| `IntensityCombined` | `Mono8` (monochrome sensors)<br>`YCbCr411_8` (color sensors) | 1280×1920 | Left rectified camera image stacked on right rectified camera image |
| `Disparity` | `Coord3D_C16` | 640×480<br>320×240<br>214×160 | Disparity image in desired resolution, i.e., `High`, `Medium`, or `Low` |
| `Confidence` | `Confidence8` | same as `Disparity` | Confidence image |
| `Error` | 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 are required. These parameters are transmitted as GenICam features *FocalLengthFactor* and *Baseline*. The *FocalLengthFactor* value has to be multiplied by the width of the disparity image to get the focal length $f$ in pixels for the given disparity image resolution. 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 with $w$ columns and $h$ rows, the 3D reconstruction in meters can be written with the GenICam parameters as

$$
\begin{aligned}
P_x &= \left(i - \frac{w}{2}\right) \frac{\text{Baseline}}{d_{ik} \cdot \text{Scan3dCoordinateScale}}, \\
P_y &= \left(k - \frac{h}{2}\right) \frac{\text{Baseline}}{d_{ik} \cdot \text{Scan3dCoordinateScale}}, \\
P_z &= (\text{FocalLengthFactor} \cdot w) \frac{\text{Baseline}}{d_{ik} \cdot \text{Scan3dCoordinateScale}}.
\end{aligned}
$$

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

$$
z_{eps} = \frac{e_{ik} \cdot \text{Scan3dCoordinateScale} \cdot \text{FocalLengthFactor} \cdot w \cdot \text{Baseline}}{(d_{ik} \cdot \text{Scan3dCoordinateScale})^2}.
$$

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 configuration* (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",
      "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"
    ],
    "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.

**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.

**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",
    "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)",
    "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)",
  "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": [
    {
      "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
    },
    {
      "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,
  "serial": "string",
  "time": 0,
  "uptime": 0
}
```


SysInfo objects are used in the following requests:

- *GET /system*

### 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.

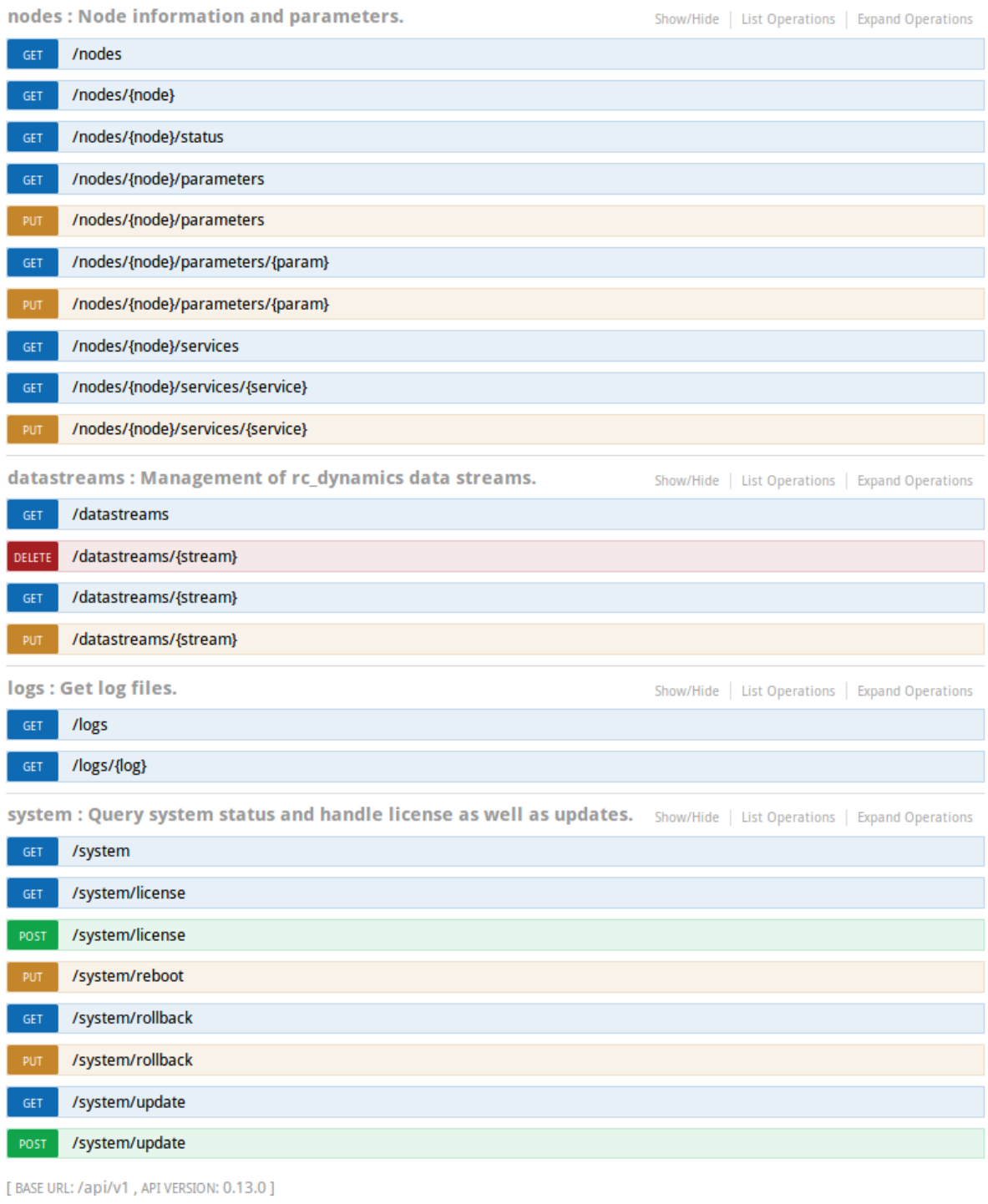



Fig. 8.1: Initial view of the *rc_visard*'s Swagger UI with its resources and requests grouped into `nodes`, `datastreams`, `logs`, and `system`

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.

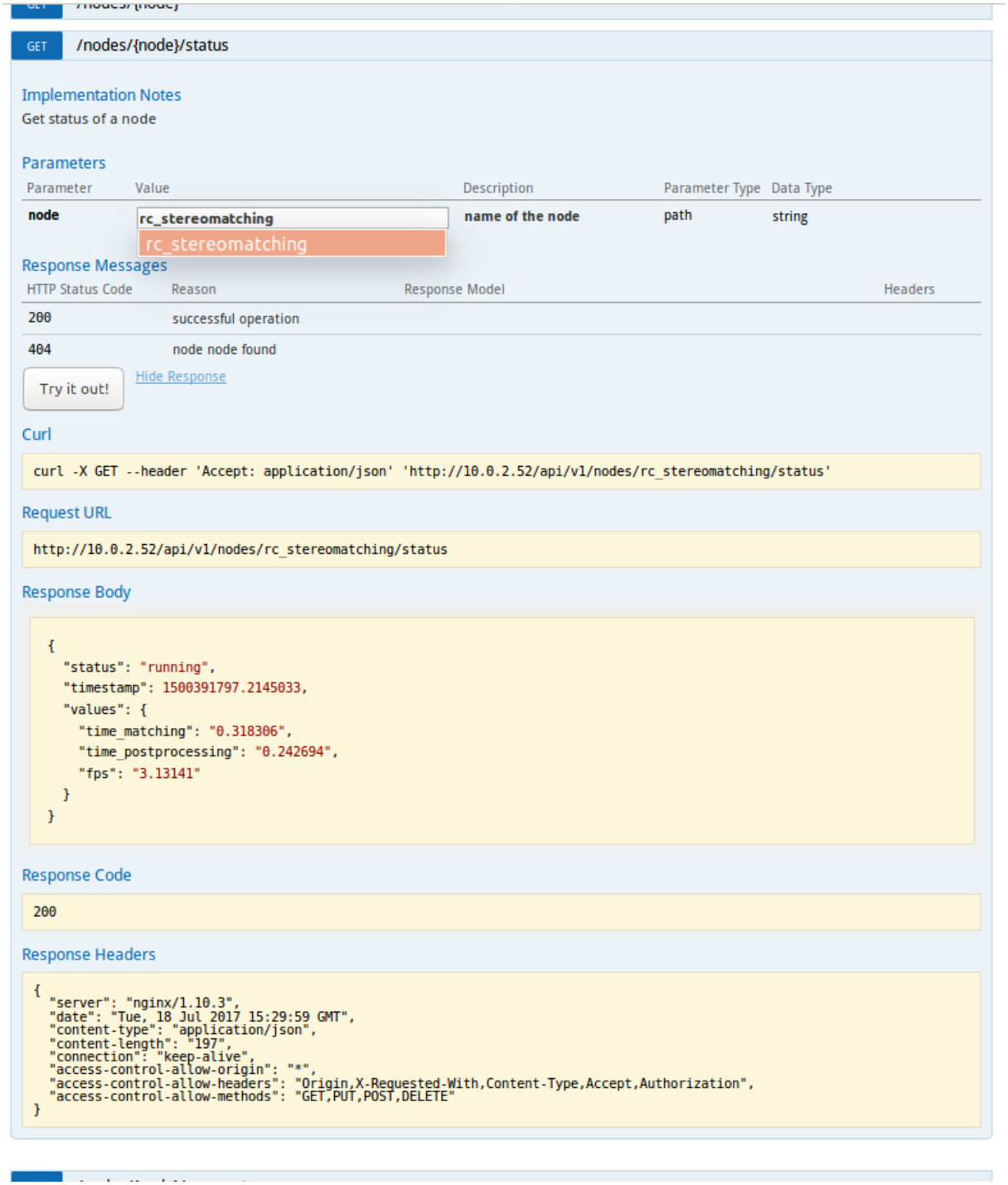



Fig. 8.2: Result of requesting the `rc_stereomatching` node's status

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.

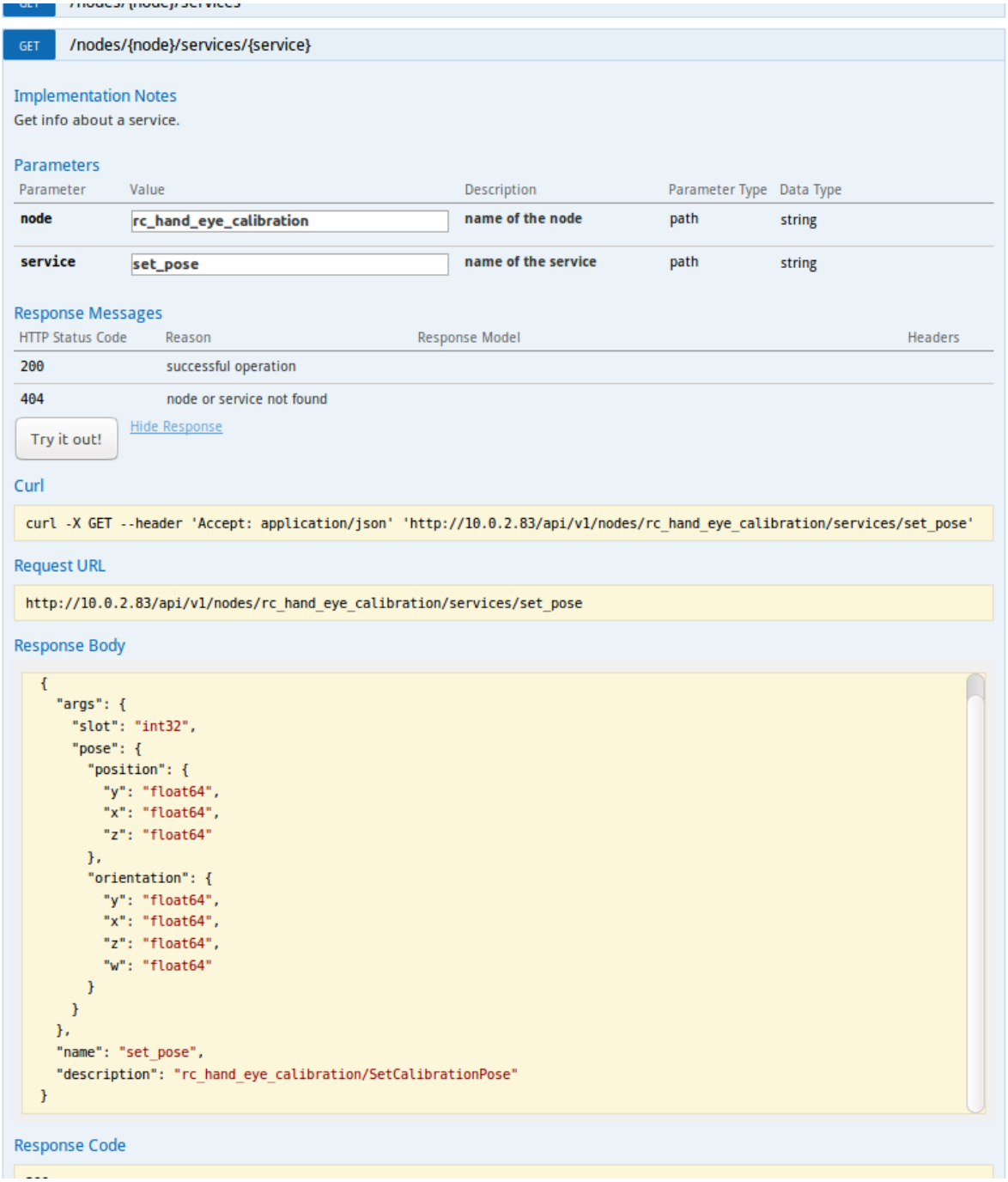



Fig. 8.3: The result of the `GET` request on the `set_pose` service shows the required arguments for this service call.

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

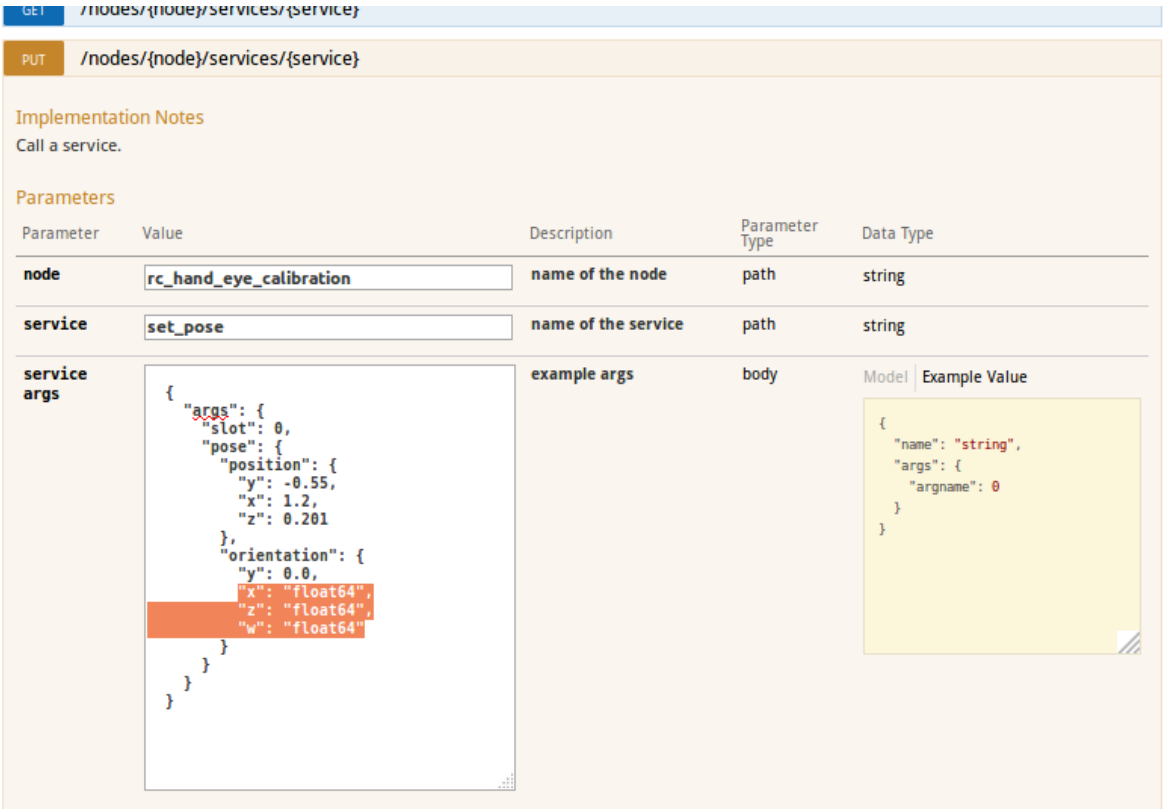



Fig. 8.4: Filling in the arguments of the `set_pose` service request

## 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 8.1: 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
}
```

- 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/s^2
  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
}
```

- 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/
↪s^2 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 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.

### 8.4.1 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.

### 8.4.2 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. 9.1). Select the desired update image file (file extension `.mender`) from the local file system and open it to start the update.

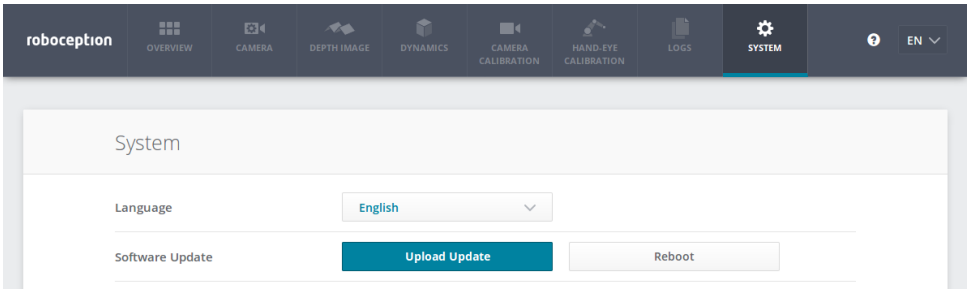



Fig. 9.1: Web GUI *System* tab

**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. 9.2.

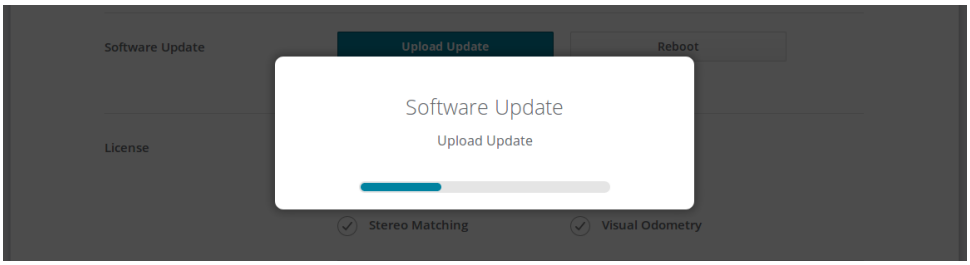



Fig. 9.2: Software update progress bar

**Note:** Depending on the web browser, the update progress status shown in Fig. 9.2 may indicate the completion of the update too early. Please wait until the context window shown in Fig. 9.3 opens. Expect an overall update time of at least five minutes.

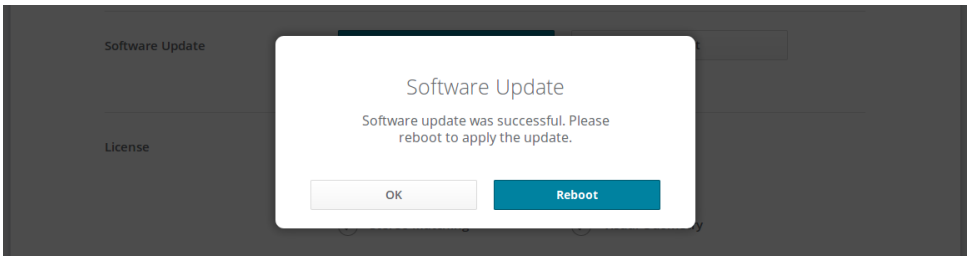



Fig. 9.3: Software update rebooting screen

> **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 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.

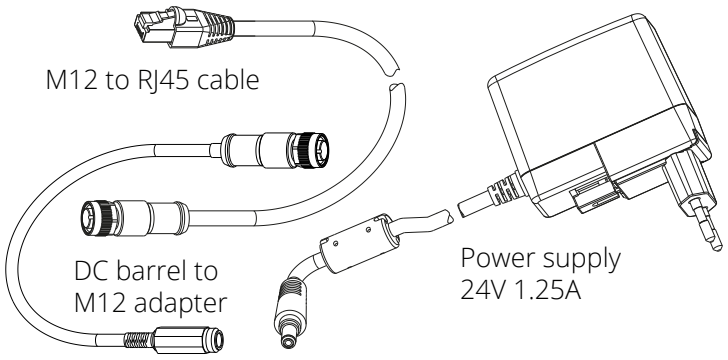



Fig. 10.1: The optional connectivity kit's components

## 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: 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 11.2: 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). 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).

- 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).
- 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 3.1).
- 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).
- 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).

**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

## /datastreams



## /logs



## /nodes



## /system

# Index

## Symbols



## A



## B



## C



## D

## E



## F



## G

## M



## N



## O



## P



## Q



## R

## S



## T



## U



## V



## W

## X