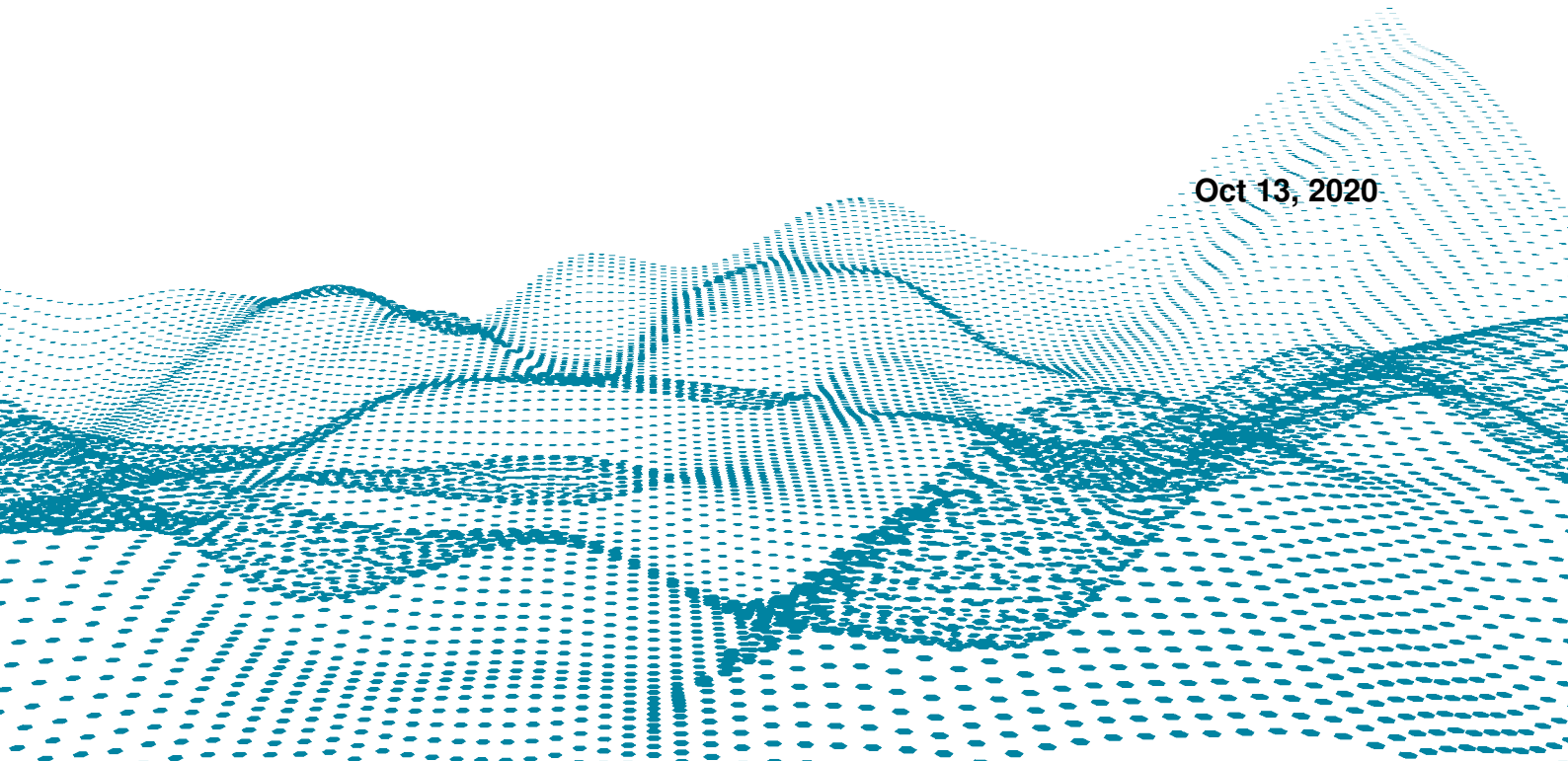
rc_visard

Documentation Revision 20.10.1

Roboception GmbH



Oct 13, 2020

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.

Documentation Revision 20.10.1 Oct 13, 2020

Applicable to *rc_visard* firmware 20.10.x

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* is an IP54-protected, self-registering stereo-camera with on-board computing capabilities.

The *rc_visard* provides real-time camera images and disparity images, which can also be used to compute depth images and 3D point clouds. Additionally, it provides confidence and error images as quality measures for each image acquisition. It offers an intuitive web UI (user interface) and a standardized GenICam interface, making it compatible with all major image processing libraries.

With optionally available *rc_reason* modules the *rc_visard* provides out-of-the-box solutions for object detection and robotic pick-and-place applications.

The *rc_visard* also 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 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.

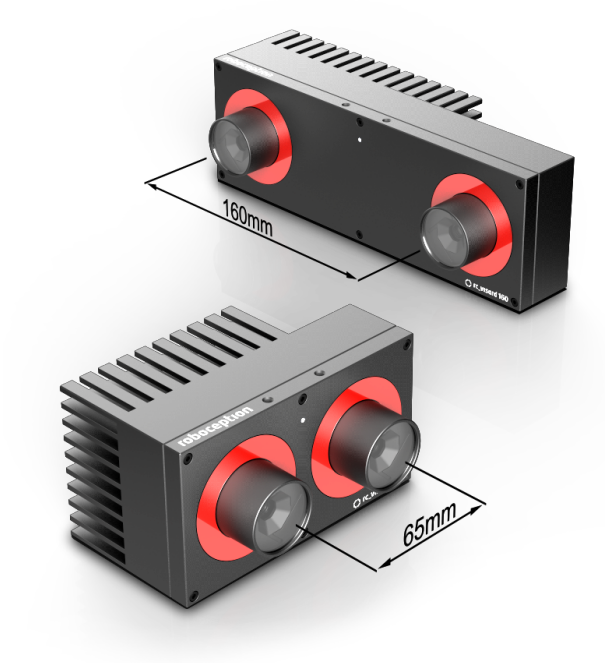


Fig. 1.1: *rc_visard* 65 and *rc_visard* 160

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 to the hard- and software not expressly approved by Roboception could void the user's warranty and guarantee rights.

Warning: The *rc_visard* utilizes complex hardware and software technology that may behave in a way not intended by the user. The purchaser must design its application to ensure that any failure or the *rc_visard* 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

NRTL certification by TÜV Süd

FC

Changes or modifications not expressly approved by the manufacturer could void the user's authority to operate the equipment. This device complies with Part 15 of the FCC rules. Operation is subject to the following two conditions:

1. This device may not cause harmful interference, and
2. this device must accept any interference received, including interference that may cause undesired operation.

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
- Registered FCC ID: 2AVMTRCV17
- Certified for Canada according to CAN ICES-3(B)/NMB-3(B)

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.

DOF The Degrees Of Freedom (DOF) are the number of independent parameters for translation and rotation. In 3D space, 6DOF (i.e. three for translation and three rotation) are sufficient to describe an arbitrary position and orientation.

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* from the 169.254.0.0/16 IP range 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.

TCP The Tool Center Point (TCP) is the position of the tool at the end effector of a robot. The position and orientation of the TCP determines the position and orientation of the tool in 3D space.

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 of 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://<ip>/api/v1/nodes/rc_stereocamera/parameters/fps` where `<ip>` 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 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* product.

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 device 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 product 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 may 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 4.6), 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.

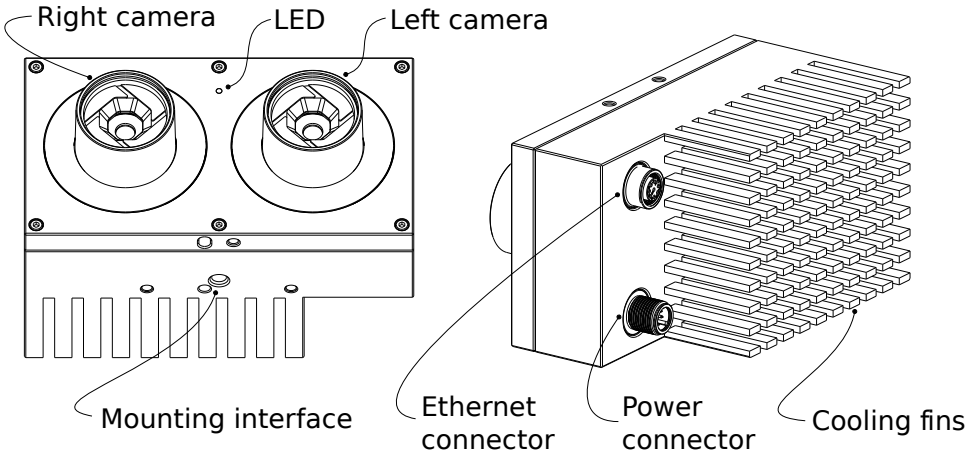


Fig. 3.1: Parts description

3.2 Technical specification

The common technical specifications for the *rc_visard* variants are given in Table 3.1. The *rc_visard* 160 is available with two different types of lenses: 4 mm and 6 mm focal length. The *rc_visard* 65 is only available with 4 mm lenses.

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

	rc_visard 65 / *rc_visard* 160
Image resolution	1280 x 960 pixel, color or monochrome
Field of view	4 mm lens: Horizontal: 61°, Vertical: 48° 6 mm lens: Horizontal: 43°, Vertical: 33°
IR Cutoff	650 nm
Depth image	1280 x 960 pixel (Full) @ 1 Hz (with StereoPlus license) 640 x 480 pixel (High) @ 3 Hz 320 x 240 pixel (Medium) @ 15 Hz 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
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 combination of baselines and lens types leads to different resolutions and accuracies.

Table 3.3: Resolution and accuracy of the *rc_visard* variants in millimeters with full resolution stereo matching and random dot projection on non-reflective and non-transparent objects.

	distance (mm)	*rc_visard* 65-4	*rc_visard* 160-4	*rc_visard* 160-6
lateral resolution (mm)	200	0.2	-	-
	500	0.5	0.5	0.3
	1000	0.9	0.9	0.6
	2000	1.9	1.9	1.3
	3000	2.8	2.8	1.9
depth resolution (mm)	200	0.04	-	-
	500	0.2	0.1	0.06
	1000	0.9	0.4	0.3
	2000	3.6	1.5	1.0
	3000	8.0	3.3	2.2
Average depth accuracy (mm)	200	0.2	-	-
	500	0.9	0.4	0.3
	1000	3.6	1.5	1.0
	2000	14.2	5.8	3.9
	3000	32.1	13.0	8.8

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

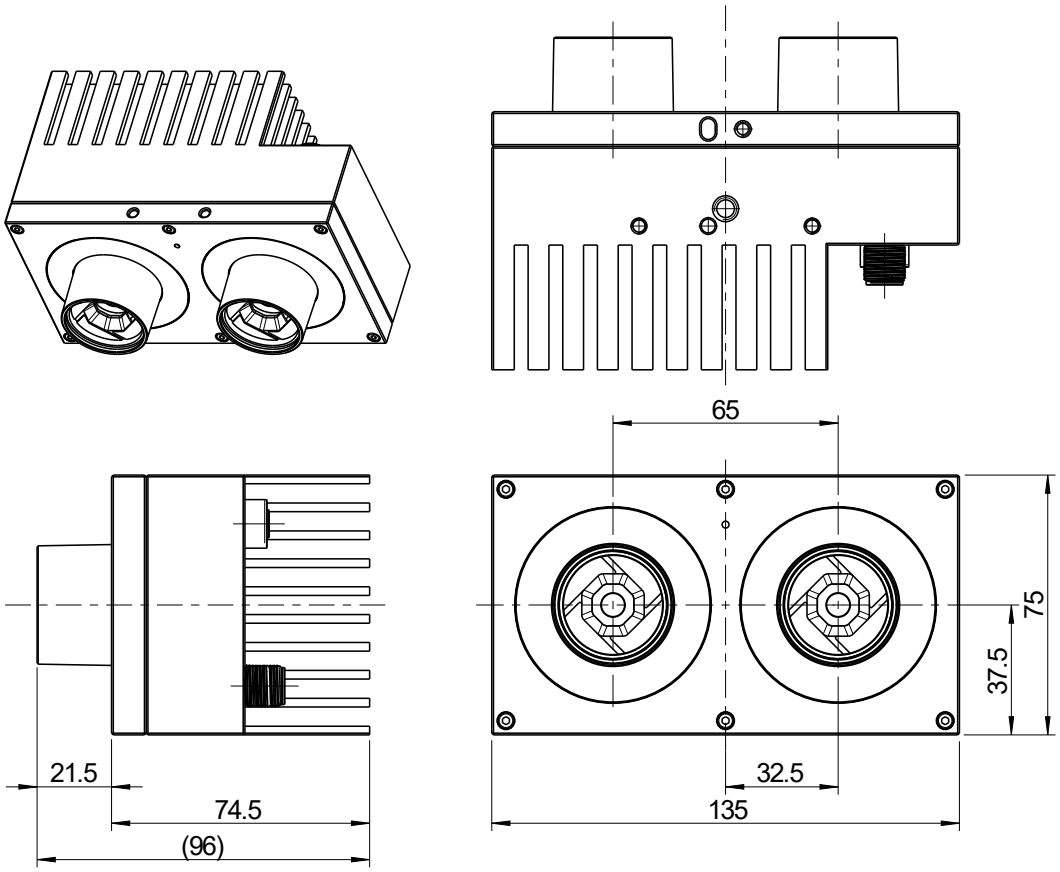


Fig. 3.2: Overall dimensions of the *rc_visard* 65

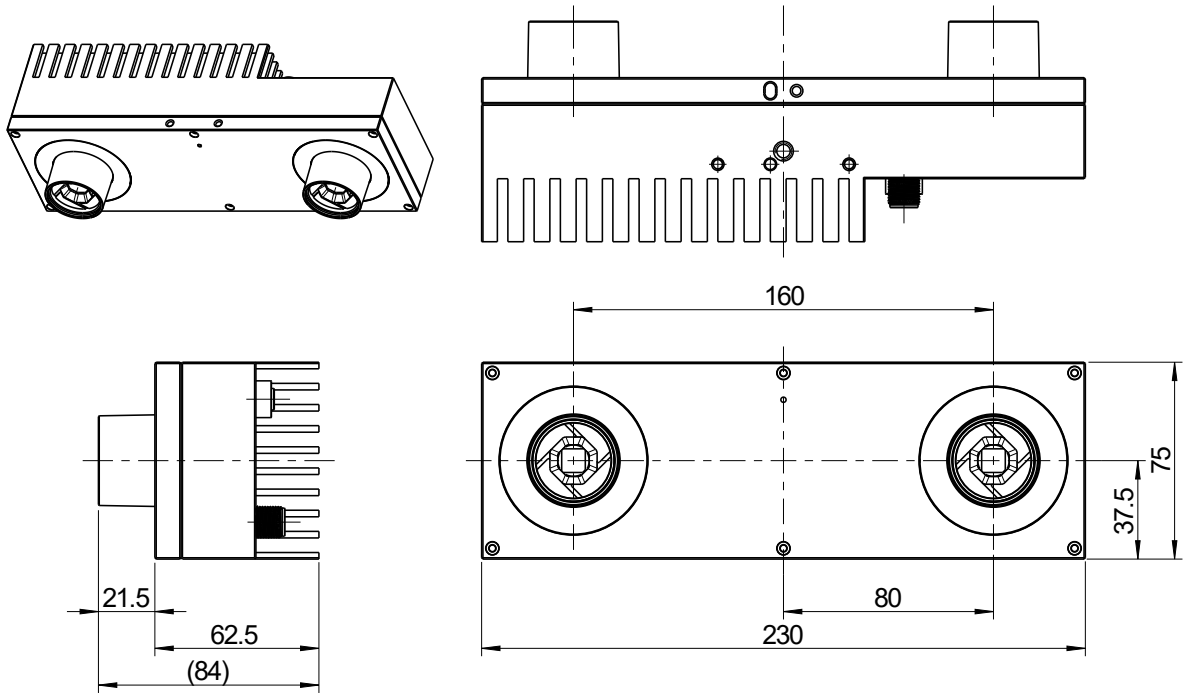


Fig. 3.3: Overall dimensions of the *rc_visard* 160

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 actual 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 3.4.

Table 3.4: 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 • Free from explosive liquids or gases • Free from powerful electromagnetic interference

The *rc_visard* is designed for an operating temperature (surrounding environment) 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 all parts 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. 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 3.5: 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	see *Standards* (Section 1.3.3)		

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.

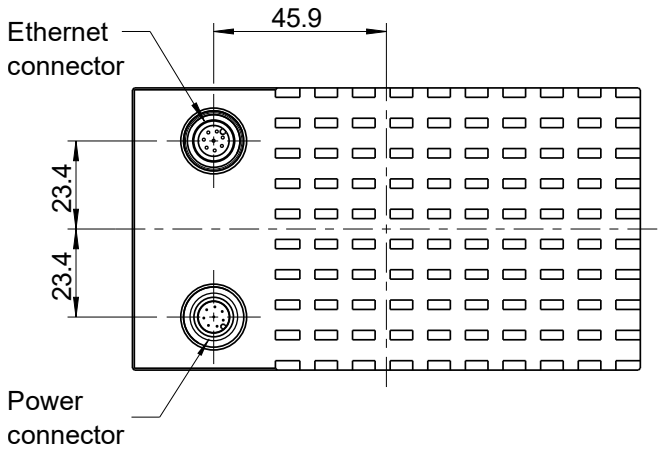


Fig. 3.4: Locations of the electrical connections for the *rc_visard* 65, with Ethernet on top and power on the bottom

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

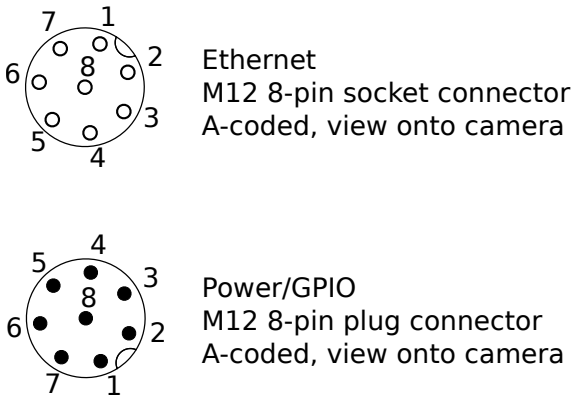


Fig. 3.5: Pin positions for power and Ethernet connector

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

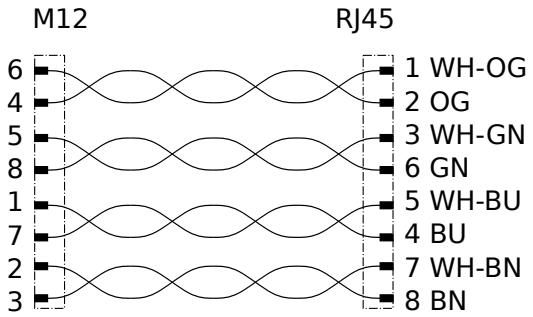


Fig. 3.6: Pin assignments for M12 to Ethernet cabling

Pin assignments for the power connector are given in Table 3.6.

Table 3.6: 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 Out 1* by default provides an exposure sync signal with a logic high level for the duration of the image exposure. All GPIOs can be controlled via the optional IOControl component (*IO and Projector Control*, Section 6.8). Pins of unused GPIOs 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 circuitry and specifications are shown in Fig. 3.7. The maximum rated voltage for *GPIO In* and *GPIO Vcc* is 30 V.

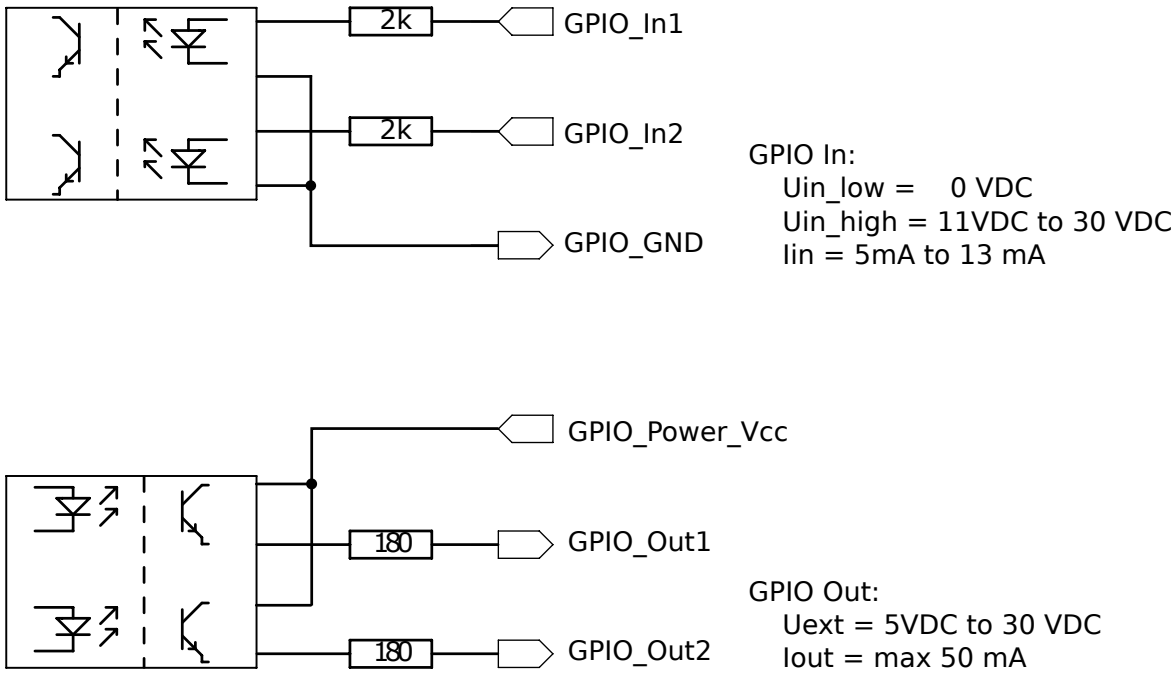


Fig. 3.7: GPIO circuitry and specifications – do not connect signals higher than 30 V

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.

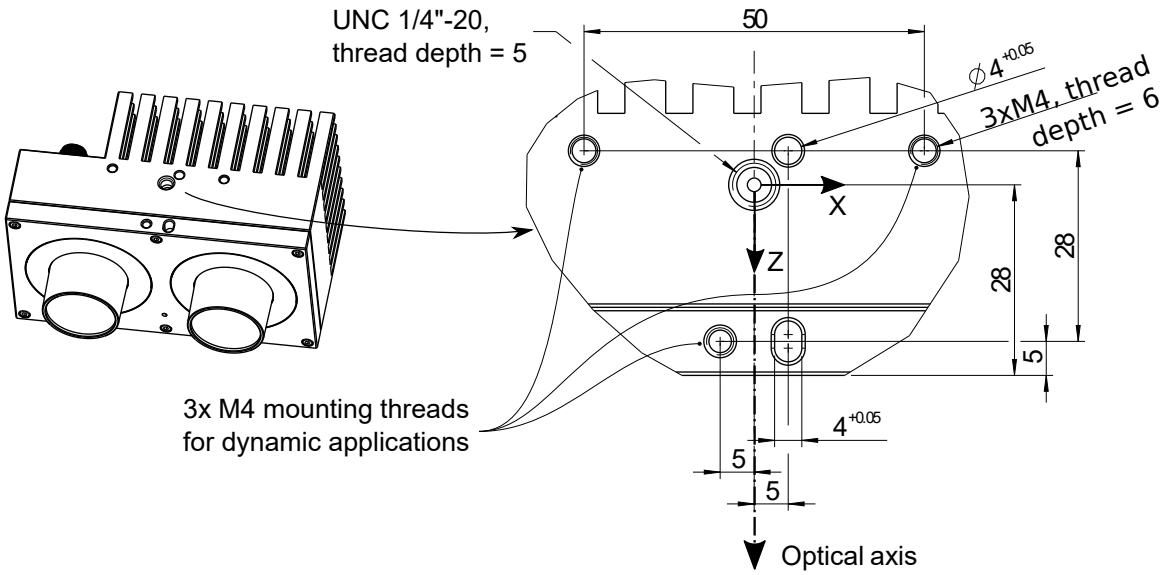


Fig. 3.8: Mounting-point for connecting the *rc_visard* to robots or other mountings

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.

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).

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.

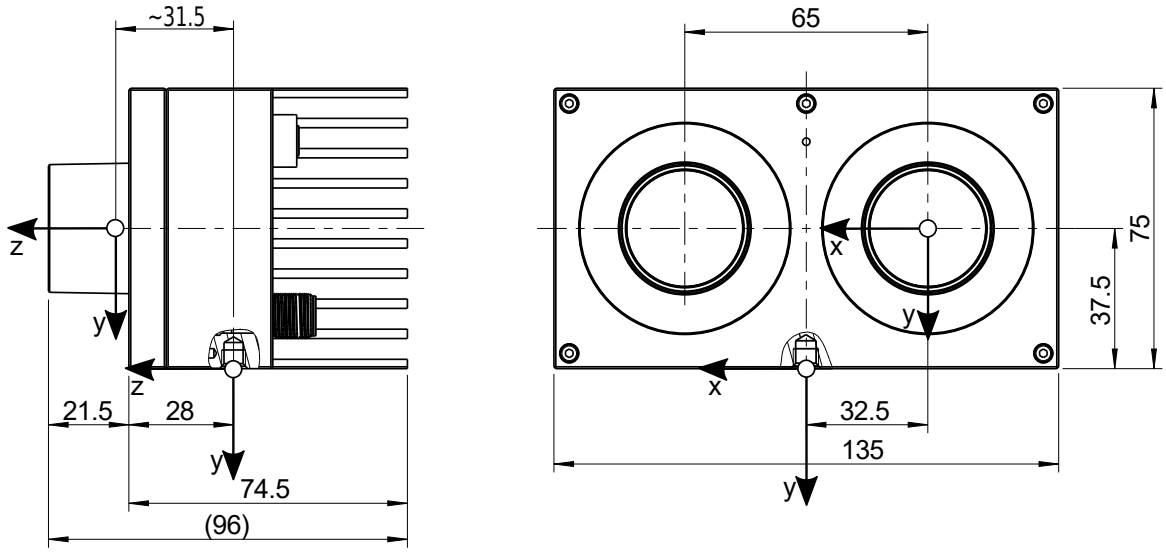


Fig. 3.9: Approximate location of sensor/camera coordinate frame (inside left lens) and mounting-point frame (at tripod thread) for the *rc_visard* 65

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

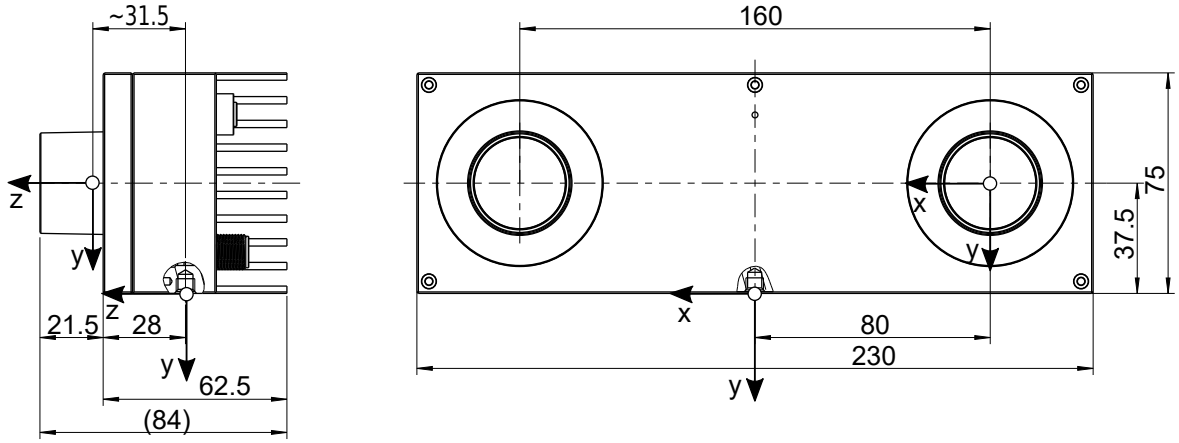


Fig. 3.10: Approximate locations of sensor/camera coordinate frame (inside left lens) and mounting-point frame (at tripod thread) for the *rc_visard* 160

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 Software license

Every *rc_visard* device ships with a pre-installed license file for licensing and protection of the installed software packages. The license is bound to that specific *rc_visard* device and cannot be used or transferred to other devices.

The functionality of the *rc_visard* can be enhanced anytime by *upgrading the license* (Section 9.6), e.g., for optionally available software components.

Note: The *rc_visard* requires to be rebooted whenever the installed licenses have changed.

Note: The license status can be retrieved via the *rc_visard*'s various interfaces such as the *System* tab of the *Web GUI* (Section 4.6).

4.3 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.

4.4 Discovery of *rc_visard* devices

Roboception devices that are powered up and connected to the local network or directly to a computer 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.

At startup, all available GigE Vision® devices – including *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 application host may also be listed. A tickmark in the discovery tool indicates whether devices are actually reachable via a web browser.

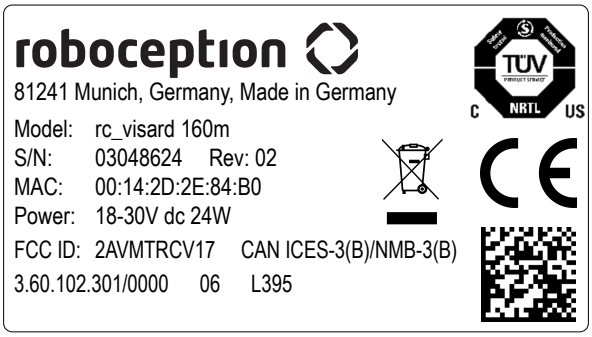


Fig. 4.1: Label on the *rc_visard* indicating model, serial number and MAC address

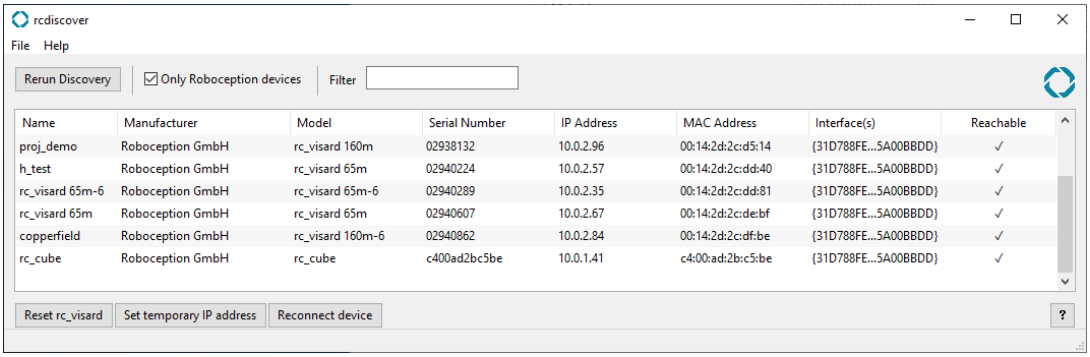


Fig. 4.2: `rcdiscover-gui` tool for finding connected GigE Vision® devices

After successful discovery, a double click on the device row opens the *Web GUI* (Section 4.6) of the device in the operating system's default web browser. Google Chrome or 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.

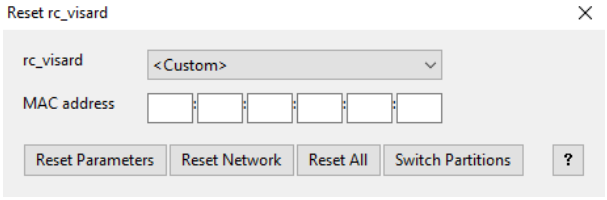


Fig. 4.3: Reset dialog of the `rcdiscover-gui` tool

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 4.6).
- *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 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 either manually via a user-configurable persistent IP address, or automatically via *DHCP*. If none of these IP configuration methods apply, the *rc_visard* falls back to a *Link-Local* IP address.

Following the *GigE Vision*® standard, the priority of IP configuration methods on the *rc_visard* is

1. Persistent IP (if enabled)
2. DHCP (if enabled)
3. Link-Local

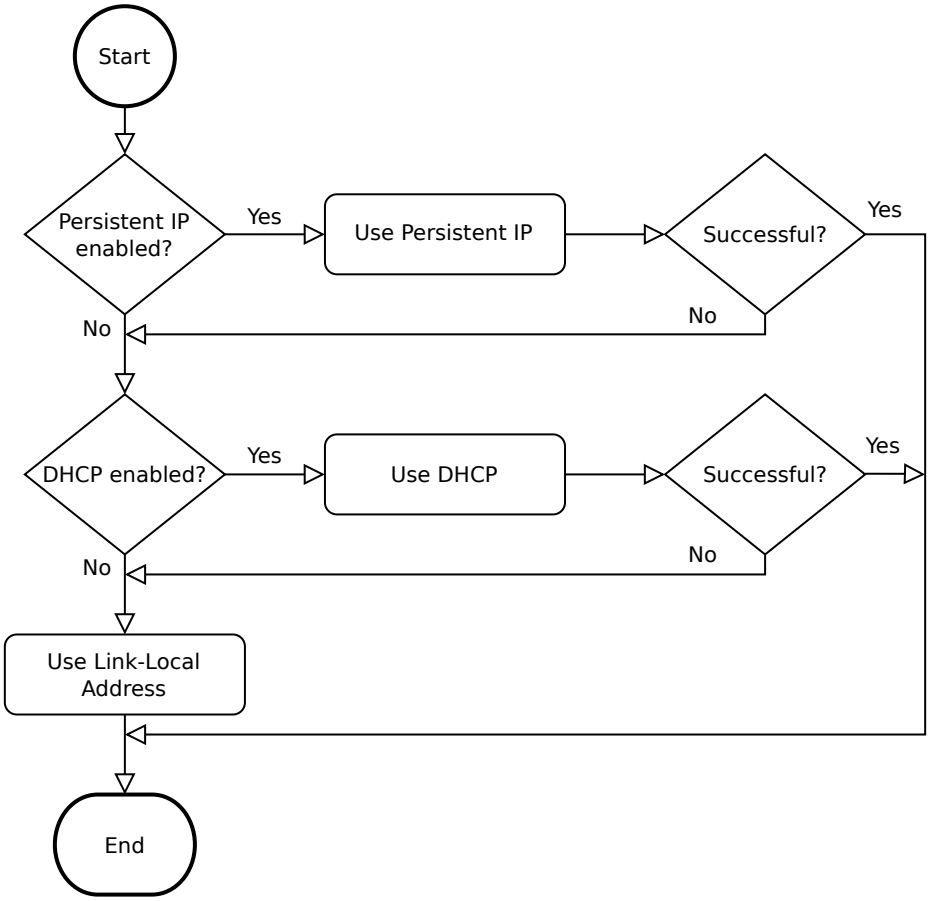


Fig. 4.4: *rc_visard*'s IP configuration method selection flowchart

Options for changing the *rc_visard*'s network settings and IP configuration are:

- the *System tab* of the *rc_visard*'s Web GUI – if it is reachable in the local network already, see *Web GUI* (Section 4.6)
- any configuration tool compatible with *GigE Vision*® 2.0, or Roboception's command-line tool `gc_config`. Typically, these tools scan for all available GigE Vision® devices 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.
- temporarily changing or completely resetting the *rc_visard*'s network configuration via Roboception's `rcdiscover-gui` tool, see *Discovery of rc_visard devices* (Section 4.4)

Note: The command-line tool `gc_config` is part of Roboception's open-source convenience layer `rc_genicam_api`, which can be downloaded free of charge for Windows and Linux from http://www.roboception.com/download.

4.5.1 Host name

The *rc_visard*'s host name is based on its serial number, which is printed on the device, and is defined as `rc-visard-<serial number>`.

4.5.2 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, the device tries to contact a DHCP server at startup and every time the

network cable is being plugged in. If a DHCP server is available on the network, 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 device label, needs to be configured in the DHCP server. At the same time, the *rc_visard*'s host name can also be set in the Domain Name Server (*DNS*). Both MAC address and host name should be sent to the network administrator for configuration.

If the *rc_visard* cannot contact a DHCP server within about 15 seconds after startup, or after plugging in the network cable, it assigns 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.

4.5.3 Manual configuration

Specifying a persistent, i.e. static IP address manually might be useful in some cases. This address is stored on the *rc_visard* to be used on device startup or network reconnection. Please make sure the selected IP address, subnet mask and gateway will not cause any conflicts on the network.

Warning: The IP address must be unique within the local network 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 4.5.2).

If this IP address cannot be assigned, e.g. because it is already used by another device in the network, IP configuration will fall back to automatic configuration via *DHCP* (if enabled) or a *Link-Local* address.

4.6 Web GUI

The *rc_visard*'s Web GUI can be used to test, calibrate, and configure the device. It can be accessed from any web browser, such as Firefox, Google Chrome, or Microsoft Edge, via the *rc_visard*'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://<host-name>` by replacing `<host-name>` with the device's host name.

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://<host-name>.local`.

The Web GUI's overview page gives the most important information about the device and the software components.

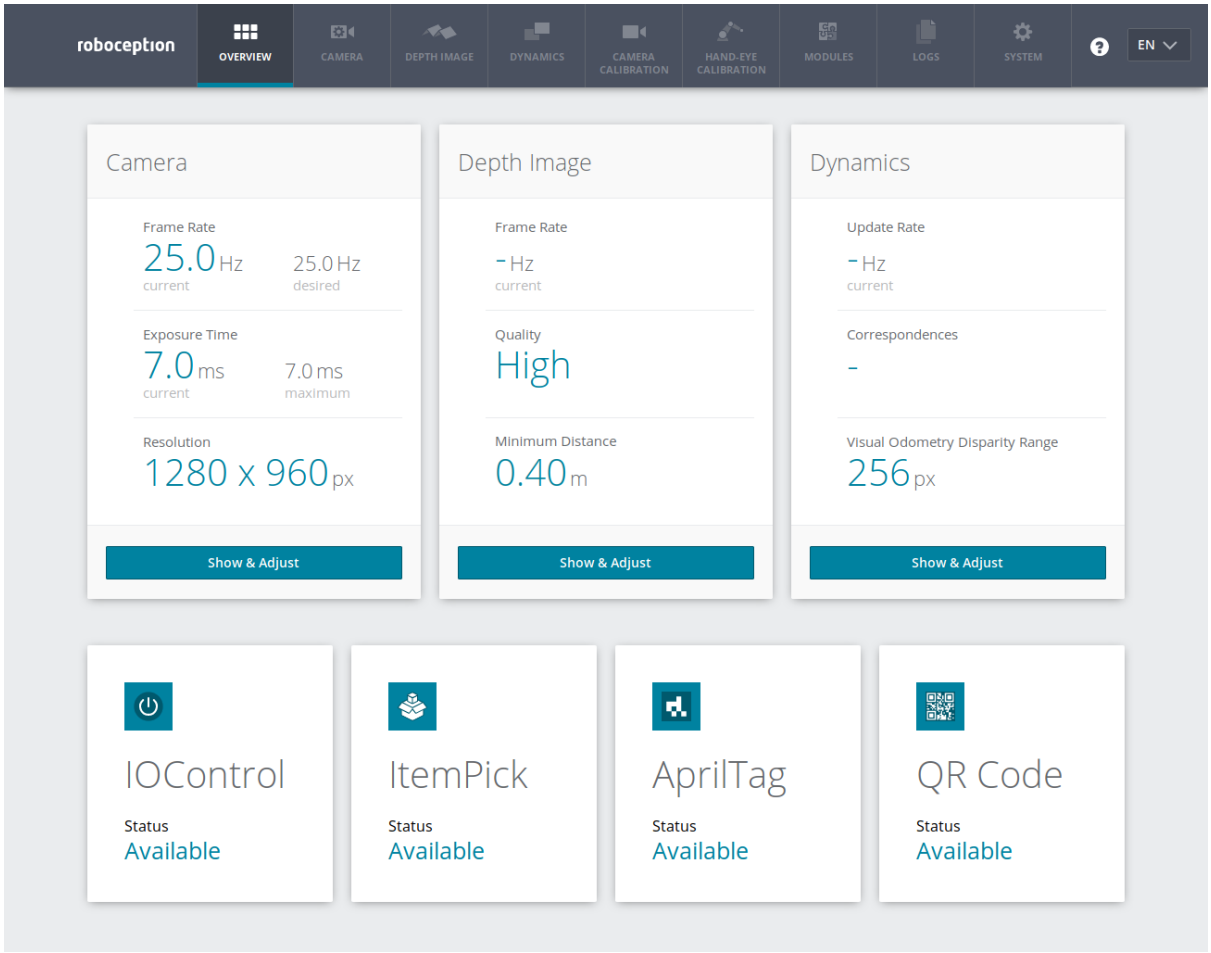


Fig. 4.5: Overview page of the *rc_visard*'s Web GUI

The page's top row permits access to the individual pages of the *rc_visard*'s Web GUI:

Camera shows a live stream of the left and right rectified camera 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 6.1.4) for more information.

Depth Image 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.

Dynamics 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.

Camera Calibration 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.

Hand-Eye-Calibration allows the computation of the static transformation between the camera and a coordinate system known in the robot system. This can be the flange coordinate system of a robotic arm if the camera is attached to the flange. Alternatively, the camera 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.

Modules gives access to the optional software components of the *rc_visard* (see *Optional software components*, Section 7).

Logs permits access to the log files on the *rc_visard*. The log files are typically checked if errors are suspected.

System gives access to general settings and device information, and permits the firmware or the license file to be updated.

Note: 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.

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

5 Measurement principles

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.

In the following, the underlying measurement principles are explained in more detail.

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.

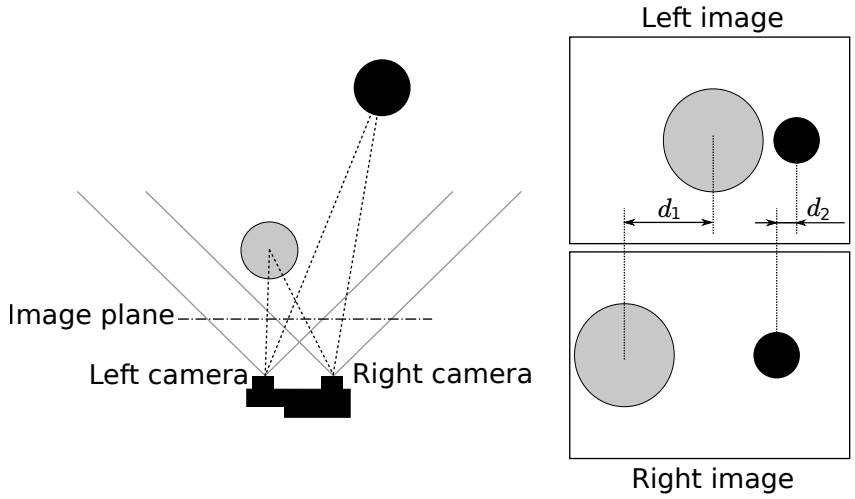


Fig. 5.1: Sketch of the stereo-vision principle: The more distant object (black) exhibits a smaller disparity d_2 than that of the close object (gray), d_1.

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. Thus, the Roboception products utilizing this sensing principle can work indoors and outdoors and multiple 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.

The following 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).

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:

- *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.

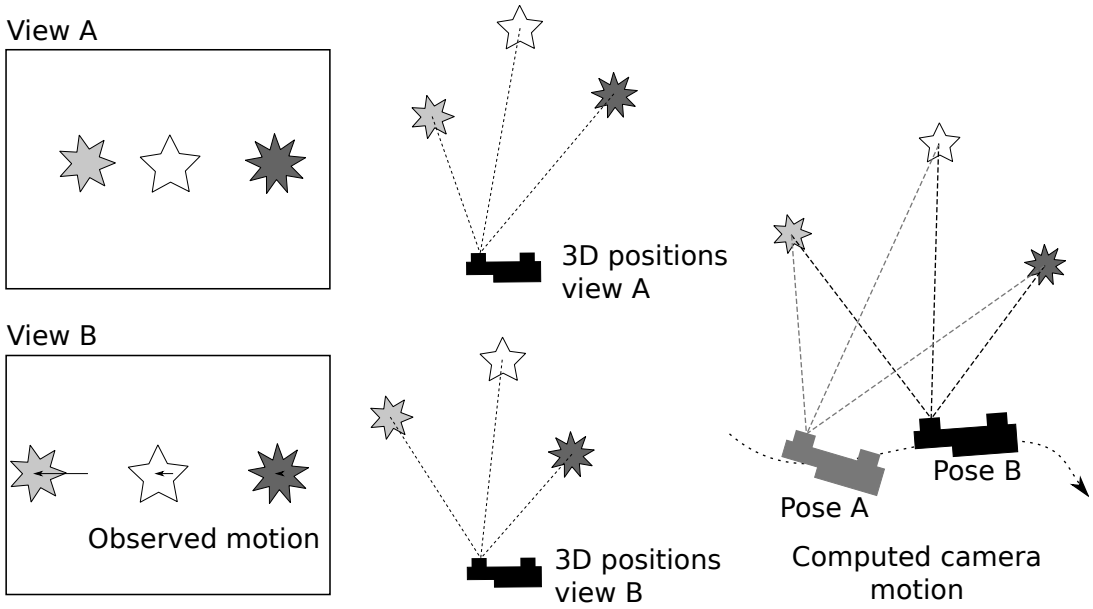


Fig. 5.2: Simplified sketch of the stereo visual odometry principle for 2D motions: Camera motion is computed from the observed motion of characteristic image points.

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).

6 Software components

The *rc_visard* comes with several on-board software components, each of which corresponds to a certain functionality and can be interfaced via its respective *node* in the *REST-API interface* (Section 8.2).

The *Stereo camera* and the *Stereo matching* components, which acquire stereo image pairs and compute 3D depth information such as disparity, error, and confidence images, are also accessible via the *rc_visard*'s *GigE Vision/GenICam interface*.

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).

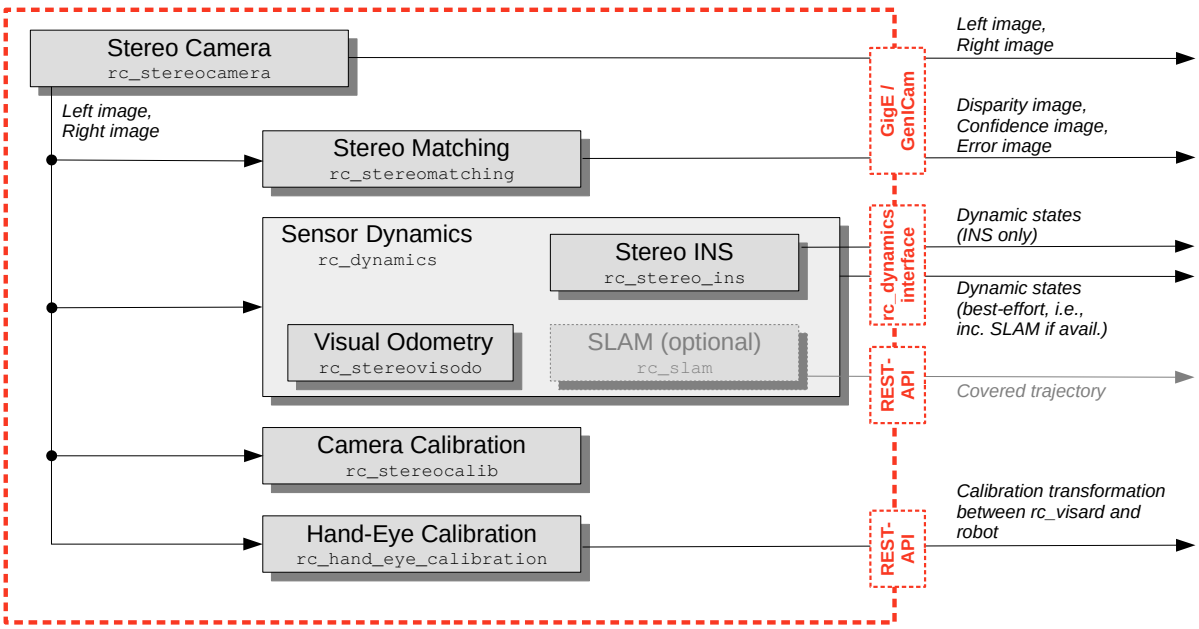


Fig. 6.1: Flowchart of the software components with their node names and the most important outputs

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

The *rc_visard*'s base 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 pairs 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_stereocalib`, 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 4.6).
- ***Hand-eye calibration*** **(`rc_hand_eye_calibration`, Section 6.7)** enables the user to calibrate the camera with respect to a robot, either via the Web GUI or the REST-API.
- ***IO and Projector Control*** **(`rc_iocontrol`, Section 6.8)** provides control over the *rc_visard*'s general purpose inputs and outputs with special modes for controlling an external random dot projector.
- ***TagDetect*** **(`rc_april_tag_detect` and `rc_qr_code_detect`, Section 6.9)** allows the detection of AprilTags and QR codes, as well as the estimation of their poses.

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 exactly the same time. The time at the middle of the image exposure is attached to the images as a timestamp. This timestamp becomes important for dynamic applications in which the cameras or the scene moves.

Note: The period during which the shutter is open is signaled via the *rc_visard*'s *GPIO out 1* (Section 3.5).

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 exposure time is chosen automatically, up to a user defined maximum. The permitted maximum is meant to limit the motion blur that occurs when taking images while the camera 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 *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.

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 Accessing Images

The *rc_visard* provides the time-stamped, rectified left and right images over the GenICam interface (see *Provided image streams*, Section 8.1.6). Live streams of the images are provided with reduced quality in the *Web GUI* (Section 4.6).

The *Web GUI* (Section 4.6) also provides the opportunity to download a snapshot of the current scene as a .tar.gz file by clicking on the camera icon below the image live streams on the *Camera* tab. This snapshot contains:

- the rectified left and right camera images in full resolution as .png files,
- a camera parameter file containing the camera matrix, image dimensions, exposure time, gain value and the stereo baseline,
- the current IMU readings as imu.csv file,
- a nodes.json file containing information about all nodes running on the *rc_visard*,
- a system_info.json file containing system information about the *rc_visard*.

6.1.4 Parameters

The stereo-camera software component is called `rc_stereocamera` and is represented by the *Camera* tab in the *Web GUI* (Section 4.6). 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_auto_average_max	float64	0.0	1.0	0.75	Maximum average intensity if exp_auto is true
exp_auto_average_min	float64	0.0	1.0	0.25	Minimum average intensity if exp_auto is true
exp_auto_mode	string	-	-	Normal	Auto-exposure mode, i.e. Normal or AdaptiveOut1
exp_height	int32	0	959	0	Height of auto exposure region. 0 for whole image.
exp_max	float64	6.6e-05	0.018	0.018	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)

Description of run-time parameters

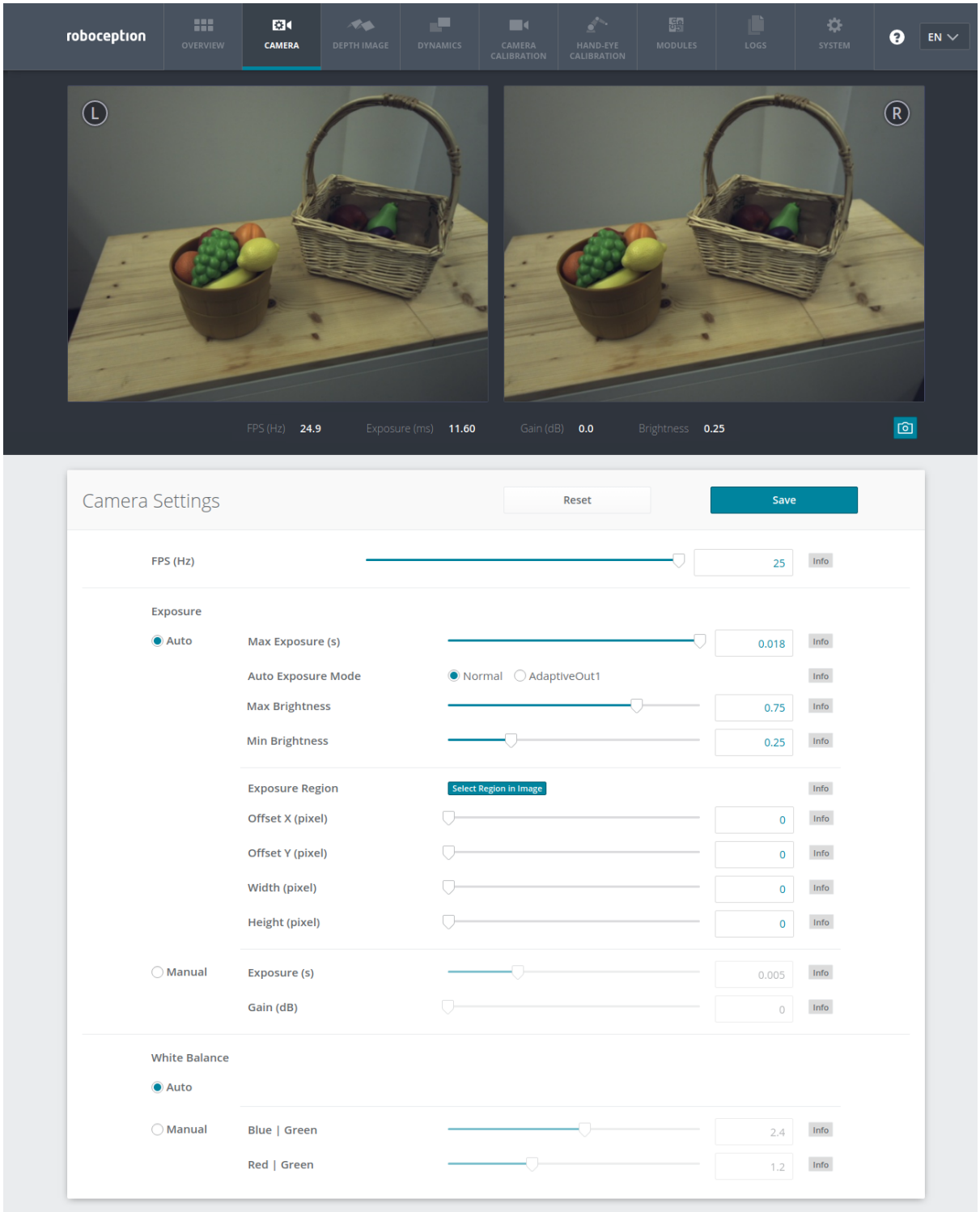


Fig. 6.2: The Web GUI's *Camera* tab

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.

The camera always runs with 25 Hz to ensure proper working of internal modules such as visual odometry

that need a constant frame rate. The user frame rate setting is implemented by excluding frames for stereo matching and transmission via GigE Vision to reduce bandwidth as shown in figure Fig. 6.3.

Fig. 6.3: Images are internally always captured with 25 Hz. The `fps` parameter determines how many of them are sent as camera images via GigE Vision.

`exp_auto` (*Exposure Auto* or *Manual*) This value can be set to *true* for auto-exposure mode, or to *false* 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_auto_mode` (*Auto Exposure Mode*) The auto exposure mode can be set to *Normal* or *AdaptiveOut1*. These modes are relevant when the *rc_visard* is used with an external light source or projector connected to the *rc_visard*'s GPIO Out1, which can be controlled by the optional IOControl component (*IO and Projector Control*, Section 6.8).

Normal: All images are considered for exposure control, except if the IOControl mode for GPIO Out1 is *ExposureAlternateActive*: then only images where GPIO Out1 is HIGH will be considered, since these images may be brighter in case GPIO Out1 is used to trigger an external light source.

AdaptiveOut1: This exposure mode optimizes the exposure time for GPIO Out1 being HIGH. This mode is recommended for using the `acquisition_mode SingleFrameOut1` in the stereo matching component as described in *Stereo Matching Parameters* (Section 6.2.4) and having an external projector connected to GPIO Out1. The exposure difference between images with GPIO Out1 LOW and HIGH is tracked. While the IOControl mode for GPIO Out1 is LOW, the images are under-exposed by this amount to avoid over-exposure for when GPIO Out1 triggers an external projector.

`exp_max` (*Max Exposure*) This value is the maximal exposure time in auto-exposure mode in seconds. 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.

`exp_auto_average_max` (*Max Brightness*) and `exp_auto_average_min` (*Min Brightness*) The auto-exposure tries to set the exposure time and gain factor such that the average intensity (i.e. brightness) in the image or exposure region is between a maximum and a minimum. The maximum brightness will be used if there is no saturation, e.g. no over-exposure due to bright surfaces or reflections. In case of saturation, the exposure time and gain factor are reduced, but only down to the minimum brightness.

The maximum brightness has precendence over the minimum brightness parameter. If the minimum brightness is larger than the maximum brightness, the auto-exposure always tries to make the average intensity equal to the maximum brightness.

The current brightness is always shown in the status bar below the images.

`exp_offset_x`, `exp_offset_y`, `exp_width`, `exp_height` (*Exposure Region*) These values define a rectangular region in the left rectified image for limiting the area used for computing the 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. 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.

The region is visualized in the Web GUI by a rectangle in the left rectified image. It can be defined using the sliders or by selecting it in the image after pressing the button `Select Region in Image`.

`exp_value` (*Exposure*) This value is the exposure time in manual exposure mode in seconds. This exposure time is kept constant even if the images are underexposed.

gain_value (*Gain*) 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.

wb_auto (*White Balance Auto* or *Manual*) This value can be set to *true* for automatic white balancing or *false* 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 and will not be displayed in the Web GUI in this case.

wb_ratio_blue and wb_ratio_red (*Blue | Green* and *Red | Green*) These values are used to set blue to green and red to green ratios for manual white balance. White balancing is without function for monochrome cameras and will not be displayed in the Web GUI in this case.

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).

6.1.5 Status values

This component reports the following status values:

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

Name	Description
`adaptive_out1_reduction`	Fraction of reduction (0.0 - 1.0) of exposure time for images with GPIO Out1=Low in exp_auto_mode=AdaptiveOut1
`baseline`	Stereo baseline t in meters
`brightness`	Current brightness of the image as value between 0 and 1
`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
`test`	0 for live images and 1 for test images
`time`	Processing time for image grabbing in seconds
`width`	Width of the camera image in pixels

6.1.6 Services

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

`save_parameters`

With this service call, the stereo camera component's current parameter settings will be made persistent to the *rc_visard*. That means, these values are applied even after reboot.

This service has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "return_code": {
    "message": "string",
```

(continues on next page)

(continued from previous page)

```
    "value": "int16"
  }
}
```

reset_defaults

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

Warning: By calling this service, the current parameter settings for the camera component are irrecoverably lost.

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

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, an object point is guaranteed to be projected onto the same pixel row in both left and right image. 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 camera. 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 quick run times and 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.

When working with poorly textured objects or in untextured environments, 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. The *rc_visard* provides the IOControl component (see *IO and Projector Control*, Section 6.8) as optional software module which can control a pattern projector connected to the *rc_visard*.

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 camera coordinate frame 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 8.1.4).

Note: The *rc_visard*'s camera coordinate frame is defined as shown in *Coordinate frames* (Section 3.7).

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 principal point that is typically in the center of the image, and p_x, p_y refer to the middle of the pixel, i.e. by adding 0.5 to the integer pixel coordinates. The following figure shows the definition of the image coordinate frame.

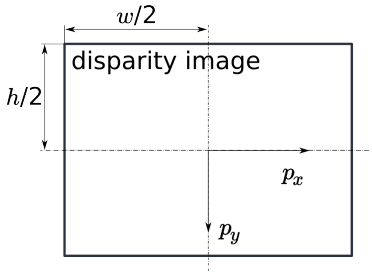


Fig. 6.4: The image coordinate frame's origin is defined to be at the image center – w is the image width and h is the image height.

The same equations, but with the corresponding GenICam parameters are given in *Image stream conversions* (Section 8.1.7).

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, additionally an error image and a confidence image are provided, 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.

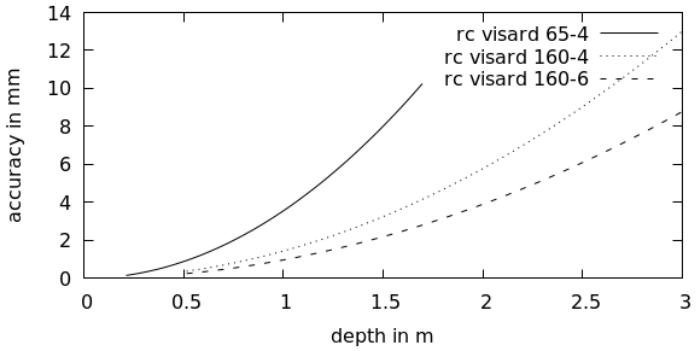
The disparity error d_{eps} (in pixels) can be converted to a depth error z_{eps} (in meters) using the focal length f (in pixels), the baseline t (in meters), and the disparity value d (in pixels) of the same pixel in the disparity image:

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

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 lengths and baselines of the different *rc_visard* models and the typical combined calibration and stereo matching error d_{eps} of 0.25 pixels, the depth accuracy 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 8.1.6). Live streams of the images are provided with reduced quality in the *Web GUI* (Section 4.6).

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 4.6). 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
acquisition_mode	string	-	-	Continuous	SingleFrame, SingleFrameOut1 or Continuous
fill	int32	0	4	3	Disparity tolerance for hole filling in pixels
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
minconf	float64	0.5	1.0	0.5	Minimum confidence
mindepth	float64	0.1	100.0	0.1	Minimum depth in meters
quality	string	-	-	High	Full, High, Medium, or Low. Full requires 'stereo_plus' license.
seg	int32	0	4000	200	Minimum size of valid disparity segments in pixels
smooth	bool	false	true	true	Smoothing of disparity image (requires 'stereo_plus' license)
static_scene	bool	false	true	false	Accumulation of images in static scenes to reduce noise

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:

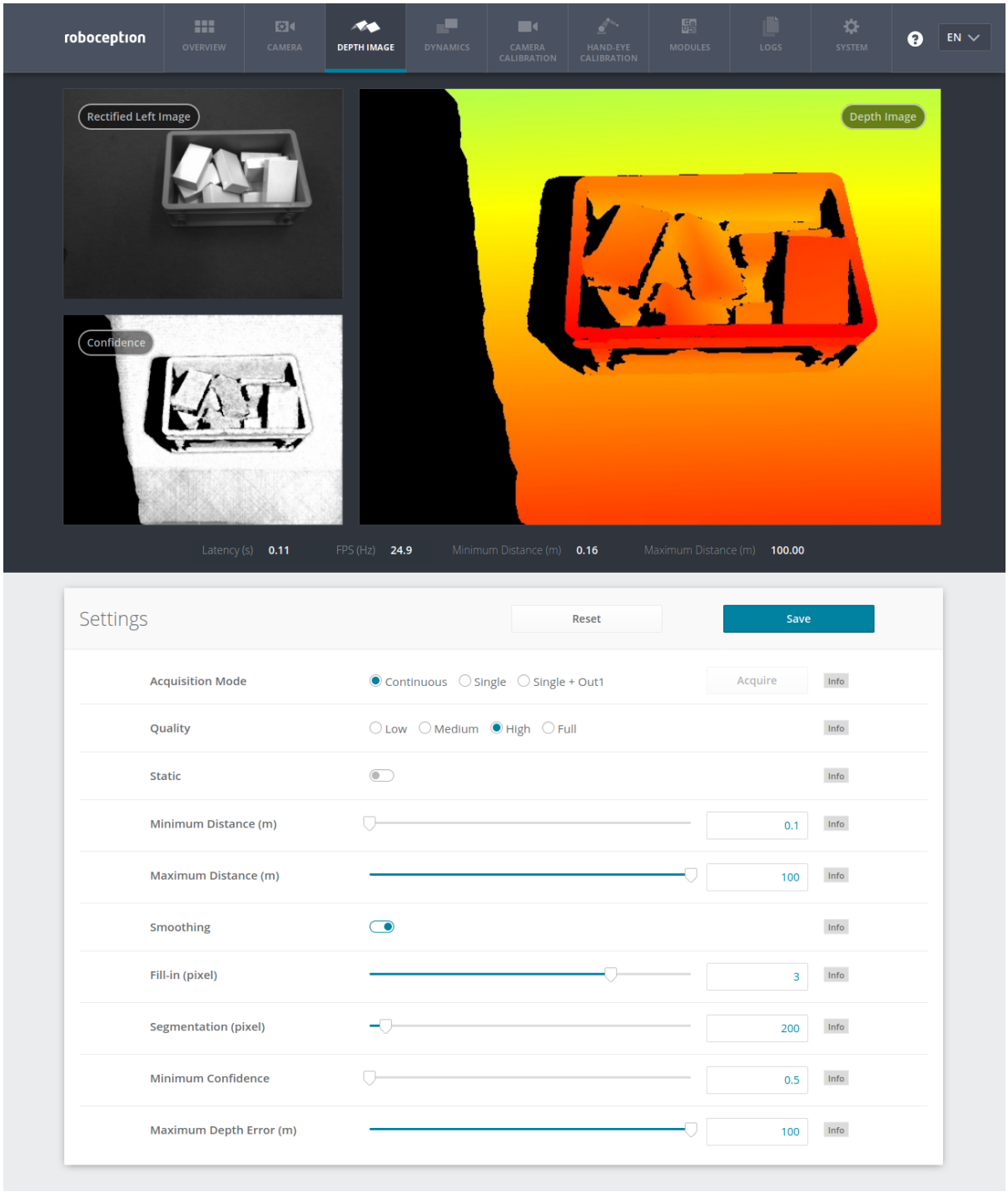


Fig. 6.5: The Web GUI's *Depth Image* tab

`acquisition_mode` (*Acquisition Mode*) The acquisition mode can be set to `Continuous`, `SingleFrame` (*Single*) or `SingleFrameOut1` (*Single + Out1*). The first one is the default, which performs stereo matching continuously according to the user defined frame rate and the available computation resources. The two other modes perform stereo matching upon each click of the *Acquire* button. The *Single + Out1* mode additionally controls an external projector that is connected to GPIO Out1 (*IO and Projector Control*, Section 6.8). In this mode, `out1_mode` of the IOControl component is automatically set to `ExposureAlternateActive` upon each trigger call and reset to `Low` after receiving images for stereo matching.

> **Note:** The `Single + Out1` mode can only change the `out1_mode` if the IOControl license is available on the *rc_visard*.

quality (***Quality***) Disparity images can be computed in 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. 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.

Additionally, full resolution matching (`Full`) with 1280 x 960 pixel is possible with a valid StereoPlus license.

If full resolution is selected, the depth range is internally limited due to limited on-board memory resources. It is recommended to adjust `mindepth` and `maxdepth` to the depth range that is required by the application.

static_scene (***Static***) This option averages 8 consecutive camera images before matching. This reduces noise, which improves the stereo matching result. The timestamp of the first image is taken as timestamp of the disparity image. This option only affects matching in full or high quality. It must only be enabled if the scene does not change during the acquisition of the 8 images.

mindepth (***Minimum Distance***) The minimum distance is the smallest distance from the camera 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.

Depending on the capabilities of the sensor, the actual minimum distance can be higher than the user setting. The actual minimum distance will be reported in the status values.

In `quality` mode `Full`, the actual minimum distance can also be higher than the user-defined minimum distance due to memory limitations. In this case, lowering the maximum distance helps to reduce the actual minimum distance.

maxdepth (***Maximum Distance***) The maximum distance is the largest distance from the camera 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.

In `quality` mode `Full`, the actual minimum distance can be higher than the user-defined minimum distance due to memory limitations. In this case, lowering the maximum distance helps to reduce the actual minimum distance.

smooth (***Smoothing***) This option activates advanced smoothing of disparity values. It is only available with a valid StereoPlus license.

fill (***Fill-in***) This option is used to fill holes in the disparity image by interpolation. The fill-in value is the maximum allowed disparity step on the border of the hole. Larger fill-in values can decrease the number of holes, but the interpolated values can have larger errors. At most 5% of pixels are interpolated. Interpolation of small holes is prefered over interpolation of larger holes. The confidence for the interpolated pixels is set to a low value of 0.5. A fill-in value of 0 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. The value is related to the high quality disparity image with 640 x 480 pixels resolution and does not have to be scaled when a different quality is chosen. Segmentation is useful for removing erroneous disparities. However, larger values may also remove real objects.

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 camera (see *Confidence and error images*, Section 6.2.3).

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 Status values

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)*.
`latency`	Time in seconds between image aquisition and publishing of disparity image
`mindepth`	Actual minimum working distance in meters
`maxdepth`	Actual maximum working distance in meters
`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`.

6.2.6 Services

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

`acquisition_trigger`

This call signals the module to perform stereo matching of the next available images, if the parameter `acquisition_mode` is set to `SingleFrame` or `SingleFrameOut1`. An error is returned if the `acquisition_mode` is set to `Continuous`.

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

Possible return codes are shown below.

Table 6.5: Possible return codes of the `acquisition_trigger` service call.

Code	Description
0	Success
-8	Triggering is only possible in SingleFrame acquisition mode
101	Trigger is ignored, because there is a trigger call pending
102	Trigger is ignored, because there are no subscribers

`save_parameters`

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

reset_defaults

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

Warning: By calling this service, the current parameter settings for the stereo matching component are irrecoverably lost.

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

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).

Note: Using *Stereo matching* (Section 6.2) in parallel to the dynamics component may lead to decreased localization accuracy. See *Visual odometry* (Section 6.4) for how to avoid this.

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.

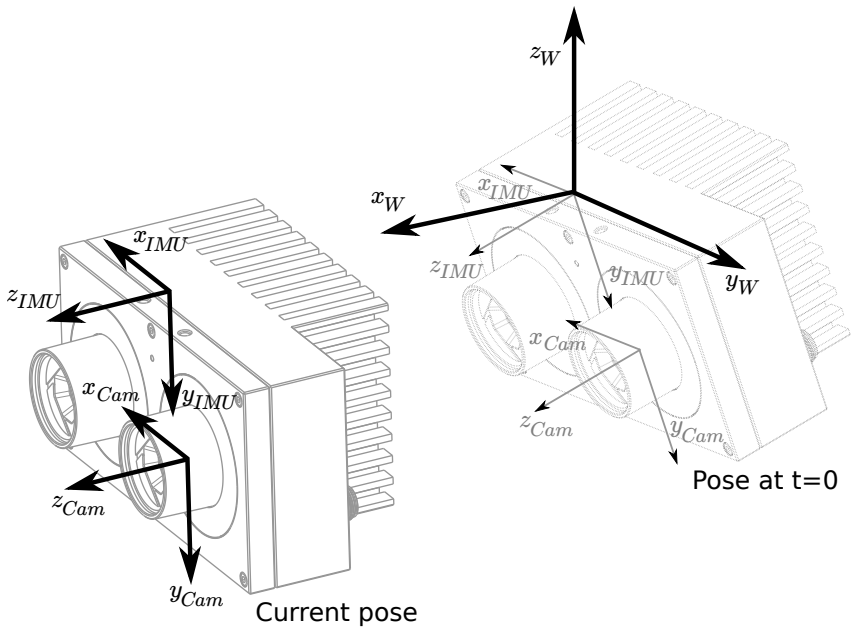


Fig. 6.6: Coordinate frames for state estimation. The IMU coordinate frame is inside the *rc_visard*'s housing. The *camera coordinate frame* (Section 3.7) is in the focal point of the left camera.

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 *SLAM* and *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 *SLAM* and *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 *SLAM* and *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 Status values

This component reports the following status values:

Table 6.6: The `rc_dynamics` component's status values

Name	Description
`state`	The current state of the rc_dynamics node

6.3.4 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 6.7.

Table 6.7: 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
STOPPING	Transitional state when going to (or through IDLE)
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 has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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 has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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` through `STOPPING` to `IDLE`.

This service has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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 has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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 `STOPPING`, `IDLE` and `WAITING_FOR_INS` to `RUNNING`.

This service has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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 `STOPPING`, `IDLE`, `WAITING_FOR_INS_AND_SLAM`, `WAITING_FOR_SLAM` to `RUNNING_WITH_SLAM`.

This service has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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.

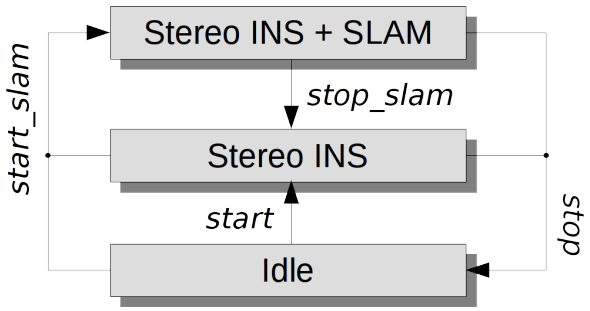


Fig. 6.7: Simplified state and transition diagram

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 6.7 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 has no arguments.

The definition for the response with corresponding datatypes is:


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


(continues on next page)

(continued from previous page)

```
  }
}
```

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.

Note: Using *Stereo matching* in parallel to the dynamics component may lead to a decreased frame rate of the visual odometry. In this case, we recommend to decrease the frame rate of the *Stereo camera* (effectively decreasing the frame rate of the depth image computation), to lower the computational load of stereo matching.

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 frequency 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 4.6). 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 6.8: 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

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* tab.

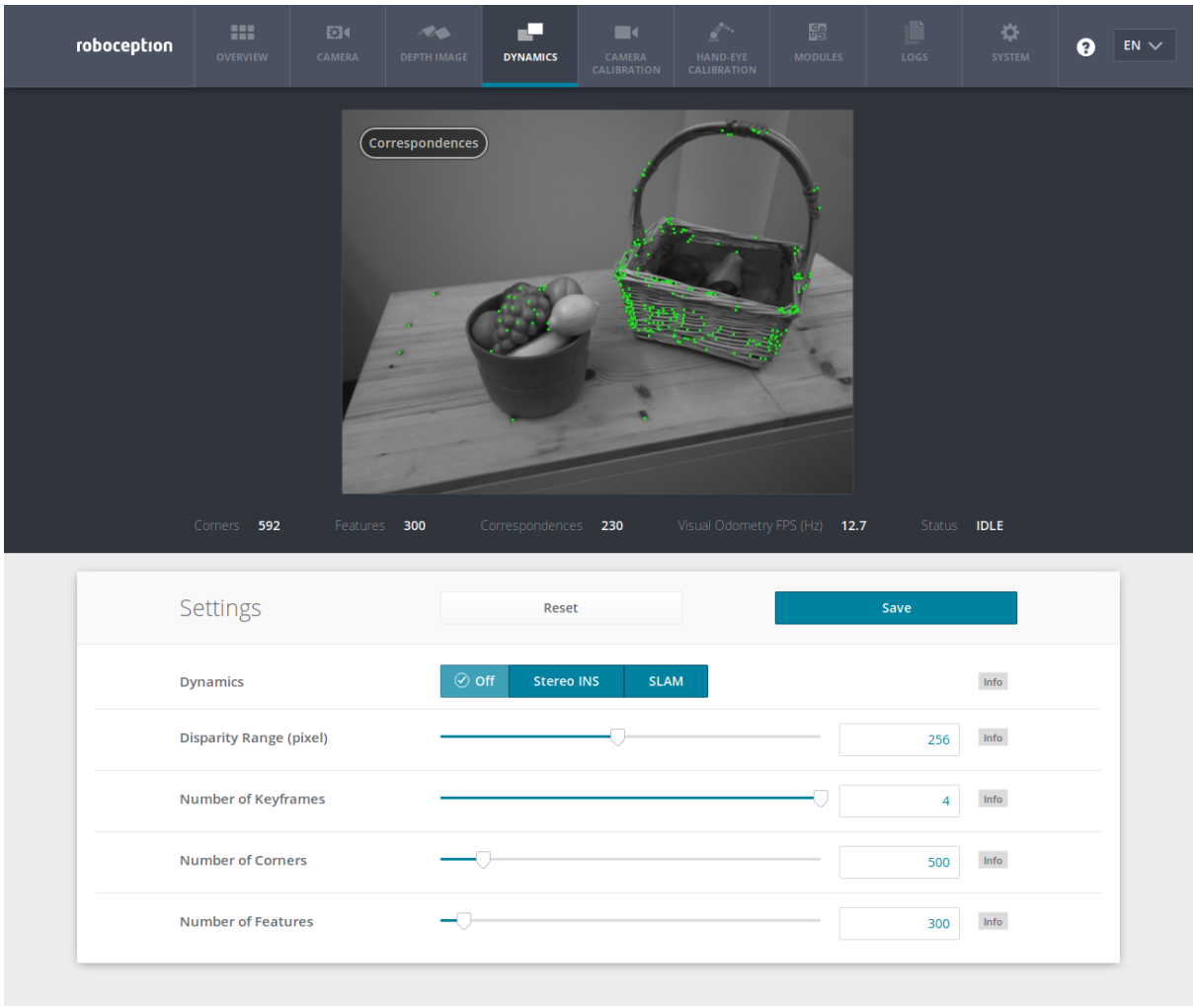


Fig. 6.8: The Web GUI's *Dynamics* tab

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* tab. 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 6.3.4).

`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* tab.

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 Status values

This component reports the following status values:

Table 6.9: 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

6.4.3 Services

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

The visual odometry component offers the following services for persisting and restoring parameter settings.

`save_parameters`

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 has no arguments.

The definition for the response with corresponding datatypes is:

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

(continues on next page)

(continued from previous page)

```
  }
}
```

reset_defaults

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

Warning: By calling this service, the current parameter settings for the visual odometry component are irrecoverably lost.

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

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.

6.5.3 Status values

This component reports the following status values:

Table 6.10: 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 re-calibration 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.

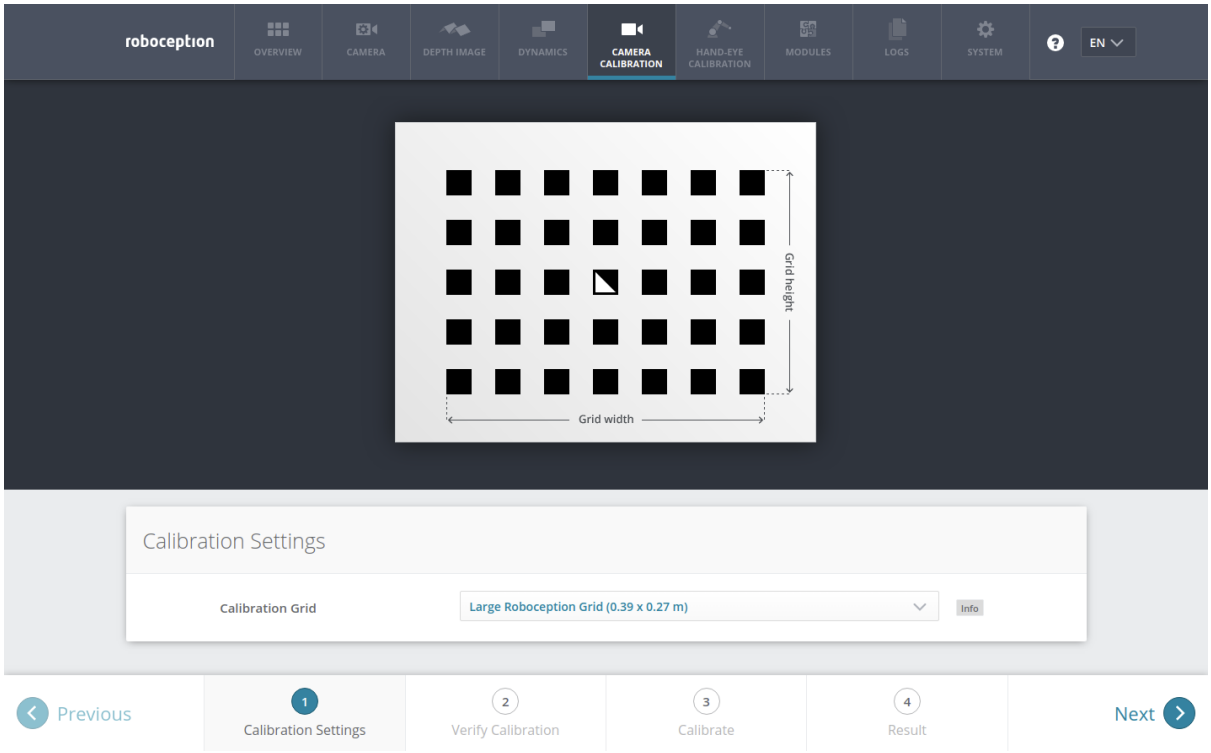


Fig. 6.9: Calibration settings

The *Camera calibration* component has to be selected in the *Web GUI* (Section 4.6) to verify or perform camera calibration. In the first step, the calibration grid must be specified. 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.

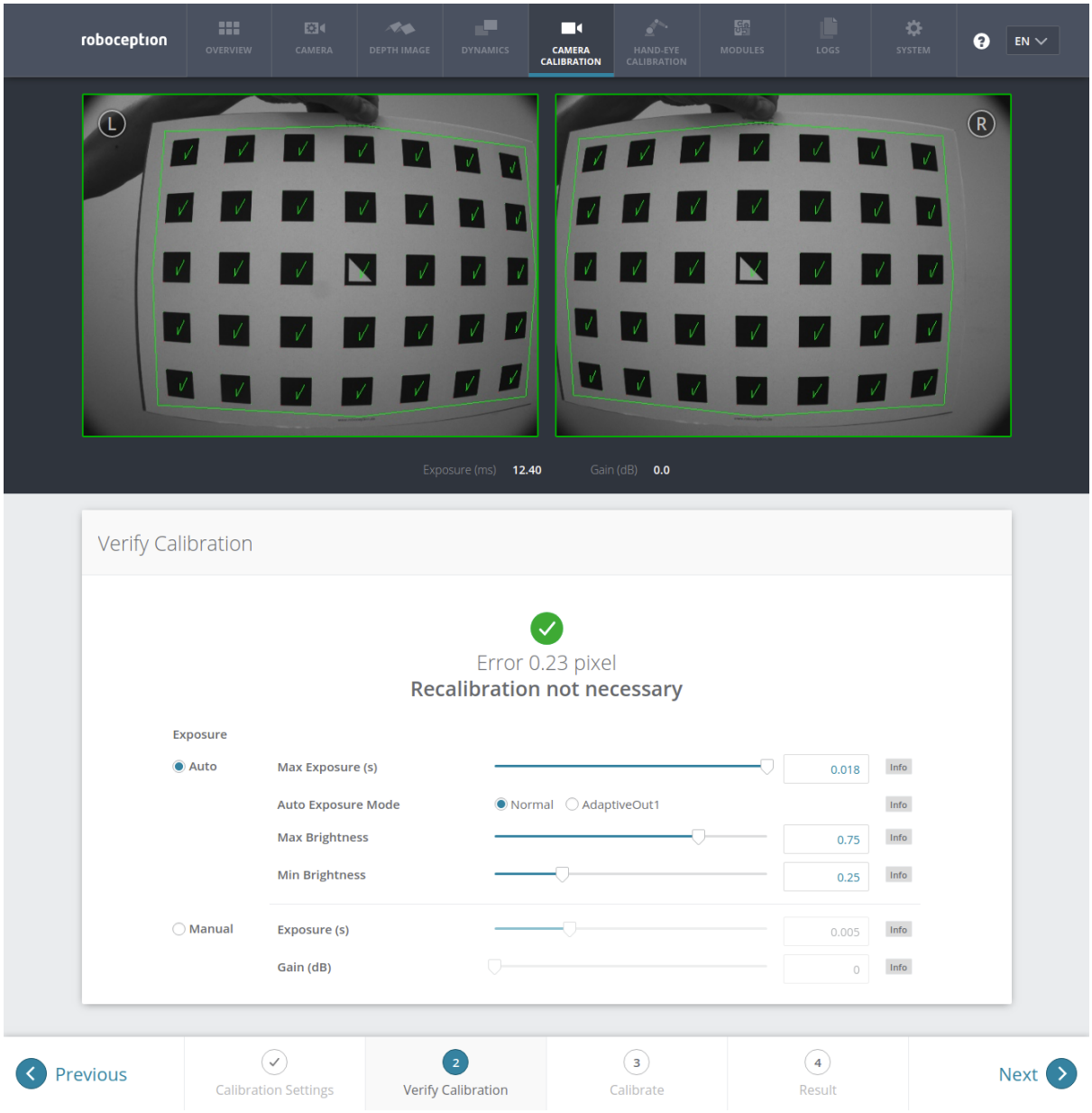


Fig. 6.10: Verification of calibration

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

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 below 0.2 pixels. If the error is in this range, 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. In case you use a custom calibration grid, 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. 6.11.

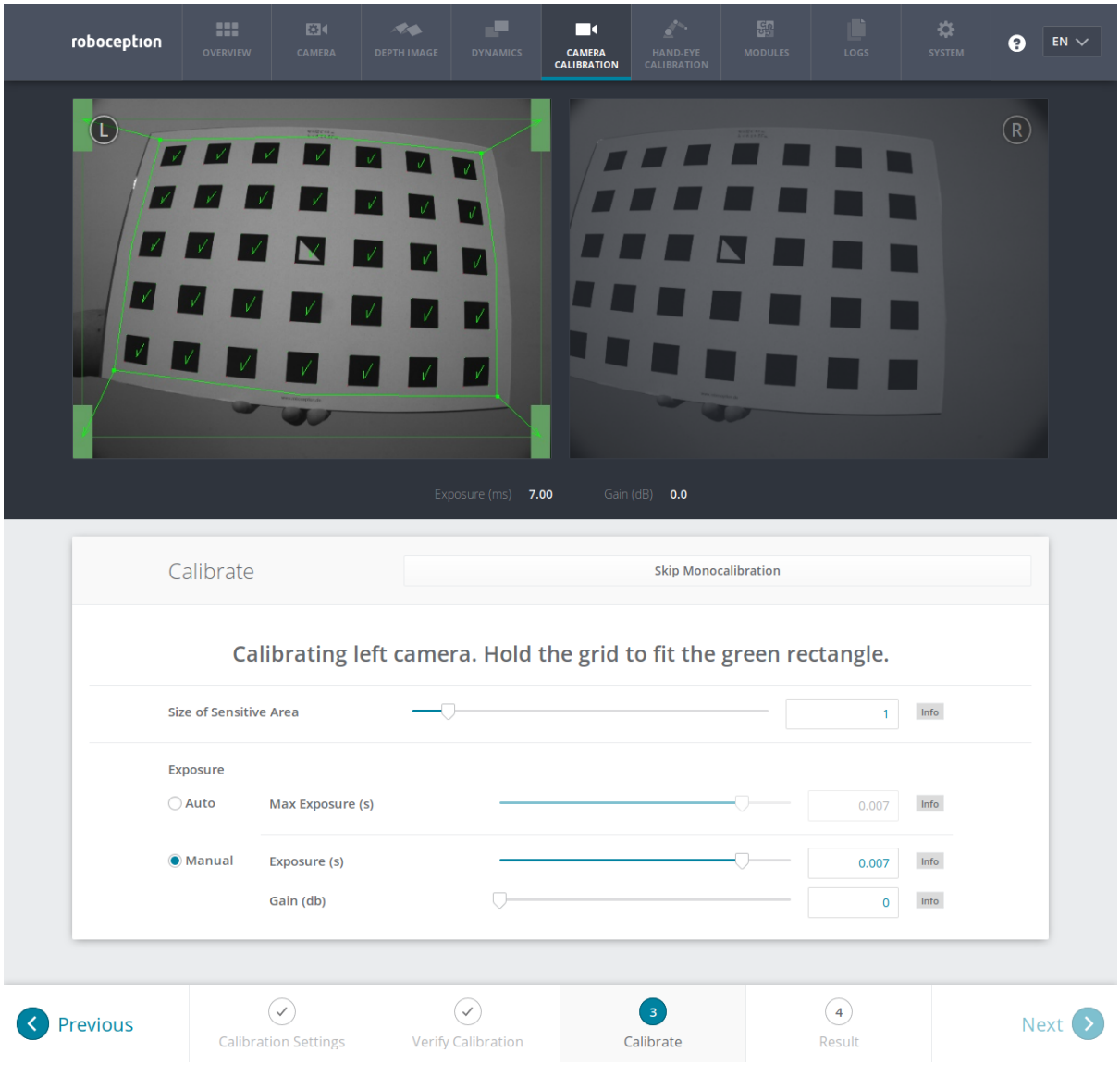


Fig. 6.11: Starting the calibration procedure

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. 6.11. However, please be aware that increasing their size too much may result in slightly lower 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. 6.12.

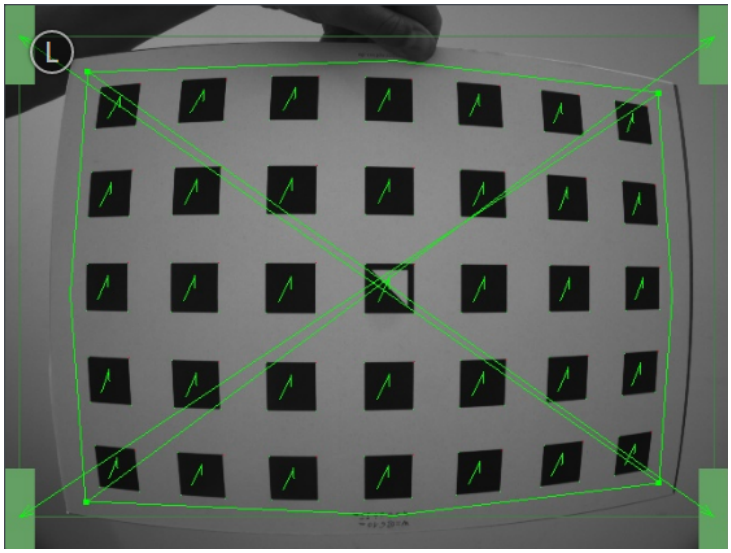


Fig. 6.12: Wrongly holding the grid upside down leads to crossed green lines.

Note: Calibration might appear cumbersome as it involves holding the grid in certain predefined poses. However, these poses are required to ensure an unbiased, high-quality calibration result.

Monocalibration

Full calibration consists of calibrating each camera individually (monocalibration) 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 the next Section *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. 6.13.

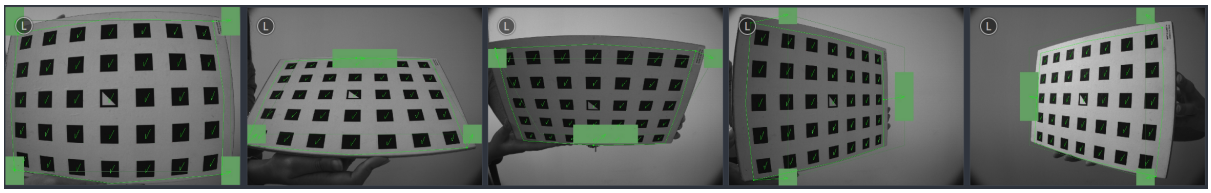


Fig. 6.13: Poses required for monocamera calibration

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 that the algorithm accepted the pose.

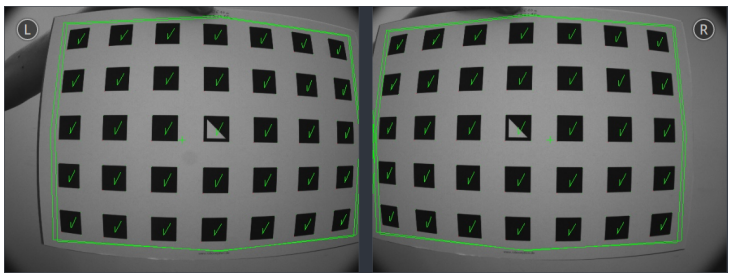


Fig. 6.14: Holding the grid closer than 40 cm during stereo calibration

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 that the algorithm accepted the pose.

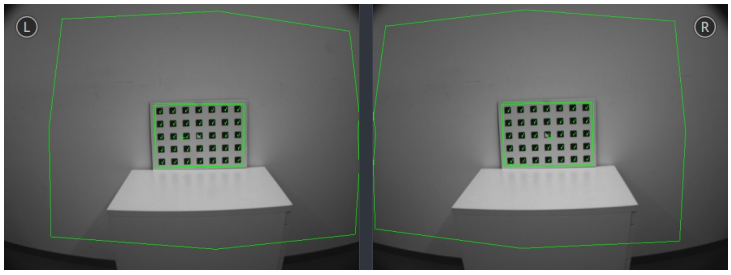


Fig. 6.15: Holding the grid farther away than 1 m during stereo calibration

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 indicated result is the mean reprojection error of all calibration points. It is given in pixels and typically has a value below 0.2.

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

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.

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 camera 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 camera 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 which 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 *rc_visard*'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 4.6) 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 or handheld device.

6.7.2 Camera mounting

As illustrated in Fig. 6.16 and Fig. 6.18, two different use cases w.r.t. to the mounting of the camera generally have to be considered:

a. The camera is **mounted on the robot**, i.e., it is mechanically fixed to a robot link (e.g., at its flange or a flange-mounted tool), and hence moves with the robot.

b. The camera 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 camera When calibrating a robot-mounted camera 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. 6.16.

> **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 from the *camera* frame into the user-selected *robot* frame such that

$$\mathbf{p}_{\text{robot}} = \mathbf{R}_{\text{camera}}^{\text{robot}} \cdot \mathbf{p}_{\text{camera}} + \mathbf{t}_{\text{camera}}^{\text{robot}} , \tag{6.3}$$

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 $\mathbf{R}_{\text{camera}}^{\text{robot}}$ as well as $\mathbf{t}_{\text{camera}}^{\text{robot}}$ are the corresponding 3×3 rotation matrix and 3×1 translation vector of the pose $\mathbf{T}_{\text{camera}}^{\text{robot}}$, respectively. In practise, in the calibration result and in the provided robot poses, the rotation is defined by Euler angles or as quaternion instead of a rotation matrix (see *Pose formats*, Section 13.1).

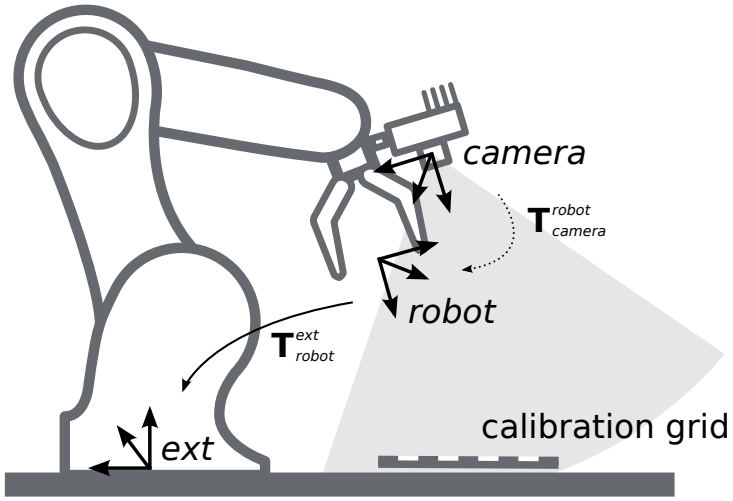


Fig. 6.16: Important frames and transformations for calibrating a camera that is mounted on a general robot. The camera is mounted with a fixed relative position to a user-defined *robot* frame (e.g., flange or TCP). It is important that the pose $\mathbf{T}_{\text{robot}}^{\text{ext}}$ of this *robot* 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.

Additional user input is required if the movement of the robot is constrained and the robot can rotate the Tool Center Point (TCP) only around one axis. This is typically the case for robots with four Degrees Of Freedom (4DOF) that are often used for palletizing tasks. In this case, the user must specify which axis of the *robot* frame is the rotation axis of the TCP. Further, the signed offset from the TCP to the *camera* coordinate system along the TCP rotation axis has to be provided. Fig. 6.17 illustrates the situation.

For *rc_visard*, the camera coordinate system is located in the optical center of the left camera. The approximate location is given in section *Coordinate frames* (Section 3.7).

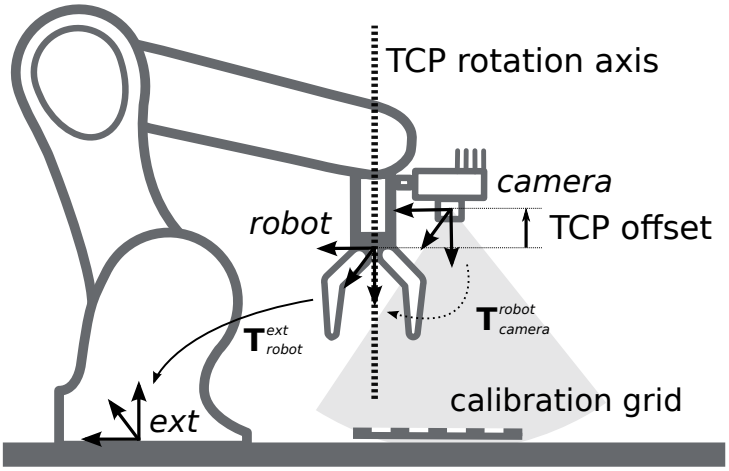


Fig. 6.17: In case of a 4DOF robot, the TCP rotation axis and the offset from the TCP to the camera coordinate system along the TCP rotation axis must be provided. In the illustrated case, this offset is negative.

Calibration with a statically-mounted camera In use cases where the camera is positioned statically w.r.t. the robot, the calibration grid needs to be mounted to the robot as shown for example in Fig. 6.18 and Fig. 6.19.

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 means, 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. 6.19.

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).

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 *camera* frame and the user-selected external reference frame *ext* such that

$$\mathbf{p}_{\text{ext}} = \mathbf{R}^{\text{ext}}_{\text{camera}} \cdot \mathbf{p}_{\text{camera}} + \mathbf{t}^{\text{ext}}_{\text{camera}} \,, \tag{6.4}$$

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 $\mathbf{R}^{\text{ext}}_{\text{camera}}$ as well as $\mathbf{t}^{\text{ext}}_{\text{camera}}$ are the corresponding 3×3 rotation matrix and 3×1 translation vector of the pose $\mathbf{T}^{\text{ext}}_{\text{camera}}$, respectively. In practise, in the calibration result and in the provided robot poses, the rotation is defined by Euler angles or as quaternion instead of a rotation matrix (see *Pose formats*, Section 13.1).

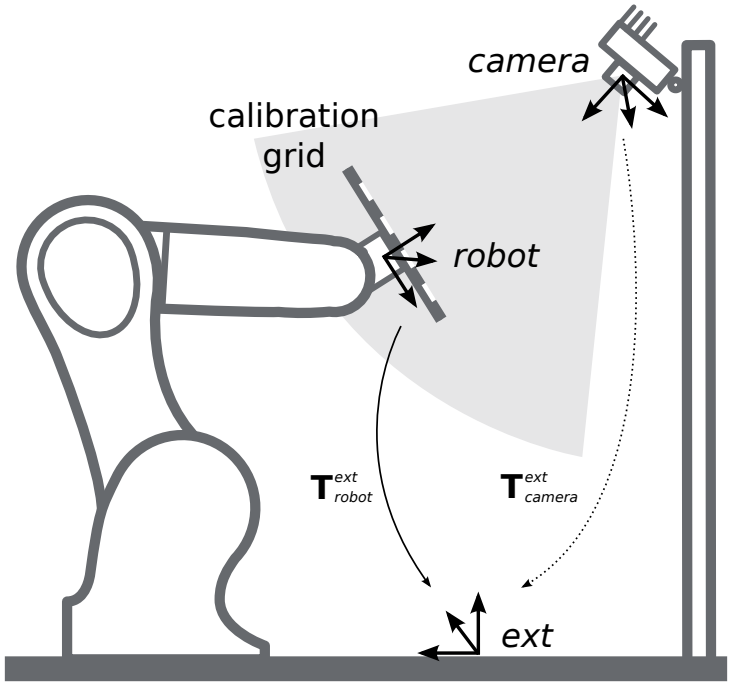


Fig. 6.18: Important frames and transformations for calibrating a statically mounted camera: The latter is mounted with a fixed position relative to a user-defined external reference frame *ext* (e.g., the world coordinate frame or the robot's mounting point). It is important that the pose $\mathbf{T}^{\text{ext}}_{\text{robot}}$ 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*.

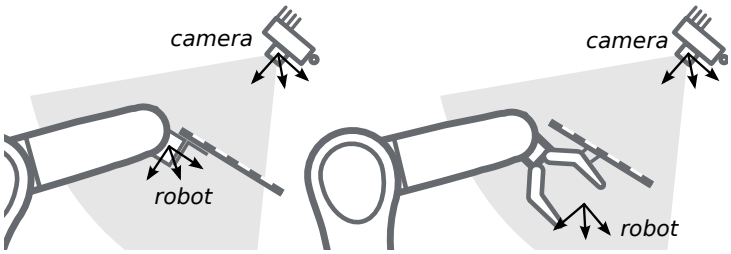


Fig. 6.19: Alternate mounting options for attaching the calibration grid to the robot

Additional user input is required if the movement of the robot is constrained and the robot can rotate the Tool Center Point (TCP) only around one axis. This is typically the case for robots with four Degrees Of Freedom (4DOF) that are often used for palletizing tasks. In this case, the user must specify which axis of the *robot* frame is the rotation axis of the TCP. Further, the signed offset from the TCP to the visible surface of the calibration grid along the TCP rotation axis has to be provided. The grid must be mounted such that the TCP rotation axis is orthogonal to the grid. Fig. 6.20 illustrates the situation.

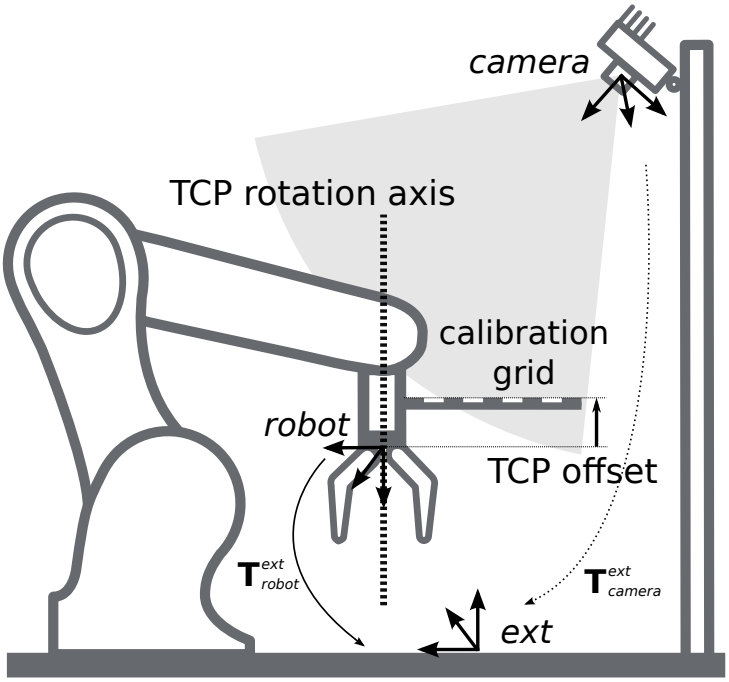


Fig. 6.20: In case of a 4DOF robot, the TCP rotation axis and the offset from the TCP to the visible surface of the grid along the TCP rotation axis must be provided. In the illustrated case, this offset is negative.

6.7.3 Calibration routine

The general hand-eye calibration routine consists of three steps, which are illustrated in Fig. 6.21. These three steps are also represented in the *Web GUI*'s guided hand-eye-calibration process (Section 4.6).

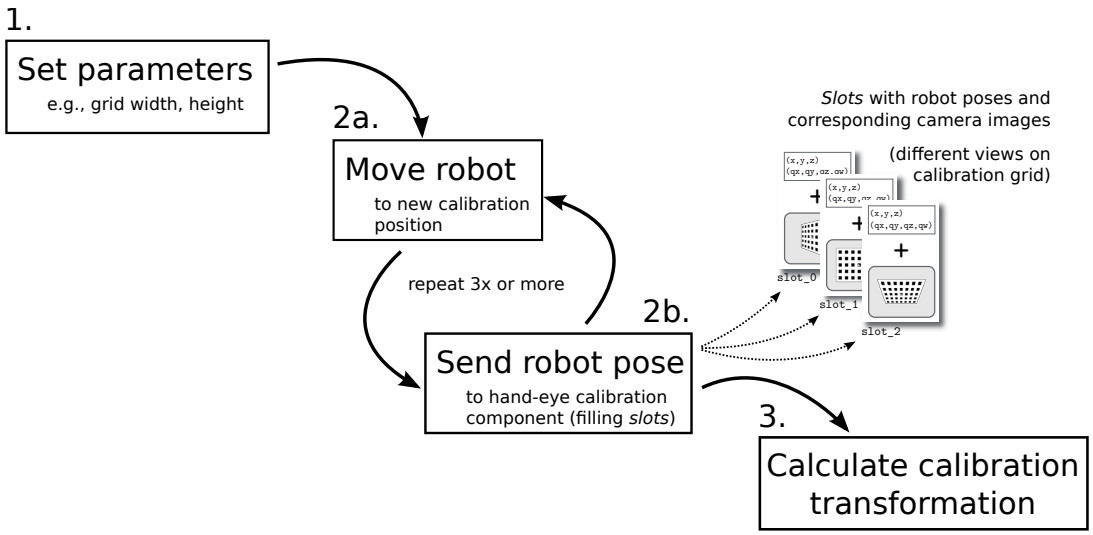


Fig. 6.21: Illustration of the three different steps involved in the hand-eye calibration routine

Step 1: Setting parameters

Before starting the actual calibration routine, the grid and 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. 6.22. In addition to grid size and camera mounting, the Web GUI also

offers settings for calibrating 4DOF robots. In this case, the rotation axis, as well as the offset from the TCP to the camera coordinate system (robot-mounted camera) or grid surface (statically mounted camera) must be given. Finally, the pose format can be chosen, which is used for setting poses in the upcoming step 2 of the calibration process. It can be set to either *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.

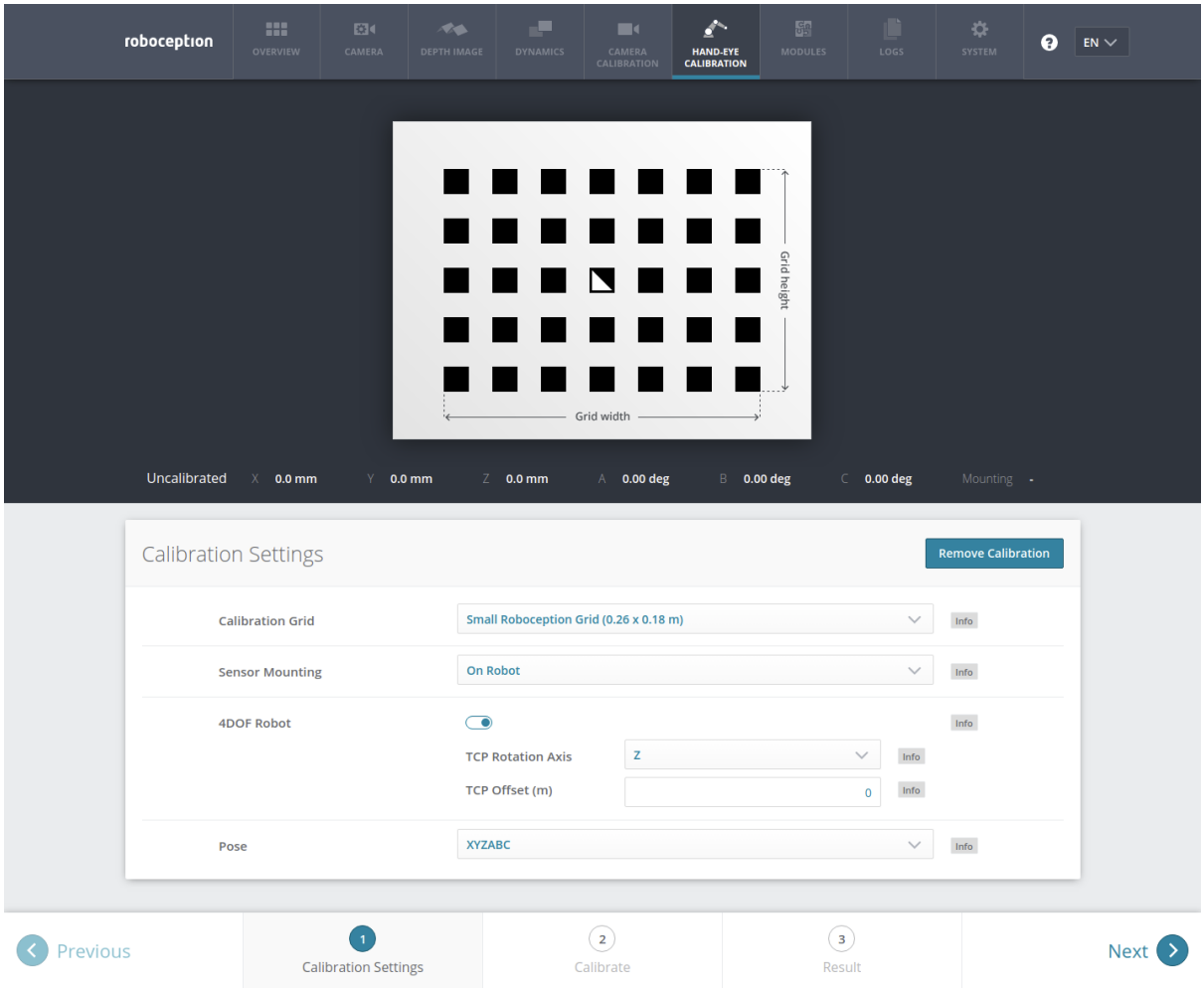


Fig. 6.22: Defining hand-eye calibration settings via the *rc_visard*'s Web GUI

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 the left camera image. Furthermore, the robot positions need to be selected properly to achieve a variety of different perspectives for the camera to perceive the calibration grid. Fig. 6.23 shows a schematic recommendation of four different view points.

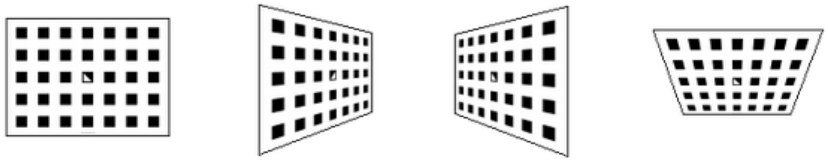


Fig. 6.23: Recommended views on the calibration grid during the calibration procedure. In case of a 4DOF robot, other views have to be chosen, which should be as different as possible.

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 $\mathbf{T}_{\text{robot}}^{\text{ext}}$ 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 corresponding left camera images. All filled slots will then be used to calculate the desired calibration transformation between the *camera* frame and either the user-defined *robot* frame (robot-mounted camera) or the user-defined external reference frame *ext* (static camera).

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.

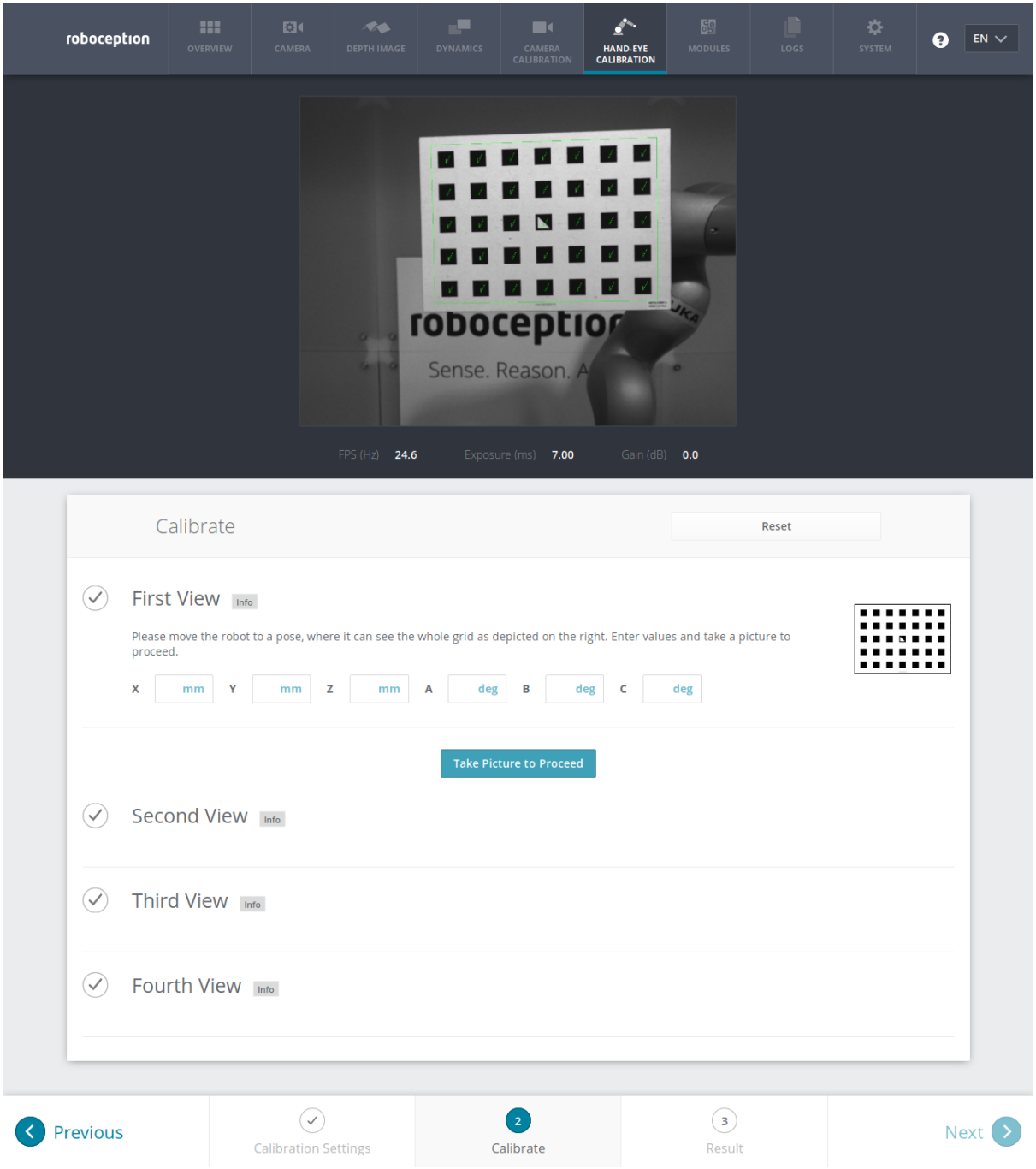


Fig. 6.24: First sample image in the hand-eye calibration process for a statically mounted camera

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. 6.23, in this example the following corresponding camera images have been sent to the hand-eye calibration component with their associated robot pose:

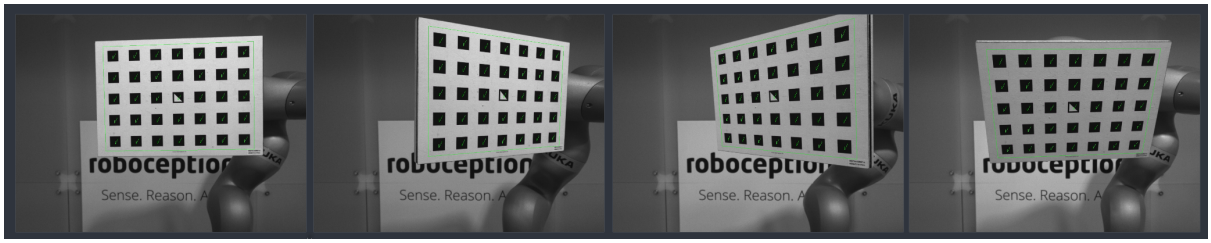


Fig. 6.25: Recorded camera images as input for the calibration procedure

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 camera 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 camera) or the user-defined external reference frame *ext* (statically mounted camera); see *Camera mounting* (Section 6.7.2).

To enable users to judge the quality of the resulting calibration transformation, the component also reports a calibration error E_{camera}. 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 value can be translated into an error E at a certain distance Z:

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

where f is the focal length in pixels.

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.

Web GUI example: The Web GUI automatically triggers computation of the calibration result after taking the last of the four pictures. The user just needs to click the *Next* button to proceed to the result. The user has the possibility to specify or correct settings related to calibration of 4DOF robots if required. After changing any settings, the recompute button needs to be pressed.

In the example that is shown in Fig. 6.26, 4DOF is turned off and the camera is mounted statically. The resulting output is the pose of the left camera in the external coordinate system of the robot. The reported error is $E_{\text{camera}} = 0.29$ pixels, which corresponds to an error of approximately 0.27 mm in a distance of 1 m.

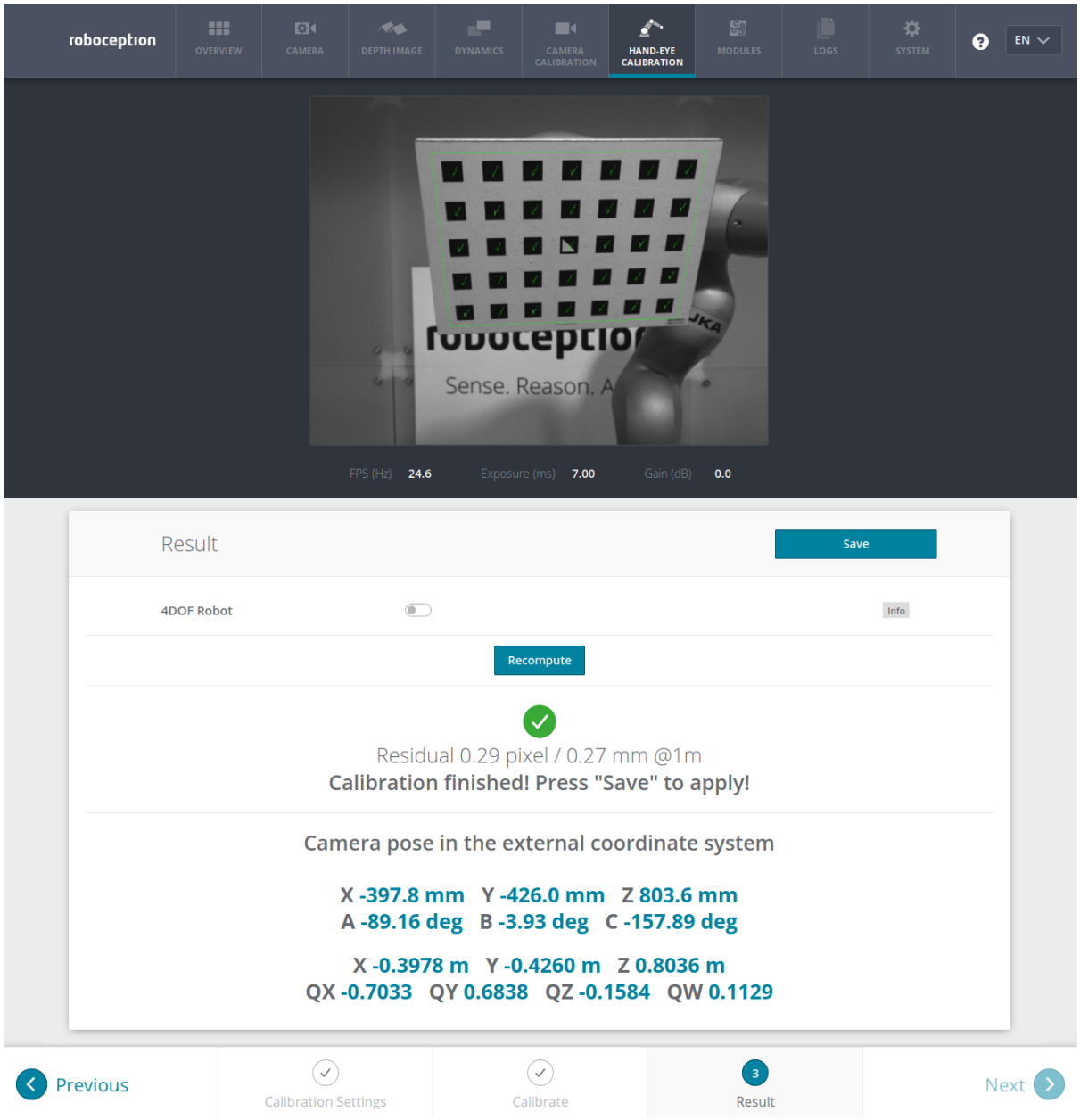


Fig. 6.26: Result of the hand-eye calibration process displayed in the Web GUI

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 4.6). 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 6.11: 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
`tcp_offset`	float64	-10.0	10.0	0.0	Offset from TCP along tcp_rotation_axis
`tcp_rotation_axis`	int32	-1	2	-1	-1 for off, 0 for x, 1 for y, 2 for z

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 given with a very high accuracy, preferably with sub-millimeter accuracy.

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

`robot_mounted` (*Camera Mounting*) If set to *true*, the camera is mounted on the robot. If set to *false*, the camera is mounted statically and the calibration grid is mounted on the robot.

`tcp_offset` (*TCP Offset*) The signed offset from the TCP to the camera coordinate system (robot-mounted sensor) or the visible surface of the calibration grid (statically mounted sensor) along the TCP rotation axis in meters. This is required if the robot's movement is constrained and it can rotate its TCP only around one axis (e.g., 4DOF robot).

`tcp_rotation_axis` (*TCP Rotation Axis*) The axis of the *robot* frame around which the robot can rotate its TCP. 0 is used for X, 1 for Y and 2 for the Z axis. This is required if the robot's movement is constrained and it can rotate its TCP only around one axis (e.g., 4DOF robot). -1 means that the robot can rotate its TCP around two independent rotation axes. `tcp_offset` is ignored in this case.

(*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 means, these values are applied even after reboot.

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

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: By calling this service, the current parameter settings for this component are irrecoverably lost.

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

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 has no arguments.

The definition for the response with corresponding datatypes is:

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

set_pose

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

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.

The definition for the request arguments with corresponding datatypes is:

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

(continues on next page)

(continued from previous page)

```
  "slot": "int32"
}
```

The definition for the response with corresponding datatypes is:

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

Table 6.12: 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

calibrate

calculates and returns the hand-eye calibration transformation with the robot poses configured by the `set_pose` service. `save_calibration` must be called to make the calibration transformation persistent.

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 has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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 6.13: Return codes of the `calibrate` service call

status	success	Description
0	true	calibration successful, returned calibration result
1	false	not enough poses to perform calibration
2	false	calibration result is invalid, please verify the input data
3	false	given calibration grid dimensions are not valid
4	false	insufficient rotation, `tcp_offset` and `tcp_rotation_axis` must be specified
5	false	sufficient rotation available, `tcp_rotation_axis` must be set to -1
6	false	poses are collinear

set_calibration

sets the hand-eye calibration transformation with arguments of this call. The calibration transformation is expected in the same format as returned by the `calibrate` and `get_calibration` calls. `save_calibration` must be called to make the calibration transformation persistent.

The definition for the request arguments with corresponding datatypes is:

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

The definition for the response with corresponding datatypes is:

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

Table 6.14: Return codes of the `set_calibration` service call

status	success	Description
0	true	setting the calibration transformation was successful
12	false	given orientation values are invalid

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 has no arguments.

The definition for the response with corresponding datatypes is:

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


Table 6.15: 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 has no arguments.

The definition for the response with corresponding datatypes is:


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


Table 6.16: Return codes of the `get_calibration` service call

status	success	Description
0	true	removed persistent calibration, device reports as uncalibrated
1	true	no persistent calibration found, device 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 has no arguments.

The definition for the response with corresponding datatypes is:


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


(continues on next page)

(continued from previous page)

```
  "robot_mounted": "bool",
  "status": "int32",
  "success": "bool"
}
```

Table 6.17: Return codes of the `get_calibration` service call

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

6.8 IO and Projector Control

The IOControl component allows reading the status of the general purpose digital inputs and controlling the digital general purpose outputs (GPIOs) of the *rc_visard*. The outputs can be set to LOW or HIGH, or configured to be HIGH for the exposure time of every image or every second image.

The purpose of the IOControl component is the control of an external light source or a projector, which is connected to one of the *rc_visard*'s GPIOs to be synchronized by the image acquisition trigger. In case a pattern projector is used to improve stereo matching, the intensity images also show the projected pattern, which might be a disadvantage for image processing tasks that are based on the intensity image (e.g. edge detection). For this reason, the IOControl component allows setting GPIO outputs to HIGH for the exposure time of every second image, so that intensity images without the projected pattern are also available.

Note: For more details on the *rc_visard*'s GPIOs please refer to *Wiring*, Section 3.5.

6.8.1 Parameters

The IOControl component is called `rc_iocontrol` in the REST-API and is represented by the *IOControl* page in the *Modules* tab of the *Web GUI* (Section 4.6). The user can change the parameters via the Web GUI, the *REST-API interface* (Section 8.2), or via GigE Vision using the DigitalIOControl parameters `LineSelector` and `LineSource` (*Category: DigitalIOControl*, Section 8.1.3).

Parameter overview

This component offers the following run-time parameters:

Table 6.18: The `rc_iocontrol` component's run-time parameters

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

Description of run-time parameters

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

`Low` sets the ouput permanently to LOW. This is the factory default.

`High` sets the output permanently to HIGH.

`ExposureActive` sets the output to HIGH for the exposure time of every image.

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

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

Fig. 6.27 shows which images are used for stereo matching and transmission via GigE Vision in `ExposureActive` mode with a user-defined frame rate of 8 Hz.

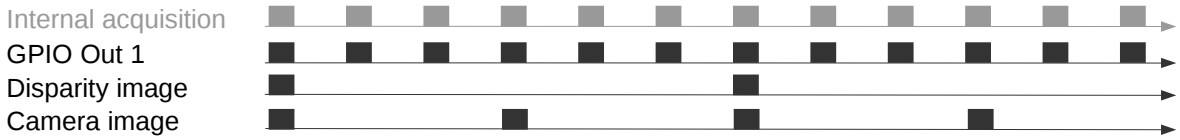


Fig. 6.27: Example of using the `ExposureActive` mode for GPIO Out 1 with a user-defined frame rate of 8 Hz. The internal image acquisition is always 25 Hz. GPIO Out 1 is HIGH for the exposure time of every image. A disparity image is computed for camera images that are sent out via GigE Vision according to the user-defined frame rate.

The mode `ExposureAlternateActive` is meant to be used when an external random dot projector is connected to the *rc_visard*'s GPIO Out 1. When setting Out 1 to `ExposureAlternateActive`, the *stereo matching* (Section 6.2) component only uses images with GPIO Out 1 being HIGH, i.e. projector is on. The maximum frame rate that is used for stereo matching is therefore half of the frame rate configured by the user (see *FPS*, Section 6.1.4). All components which make use of the intensity image, like *TagDetect* (Section 6.9) and *ItemPick* (Section 7.2), use the intensity images with GPIO Out 1 being LOW, i.e. projector is off. Fig. 6.28 shows an example.

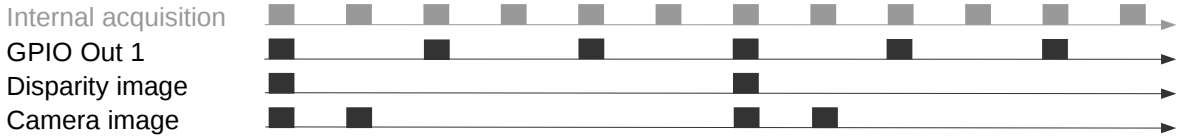


Fig. 6.28: Example of using the `ExposureAlternateActive` mode for GPIO Out 1 with a user-defined frame rate of 8 Hz. The internal image acquisition is always 25 Hz. GPIO Out 1 is HIGH for the exposure time of every second image. A disparity image is computed for images where Out 1 is HIGH and that are sent out via GigE Vision according to the user-defined frame rate. In `ExposureAlternateActive` mode, intensity images are always transmitted pairwise: one with GPIO Out 1 HIGH, for which a disparity image might be available, and one with GPIO Out 1 LOW.

Note: In `ExposureAlternateActive` mode, an intensity image with GPIO Out 1 being HIGH (i.e. with projection) is always 40 ms away from an intensity image with Out 1 being LOW (i.e. without projection), regardless of the user-defined frame rate. This needs to be considered when synchronizing disparity images and camera images without projection in this special mode.

The functionality can also be controlled by the DigitalIOControl parameters of the GenICam interface (*Category: DigitalIOControl*, Section 8.1.3).

6.8.2 Services

Each service response contains a `return_code`, which consists of a `value` plus an optional `message`. A successful service returns with a `return_code` value of `0`. Negative `return_code` values indicate that the service failed. Positive `return_code` values indicate that the service succeeded with additional information.

The IOControl component offers the following services.

get_io_values

This service call retrieves the current state of the *rc_visard*'s general purpose inputs and outputs (GPIOs).

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

The returned `timestamp` is the time of measurement.

`return_code` holds possible warnings or error codes and messages. Possible `return_code` values are shown below.

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

save_parameters

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

reset_defaults

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "return_code": {
    "message": "string",
```

(continues on next page)

(continued from previous page)

```
    "value": "int16"
  }
}
```

Warning: By calling this service, the current parameter settings for the IOControl component are irrecoverably lost.

6.9 TagDetect

6.9.1 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.

Tag detection is made up of three steps:

1. Tag reading on the 2D image pair (see *Tag reading*, Section 6.9.2).
2. Estimation of the pose of each tag (see *Pose estimation*, Section 6.9.3).
3. Re-identification of previously seen tags (see *Tag re-identification*, Section 6.9.4).

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

QR code

Fig. 6.29: 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

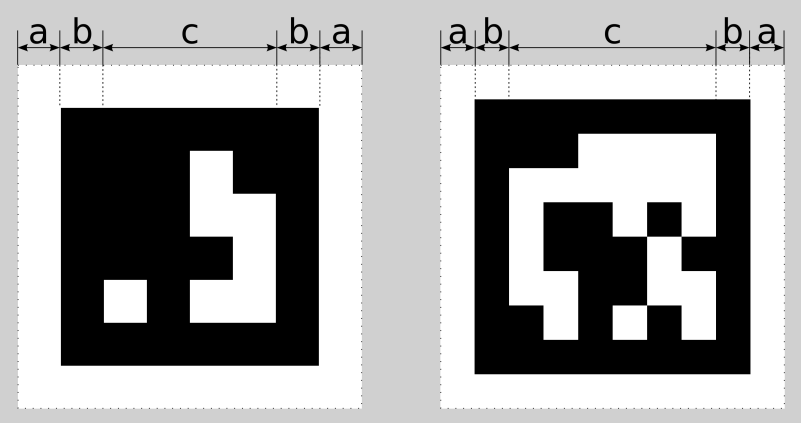


Fig. 6.30: A 16h5 tag (left) and a 36h11 tag (right). AprilTags consist of a mandatory white (a) and black (b) border and a variable amount of data bits (c).

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. 6.30 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. 6.30 for example are of family 16h5 and 36h11 and have id 0 and 11, respectively. All supported families are shown in Table 6.19.

Table 6.19: 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. 6.30), 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 6.19).

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: Firstly, fewer data modules lead to fewer individual tag IDs. Secondly, 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. 6.30). 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.

6.9.2 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. `High` yields the largest maximum detection distance and highest precision, but also the highest processing time. `Low` results in the smallest maximum detection distance and lowest precision, but processing requires less than half of the time. `Medium` lies in between. Please note that this quality parameter has no relation to the quality parameter of *Stereo matching* (Section 6.2).

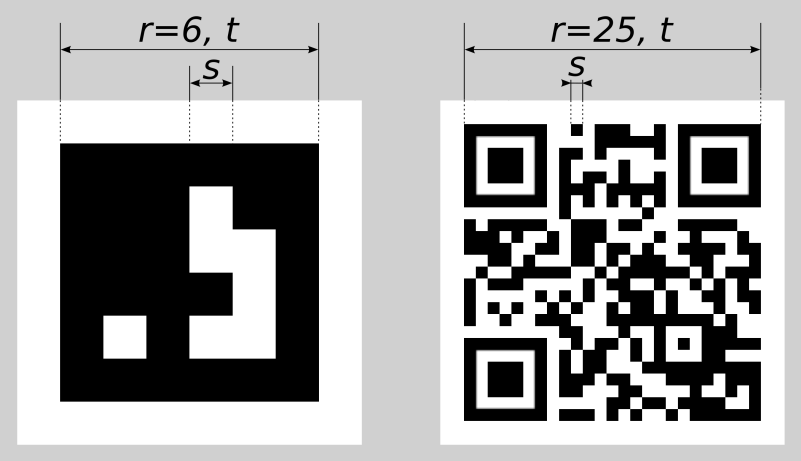


Fig. 6.31: Visualization of module size s, size of a tag in modules r, and size of a tag in meters t for AprilTags (left) and QR codes (right)

The maximum detection distance z at quality `High` 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. 6.31 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 `High`.

Table 6.20: 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 6.21: 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

6.9.3 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.

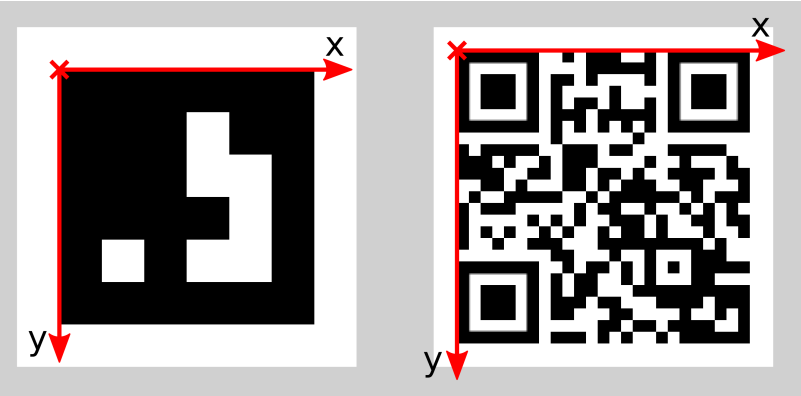


Fig. 6.32: Coordinate frames of AprilTags (left) and QR codes (right)

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

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

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

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

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

Below tables give approximate precisions of the estimated poses of AprilTags and QR codes. We distinguish between lateral precision (i.e., in x and y direction) and precision in z direction. It is assumed that `quality` is set to `High` and that the camera'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 6.22: 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 6.23: 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

6.9.4 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. 6.33. 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.

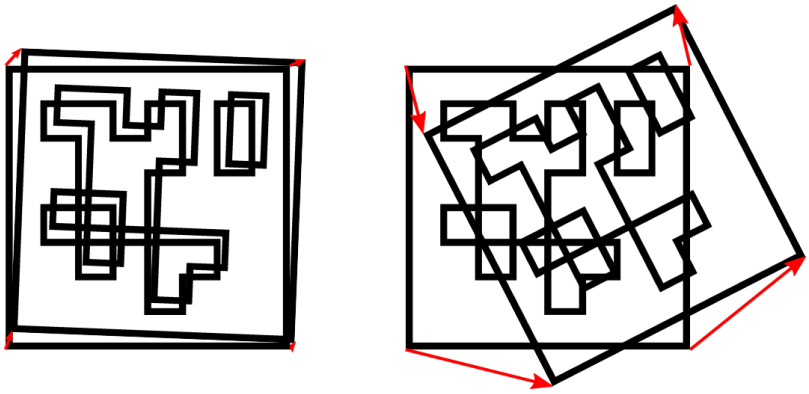


Fig. 6.33: Simplified visualization of tag re-identification. Euclidean distances between associated tag corners in 3D are compared (red arrows).

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`.

6.9.5 Hand-eye calibration

In case the *rc_visard* has been calibrated to a robot, the TagDetect component can automatically provide poses in the robot coordinate frame. For the TagDetect node's *Services* (Section 6.9.8), the frame of the output poses can be controlled with the `pose_frame` argument.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). All poses provided by the component are in the camera frame.
2. **External frame** (`external`). All poses provided by the component are in the external frame, configured by the user during the hand-eye calibration process. The component relies on the on-board *Hand-eye calibration component* (Section 6.7) to retrieve the sensor mounting (static or robot mounted) and the hand-eye transformation. If the sensor mounting is static, no further information is needed. If the sensor is robot-mounted, the `robot_pose` is required to transform poses to and from the `external` frame.

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

6.9.6 Parameters

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

In addition to the *REST-API interface* (Section 8.2), the TagDetect components provide tabs on the Web GUI, on which they can be tried out and configured manually.

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

This component offers the following run-time parameters:

Table 6.24: The rc_qr_code_detect component's run-time parameters

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

6.9.7 Status values

These TagDetect components reports the following status values:

Table 6.25: The rc_qr_code_detect and rc_april_tag_detect component's status values

Name	Description
data_acquisition_time	Time in seconds required to acquire image pair
last_timestamp_processed	The timestamp of the last processed image pair
state	The current state of the node
tag_detection_time	Processing time of the last tag detection in seconds

The reported state can take one of the following values.

Table 6.26: 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.

6.9.8 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, the component should only be running when necessary.

This service has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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 has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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 has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "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`.

Request:

The definition for the request arguments with corresponding datatypes is:

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

(continues on next page)

(continued from previous page)

```
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  },
  "tags": [
    {
      "id": "string",
      "size": "float64"
    }
  ]
}
```

Optional arguments:

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

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

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

In addition to the ID, the approximate size ($\pm 10\%$) of a tag can be set with the `size` parameter. As described in *Pose estimation*, Section 6.9.3, this information helps to resolve ambiguities in pose estimation that may arise in certain situations.

`pose_frame` controls whether the poses of the detected tags are returned in the camera or external frame, as detailed in *Hand-eye calibration*, Section 6.9.5. The default is `camera`.

Response:

The definition for the response with corresponding datatypes is:

```
{
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "tags": [
    {
      "id": "string",
      "instance_id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
```

(continues on next page)

(continued from previous page)


```
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "size": "float64",
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      }
    }
  ],
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```


`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 either be "camera" or "external".

`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 during tag detection
-102	There was a backwards jump of system time
-103	Internal error during tag pose estimation
-200	A fatal internal error occurred
200	Multiple warnings occurred; see list in `message`
201	The component was not in state `RUNNING`

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

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

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

save_parameters

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

reset_defaults

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

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).
- ***ItemPick and BoxPick*** **(`rc_itempick` and `rc_boxpick`, Section 7.2)** provides an out-of-the-box perception solution for robotic pick-and-place applications of unknown objects or boxes.
- ***SilhouetteMatch*** **(`rc_silhouettematch`, Section 7.3)** provides an object detection solution for objects placed on a plane.
- ***CollisionCheck*** **(`rc_collision_check`, Section 7.4)** provides an easy way to check if a gripper is in collision with load carrier.

The *rc_visard* provides some common functionalities which are used in multiple optional software components. The common functionalities are:

- ***Load carrier functionality*** **(Section 7.5.1)** allows setting and retrieving load carriers, as well as detecting load carriers and their filling level.
- ***Region of interest functionality*** **(Section 7.5.2)** allows setting and retrieving 3D regions of interest.

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 6.3.4) 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 7.1.5) below for details.

To store the feature map on the *rc_visard*, the SLAM component provides the service `save_map`, which can be used only during runtime (state "RUNNING") or after stopping (state "HALTED").

A stored map can be loaded before startup using the service `load_map`, which is only applicable in state "IDLE" (use the `reset` service to go back to "IDLE" when SLAM is in state "HALTED"). Note that mistaken localization at (visually) similar places may happen more easily when initially localizing in a loaded map than when localizing during continuous operation. Choosing a starting point with a unique visual appearance avoids this problem. The SLAM component will therefore assume that the *rc_visard* is started in the rough vincinity (a few meters) of the origin of the map. The origin of the map is where the Stereo-INS component was started when the map was recorded.

7.1.2 Memory limitations

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

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

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

The second option is to keep running until the memory is exhausted. In that case, the SLAM component will be restarted. If the `autorecovery` parameter is set to true, the SLAM component will recover its previous position and resume mapping. Otherwise it will go to FATAL state, requiring to be restarted via the rc_dynamics interface (see *Services*, Section 6.3.4).

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

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

7.1.3 Parameters

The SLAM component is called `rc_slam` in the REST-API. The user can change the SLAM parameters using the *REST-API interface* (Section 8.2).

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

7.1.4 Status values

This component reports the following status values:

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

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

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

Table 7.3: 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.

7.1.5 Services

Note: Activation and deactivation of the SLAM component is done via the service interface of rc_dynamics (see *Services*, Section 6.3.4).

Each service response (except for the `reset` service) contains a `return_code`, which consists of a `value` plus an optional `message`. A successful service returns with a `return_code` value of `0`. Negative `return_code` values indicate that the service failed. Positive `return_code` values indicate that the service succeeded with additional information.

The SLAM component offers the following services.

`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 6.3.4). 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 has no arguments.

The definition for the response with corresponding datatypes is:


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


get_trajectory

returns the trajectory.

The definition for the request arguments with corresponding datatypes is:


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


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

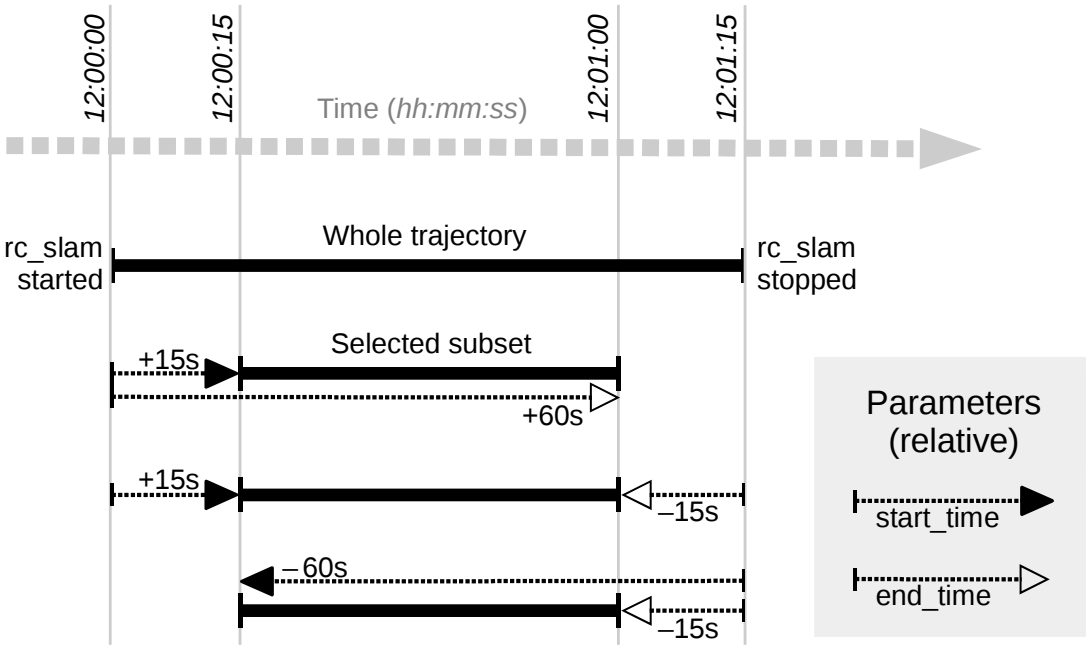


Fig. 7.1: Examples for combinations of relative start and end times for the `get_trajectory` service. All combinations shown select the same subset of the trajectory.

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

The definition for the response with corresponding datatypes is:


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


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

The field `return_code` holds possible warnings or error codes and messages. The following table contains a list of possible `return_code` values:

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

`save_map`

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

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

Warning: The map is lost on software updates or rollbacks

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

load_map

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

remove_map

removes the stored map from the persistent memory.

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

7.2 ItemPick and BoxPick

7.2.1 Introduction

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

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

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

In addition, both components offer:

- A dedicated page on the *rc_visard Web GUI* (Section 4.6) for easy setup, configuration, testing, and application tuning.
- The definition of regions of interest to select relevant volumes in the scene (see *Region of interest functionality*, Section 7.5.2).
- A load carrier detection functionality for bin-picking applications (see *Load carrier functionality*, Section 7.5.1), to provide grasps for items inside a bin only.
- The definition of compartments inside a load carrier to provide grasps for specific volumes of the bin only.
- Support for static and robot-mounted cameras and optional integration with the *Hand-eye calibration* (Section 6.7) component, to provide grasps in the user-configured external reference frame.
- A quality value associated to each suggested grasp and related to the flatness of the grasping surface.
- Sorting of grasps according to gravity and size so that items on top of a pile are grasped first.

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

7.2.2 Detection of items (BoxPick)

The BoxPick component supports the detection of multiple `item_models` of `type` `RECTANGLE`. Each item model is defined by its minimum and maximum size, with the minimum dimensions strictly smaller than the maximum dimensions. The dimensions should be given fairly accurately to avoid misdetections, while still considering a certain tolerance to account for possible production variations and measurement inaccuracies.

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

- The ID of the load carrier which contains the items to be detected.
- A compartment inside the load carrier where to detect items.
- The ID of the region of interest where to search for the load carriers if a load carrier is set. Otherwise, the ID of the region of interest where to search for the items.
- The current robot pose in case the camera is mounted on the robot and the chosen coordinate frame for the poses is `external` or the chosen region of interest is defined in the external frame.

The detected `item` poses are given relative to the centers of the rectangles, with the z axis pointing towards the camera. Each detected item includes a `uuid` (Universally Unique Identifier) and the `timestamp` of the oldest image that was used to detect it.

7.2.3 Computation of grasps

The ItemPick and BoxPick components offer a service for computing grasps for suction grippers. The gripper is defined by its suction surface length and width.

The ItemPick component identifies flat surfaces in the scene and supports flexible and/or deformable items. The `type` of these `item_models` is called `UNKNOWN` since they don't need to have a standard geometrical shape. Optionally, the user can also specify the minimum and maximum size of the item.

For BoxPick, the grasps are computed on the detected rectangular `items` (see *Detection of items (BoxPick)*, Section 7.2.2).

Optionally, further information can be given to the components in a grasp computation request:

- The ID of the load carrier which contains the items to be grasped.
- A compartment inside the load carrier where to compute grasps. The `load_carrier_compartment` is a box whose `pose` is defined with respect to the load carrier reference frame.

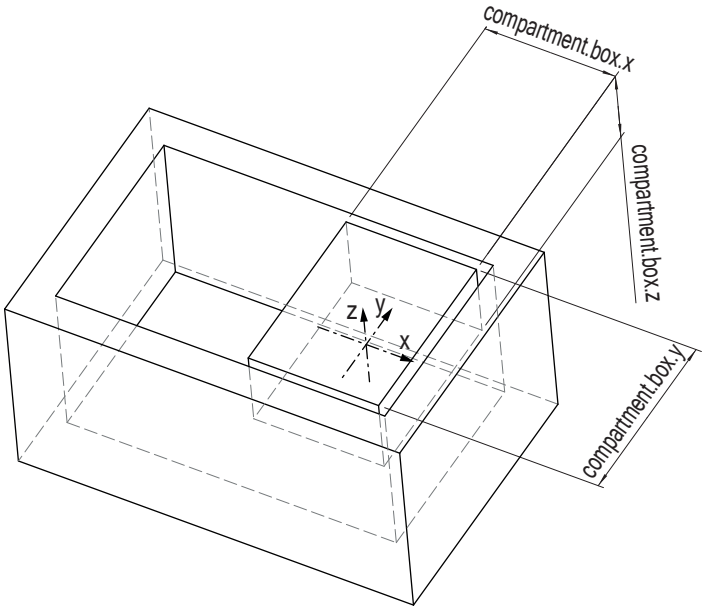


Fig. 7.2: Compartment inside a load carrier.

- The ID of the region of interest where to search for the load carriers if a load carrier is set. Otherwise, the ID of the region of interest where to compute grasps.
- Collision detection information: The ID of the gripper to enable collision checking and optionally a pre-grasp offset to define a pre-grasp position. The collision check requires a separate CollisionCheck *license* (Section 9.6) to be purchased. Details on collision checking are given below in *CollisionCheck* (Section 7.2.4).

A grasp provided by the ItemPick and BoxPick components represents the recommended pose of the TCP (Tool Center Point) of the suction gripper. The grasp `type` is always set to `SUCTION`. The computed grasp pose is the center of the biggest ellipse that can be inscribed in each surface. The grasp orientation is a right-handed coordinate system and is defined such that its z axis is normal to the surface pointing inside the object at the grasp position and its x axis is directed along the maximum elongation of the ellipse.

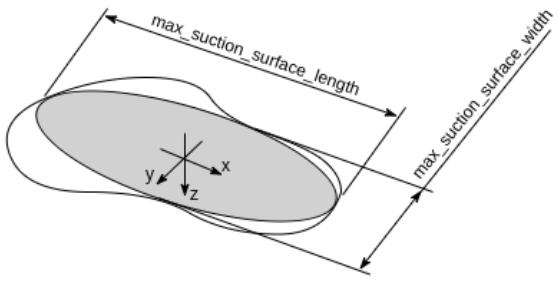


Fig. 7.3: Illustration of suction grasp with coordinate system and ellipse representing the maximum suction surface.

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

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

Each grasp also includes a `quality` value, which gives an indication of the flatness of the grasping surface. The `quality` value varies between 0 and 1, where higher numbers correspond to a flatter reconstructed surface.

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

7.2.4 Interaction with other components

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

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

Stereo camera and Stereo matching

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

- Rectified images from the *Stereo camera* component (`rc_stereocamera`, Section 6.1);
- Disparity, error, and confidence images from the *Stereo matching* component (`rc_stereomatching`, Section 6.2).

All processed images are guaranteed to be captured after the component trigger time.

Estimation of gravity vector

For each load carrier detection and grasp computation, the components estimate the gravity vector by subscribing to the *rc_visard*'s IMU data stream.

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

IO and Projector Control

In case the *rc_visard* is used in conjunction with an external random dot projector and the *IO and Projector Control* component (`rc_iocontrol`, Section 6.8), it is recommended to connect the projector to GPIO Out 1 and set the stereo-camera component's acquisition mode to `SingleFrameOut1` (see *Stereo matching parameters*, Section 6.2.4), so that on each image acquisition trigger an image with and without projector pattern is acquired.

Alternatively, the output mode for the GPIO output in use should be set to `ExposureAlternateActive` (see *Description of run-time parameters*, Section 6.8.1).

In either case, the *Auto Exposure Mode* `exp_auto_mode` should be set to `AdaptiveOut1` to optimize the exposure of both images (see *Stereo camera parameters*, Section 6.1.4).

Hand-eye calibration

In case the camera has been calibrated to a robot, the ItemPick and BoxPick components can automatically provide poses in the robot coordinate frame. For the ItemPick and BoxPick nodes' *Services* (Section 7.2.7), the frame of the output poses can be controlled with the `pose_frame` argument.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). All poses provided by the components are in the camera frame, and no prior knowledge about the pose of the camera in the environment is required. This means that the configured regions of interest and load carriers move with the camera. It is the user's responsibility to update the configured poses if the camera frame moves (e.g. with a robot-mounted camera).
2. **External frame** (`external`). All poses provided by the components are in the external frame, configured by the user during the hand-eye calibration process. The component relies on the on-board *Hand-eye calibration component* (Section 6.7) to retrieve the sensor mounting (static or robot mounted) and the hand-eye transformation. If the mounting is static, no further information is needed. If the sensor is robot-mounted, the `robot_pose` is required to transform poses to and from the `external` frame.

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

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

CollisionCheck

In case a CollisionCheck license is available, the collision checking can be easily enabled for grasp computation of the ItemPick and BoxPick components by passing the ID of the used gripper and optionally a pre-grasp offset to the `compute_grasps` service call. The gripper has to be defined in the CollisionCheck component (see *Setting a gripper* (Section 7.4.2)) and details about collision checking are given in *Collision checking within other modules* (Section 7.4.3).

If collision checking is enabled, only grasps which are collision free will be returned. However, the visualization images on the *ItemPick* or *BoxPick* tab of the Web GUI also show colliding grasp points as black ellipses.

The CollisionCheck module's run-time parameters affect the collision detection as described in *CollisionCheck Parameters* (Section 7.4.4).

7.2.5 Parameters

The ItemPick and BoxPick components are called `rc_itempick` and `rc_boxpick` in the REST-API and are represented by the *ItemPick* and *BoxPick* pages in the *Modules* tab of the *Web GUI* (Section 4.6). The user can explore and configure the `rc_itempick` and `rc_boxpick` component's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 8.2).

Parameter overview

This component offers the following run-time parameters:

Table 7.4: The `rc_itempick` and `rc_boxpick` components application parameters

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

Table 7.5: The `rc_itempick` and `rc_boxpick` components load carrier detection parameters

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

Table 7.6: The `rc_itempick` and `rc_boxpick` components surface clustering parameters

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

Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's *ItemPick* or *BoxPick* page in the *Modules* 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:

max_grasps (*Maximum Grasps*) sets the maximum number of provided grasps.

load_carrier_model_tolerance (*Model Tolerance*) see *Parameters of the load carrier functionality* (Section 7.5.1).

load_carrier_crop_distance (*Crop Distance*) see *Parameters of the load carrier functionality* (Section 7.5.1).

cluster_max_dimension (Only for ItemPick, *Cluster Maximum Dimension*) sets the diameter of the largest circle enclosing each cluster in meters. Clusters larger than this value are filtered out before grasp computation.

cluster_max_curvature (*Cluster Maximum Curvature*) is the maximum curvature allowed within one cluster. The smaller this value, the more clusters will be split apart.

clustering_patch_size (Only for ItemPick, *Patch Size*) is the size of the square patches the depth map is subdivided into during the first clustering step in pixels.

clustering_discontinuity_factor (*Discontinuity Factor*) is the factor used to discriminate depth discontinuities within a patch. The smaller this value, the more clusters will be split apart.

clustering_max_surface_rmse (*Maximum Surface RMSE*) is the maximum root-mean-square error (RMSE) in meters of points belonging to a surface.

7.2.6 Status values

The rc_itempick and rc_boxpick components report the following status values:

Table 7.7: The rc_itempick and rc_boxpick components status values

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

The reported state can take one of the following values.

Table 7.8: Possible states of the ItemPick and BoxPick components

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

7.2.7 Services

The user can explore and call the rc_itempick and rc_boxpick component's services, e.g. for development and testing, using the *REST-API interface* (Section 8.2) or the *rc_visard Web GUI* (Section 4.6).

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

The following table contains a list of common codes:

Table 7.9: Return codes of the ItemPick and BoxPick services

Code	Description
0	Success
-1	An invalid argument was provided
-4	Data acquisition took longer than the maximum allowed time of 5.0 seconds
-10	New element could not be added as the maximum storage capacity of load carriers or regions of interest has been exceeded
-200	Fatal internal error
-301	More than one item model of type `UNKNOWN` provided to the `compute_grasps` service
-302	More than one load carrier provided to the `detect_load_carriers` or `detect_filling_level` services, but only one is supported
10	The maximum storage capacity of load carriers or regions of interest has been reached
11	An existent persistent model was overwritten by the call to `set_load_carrier` or `set_region_of_interest`
100	The requested load carriers were not detected in the scene
101	No valid surfaces or grasps were found in the scene
102	The detected load carrier is empty
103	All computed grasps are in collision with the load carrier
200	The component is in state `IDLE`
300	A valid `robot_pose` was provided as argument but it is not required
400	No `item_models` were provided to the `compute_grasps` service request

The ItemPick and BoxPick components offer the following services.

`start`

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

`stop`

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

`set_region_of_interest`

see *set_region_of_interest* (Section 7.5.2).

get_regions_of_interest

see *get_regions_of_interest* (Section 7.5.2).

delete_regions_of_interest

see *delete_regions_of_interest* (Section 7.5.2).

set_load_carrier

see *set_load_carrier* (Section 7.5.1).

get_load_carriers

see *get_load_carriers* (Section 7.5.1).

delete_load_carriers

see *delete_load_carriers* (Section 7.5.1).

detect_load_carriers

see *detect_load_carriers* (Section 7.5.1).

detect_filling_level

see *detect_filling_level* (Section 7.5.1).

detect_items (BoxPick only)

Triggers the detection of rectangles as described in *Detection of items (BoxPick)* (Section 7.2.2).

Request:

The definition for the request arguments with corresponding datatypes is:

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

(continues on next page)

(continued from previous page)

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

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 7.2.4).

`item_models`: list of rectangles with minimum and maximum size, with the minimum dimensions strictly smaller than the maximum dimensions. The dimensions should be given fairly accurately to avoid misdetections, while still considering a certain tolerance to account for possible production variations and measurement inaccuracies.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 7.2.4).

Optional arguments:

`load_carrier_id`: ID of the load carrier which contains the items to be detected.

`load_carrier_compartment`: compartment inside the load carrier where to detect items.

`region_of_interest_id`: if `load_carrier_id` is set, ID of the region of interest where to search for the load carriers. Otherwise, ID of the region of interest where to search for the items.

Response:

The definition for the response with corresponding datatypes is:

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


(continues on next page)

(continued from previous page)

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

`load_carriers`: list of detected load carriers.

`items`: list of detected rectangles.

`timestamp`: timestamp of the image set the detection ran on.

`return_code`: holds possible warnings or error codes and messages.

`compute_grasps` (for ItemPick)

Triggers the computation of grasping poses for a suction device as described in *Computation of grasps* (Section 7.2.3).

Request:

The definition for the request arguments with corresponding datatypes is:

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

(continues on next page)

(continued from previous page)

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

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 7.2.4).

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

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

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 7.2.4).

Optional arguments:

`load_carrier_id`: ID of the load carrier which contains the items to be grasped.

`load_carrier_compartment`: compartment inside the load carrier where to compute grasps.

`region_of_interest_id`: if `load_carrier_id` is set, ID of the region of interest where to search for the load carriers. Otherwise, ID of the region of interest where to compute grasps.

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

`collision_detection`: see *Collision checking within other modules* (Section 7.4.3). The collision check requires a separate CollisionCheck *license* (Section 9.6) to be purchased.

Response:

The definition for the response with corresponding datatypes is:

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


(continues on next page)

(continued from previous page)

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

`load_carriers`: list of detected load carriers.

`grasps`: sorted list of suction grasps.

`timestamp`: timestamp of the image set the detection ran on.

`return_code`: holds possible warnings or error codes and messages.

`compute_grasps` (for BoxPick)

Triggers the detection of rectangles and the computation of grasping poses for the detected rectangles as described in *Computation of grasps* (Section 7.2.3).

Request:

The definition for the request arguments with corresponding datatypes is:

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

(continues on next page)

(continued from previous page)

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

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 7.2.4).

`item_models`: list of rectangles with minimum and maximum size, with the minimum dimensions strictly smaller than the maximum dimensions. The dimensions should be given fairly accurately to avoid misdetections, while still considering a certain tolerance to account for possible production variations and measurement inaccuracies.

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

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

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 7.2.4).

Optional arguments:

`load_carrier_id`: ID of the load carrier which contains the items to be grasped.

`load_carrier_compartment`: compartment inside the load carrier where to compute grasps.

`region_of_interest_id`: if `load_carrier_id` is set, ID of the region of interest where to search for the load carriers. Otherwise, ID of the region of interest where to compute grasps.

`collision_detection`: see *Collision checking within other modules* (Section 7.4.3). The collision check requires a separate CollisionCheck *license* (Section 9.6) to be purchased.

Response:

The definition for the response with corresponding datatypes is:

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


(continues on next page)

(continued from previous page)


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


`load_carriers`: list of detected load carriers.

`items`: sorted list of suction grasps on the detected rectangles.

`grasps`: sorted list of suction grasps.

`timestamp`: timestamp of the image set the detection ran on.

`return_code`: holds possible warnings or error codes and messages.

save_parameters

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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


reset_defaults

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

This service has no arguments.

The definition for the response with corresponding datatypes is:


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


7.3 SilhouetteMatch

7.3.1 Introduction

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

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

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

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

The SilhouetteMatch component offers:

- A dedicated page on the *rc_visard Web GUI* (Section 4.6) for easy setup, configuration, testing, and application tuning.
- A *REST-API interface* (Section 8.2) and a *KUKA Ethernet KRL Interface* (Section 8.4).
- The definition of 2D regions of interest to select relevant parts of the camera image (see *Setting a region of interest*, Section 7.3.3).
- A load carrier detection functionality for bin-picking applications (see *Load carrier functionality*, Section 7.5.1), to provide grasps for objects inside a bin only.
- The definition of grasp points for each template via an interactive visualization in the Web GUI.
- Support for static and robot-mounted cameras and optional integration with the *Hand-eye calibration* (Section 6.7) component, to provide grasps in the user-configured external reference frame.

- Sorting of grasps according to reachability so that the ones which are closest to the camera along the z axis of the preferred orientation of the TCP are returned first.

Suitable objects

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

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

Suitable scene

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

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

7.3.2 Base-plane calibration

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

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

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

- AprilTag based
- Stereo based
- Manual

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

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

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

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

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

AprilTag based base-plane calibration

AprilTag detection (ref. *TagDetect*, Section 6.9) is used to find AprilTags in the scene and fit a plane through them. At least three AprilTags must be placed on the base plane so that they are visible in the left and right camera images. The tags should be placed such that they are spanning a triangle that is as large as possible. The larger the triangle, the more accurate is the resulting base-plane estimate. Use this method if the base plane is untextured and no external random dot projector is available. This calibration mode is available via the *REST-API interface* (Section 8.2) and the *rc_visard* Web GUI.

Stereo based base-plane calibration

The 3D point cloud computed by the stereo matching component is used to fit a plane through its 3D points. Therefore, the region of interest (ROI) for this method must be set such that only the relevant base plane is included. The `plane_preference` parameter allows to select whether the plane closest to or farthest from the camera should be used as base plane. Selecting the closest plane can be used in scenarios where the base plane is completely occluded by objects or not accessible for calibration. Use this method if the base plane is well textured or you can make use of a random dot projector to project texture on the base plane. This calibration mode is available via the *REST-API interface* (Section 8.2) and the *rc_visard* Web GUI.

Manual base-plane calibration

The base plane can be set manually if its parameters are known, e.g. from previous calibrations. This calibration mode is only available via the *REST-API interface* (Section 8.2) and not the *rc_visard* Web GUI.

7.3.3 Setting a region of interest

If objects are to be detected only in part of the camera's field of view, a region of interest (ROI) can be set accordingly. This ROI is defined as a rectangular part of the left camera image, and can be set via the *REST-API interface* (Section 8.2) or the *rc_visard* Web GUI. The Web GUI offers an easy-to-use selection tool. Up to 50 ROIs can be set and stored persistently on the *rc_visard*. Each ROI must have a unique name to address a specific ROI in the base-plane calibration or object detection process.

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

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

7.3.4 Setting of grasp points

To use SilhouetteMatch directly in a robot application, grasp points can be defined for each template. A grasp point represents the desired position and orientation of the robot's TCP (Tool Center Point) to grasp an object as shown in Fig. 7.4

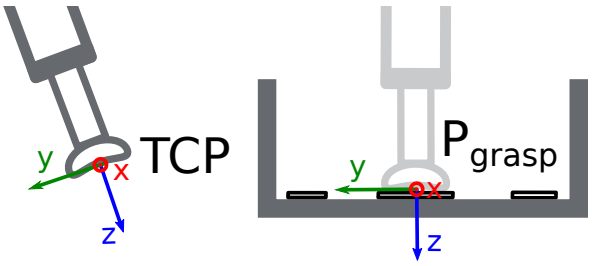


Fig. 7.4: Definition of grasp points with respect to the robot's TCP

Each grasp consists of an `id` which must be unique within all grasps for an object template, the `template_id` representing the template to which the grasp should be attached, and the `pose` in the coordinate frame of the object template. Grasp points can be set via the *REST-API interface* (Section 8.2), or by using the interactive visualization in the Web GUI. The *rc_visard* can store up to 50 grasp points per template.

Setting grasp points in the Web GUI

The *rc_visard* Web GUI provides an intuitive and interactive way of defining grasp points for object templates. In a first step, the object template has to be uploaded to the *rc_visard*. This can be done on the *SilhouetteMatch* page in the *Modules* tab of the Web GUI by clicking on *add new Template* in the *Templates and Grasps* section of the *SilhouetteMatch* page. Once the template upload is complete, a dialog with a 3D visualization of the object tempate is shown for adding or editing grasp points. The same dialog appears when editing an existing template.

This dialog provides two ways for setting grasp points:

1. **Adding grasps manually**: By clicking on the + symbol, a new grasp is placed in the object origin. The grasp can be given a unique name which corresponds to its ID. The desired pose of the grasp can be entered in the fields for *Position* and *Roll/Pitch/Yaw* which are given in the coordinate frame of the object template represented by the long x, y and z axes in the visualization. The grasp point can be placed freely with respect to the object template - inside, outside or on the surface. The grasp point and its orientation are visualized in 3D for verification.
2. **Adding grasps interactively**: Grasp points can be added interactively by first clicking on the *Add Grasp* button in the upper left corner of the visualization and then clicking on the desired point on the object template visualization. The grasp is attached to the template surface. The grasp orientation is a right-handed coordinate system and is chosen such that its z axis is perpendicular to the surface pointing inside the template at the grasp position. The position and orientation in the object coordinate frame is displayed on the right. The position and orientation of the grasp can also be changed interactively. In case *Snap to surface* is enabled in the visualization (default), the grasp can be moved along the template surface by clicking on the *Translate* button in the visualization and then clicking on the grasp point and dragging it to the desired position. The orientation of the grasp around the surface normal can also be changed by choosing *Rotate* and then rotating the grasp with the cursor. In case *Snap to surface* is disabled, the grasp can be translated and rotated freely in all three dimensions.

If the object template has symmetries, the grasps which are symmetric to the defined grasps can be displayed by clicking on *Show symmetric grasps*.

Setting grasp points via the REST-API

Grasp points can be set via the *REST-API interface* (Section 8.2) using the `set_grasp` or `set_all_grasps` services (see *Services*, Section 7.3.10). In the SilhouetteMatch component a grasp consists of the `template_id` of the template to which the grasp should be attached, an `id` uniquely identifying the grasp point and the `pose`. The pose is given in the coordinate frame of the object template and consists of a `position` in meters and an `orientation` as quaternion.

7.3.5 Setting the preferred orientation of the TCP

The SilhouetteMatch component determines the reachability of grasp points based on the *preferred orientation* of the gripper or TCP. The preferred orientation can be set via the `set_preferred_orientation` service or on the *SilhouetteMatch* page in the Web GUI. The resulting direction of the TCP's z axis is used to reject grasps which cannot be reached by the gripper. Furthermore, it is used to sort the reachable grasps such that the closest grasps to the camera along the Z axis of the preferred orientation of the TCP are returned first.

The preferred orientation can be set in the camera coordinate frame or in the external coordinate frame, in case a hand-eye calibration is available. If the preferred orientation is specified in the external coordinate frame and the sensor is robot mounted, the current robot pose has to be given to each object detection call, so that the preferred orientation can be used for filtering and sorting the grasps on the detected objects. If no preferred orientation is set, the z axis of the left camera is used as the preferred orientation of the TCP.

7.3.6 Detection of objects

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

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

- The template of the object to be detected in the scene.
- The coordinate frame in which the poses of the detected objects shall be returned (ref. *Hand-eye calibration*, Section 7.3.7).

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

- An offset in case the objects are lying not on the base plane but on a plane parallel to it. The offset is the distance between both planes given in the direction towards the camera. If omitted, an offset of 0 is assumed.
- The ID of the load carrier which contains the objects to be detected.
- The ID of the region of interest where to search for the load carrier if a load carrier is set. Otherwise, the ID of the region of interest where the objects should be detected. If omitted, objects are matched in the whole image.
- The current robot pose in case the camera is mounted on the robot and the chosen coordinate frame for the poses is `external` or the preferred orientation is given in the external frame.
- Collision detection information: The ID of the gripper to enable collision checking and optionally a pre-grasp offset to define a pre-grasp position. The collision check requires a separate CollisionCheck *license* (Section 9.6) to be purchased. Details on collision checking are given below in *CollisionCheck* (Section 7.3.7).

On the Web GUI the detection can be tested in the *Try Out* section of the SilhouetteMatch component's tab. The result is visualized as shown in Fig. 7.5.

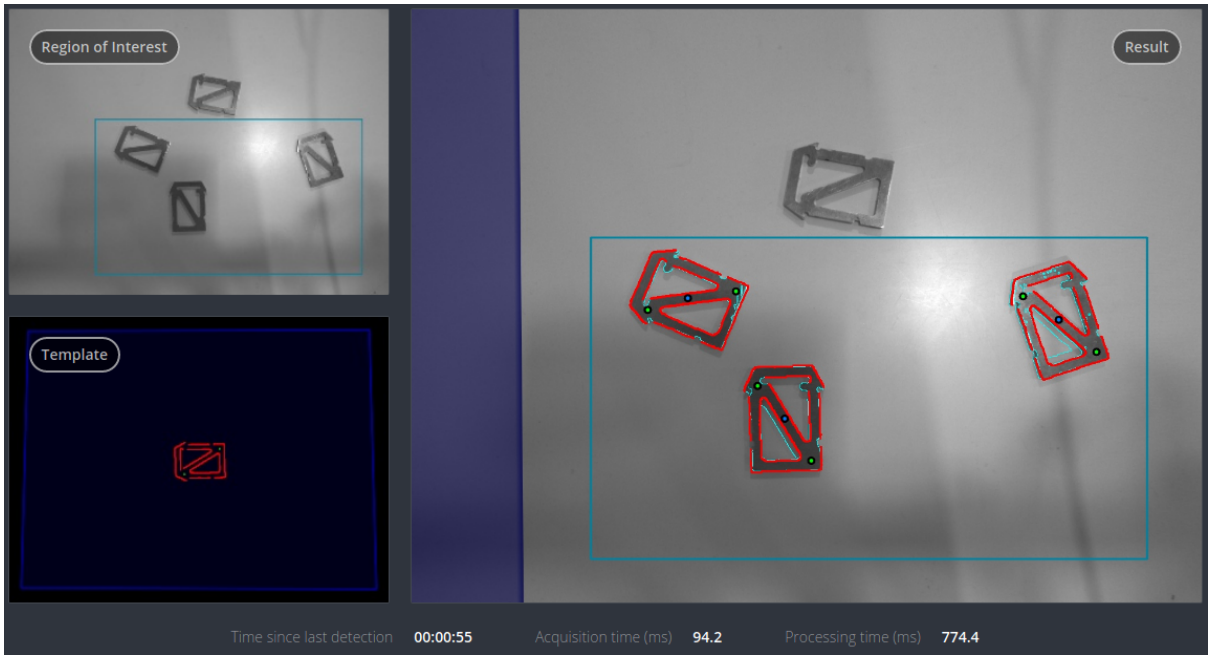


Fig. 7.5: Result image of the SilhouetteMatch component as shown in the Web GUI

The upper left image shows the selected region of interest. The lower left image shows the calibrated base plane in blue and the template to be matched in red with the defined grasp points in green (see *Setting of grasp points*, Section 7.3.4). The template is warped to the size and tilt matching objects on the calibrated base plane would have.

The right image shows the detection result. The shaded blue area on the left is the region of the left camera image which does not overlap with the right image, and in which no objects can be detected. The chosen region of interest is shown as bold petrol rectangle. The detected edges in the image are shown in light blue and the matches with the template (`instances`) are shown in red. The blue circles are the origins of the detected objects as defined in the template and the green circles are the reachable grasp points. Unreachable grasp points will be visualized as red dots (not shown in the figure).

The poses of the object origins in the chosen coordinate frame are returned as results. If the chosen template also has grasp points attached, a list of `grasps` for all objects sorted by their reachability (see *Setting the preferred orientation of the TCP*, Section 7.3.5) is returned in addition to the list of detected objects. The grasp poses are given in the desired coordinate frame. There are references between the detected object instances and the grasps via their `uuids`. In case the templates have a continuous rotational symmetry, all returned object poses will have the same orientation. For rotationally non-symmetric objects, the orientation of the detected objects is aligned with the normal of the base plane.

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

7.3.7 Interaction with other components

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

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

Stereo camera and stereo matching

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

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

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

IO and Projector Control

In case the *rc_visard* is used in conjunction with an external random dot projector and the *IO and Projector Control* component (`rc_iocontrol`, Section 6.8), the projector should be used for the stereo-based base-plane calibration.

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

Hand-eye calibration

In case the camera has been calibrated to a robot, the SilhouetteMatch component can automatically provide poses in the robot coordinate frame. For the SilhouetteMatch node's *Services* (Section 7.3.10), the frame of the input and output poses and plane coordinates can be controlled with the `pose_frame` argument.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). All poses and plane coordinates provided to and by the component are in the camera frame.
2. **External frame** (`external`). All poses and plane coordinates provided to and by the component are in the external frame, configured by the user during the hand-eye calibration process. The component relies on the on-board *Hand-eye calibration component* (Section 6.7) to retrieve the camera mounting (static or robot mounted) and the hand-eye transformation. If the sensor mounting is static, no further information is needed. If the sensor is robot-mounted, the `robot_pose` is required to transform poses to and from the `external` frame.

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

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

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

If the sensor is robot-mounted, the current `robot_pose` has to be provided depending on the value of `pose_frame` and the definition of the preferred TCP orientation:

- If `pose_frame` is set to `external`, providing the robot pose is obligatory.
- If the preferred TCP orientation is defined in `external`, providing the robot pose is obligatory.
- If `pose_frame` is set to `camera` and the preferred TCP orientation is defined in `camera`, providing the robot pose is optional.

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

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

CollisionCheck

In case a CollisionCheck license is available, the collision checking can be easily enabled for grasp computation of the SilhouetteMatch component by passing the ID of the used gripper and optionally a pre-grasp offset to the `detect_object` service call. The gripper has to be defined in the CollisionCheck component (see *Setting a gripper* (Section 7.4.2)) and details about collision checking are given in *Collision checking within other modules* (Section 7.4.3).

If collision checking is enabled, only grasps which are collision free will be returned. However, the visualization images on the *SilhouetteMatch* tab of the Web GUI also shows colliding grasp points in red.

The CollisionCheck module's run-time parameters affect the collision detection as described in *CollisionCheck Parameters* (Section 7.4.4).

7.3.8 Parameters

The SilhouetteMatch software component is called `rc_silhouettematch` in the REST-API and is represented by the *SilhouetteMatch* page in the *Modules* tab of the *Web GUI* (Section 4.6). The user can explore and configure the `rc_silhouettematch` component's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 8.2).

Parameter overview

This component offers the following run-time parameters:

Table 7.10: The rc_silhouettematch component's run-time parameters

Name	Type	Min	Max	Default	Description
edge_sensitivity	float64	0.1	1.0	0.6	sensitivity of the edge detector
load_carrier_crop_distance	float64	0.0	0.02	0.005	Safety margin in meters by which the load carrier inner dimensions are reduced to define the region of interest for detection
load_carrier_model_tolerance	float64	0.003	0.025	0.008	Indicates how much the estimated load carrier dimensions are allowed to differ from the load carrier model dimensions in meters
match_max_distance	float64	0.0	10.0	2.5	maximum allowed distance in pixels between the template and the detected edges in the image
match_percentile	float64	0.7	1.0	0.85	percentage of template pixels that must be within the maximum distance to successfully match the template
max_number_of_detected_objects	int32	1	20	10	maximum number of detected objects
quality	string	-	-	High	High, Medium, or Low

Description of run-time parameters

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

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

load_carrier_model_tolerance (***Model Tolerance***) see *Parameters of the load carrier functionality* (Section 7.5.1).

load_carrier_crop_distance (***Crop Distance***) see *Parameters of the load carrier functionality* (Section 7.5.1).

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

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

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

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

7.3.9 Status values

This component reports the following status values:

Table 7.11: The `rc_silhouettematch` component's status values

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

7.3.10 Services

The user can explore and call the `rc_silhouettematch` component's services, e.g. for development and testing, using the *REST-API interface* (Section 8.2) or the *rc_visard Web GUI* (Section 4.6).

Each service response contains a `return_code`, which consists of a `value` plus an optional `message`. A successful service returns with a `return_code` value of `0`. Negative `return_code` values indicate that the service failed. Positive `return_code` values indicate that the service succeeded with additional information.

Table 7.12: Return codes of the SilhouetteMatch component services

Code	Description
0	Success
-1	An invalid argument was provided
-3	An internal timeout occurred, e.g. during object detection
-4	Data acquisition took longer than the maximum allowed time of 5.0 seconds
-7	Data could not be read or written to persistent storage
-8	Component is not in a state in which this service can be called. E.g. `detect_object` cannot be called if there is no base-plane calibration.
-10	New element could not be added as the maximum storage capacity of regions of interest or templates has been exceeded
-100	An internal error occurred
-101	Detection of the base plane failed
-102	The hand-eye calibration changed since the last base-plane calibration
-104	Offset between the base plane normal and the camera's line of sight exceeds 10 degrees
10	The maximum storage capacity of regions of interest or templates has been reached
11	An existing element was overwritten
100	The requested load carrier was not detected in the scene
101	None of the detected grasps is reachable
102	The detected load carrier is empty
103	All detected grasps are in collision with the load carrier
107	The base plane was not transformed to the current camera pose, e.g. because no robot pose was provided during base-plane calibration
108	The template is deprecated.
151	The object template has a continuous symmetry
999	Additional hints for application development

The SilhouetteMatch component offers the following services.

`calibrate_base_plane`

Triggers the calibration of the base plane, as described in *Base-plane calibration* (Section 7.3.2). A successful base-plane calibration is stored persistently on the *rc_visard* and returned by this service. The base-plane calibration is persistent over firmware updates and rollbacks.

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

Request:

The definition for the request arguments with corresponding datatypes is:


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


(continues on next page)

(continued from previous page)

```
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  },
  "stereo": {
    "plane_preference": "string"
  }
}
```

Required arguments:

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

`pose_frame`: see *Hand-eye calibration* (Section 7.3.7).

Potentially required arguments:

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

`robot_pose`: see *Hand-eye calibration* (Section 7.3.7).

`region_of_interest_2d_id`: ID of the region of interest for base-plane calibration.

Optional arguments:

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

`plane_preference` in `stereo`: whether the plane closest to or farthest from the camera should be used as base plane. This option can be set only if `plane_estimation_method` is `STEREO`. Valid values are `CLOSEST` and `FARTHEST`. If not set, the default is `FARTHEST`.

Response:

The definition for the response with corresponding datatypes is:

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

`plane`: calibrated base plane.

`timestamp`: timestamp of the image set the calibration ran on.

`return_code`: holds possible warnings or error codes and messages.

`get_base_plane_calibration`

Returns the configured base-plane calibration.

Request:

The definition for the request arguments with corresponding datatypes is:

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

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 7.3.7).

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 7.3.7).

Response:

The definition for the response with corresponding datatypes is:

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

`delete_base_plane_calibration`

Deletes the configured base-plane calibration.

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

set_region_of_interest_2d

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

The definition for the request arguments with corresponding datatypes is:

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

`region_of_interest_2d`: see *Setting a region of interest* (Section 7.3.3).

The definition for the response with corresponding datatypes is:

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

get_regions_of_interest_2d

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

The definition for the request arguments with corresponding datatypes is:

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

The definition for the response with corresponding datatypes is:

```
{
  "regions_of_interest": [
    {
      "height": "uint32",
      "id": "string",
      "offset_x": "uint32",
      "offset_y": "uint32",
      "width": "uint32"
```

(continues on next page)

(continued from previous page)

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

delete_regions_of_interest_2d

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

The definition for the request arguments with corresponding datatypes is:

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

The definition for the response with corresponding datatypes is:

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

set_load_carrier

see *set_load_carrier* (Section 7.5.1).

get_load_carriers

see *get_load_carriers* (Section 7.5.1).

delete_load_carriers

see *delete_load_carriers* (Section 7.5.1).

detect_load_carriers

see *detect_load_carriers* (Section 7.5.1).

detect_filling_level

see *detect_filling_level* (Section 7.5.1).

set_preferred_orientation

Persistently stores the preferred orientation of the gripper to compute the reachability of the grasps, which is used for filtering and sorting the grasps returned by the `detect_object` service (see *Setting the preferred orientation of the TCP*, Section 7.3.5).

The definition for the request arguments with corresponding datatypes is:

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

The definition for the response with corresponding datatypes is:

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

get_preferred_orientation

Returns the preferred orientation of the gripper to compute the reachability of the grasps, which is used for filtering and sorting the grasps returned by the `detect_object` service (see *Setting the preferred orientation of the TCP*, Section 7.3.5).

This service has no arguments.

The definition for the response with corresponding datatypes is:

```
{
  "orientation": {
    "w": "float64",
    "x": "float64",
    "y": "float64",
    "z": "float64"
  },
  "pose_frame": "string",
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

set_grasp

Persistently stores a grasp for the given object template on the *rc_visard*. All configured grasps are persistent over firmware updates and rollbacks.

The definition for the request arguments with corresponding datatypes is:

```
{
  "grasp": {
```

(continues on next page)

(continued from previous page)

```
    "id": "string",
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "template_id": "string"
  }
}
```

The definition for the response with corresponding datatypes is:

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

Details for the definition of the `grasp` type are given in *Setting of grasp points* (Section 7.3.4).

`set_all_grasps`

Replaces the list of grasps for the given object template on the *rc_visard*.

The definition for the request arguments with corresponding datatypes is:

```
{
  "grasps": [
    {
      "id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "template_id": "string"
    }
  ],
  "template_id": "string"
}
```

The definition for the response with corresponding datatypes is:

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

Details for the definition of the `grasp` type are given in *Setting of grasp points* (Section 7.3.4).

get_grasps

Returns all configured grasps which have the requested `grasp_ids` and belong to the requested `template_ids`. If no `grasp_ids` are provided, all grasps belonging to the requested `template_ids` are returned. If no `template_ids` are provided, all grasps with the requested `grasp_ids` are returned. If neither IDs are provided, all configured grasps are returned.

The definition for the request arguments with corresponding datatypes is:

```
{
  "grasp_ids": [
    "string"
  ],
  "template_ids": [
    "string"
  ]
}
```

The definition for the response with corresponding datatypes is:

```
{
  "grasps": [
    {
      "id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "template_id": "string"
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

delete_grasps

Deletes all grasps with the requested `grasp_ids` that belong to the requested `template_ids`. If no `grasp_ids` are provided, all grasps belonging to the requested `template_ids` are deleted. The `template_ids` list must not be empty.

The definition for the request arguments with corresponding datatypes is:

```
{
  "grasp_ids": [
    "string"
  ],
  "template_ids": [
    "string"
  ]
}
```

The definition for the response with corresponding datatypes is:

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

get_symmetric_grasps

Returns all grasps that are symmetric to the given grasp. The first grasp in the returned list is the one that was passed with the service call. If the object template does not have an exact symmetry, only the grasp passed with the service call will be returned. If the object template has a continuous symmetry (e.g. a cylindrical object), only 12 equally spaced sample grasps will be returned.

The definition for the request arguments with corresponding datatypes is:

```
{
  "grasp": {
    "id": "string",
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "template_id": "string"
  }
}
```

The definition for the response with corresponding datatypes is:

```
{
  "grasps": [
    {
      "id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
```

(continues on next page)

(continued from previous page)

```
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "template_id": "string"
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```

Details for the definition of the `grasp` type are given in *Setting of grasp points* (Section 7.3.4).

`detect_object`

Triggers an object detection as described in *Detection of objects* (Section 7.3.6) and returns the pose of all found object instances. The maximum number of returned instances can be controlled with the `max_number_of_detected_objects` parameter.

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

Request:

The definition for the request arguments with corresponding datatypes is:

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

Required arguments:

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

`pose_frame`: see *Hand-eye calibration* (Section 7.3.7).

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 7.3.7).

Optional arguments:

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

`load_carrier_id`: ID of the load carrier which contains the items to be detected.

`collision_detection`: see *Collision checking within other modules* (Section 7.4.3). The collision check requires a separate CollisionCheck *license* (Section 9.6) to be purchased.

Response:

The definition for the response with corresponding datatypes is:


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


(continues on next page)

(continued from previous page)


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


`object_id`: ID of the detected template.

`instances`: list of detected object instances.

`grasps`: list of grasps on the detected objects. The grasps are ordered by their reachability

starting with the grasp that can be reached most easily by the robot. The `instance_uuid` gives the reference to the detected object in `instances` this grasp belongs to.

`load_carriers`: list of detected load carriers.

`timestamp`: timestamp of the image set the detection ran on.

`return_code`: holds possible warnings or error codes and messages.

save_parameters

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

reset_defaults

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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

7.3.11 Template Upload

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

`GET /nodes/rc_silhouettematch/templates`
Get list of all rc_silhouettematch templates.

Template request

```
GET /api/v1/nodes/rc_silhouettematch/templates HTTP/1.1
```

Template response

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

[
  {
```

(continues on next page)

(continued from previous page)

```
    "id": "string"
  }
]
```

Response Headers

- Content-Type – application/json

Status Codes

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

Referenced Data Models

- *Template* (Section 8.2.3)

GET /nodes/rc_silhouettematch/templates/{id}

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

Template request

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

Template response

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

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

Parameters

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

Response Headers

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

Status Codes

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

Referenced Data Models

- *Template* (Section 8.2.3)

PUT /nodes/rc_silhouettematch/templates/{id}

Create or update a rc_silhouettematch template.

Template request

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

Template response

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

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

Parameters

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

Form Parameters

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

Request Headers

- Accept – multipart/form-data application/json

Response Headers

- Content-Type – application/json

Status Codes

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

Referenced Data Models

- *Template* (Section 8.2.3)

DELETE /nodes/rc_silhouettematch/templates/{id}

Remove a rc_silhouettematch template.

Template request

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

Parameters

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

Request Headers

- Accept – application/json

Response Headers

- Content-Type – application/json

Status Codes

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

7.4 CollisionCheck

7.4.1 Introduction

The CollisionCheck component is an optional on-board component of the *rc_visard*.

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

The component provides an easy way to check if a gripper is in collision with a load carrier. It is integrated with the *ItemPick and BoxPick* (Section 7.2) and *SilhouetteMatch* (Section 7.3) modules, but can be used as standalone product.

Warning: Collisions are checked only between the load carrier and the gripper, not the robot itself, the flange, other items, or the item located in the robot gripper.

7.4.2 Setting a gripper

The gripper is a collision geometry used to determine whether the grasp is in collision with the load carrier. The gripper consists of up to 15 elements connected to each other.

At this point, the gripper can be built of elements of the following types:

- `BOX`, with dimensions `box.x`, `box.y`, `box.z`.
- `CYLINDER`, with radius `cylinder.radius` and height `cylinder.height`.

Additionally, for each gripper the flange radius, and information about the Tool Center Point (TCP) have to be defined.

The configuration of the gripper is normally performed offline during the setup of the desired application. This can be done via the *REST-API interface* (Section 8.2) or the *rc_visard Web GUI* (Section 4.6).

Robot flange radius

Collisions are checked only between the gripper and the load carrier. The robot body is not considered. As a safety feature, to prevent collisions between the load carrier and the robot, all grasps having any part of the robot's flange inside the load carrier can be designated as colliding (see Fig. 7.6). This check is based on the defined gripper geometry and the flange radius value. It is optional to use this functionality, and it can be turned on and off with the run-time parameter `check_flange` as described in *Parameter overview* (Section 7.4.4).

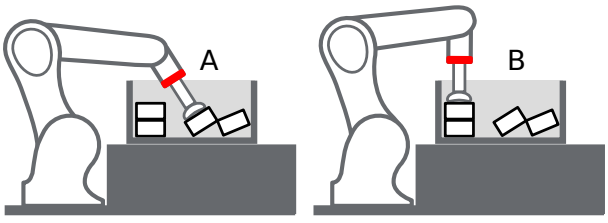


Fig. 7.6: Case A would be marked as collision only if `check_flange` is true, because the robot's flange (red) is inside the load carrier. Case B is collision free independent of `check_flange`.

Creating a gripper via the REST-API

When creating a gripper via the *REST-API interface* (Section 8.2), each element of the gripper has a *parent* element, which defines how they are connected. The gripper is always built in the direction from the robot flange to the TCP, and at least one element must have 'flange' as parent. The elements' IDs must be unique and must not be

'tcp' or 'flange'. The pose of the child element has to be given in the coordinate frame of the parent element. In the REST-API representation, the coordinate frame of an element is always in its geometric center. Accordingly, for a child element to be exactly below the parent element, the position of the child element must be computed from the heights of both parent and child element (see Fig. 7.7).

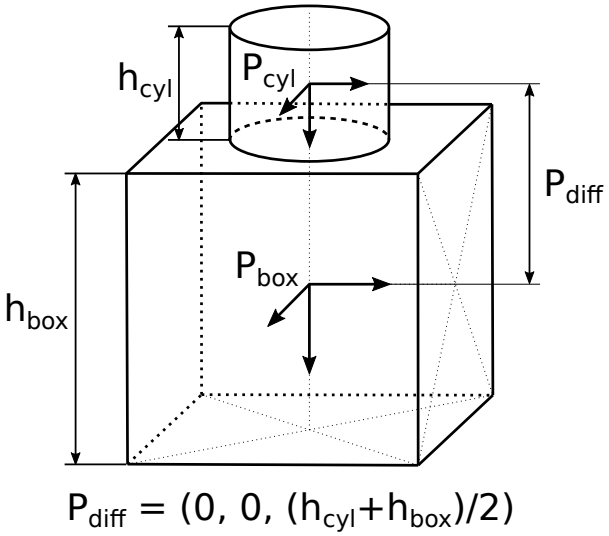


Fig. 7.7: Reference frames for gripper creation via the REST-API

The reference frame for the first element for the gripper creation is always the center of the robot's flange with the z axis pointing outwards. Via the REST-API it is possible to create a gripper with a tree structure, corresponding to multiple elements having the same parent element, as long as they are all connected.

Creating a gripper in the Web GUI

The tab *CollisionCheck* on the *rc_visard Web GUI* (Section 4.6) offers a simplified interface to create grippers. It is possible to select the type, the size, as well as the position of each element. In the Web GUI representation the position of each element originates from the bottom of the parent element. Therefore, a child element with position (0, 0, 0) will always be placed exactly below its parent element, irrespective of the elements' heights. Grippers which have a tree structure or which have rotated elements cannot be created via the Web GUI.

Calculated TCP position

After gripper creation via the `set_gripper` service call, the TCP position in the flange coordinate system is calculated and returned as `tcp_pose_flange`. It is important to check if this value is the same as the robot's true TCP position.

Creating rotationally asymmetric grippers

For grippers which are not rotationally symmetric around the z axis, it is crucial to ensure that the gripper is properly mounted, so that the representation stored in the CollisionCheck module corresponds to reality.

7.4.3 Collision checking

Stand-alone collision checking

The `check_collision` service call triggers collision checking between the chosen gripper and the provided load carriers for each of the provided grasps. The CollisionCheck module checks if the chosen gripper is in collision with at least one of the load carriers, when the TCP of the gripper is positioned in the grasp position. It is possible

to check the collision with multiple load carriers simultaneously. The grasps which are in collision with any of the defined load carriers will be returned as colliding.

The `pre_grasp_offset` can be used for additional collision checking. The pre-grasp offset P_{off} is the offset between the grasp point P_{grasp} and the pre-grasp position P_{pre} in the grasp's coordinate frame (see Fig. 7.8). If the pre-grasp offset is defined, the grasp will be detected as colliding if the gripper is in collision with the load carrier at any point during motion from the pre-grasp position to the grasp position (assuming a linear movement).

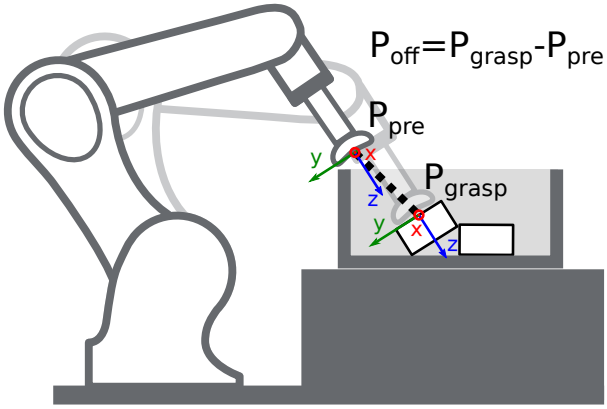


Fig. 7.8: Illustration of the pre-grasp offset parameter for collision checking. In this case, the pre-grasp position as well as the grasp position are collision free. However, the trajectory between these poses would have collisions. Thus, this grasp pose would be marked as colliding.

Collision checking within other modules

Collision checking is integrated in the following modules' services:

- *ItemPick and BoxPick* (Section 7.2): `compute_grasps` (see *compute_grasps for ItemPick*, Section 7.2.7 and *compute_grasps for BoxPick*, Section 7.2.7)
- *SilhouetteMatch* (Section 7.3): `detect_object` (see *detect_object*, Section 7.3.10)

Each of these services can take a `collision_detection` argument consisting of the `gripper_id` of the gripper and optionally the `pre_grasp_offset` as described in the previous section *Stand-alone collision checking* (Section 7.4.3). When the `collision_detection` argument is given, these services only return the grasps at which the gripper is not in collision with the load carrier detected by these services. For this, a load carrier ID has to be provided to these services as well.

Warning: Collisions are checked only between the load carrier and the gripper, not the robot itself, the flange, other objects or the item located in the robot gripper.

The collision-check results are affected by run-time parameters, which are listed and explained further below.

7.4.4 Parameters

The CollisionCheck component is called `rc_collision_check` in the REST-API and is represented by the *CollisionCheck* page in the *Modules* tab of the *Web GUI* (Section 4.6). The user can explore and configure the `rc_collision_check` component's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 8.2).

Parameter overview

This component offers the following run-time parameters:

Table 7.13: The rc_collision_check component's run-time parameters

Name	Type	Min	Max	Default	Description
check_bottom	bool	false	true	true	Check collisions with the bottom of the load carrier
check_flange	bool	false	true	true	Position is in collision if robot flange is inside the load carrier
collision_dist	float64	0.0	0.1	0.01	Minimal distance in meters between any part of the gripper and any of the load carrier's walls for grasp to be considered collision free

Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's Module tab. The name in the Web GUI is given in brackets behind the parameter name:

collision_dist (*Collision Distance*) Minimal distance in meters between any part of the gripper and any of the load carrier's walls for a grasp to be considered collision free.

check_flange (*Check Flange*) Performs an additional safety check as described in *Robot flange radius* (Section 7.4.2). If this parameter is set, all positions in which any part of the robot's flange is inside the load carrier are marked as colliding.

check_bottom (*Check Bottom*) When this check is enabled the collisions will be checked not only with the side walls of the load carrier but also with its bottom. It might be necessary to disable this check if the TCP is inside the collision geometry (e.g. is defined inside a suction cup).

7.4.5 Status values

The rc_collision_check component reports the following status values:

Table 7.14: The rc_collision_check component status values

Name	Description
last_evaluated_grasps	Number of evaluated grasps
last_collision_free_grasps	Number of collision-free grasps

7.4.6 Services

The user can explore and call the rc_collision_check component's services, e.g. for development and testing, using *REST-API interface* (Section 8.2) or the *rc_visard Web GUI* (Section 4.6).

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

The following table contains a list of common codes:

Table 7.15: Return codes of the CollisionCheck services

Code	Description
0	Success
-1	An invalid argument was provided
-7	Data could not be read or written to persistent storage
-9	No valid license for the module
-10	New gripper could not be added as the maximum storage capacity of grippers has been exceeded
10	The maximum storage capacity of grippers has been reached
11	Existing gripper was overwritten

The CollisionCheck component offers the following services.

set_gripper

Persistently stores a gripper on the *rc_visard*. All configured grippers are persistent over firmware updates and rollbacks.

Request:

The definition for the request arguments with corresponding datatypes is:


```
{
  "elements": [
    {
      "box": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "cylinder": {
        "height": "float64",
        "radius": "float64"
      },
      "id": "string",
      "parent_id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "type": "string"
    }
  ],
  "flange_radius": "float64",
  "id": "string",
  "tcp_parent_id": "string",
  "tcp_pose_parent": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
```


(continues on next page)

(continued from previous page)

```
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  }
}
```

Required arguments:

`elements`: list of geometric elements for the gripper. Each element must be of `type` 'CYLINDER' or 'BOX' with the corresponding dimensions in the `cylinder` or `box` field. The pose of each element must be given in the coordinate frame of the parent element (see *Setting a gripper* (Section 7.4.2) for an explanation of the coordinate frames). The element's `id` must be unique and must not be 'tcp' or 'flange'. The `parent_id` is the ID of the parent element. It can either be 'flange' or it must correspond to another element in list.

`flange_radius`: radius of the flange used in case the `check_flange` run-time parameter is active.

`id`: unique name of the gripper

`tcp_parent_id`: ID of the element on which the TCP is defined

`tcp_pose_parent`: The pose of the TCP with respect to the coordinate frame of the element specified in `tcp_parent_id`.

Response:

The definition for the response with corresponding datatypes is:

```
{
  "gripper": {
    "elements": [
      {
        "box": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "cylinder": {
          "height": "float64",
          "radius": "float64"
        },
        "id": "string",
        "parent_id": "string",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "type": "string"
      }
```

(continues on next page)

(continued from previous page)


```
    ],
    "flange_radius": "float64",
    "id": "string",
    "tcp_parent_id": "string",
    "tcp_pose_flange": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "tcp_pose_parent": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "type": "string"
  },
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```


`gripper`: returns the gripper as defined in the request with an additional field `tcp_pose_flange`. This gives the coordinates of the TCP in the flange coordinate frame for comparison with the true settings of the robot's TCP.

`return_code`: holds possible warnings or error codes and messages.

get_grippers

Returns the configured grippers with the requested `gripper_ids`. If no `gripper_ids` are provided, all configured grippers are returned.

The definition for the request arguments with corresponding datatypes is:


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


The definition for the response with corresponding datatypes is:

```
{
  "grippers": [
    {
      "elements": [
        {
          "box": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "cylinder": {
            "height": "float64",
            "radius": "float64"
          },
          "id": "string",
          "parent_id": "string",
          "pose": {
            "orientation": {
              "w": "float64",
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "position": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            }
          },
          "type": "string"
        }
      ],
      "flange_radius": "float64",
      "id": "string",
      "tcp_parent_id": "string",
      "tcp_pose_flange": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "tcp_pose_parent": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "type": "string"
```


(continues on next page)

(continued from previous page)

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

delete_grippers

Deletes the configured grippers with the requested `gripper_ids`. All grippers to be deleted must be explicitly stated in `gripper_ids`.

The definition for the request arguments with corresponding datatypes is:

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

The definition for the response with corresponding datatypes is:

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

check_collision

Triggers a collision check.

Request:

The definition for the request arguments with corresponding datatypes is:

```
{
  "grasps": [
    {
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "uuid": "string"
    }
  ],
  "gripper_id": "string",
```

(continues on next page)

(continued from previous page)

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

Required arguments:

`grasps`: list of grasps that should be checked.

`load_carriers`: list of load carriers against which the collision should be checked. The fields of the load carrier definition are described in *Detection of load carriers* (Section 7.5.1). The position frame of the grasps and load carriers has to be the same.

`gripper_id`: the id of the gripper that is used to check the collisions. The gripper has to be configured beforehand.

Optional arguments:

`pre_grasp_offset`: the offset in meters from the grasp position to the pre-grasp position in the grasp frame. If this argument is set, the collisions will not only be checked in the grasp point, but also on the path from the pre-grasp position to the grasp position (assuming a linear movement).

Response:

The definition for the response with corresponding datatypes is:

```
{
  "colliding_grasps": [
    {
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "uuid": "string"
    }
  ],
  "collision_free_grasps": [
    {
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "uuid": "string"
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  }
}
```


`colliding_grasps`: list of grasps in collision with one or more load carriers.

`collision_free_grasps`: list of collision-free grasps.

`return_code`: holds possible warnings or error codes and messages.

save_parameters

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

This service has no arguments.

The definition for the response with corresponding datatypes is:

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


reset_defaults

This service resets all parameters of the component to its default values, as listed in above table. The reset does not apply to grippers.

This service has no arguments.

The definition for the response with corresponding datatypes is:


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


7.5 Common functionalities

The *rc_visard* provides some common functionalities which are used in multiple optional software components. The common functionalities are:

- ***Load carrier functionality*** **(Section 7.5.1)** allows setting and retrieving load carriers, as well as detecting load carriers and their filling level.
- ***Region of interest functionality*** **(Section 7.5.2)** allows setting and retrieving 3D regions of interest.

All services and parameters of the common functionalities are only available through the software component that uses them. All stored load carriers and regions of interest are available to all software components that use this respective functionality.

7.5.1 Load carrier functionality

Introduction

The load carrier functionality is contained in an internal load carrier component and can only be used through the software components providing a load carrier functionality.

The load carrier functionality is provided by the *ItemPick and BoxPick* (Section 7.2) and *SilhouetteMatch* (Section 7.3) modules.

Load Carrier

A load carrier (bin) is a container with four walls, a floor and a rectangular rim, which can contain objects.

A load carrier is defined by its `outer_dimensions` and `inner_dimensions`. The maximum `outer_dimensions` are 2.0 meters in every dimension.

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

The *rc_visard* can persistently store up to 50 different load carrier models, each one identified by a different `id`. The configuration of a load carrier model is normally performed offline, during the set up the desired application. This can be done via the *REST-API interface* (Section 8.2) or in the *rc_visard* Web GUI.

Note: The configured load carrier models are persistent even over firmware updates and rollbacks.

Detection of load carriers

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

The origin of a detected load carrier is in the center of the load carrier outer box and its z axis is perpendicular to the load carrier floor. The detection functionality also determines if the detected load carrier is `overfilled`.

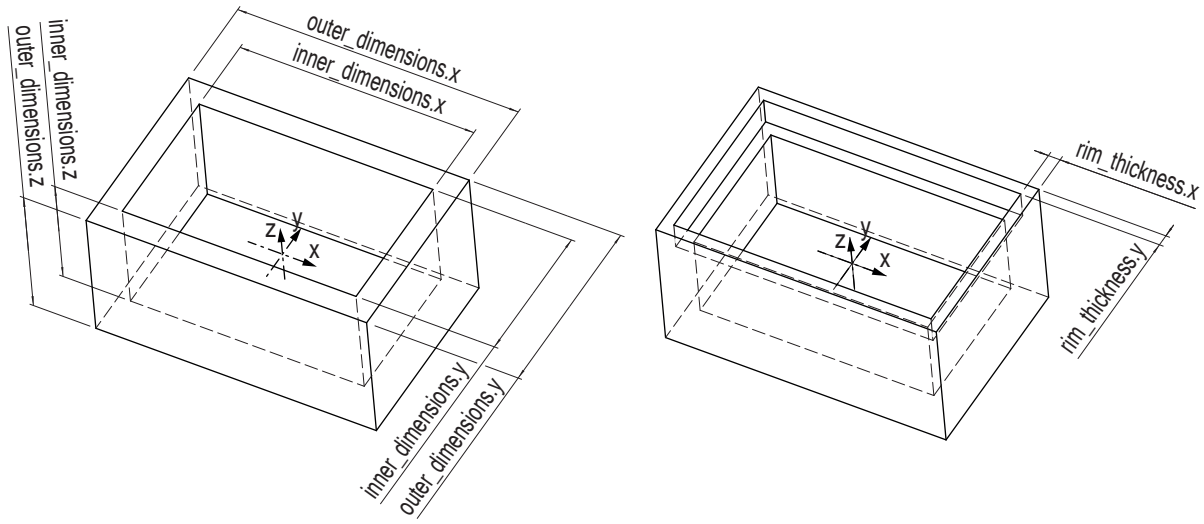


Fig. 7.9: Load carrier models and reference frame.

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

Detection of filling level

The load carrier functionality contains the `detect_filling_level` service to compute the filling level of a detected load carrier.

The load carrier is subdivided in a configurable number of cells in a 2D grid. The maximum number of cells is 10x10. For each cell, the following values are reported:

- `level_in_percent`: minimum, maximum and mean cell filling level in percent from the load carrier floor. These values can be larger than 100% if the cell is overfilled.
- `level_free_in_meters`: minimum, maximum and mean cell free level in meters from the load carrier rim. These values can be negative if the cell is overfilled.
- `cell_size`: dimensions of the 2D cell in meters.
- `cell_position`: position of the cell center in meters (either in `camera` or `external` frame, see *Hand-eye calibration*, Section 7.5.1). The z-coordinate is on the level of the load carrier rim.

- `coverage`: represents the proportion of valid pixels in this cell. It varies between 0 and 1 with steps of 0.1. A low coverage indicates that the cell contains several missing data (i.e. only a few points were actually measured in this cell).

These values are also calculated for the whole load carrier itself. If no cell subdivision is specified, only the overall filling level is computed.

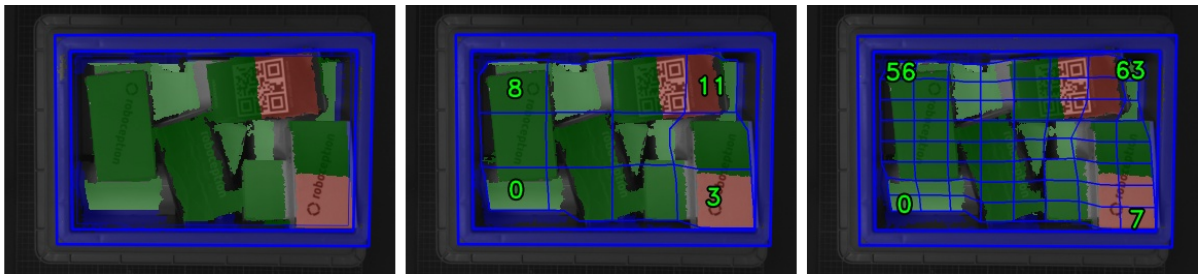


Fig. 7.10: Visualizations of the load carrier filling level: overall filling level without grid (left), 4x3 grid (center), 8x8 grid (right). The load carrier content is shown in a green gradient from white (on the load carrier floor) to dark green. The overfilled regions are visualized in red. Numbers indicate cell IDs.

Interaction with other components

Internally, the load carrier functionality depends on, and interacts with other on-board components as listed below.

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

Stereo camera and Stereo matching

The load carrier component makes internally use of the following data:

- Rectified images from the *Stereo camera* component (`rc_stereocamera`, Section 6.1);
- Disparity, error, and confidence images from the *Stereo matching* component (`rc_stereomatching`, Section 6.2).

All processed images are guaranteed to be captured after the component trigger time.

Estimation of gravity vector

For each load carrier detection, the component estimates the gravity vector by subscribing to the *rc_visard*'s IMU data stream.

Note: The gravity vector is estimated from linear acceleration readings from the on-board IMU. For this reason, the load carrier component requires the *rc_visard* to remain still while the gravity vector is being estimated.

IO and Projector Control

In case the *rc_visard* is used in conjunction with an external random dot projector and the *IO and Projector Control* component (`rc_iocontrol`, Section 6.8), it is recommended to connect the projector to GPIO Out 1 and set the stereo-camera component's acquisition mode to `SingleFrameOut1` (see *Stereo matching parameters*, Section 6.2.4), so that on each image acquisition trigger an image with and without projector pattern is acquired.

Alternatively, the output mode for the GPIO output in use should be set to `ExposureAlternateActive` (see *Description of run-time parameters*, Section 6.8.1).

In either case, the *Auto Exposure Mode* `exp_auto_mode` should be set to `AdaptiveOut1` to optimize the exposure of both images (see *Stereo camera parameters*, Section 6.1.4).

No additional changes are required to use the load carrier component in combination with a random dot projector.

Hand-eye calibration

In case the camera has been calibrated to a robot, the loadcarrier component can automatically provide poses in the robot coordinate frame. For the loadcarrier nodes' *Services* (Section 7.5.1), the frame of the output poses can be controlled with the `pose_frame` argument.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). All poses provided by the components are in the camera frame, and no prior knowledge about the pose of the camera in the environment is required. This means that the configured load carriers move with the camera. It is the user's responsibility to update the configured poses if the camera frame moves (e.g. with a robot-mounted camera).
2. **External frame** (`external`). All poses provided by the components are in the external frame, configured by the user during the hand-eye calibration process. The component relies on the on-board *Hand-eye calibration component* (Section 6.7) to retrieve the sensor mounting (static or robot mounted) and the hand-eye transformation. If the mounting is static, no further information is needed. If the sensor is robot-mounted, the `robot_pose` is required to transform poses to and from the `external` frame.

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

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

Parameters

The load carrier functionality is used internally by several other components and their parameters and services are provided through these nodes. They can also be used in the *Web GUI* (Section 4.6) on the page of the corresponding module. The user can explore and configure the load carrier component's run-time parameters, e.g. for development and testing, using the corresponding module page in the Web GUI or the *REST-API interface* (Section 8.2).

Parameter overview

This component offers the following run-time parameters:

Table 7.16: The load carrier component's parameters

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

Description of run-time parameters

Each run-time parameter is represented by a row on the Settings section of the Web GUI's module page in the subsection *Load Carrier Detection Parameters*. 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:

`load_carrier_model_tolerance` (*Model Tolerance*) indicates how much the estimated load carrier dimensions are allowed to differ from the load carrier model dimensions in meters.

`load_carrier_crop_distance` (*Crop Distance*) sets the safety margin in meters by which the load carrier's inner dimensions are reduced to define the region of interest for detection.

Services

The user can explore and call the load carrier component's services, e.g. for development and testing, using the *REST-API interface* (Section 8.2) or the *rc_visard Web GUI* (Section 4.6) on the page of the module offering the load carrier functionality.

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

The following table contains a list of common codes:

Table 7.17: Return codes of the load carrier services

Code	Description
0	Success
-1	An invalid argument was provided
-4	Data acquisition took longer than the maximum allowed time of 5.0 seconds
-10	New element could not be added as the maximum storage capacity of load carriers has been exceeded
-302	More than one load carrier provided to the `detect_load_carriers` or `detect_filling_level` services, but only one is supported
10	The maximum storage capacity of load carriers has been reached
11	An existent persistent model was overwritten by the call to `set_load_carrier`
100	The requested load carriers were not detected in the scene
102	The detected load carrier is empty
300	A valid `robot_pose` was provided as argument but it is not required

All software components providing the load carrier functionality offer the following services.

`set_load_carrier`

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

The definition for the request arguments with corresponding datatypes is:

```
{
  "load_carrier": {
    "id": "string",
    "inner_dimensions": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "outer_dimensions": {
```

(continues on next page)

(continued from previous page)

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

Details for the definition of the `load_carrier` type are given in *Detection of load carriers* (Section 7.5.1).

The definition for the response with corresponding datatypes is:

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

get_load_carriers

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

The definition for the request arguments with corresponding datatypes is:

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

The definition for the response with corresponding datatypes is:

```
{
  "load_carriers": [
    {
      "id": "string",
      "inner_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
```

(continues on next page)

(continued from previous page)


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


delete_load_carriers

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

The definition for the request arguments with corresponding datatypes is:


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


The definition for the response with corresponding datatypes is:


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


detect_load_carriers

Triggers a load carrier detection as described in *Detection of load carriers* (Section 7.5.1).

Request:

The definition for the request arguments with corresponding datatypes is:


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


Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 7.5.1).

`load_carrier_ids`: IDs of the load carriers which should be detected.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 7.5.1).

Optional arguments:

`region_of_interest_id`: ID of the region of interest where to search for the load carriers.

Response:

The definition for the response with corresponding datatypes is:


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


(continues on next page)

(continued from previous page)


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


`load_carriers`: list of detected load carriers.

`timestamp`: timestamp of the image set the detection ran on.

`return_code`: holds possible warnings or error codes and messages.

`detect_filling_level`

Triggers a load carrier filling level detection as described in *Detection of filling level* (Section 7.5.1).

Request:

The definition for the request arguments with corresponding datatypes is:


```
{
  "filling_level_cell_count": {
    "x": "uint32",
    "y": "uint32"
  },
  "load_carrier_ids": [
    "string"
  ],
  "pose_frame": "string",
  "region_of_interest_id": "string",
  "robot_pose": {
    "orientation": {
      "w": "float64",
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "position": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    }
  }
}
```


Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 7.5.1).

`load_carrier_ids`: IDs of the load carriers which should be detected.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 7.5.1).

Optional arguments:

`region_of_interest_id`: ID of the region of interest where to search for the load carriers.

`filling_level_cell_count`: Number of cells in the filling level grid.

Response:

The definition for the response with corresponding datatypes is:


```
{
  "load_carriers": [
    {
      "cells_filling_levels": [
        {
          "cell_position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "cell_size": {
            "x": "float64",
            "y": "float64"
          },
          "coverage": "float64",
          "level_free_in_meters": {
            "max": "float64",
            "mean": "float64",
            "min": "float64"
          },
          "level_in_percent": {
            "max": "float64",
            "mean": "float64",
            "min": "float64"
          }
        }
      ],
      "filling_level_cell_count": {
        "x": "uint32",
        "y": "uint32"
      },
      "id": "string",
      "inner_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "outer_dimensions": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "overall_filling_level": {
        "cell_position": {
          "x": "float64",
          "y": "float64",
```


(continues on next page)

(continued from previous page)


```
          "z": "float64"
        },
        "cell_size": {
          "x": "float64",
          "y": "float64"
        },
        "coverage": "float64",
        "level_free_in_meters": {
          "max": "float64",
          "mean": "float64",
          "min": "float64"
        },
        "level_in_percent": {
          "max": "float64",
          "mean": "float64",
          "min": "float64"
        }
      },
      "overfilled": "bool",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "rim_thickness": {
        "x": "float64",
        "y": "float64"
      }
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
    "nsec": "int32",
    "sec": "int32"
  }
}
```


`load_carriers`: list of detected load carriers and their filling level.

`timestamp`: timestamp of the image set the detection ran on.

`return_code`: holds possible warnings or error codes and messages.

7.5.2 Region of interest functionality

Introduction

The region of interest (ROI) functionality is contained in an internal ROI component and can only be used through the software components providing a ROI functionality.

The ROI functionality is provided by the *ItemPick and BoxPick* (Section 7.2) modules.

Region of interest

A region of interest (ROI) defines a volume in space which is of interest for a specific user-application. A ROI can narrow the volume where a load carrier is searched for, or select a volume which only contains items to be detected and/or grasped. Processing times can significantly decrease when using a ROI.

Currently, regions of interest of the following types (`type`) are supported:

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

The user can specify the region of interest `pose` in the `camera` or the `external` coordinate system. `External` can only be chosen if a *Hand-eye calibration* (Section 6.7) is available. When the *rc_visard* is robot mounted, and the region of interest is defined in the external frame, the current robot pose must be given to every detect service call that uses this region of interest.

The *rc_visard* can persistently store up to 50 different regions of interest, each one identified by a different `id`. The configuration of regions of interest is normally performed offline, during the set up of the desired application. This can be done via the *REST-API interface* (Section 8.2) or in the *rc_visard* Web GUI on the page of the module offering the region of interest functionality.

Note: The configured regions of interest are persistent even over firmware updates and rollbacks.

Parameters

The ROI component does not have any parameters.

Services

The user can explore and call the ROI component's services, e.g. for development and testing, using the *REST-API interface* (Section 8.2) or the *rc_visard Web GUI* (Section 4.6) page of the module offering the ROI functionality.

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

The following table contains a list of common codes:

Table 7.18: Return codes of the region of interest services

Code	Description
0	Success
-1	An invalid argument was provided
-10	New element could not be added as the maximum storage capacity of regions of interest has been exceeded
10	The maximum storage capacity of regions of interest has been reached
11	An existent persistent model was overwritten by the call to `set_region_of_interest`

All software components providing the ROI functionality offer the following services.

set_region_of_interest

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

The definition for the request arguments with corresponding datatypes is:

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

Details for the definition of the `region_of_interest` type are given in *Region of interest* (Section 7.5.2).

The definition for the response with corresponding datatypes is:

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

get_regions_of_interest

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

The definition for the request arguments with corresponding datatypes is:

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

The definition for the response with corresponding datatypes is:

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


delete_regions_of_interest

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

The definition for the request arguments with corresponding datatypes is:


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


The definition for the response with corresponding datatypes is:


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

8 Interfaces

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

- *GigE Vision 2.0/GenICam* (Section 8.1)

  Images and camera related configuration.

- *REST API* (Section 8.2)

  API to configure the *rc_visard*, query status information, do service calls, etc.

- *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.

- *Ethernet KRL Interface (EKI)* (Section 8.4)

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

- *Time synchronization* (Section 8.5)

  Time synchronization between the *rc_visard* and the application host.

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 GigE Vision ports

GigE Vision is a UDP based protocol. On the *rc_visard* the UDP ports are fixed and known:

- UDP port 3956: GigE Vision Control Protocol (GVCP). Used for discovery, control and configuration.
- UDP port 50010: Stream channel source port for GigE Vision Stream Protocol (GVSP) used for image streaming.

8.1.2 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.

8.1.3 Important standard GenICam features

Category: ImageFormatControl

`ComponentSelector`

- type: Enumeration, one of `Intensity`, `IntensityCombined`, `Disparity`, `Confidence`, or `Error`
- default: -
- description: Allows the user to select one of the five image streams for configuration (see *Provided image streams*, Section 8.1.6).

`ComponentIDValue` (read-only)

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

`ComponentEnable`

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

`Width` (read-only)

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

`Height` (read-only)

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

`WidthMax` (read-only)

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

`HeightMax` (read-only)

- type: Integer

- description: Maximum height of an image in the streams. This is always 1920 pixels due to the stacked left and right images in the `IntensityCombined` stream (see *Provided image streams*, Section 8.1.6).

`PixelFormat`

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

Category: AcquisitionControl

`AcquisitionFrameRate`

- type: Float, ranges from 1 Hz to 25 Hz
- default: 25 Hz
- description: Frame rate of the camera (*FPS*, Section 6.1.4).

`ExposureAuto`

- type: Enumeration, one of `Continuous`, `AdaptiveOut1` or `Off`
- default: `Continuous`
- description: Can be set to `Off` for manual exposure mode, to `Continuous` or `AdaptiveOut1` for *auto exposure* (Section 6.1.4). The value `Continuous` maps to the value *Normal* of the `exp_auto_mode` (*auto exposure mode*, Section 6.1.4) and `AdaptiveOut1` to the mode of the same name.

`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 (*Exposure*, Section 6.1.4).

Category: AnalogControl

`GainSelector` (read-only)

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

`Gain`

- type: Float, ranges from 0 dB to 18 dB
- default: 0 dB
- description: The cameras' gain value in decibel that is used in manual exposure mode (*Gain*, Section 6.1.4).

`BalanceWhiteAuto` (color cameras 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 cameras (*wb_auto*, Section 6.1.4).

`BalanceRatioSelector` (color cameras 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 cameras.

`BalanceRatio` (color cameras 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 cameras (*wb_ratio*, Section 6.1.4).

Category: DigitalIOControl

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

`LineSelector`

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

`LineStatus` (read-only)

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

`LineStatusAll` (read-only)

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

Table 8.1: Meaning of bits of `LineStatusAll` field.

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

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

- type: Enumeration, one of `ExposureActive`, `ExposureAlternateActive`, `Low` or `High`
- default: `Low`
- description: Mode for output line selected by `LineSelector` as described in the IOControl module (*out1_mode and out2_mode*, Section 6.8.1). See also parameter `AcquisitionAlternateFilter` for filtering images in `ExposureAlternateActive` mode.

Category: TransportLayerControl / PtpControl

`PtpEnable`

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

Category: Scan3dControl

`Scan3dDistanceUnit` (read-only)

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

`Scan3dOutputMode` (read-only)

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

`Scan3dFocalLength` (read-only)

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

`Scan3dBaseline` (read-only)

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

`Scan3dPrincipalPointU` (read-only)

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

`Scan3dPrincipalPointV` (read-only)

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

`Scan3dCoordinateScale` (read-only)

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

`Scan3dCoordinateOffset` (read-only)

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

`Scan3dInvalidDataFlag` (read-only)

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

`Scan3dInvalidDataValue` (read-only)

- type: Float

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

Category: ChunkDataControl

`ChunkModeActive`

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

8.1.4 Custom GenICam features of the *rc_visard*

Category: AcquisitionControl

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

- type: Enumeration, one of `Off`, `OnlyHigh` or `OnlyLow`
- default: `Off`
- description: If this parameter is set to `OnlyHigh` (or `OnlyLow`) and the `LineSource` is set to `ExposureAlternateActive` for any output, then only camera images are delivered that are captured while the output is high, i.e. a potentially connected projector is on (or low, i.e. a potentially connected projector is off). This parameter is a simple means for only getting images without projected pattern. The minimal time difference between camera and disparity images will be about 40 ms in this case (see *IOControl*, Section 6.8.1).

`AcquisitionMultiPartMode`

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

`ExposureTimeAutoMax`

- type: Float, ranges from 66 µs to 18000 µs
- default: 18000 µs
- description: Maximal exposure time in auto exposure mode (*Max Exposure*, Section 6.1.4).

`ExposureRegionOffsetX`

- type: Integer in the range of 0 to 1280
- default: 0
- description: Horizontal offset of *exposure region* (Section 6.1.4) in pixel.

`ExposureRegionOffsetY`

- type: Integer in the range of 0 to 960
- default: 0

- description: Vertical offset of *exposure region* (Section 6.1.4) in pixel.

`ExposureRegionWidth`

- type: Integer in the range of 0 to 1280
- default: 0
- description: Width of *exposure region* (Section 6.1.4) in pixel.

`ExposureRegionHeight`

- type: Integer in the range of 0 to 960
- default: 0
- description: Height of *exposure region* (Section 6.1.4) in pixel.

`RcExposureAutoAverageMax`

- type: Float in the range of 0 to 1
- default: 0.75
- description: Maximum brightness for the *auto exposure function* (Section 6.1.4) as value between 0 (dark) and 1 (bright).

`RcExposureAutoAverageMin`

- type: Float in the range of 0 to 1
- default: 0.25
- description: Minimum brightness for the *auto exposure function* (Section 6.1.4) as value between 0 (dark) and 1 (bright).

Category: Scan3dControl

`FocalLengthFactor` (read-only)

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

`Baseline` (read-only)

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

Category: DepthControl

`DepthAcquisitionMode`

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

`DepthAcquisitionTrigger`

- type: Command

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

`DepthQuality`

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

`DepthStaticScene`

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

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

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

`DepthFill`

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

`DepthSeg`

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

`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).

8.1.5 Chunk data

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

Particularly useful chunk parameters are:

- `ChunkComponentSelector` selects for which component to extract the chunk data in MultiPart mode.
- `ChunkComponentID` and `ChunkComponentIDValue` provide the relation of the image to its component (e.g. camera image or disparity image) without guessing from the image format or size.
- `ChunkLineStatusAll` provides the status of all GPIOs at the time of image acquisition. See `LineStatusAll` above for a description of bits.
- `ChunkScan3d...` parameters are useful for 3D reconstruction as described in Section *Image stream conversions* (Section 8.1.7).
- `ChunkPartIndex` provides the index of the image part in this MultiPart block for the selected component (`ChunkComponentSelector`).

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

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

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

8.1.7 Image stream conversions

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

Note: The *rc_visard*'s camera coordinate frame is defined as shown in *sensor coordinate frame* (Section 3.7).

Assuming that $d16_{ik}$ is the 16 bit disparity value at column i and row k of a disparity image, the float disparity in pixels d_{ik} is given by

$$d_{ik} = d16_{ik} \cdot \text{Scan3dCoordinateScale}$$

The 3D reconstruction in meters can be written with the GenICam parameters as:

$$\begin{aligned} P_x &= (i + 0.5 - \text{Scan3dPrincipalPointU}) \frac{\text{Scan3dBaseline}}{d_{ik}}, \\ P_y &= (k + 0.5 - \text{Scan3dPrincipalPointV}) \frac{\text{Scan3dBaseline}}{d_{ik}}, \\ P_z &= \text{Scan3dFocalLength} \frac{\text{Scan3dBaseline}}{d_{ik}}. \end{aligned}$$

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

$$\begin{aligned} d_{ik} &= d16_{ik} \cdot \text{Scan3dCoordinateScale}, \\ z_{eps} &= \frac{e_{ik} \cdot \text{Scan3dCoordinateScale} \cdot \text{Scan3dFocalLength} \cdot \text{Scan3dBaseline}}{(d_{ik})^2}. \end{aligned}$$

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

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

8.2 REST-API interface

Aside from 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 4.6), the REST-API serves rather as a machine-to-machine interface to the *rc_visard*, e.g., to programmatically

- set and get run-time parameters of computation nodes, e.g., of cameras or image processing components;
- do service calls, e.g., to start and stop individual computational nodes, or to use offered services such as the hand-eye calibration;

- read the current state of the system and individual computational nodes; or
- 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 hand-eye calibration node.

8.2.1 General API structure

The general **entry point** to the *rc_visard*'s API is `http://<host>/api/`, where `<host>` is either the device's IP address or its *host name* as known by the respective DHCP server, as explained in *network configuration* (Section 4.5). 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://<host>/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://<host>/api/v1/nodes/rc_stereomatching'
```

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

```
curl -X GET 'http://<host>/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://<host>/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://<host>/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 *rc_dynamics interface* (Section 8.3).
- **`/logs`** Access the log files on the *rc_visard*.
- **`/system`** Access the system state, set network configuration 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://<host>/api/v1/nodes
```

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

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

Its `quality` parameter could be set to `Full` using

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

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 4.6).

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 or Web GUI 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 the hand-eye calibration component* (Section 6.7.5) could be listed using

```
curl -X GET http://<host>/api/v1/nodes/rc_hand_eye_calibration/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, the stereo matching component can be triggered to do an acquisition by:

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

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
```

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",
```

(continues on next page)

(continued from previous page)

```
    "parameters": [],
    "services": [],
    "status": "stale"
  },
  {
    "name": "rc_stereomatching",
    "parameters": [
      "quality",
      "seg",
      "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
```

Sample response

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

{
```

(continues on next page)

(continued from previous page)

```
  "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
```

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
  },
```

(continues on next page)

(continued from previous page)

```
  {
    "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
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
  },
  {
```

(continues on next page)

(continued from previous page)


```
    "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** (*ParameterNameValue*) – 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)*
- 400 Bad Request – invalid parameter value
- 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.
- 404 Not Found – node not found

Referenced Data Models

- *Parameter* (Section 8.2.3)
- *ParameterNameValue* (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
```

Sample response


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

{
  "default": "H",
```


(continues on next page)

(continued from previous page)


```
  "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
Accept: application/json

{
  "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** (*ParameterValue*) – 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)*
- 400 Bad Request – invalid parameter value
- 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.
- 404 Not Found – node or parameter not found

Referenced Data Models

- *ParameterValue* (Section 8.2.3)
- *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
```

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
```

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

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
```

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 *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://<host>/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://<host>/api/v1/datastreams/pose
```

- and stopping data streams, e.g.,

```
curl -X DELETE http://<host>/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
```

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
```

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
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
```

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
```

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
```

Sample response

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

(continues on next page)

(continued from previous page)


```
{
  "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
```

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
```

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
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

GET /system/network

Get current network configuration.

Template request

```
GET /api/v1/system/network HTTP/1.1
```

Sample response

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

{
  "current_method": "DHCP",
  "default_gateway": "10.0.3.254",
  "ip_address": "10.0.1.41",
  "settings": {
    "dhcp_enabled": true,
    "persistent_default_gateway": "",
    "persistent_ip_address": "192.168.0.10",
    "persistent_ip_enabled": false,
    "persistent_subnet_mask": "255.255.255.0"
  },
  "subnet_mask": "255.255.252.0"
}
```

Response Headers

- Content-Type – application/json

Status Codes

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

Referenced Data Models

- *NetworkInfo* (Section 8.2.3)

GET /system/network/settings

Get current network settings.

Template request

```
GET /api/v1/system/network/settings HTTP/1.1
```

Sample response

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

{
  "dhcp_enabled": true,
  "persistent_default_gateway": "",
  "persistent_ip_address": "192.168.0.10",
  "persistent_ip_enabled": false,
  "persistent_subnet_mask": "255.255.255.0"
}
```

Response Headers

- Content-Type – application/json

Status Codes

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

Referenced Data Models

- *NetworkSettings* (Section 8.2.3)

PUT /system/network/settings

Set current network settings.

Template request

```
PUT /api/v1/system/network/settings HTTP/1.1
Accept: application/json

{}
```

Sample response

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

{
  "dhcp_enabled": true,
  "persistent_default_gateway": "",
  "persistent_ip_address": "192.168.0.10",
  "persistent_ip_enabled": false,
  "persistent_subnet_mask": "255.255.255.0"
}
```

Request JSON Object

- **settings** (*NetworkSettings*) – network settings to apply *(required)*

Request Headers

- Accept – application/json

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation *(returns NetworkSettings)*
- 400 Bad Request – invalid/missing arguments
- 403 Forbidden – Changing network settings forbidden because this is locked by a running GigE Vision application.

Referenced Data Models

- *NetworkSettings* (Section 8.2.3)

PUT /system/reboot

Reboot the sensor.

Template request

```
PUT /api/v1/system/reboot HTTP/1.1
```

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
```

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
```

Status Codes

- 200 OK – successful operation
- 400 Bad Request – already set to use inactive partition on next boot
- 500 Internal Server Error – internal error

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
```

Sample response

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

{
  "active_image": {
    "image_version": "rc_visard_v1.1.0"
```

(continues on next page)

(continued from previous page)

```
  },
  "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
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*.

LicenseComponentConstraint: Constraints on the component version.

An object of type LicenseComponentConstraint has the following properties:

- **max_version** (string) - optional maximum supported version (exclusive)
- **min_version** (string) - optional minimum supported version (inclusive)

Template object

```
{
  "max_version": "string",
  "min_version": "string"
}
```

LicenseComponentConstraint objects are nested in *LicenseConstraints*.

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*.

LicenseConstraints: Version constrains for components.

An object of type LicenseConstraints has the following properties:

- **image_version** (*LicenseComponentConstraint*) - see description of *LicenseComponentConstraint*

Template object


```
{
  "image_version": {
    "max_version": "string",
    "min_version": "string"
  }
}
```


LicenseConstraints 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*
- **components_constraints** (*LicenseConstraints*) - see description of *LicenseConstraints*
- **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
  },
  "components_constraints": {
    "image_version": {
      "max_version": "string",
      "min_version": "string"
    }
  },
  "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) - name 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*

NetworkInfo: Current network configuration.

An object of type NetworkInfo has the following properties:

- **current_method** (string) - method by which current settings were applied (one of `INIT`, `LinkLocal`, `DHCP`, `PersistentIP`, `TemporaryIP`)
- **default_gateway** (string) - current default gateway
- **ip_address** (string) - current IP address
- **settings** (*NetworkSettings*) - see description of *NetworkSettings*
- **subnet_mask** (string) - current subnet mask

Template object

```
{
  "current_method": "string",
  "default_gateway": "string",
  "ip_address": "string",
  "settings": {
    "dhcp_enabled": false,
    "persistent_default_gateway": "string",
    "persistent_ip_address": "string",
    "persistent_ip_enabled": false,
    "persistent_subnet_mask": "string"
  },
  "subnet_mask": "string"
}
```

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

- *GET /system/network*

NetworkSettings: Current network settings.

An object of type NetworkSettings has the following properties:

- **dhcp_enabled** (boolean) - DHCP enabled
- **persistent_default_gateway** (string) - Persistent default gateway
- **persistent_ip_address** (string) - Persistent IP address
- **persistent_ip_enabled** (boolean) - Persistent IP enabled
- **persistent_subnet_mask** (string) - Persistent subnet mask

Template object

```
{
  "dhcp_enabled": false,
  "persistent_default_gateway": "string",
  "persistent_ip_address": "string",
  "persistent_ip_enabled": false,
```

(continues on next page)

(continued from previous page)

```
  "persistent_subnet_mask": "string"
}
```

NetworkSettings objects are nested in *NetworkInfo*, and are used in the following requests:

- *GET /system/network/settings*
- *PUT /system/network/settings*

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}*

ParameterNameValue: Parameter name and value. 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 ParameterNameValue has the following properties:

- **name** (string) - name of the parameter
- **value** (type not defined) - the parameter's current value

Template object

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

ParameterNameValue objects are used in the following requests:

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

ParameterValue: Parameter value. 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 ParameterValue has the following properties:

- **value** (type not defined) - the parameter's current value

Template object

```
{
  "value": {}
}
```

ParameterValue objects are used in the following requests:

- *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
- **network** (*NetworkInfo*) - see description of *NetworkInfo*
- **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",
  "network": {
    "current_method": "string",
    "default_gateway": "string",
    "ip_address": "string",
    "settings": {
      "dhcp_enabled": false,
      "persistent_default_gateway": "string",
      "persistent_ip_address": "string",
      "persistent_ip_enabled": false,
      "persistent_subnet_mask": "string"
    },
    "subnet_mask": "string"
  },
  "ntp_status": {
    "accuracy": "string",
    "synchronized": false
  },
  "ptp_status": {
    "master_ip": "string",
    "offset": 0,
    "offset_dev": 0,
    "offset_mean": 0,
    "state": "string"
  },
  "ready": false,
```


(continues on next page)

(continued from previous page)

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

SysInfo objects are used in the following requests:

- *GET /system*

Template: rc_silhouettematch template

An object of type Template has the following properties:

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

Template object

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

Template objects are used in the following requests:

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

8.2.4 Swagger UI

The *rc_visard*'s Swagger UI allows developers to easily visualize and interact with the REST-API, e.g., for development and testing. Accessing `http://<host>/api/` or `http://<host>/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.

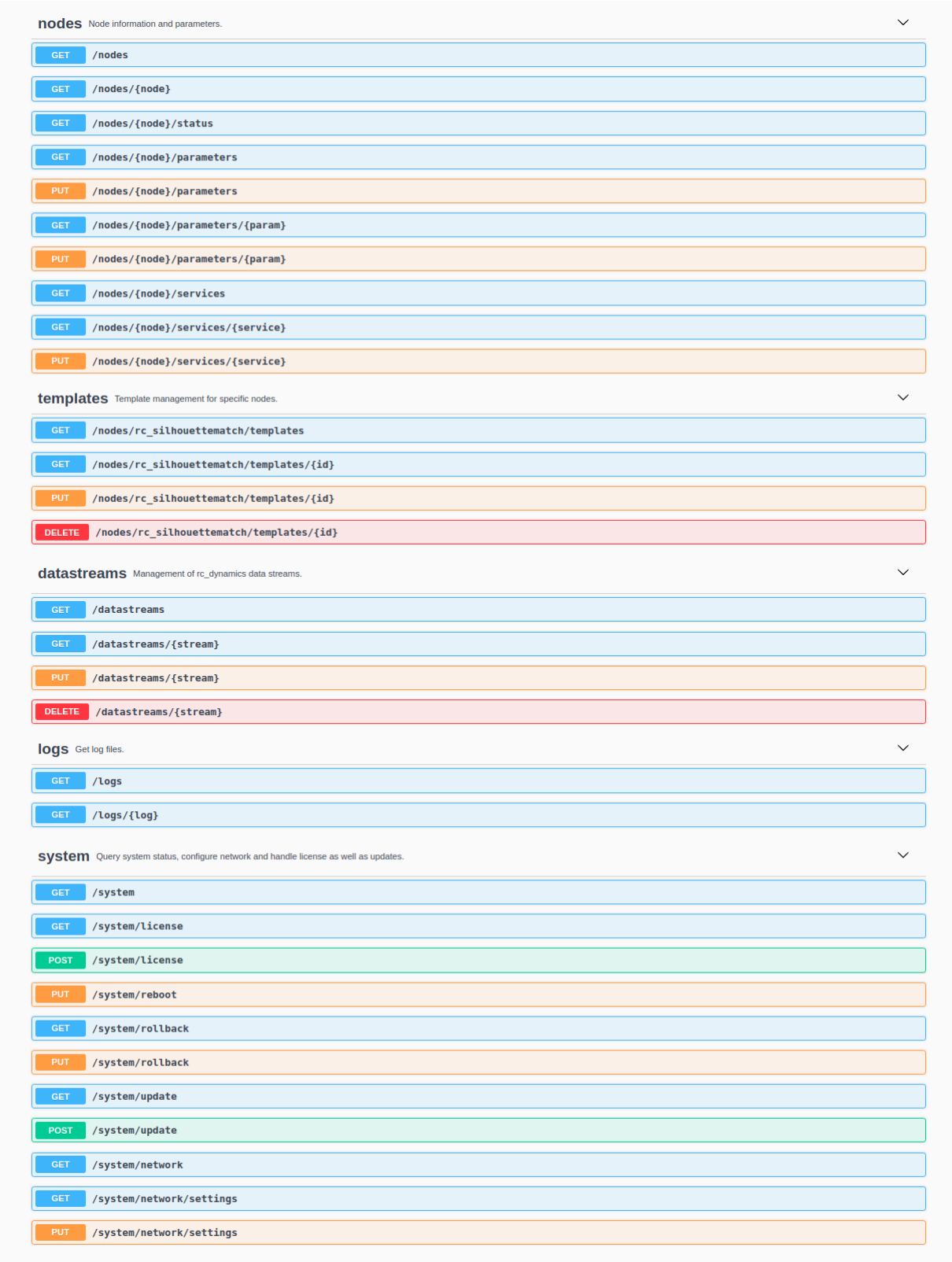


Fig. 8.1: Initial view of the *rc_visard*'s Swagger UI with its resources and requests grouped into `nodes`, `templates`, `datastreams`, `logs`, and `system`

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 clicking the *Try it out!* button, filling in the necessary parameter (`node` name) and clicking *Execute*. 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.

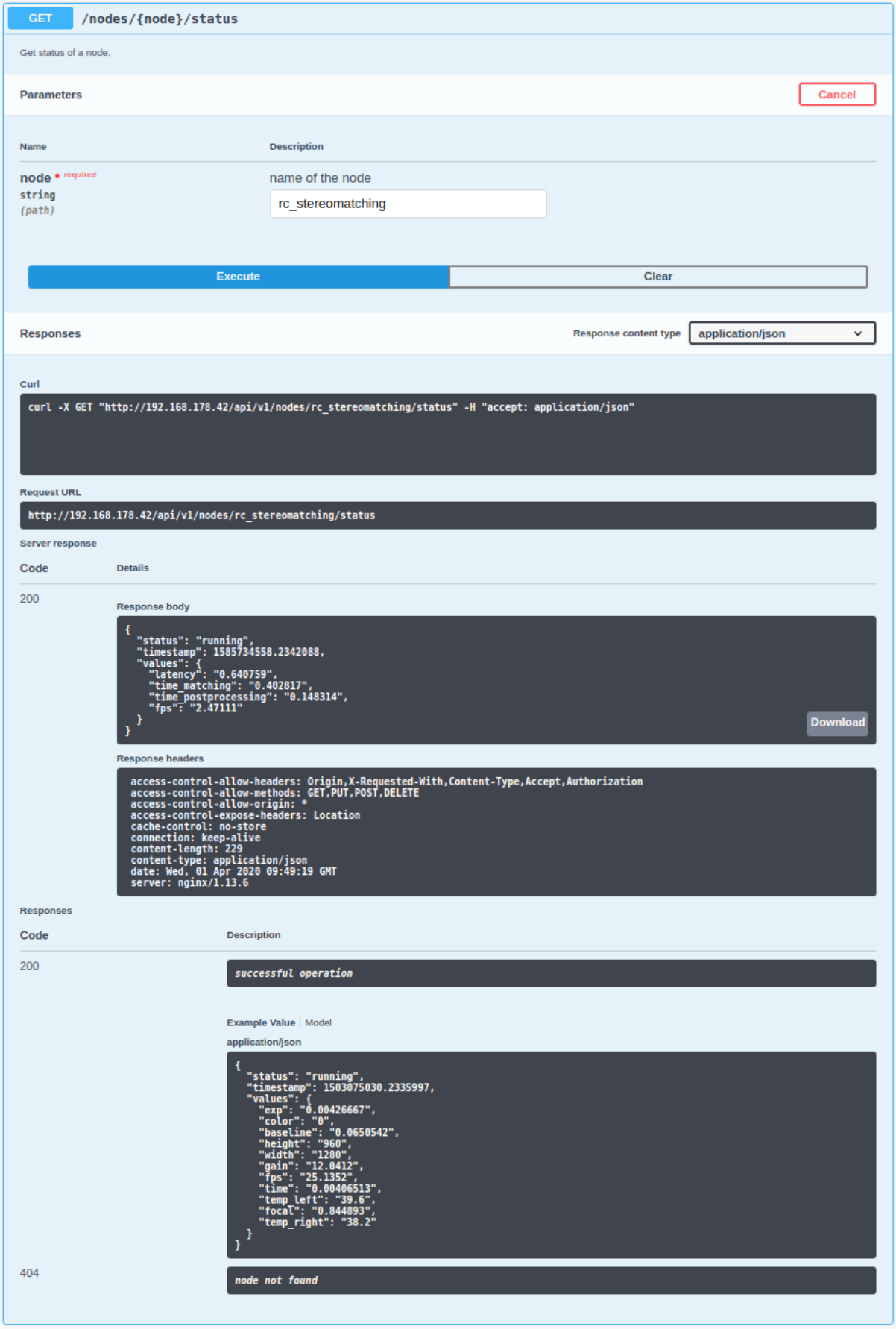


Fig. 8.2: Result of requesting the `rc_stereomatching` node's status

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.

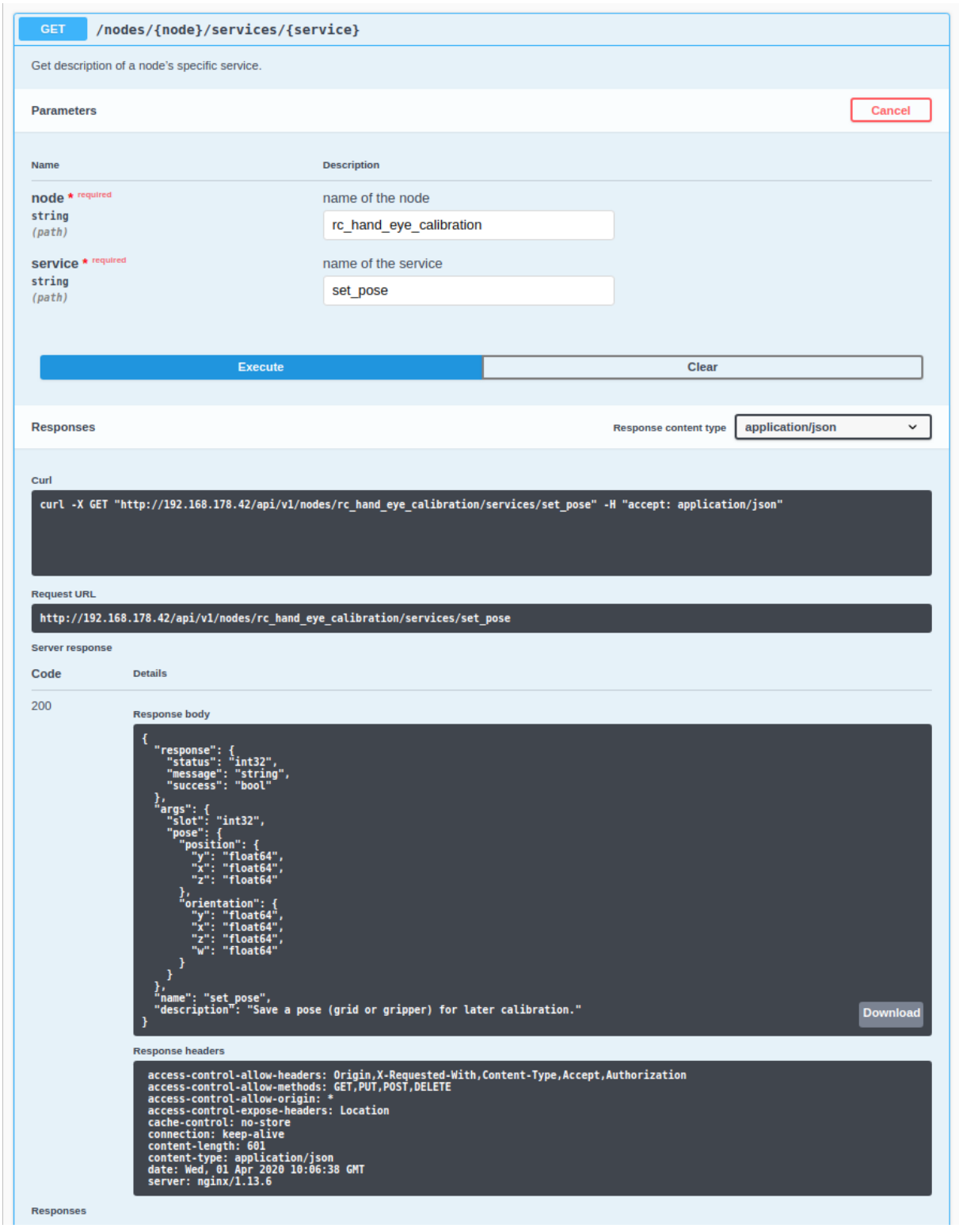


Fig. 8.3: The result of the `GET` request on the `set_pose` service shows the required arguments for this service call.

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

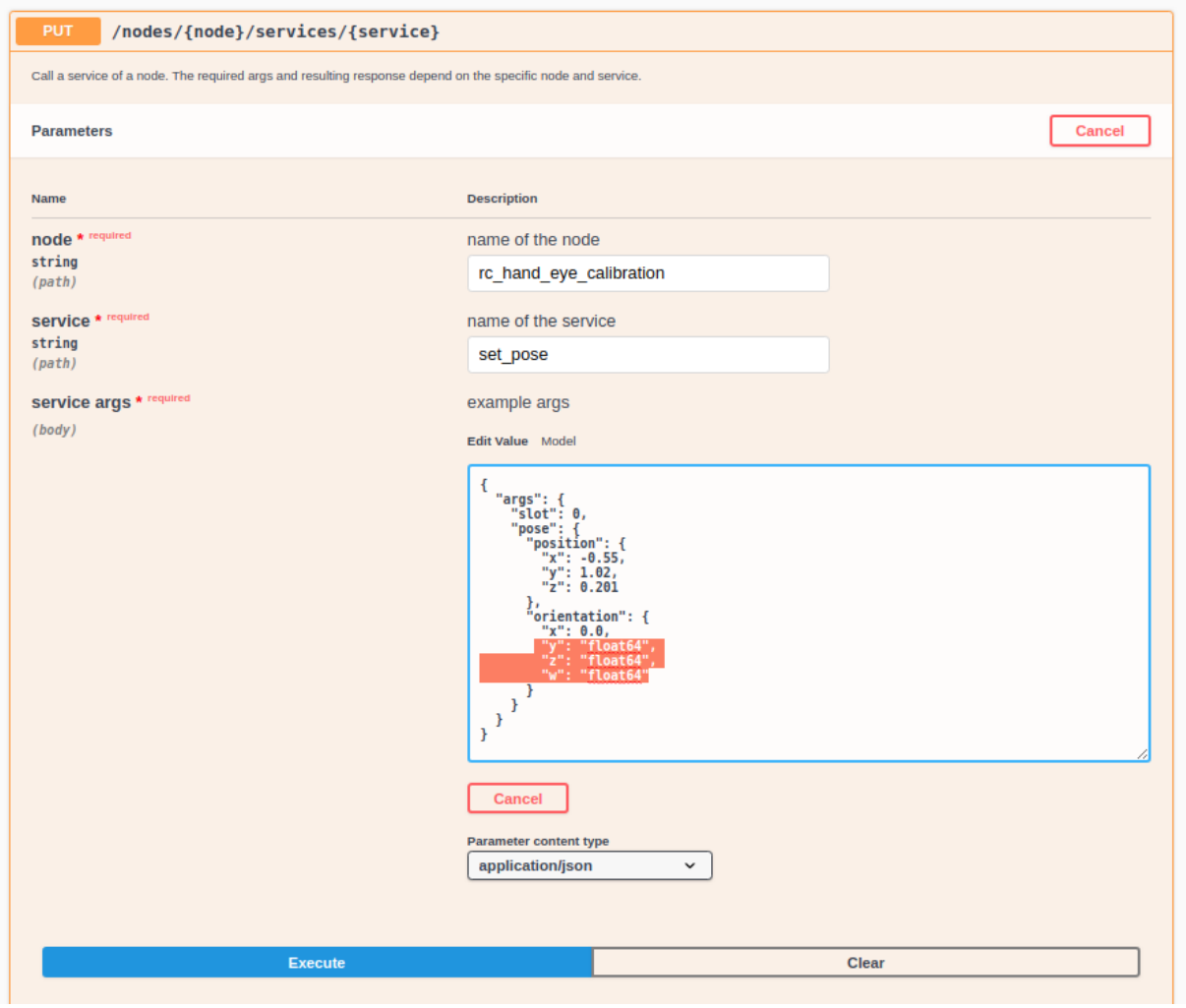


Fig. 8.4: Filling in the arguments of the `set_pose` service request

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 w.r.t. 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 8.2: 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)
`imu`	UDP	*Imu*	Raw IMU (Inertial Measurement Unit) values 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_ins`	UDP	*Frame*	Pose of left camera from stereo INS 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_rt_ins`	UDP	*Frame*	Pose of left camera from stereo INS at realtime frequency (IMU rate)

The general procedure for working with the rc_dynamics interface is the following:

1. **Request a data stream via REST-API.** The following sample `curl` command issues a *`PUT /datastreams/{stream}`* request to initiate a stream of type `pose_rt` from the *rc_visard* to client host `10.0.1.14` at port `30000`:

   ```
   curl -X PUT --header 'Content-Type: application/x-www-form-urlencoded' --header
   ↪'Accept: application/json' -d 'destination=10.0.1.14:30000' 'http://<rcvisard>/api/v1/
   ↪datastreams/pose_rt'
   ```

2. **Receive and deserialize data.** With a successful request, the stream is initiated and data of the specified stream type is continuously sent to the client host. According to the *Data-stream protocol* (Section 8.3.3), the client needs to receive, deserialize and process the data.

3. **Stop a requested data stream via REST-API.** The following sample `curl` command issues a *`DELETE /datastreams/{stream}`* request to delete, i.e., stop, the previously requested stream of type `pose_rt` with destination `10.0.1.14:30000`:

   ```
   curl -X DELETE --header 'Accept: application/json' 'http://<rcvisard>/api/v1/
   ↪datastreams/pose_rt?destination=10.0.1.14:30000'
   ```

   To remove all destinations for a stream, simply omit the destination parameter.

Warning: Data streams can not be deleted automatically, i.e., the *rc_visard* keeps streaming data even if the client-side is disconnected or has stopped consuming the sent datagrams. A maximum of 10 destinations per stream are allowed. It is therefore strongly recommended to stop data streams via the REST-API when they are or no longer used.

8.3.3 Data-stream protocol

Once a data stream is established, data is continuously sent to the specified client host and port (`destination`) via the following protocol:

Network protocol: The only currently supported network protocol is *UDP*, i.e., data is sent as UDP datagrams.

Data serialization: The data being sent is serialized via Google protocol buffers. The following message type definitions are used.

- The *camera-pose streams* and *real-time camera-pose streams* (Section 6.3.2) are serialized using the `Frame` message type:

```
message Frame
{
  optional PoseStamped pose  = 1;
  optional string parent     = 2; // Name of the parent frame
  optional string name       = 3; // Name of the frame
  optional string producer   = 4; // Name of the producer of this data
}
```

  The `producer` field can take the values `ins`, `slam`, `rt_ins`, and `rt_slam`, indicating whether the data was computed by SLAM or Stereo INS, and is real-time (rt) or not.

- The *real-time dynamics stream* (Section 6.3.2) is serialized using the `Dynamics` message type:

```
message Dynamics
{
  optional Time timestamp                    = 1; // Time when the data was␣
↪captured
  optional Pose pose                         = 2;
  optional string pose_frame                 = 3; // Name of the frame that␣
↪the pose is given in
  optional Vector3d linear_velocity          = 4; // Linear velocity in m/s
  optional string linear_velocity_frame      = 5; // Name of the frame that␣
↪the linear_velocity is given in
  optional Vector3d angular_velocity         = 6; // Angular velocity in rad/s
  optional string angular_velocity_frame     = 7; // Name of the frame that␣
↪the angular_velocity is given in
  optional Vector3d linear_acceleration      = 8; // Gravity compensated␣
↪linear acceleration in m/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
  optional string producer                   = 13; // Name of the producer of␣
↪this data
}
```

  The `producer` field can take the values `rt_ins` and `rt_slam`, indicating whether the data was computed by SLAM or Stereo INS.

- The *IMU stream* (Section 6.3.2) is serialized using the `Imu` message type:

```
message Imu
{
  optional Time timestamp                    = 1; // Time when the data was␣
↪captured
  optional Vector3d linear_acceleration      = 2; // Linear acceleration in m/
↪s^2 measured by the IMU
```

(continues on next page)

(continued from previous page)

```
  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;
}
```

8.3.4 rc_dynamics_api

Our open-source `rc_dynamics_api` package provides a simple, convenient C++ wrapper to request and parse rc_dynamics streams. See http://www.roboception.com/download.

8.4 KUKA Ethernet KRL Interface

The *rc_visard* provides an Ethernet KRL Interface (EKI Bridge), which allows communicating with the *rc_visard* from KUKA KRL via KUKA.EthernetKRL XML.

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

Note: The KUKA.EthernetKRL add-on software package version 2.2 or newer must be activated on the robot controller to use this component.

The EKI Bridge can be used to programmatically

- do service calls, e.g. to start and stop individual computational nodes, or to use offered services such as the hand-eye calibration or the computation of grasp poses;
- set and get run-time parameters of computation nodes, e.g. of the camera, or disparity calculation.

8.4.1 Ethernet connection configuration

The EKI Bridge listens on port 7000 for EKI XML messages and transparently bridges the *rc_visard*'s *REST-API* (Section 8.2). The received EKI messages are transformed to JSON and forwarded to the *rc_visard*'s REST-API. The response from the REST-API is transformed back to EKI XML.

The EKI Bridge gives access to run-time parameters and offered services of all computational nodes described in *Software components* (Section 6) and *Optional software components* (Section 7).

The Ethernet connection to the *rc_visard* on the robot controller is configured using XML configuration files. The EKI XML configuration files of all nodes running on the *rc_visard* are available for download at:

https://doc.rc-visard.com/latest/en/eki.html#eki-xml-configuration-files

Each node offering run-time parameters has an XML configuration file for setting and getting its parameters. These are named following the scheme `<node_name>-parameters.xml`. Each node's service has its own XML configuration file. These are named following the scheme `<node_name>-<service_name>.xml`.

All elements in the XML files are preset, except for the IP of the *rc_visard* in the network.

These files must be stored in the directory `C:\KRC\ROBOTER\Config\User\Common\EthernetKRL` of the robot controller and they are read in when a connection is initialized.

As an example, an Ethernet connection to configure the `rc_stereomatching` parameters is established with the following KRL code.

```
DECL EKI_Status RET
RET = EKI_INIT("rc_stereomatching-parameters")
RET = EKI_Open("rc_stereomatching-parameters")

; ----------- Desired operation -----------

RET = EKI_Close("rc_stereomatching-parameters")
```

Note: The EKI Bridge automatically terminates the connection to the client if the received XML telegram is invalid.

8.4.2 Generic XML structure

For data transmission, the EKI Bridge uses `<req>` as root XML element (short for request).

The root tag always includes the following elements.

- `<node>`. This includes a child XML element used by the EKI Bridge to identify the target node. The node name is already included in the XML configuration file.
- `<end_of_request>`. End of request flag that triggers the request.

The following listing shows the generic XML structure for data transmission.

```
<SEND>
  <XML>
    <ELEMENT Tag="req/node/<node_name>" Type="STRING"/>
    <ELEMENT Tag="req/end_of_request" Type="BOOL"/>
  </XML>
</SEND>
```

For data reception, the EKI Bridge uses `<res>` as root XML element (short for response). The root tag always includes a `<return_code>` child element.

```
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

Note: By default the XML configuration files uses 998 as flag to notify KRL that the response data record has been received. If this value is already in use, it should be changed in the corresponding XML configuration file.

Return code

The `<return_code>` element consists of a `value` and a `message` attribute.

As for all other components, a successful request returns with a `res/return_code/@value` of 0. Negative values indicate that the request failed. The error message is contained in `res/return_code/@message`. Positive values indicate that the request succeeded with additional information, contained in `res/return_code/@message` as well.

The following codes can be issued by the EKI Bridge component.

Table 8.3: Return codes of the EKI Bridge component

Code	Description
0	Success
-1	Parsing error in the conversion from XML to JSON
-2	Internal error
-9	Missing or invalid license for EKI Bridge component
-11	Connection error from the REST-API

Note: The EKI Bridge can also return return code values specific to individual nodes. They are documented in the respective *software component* (Section 6).

Note: Due to limitations in KRL, the maximum length of a string returned by the EKI Bridge is 512 characters. All messages larger than this value are truncated.

8.4.3 Services

For the nodes' services, the XML schema is generated from the service's arguments and response in JavaScript Object Notation (JSON) described in *Software components* (Section 6) and *Optional software components* (Section 7). The conversion is done transparently, except for the conversion rules described below.

Conversions of poses:

A pose is a JSON object that includes `position` and `orientation` keys.

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

This JSON object is converted to a KRL `FRAME` in the XML message.

```
<pose X="..." Y="..." Z="..." A="..." B="..." C="..."></pose>
```

Positions are converted from meters to millimeters and orientations are converted from quaternions to KUKA ABC (in degrees).

Note: No other unit conversions are included in the EKI Bridge. All dimensions and 3D coordinates that don't belong to a pose are expected and returned in meters.

Arrays:

Arrays are identified by adding the child element `<le>` (short for list element) to the list name. As an example, the JSON object

```
{
  "rectangles": [
    {
      "x": "float64",
      "y": "float64"
    }
  ]
}
```

is converted to the XML fragment

```
<rectangles>
  <le>
    <x>...</x>
    <y>...</y>
  </le>
</rectangles>
```

Use of XML attributes:

All JSON keys whose values are a primitive data type and don't belong to an array are stored in attributes. As an example, the JSON object

```
{
  "item": {
    "uuid": "string",
    "confidence": "float64",
    "rectangle": {
      "x": "float64",
      "y": "float64"
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

is converted to the XML fragment

```
<item uuid="..." confidence="...">
  <rectangle x="..." y="...">
  </rectangle>
</item>
```

Request XML structure

The <SEND> element in the XML configuration file for a generic service follows the specification below.

```
<SEND>
  <XML>
    <ELEMENT Tag="req/node/<node_name>" Type="STRING"/>
    <ELEMENT Tag="req/service/<service_name>" Type="STRING"/>
    <ELEMENT Tag="req/args/<argX>" Type="<argX_type>"/>
    <ELEMENT Tag="req/end_of_request" Type="BOOL"/>
  </XML>
</SEND>
```

The <service> element includes a child XML element that is used by the EKI Bridge to identify the target service from the XML telegram. The service name is already included in the configuration file.

The <args> element includes the service arguments and should be configured with EKI_Set<Type> KRL instructions.

As an example, the <SEND> element of the rc_itempick's get_load_carriers service (see *ItemPick and BoxPick*, Section 7.2) is:

```
<SEND>
  <XML>
    <ELEMENT Tag="req/node/rc_itempick" Type="STRING"/>
    <ELEMENT Tag="req/service/get_load_carriers" Type="STRING"/>
    <ELEMENT Tag="req/args/load_carrier_ids/le" Type="STRING"/>
    <ELEMENT Tag="req/end_of_request" Type="BOOL"/>
  </XML>
</SEND>
```

The <end_of_request> element allows to have arrays in the request. For configuring an array, the request is split into as many packages as the size of the array. The last telegram contains all tags, including the <end_of_request> flag, while all other telegrams contain one array element each.

As an example, for requesting two load carrier models to the rc_itempick's get_load_carriers service, the user needs to send two XML messages. The first XML telegram is:

```
<req>
  <args>
    <load_carrier_ids>
      <le>load_carrier1</le>
    </load_carrier_ids>
  </args>
</req>
```

This telegram can be sent from KRL with the EKI_Send command, by specifying the list element as path:

```
DECL EKI_STATUS RET
RET = EKI_SetString("rc_itempick-get_load_carriers", "req/args/load_carrier_ids/le",
↪"load_carrier1")
RET = EKI_Send("rc_itempick-get_load_carriers", "req/args/load_carrier_ids/le")
```

The second telegram includes all tags and triggers the request to the `rc_itempick` node:

```
<req>
  <node>
    <rc_itempick></rc_itempick>
  </node>
  <service>
    <get_load_carriers></get_load_carriers>
  </service>
  <args>
    <load_carrier_ids>
      <le>load_carrier2</le>
    </load_carrier_ids>
  </args>
  <end_of_request></end_of_request>
</req>
```

This telegram can be sent from KRL by specifying `req` as path for `EKI_Send`:

```
DECL EKI_STATUS RET
RET = EKI_SetString("rc_itempick-get_load_carriers", "req/args/load_carrier_ids/le",
↪"load_carrier2")
RET = EKI_Send("rc_itempick-get_load_carriers", "req")
```

Response XML structure

The `<RECEIVE>` element in the XML configuration file for a generic service follows the specification below:

```
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/<resX>" Type="<resX_type>"/>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

As an example, the `<RECEIVE>` element of the `rc_april_tag_detect`'s `detect` service (see *TagDetect*, Section 6.9) is:

```
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/timestamp/@sec" Type="INT"/>
    <ELEMENT Tag="res/timestamp/@nsec" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/tags/le/pose_frame" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/timestamp/@sec" Type="INT"/>
    <ELEMENT Tag="res/tags/le/timestamp/@nsec" Type="INT"/>
    <ELEMENT Tag="res/tags/le/pose/@X" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@Y" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@Z" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@A" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@B" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@C" Type="REAL"/>
```

(continues on next page)

(continued from previous page)

```
    <ELEMENT Tag="res/tags/le/instance_id" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/id" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/size" Type="REAL"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

For arrays, the response includes multiple instances of the same XML element. Each element is written into a separate buffer within EKI and can be read from the buffer with KRL instructions. The number of instances can be requested with `EKI_CheckBuffer` and each instance can then be read by calling `EKI_Get<Type>`.

As an example, the tag poses received after a call to the `rc_april_tag_detect`'s `detect` service can be read in KRL using the following code:

```
DECL EKI_STATUS RET
DECL INT i
DECL INT num_instances
DECL FRAME poses[32]

DECL FRAME pose = {X 0.0, Y 0.0, Z 0.0, A 0.0, B 0.0, C 0.0}

RET = EKI_CheckBuffer("rc_april_tag_detect-detect", "res/tags/le/pose")
num_instances = RET.Buff
for i=1 to num_instances
  RET = EKI_GetFrame("rc_april_tag_detect-detect", "res/tags/le/pose", pose)
  poses[i] = pose
endfor
RET = EKI_ClearBuffer("rc_april_tag_detect-detect", "res")
```

Note: Before each request from EKI to the *rc_visard*, all buffers should be cleared in order to store only the current response in the EKI buffers.

8.4.4 Parameters

All nodes' parameters can be set and queried from the EKI Bridge. The XML configuration file for a generic node follows the specification below:

```
<SEND>
  <XML>
    <ELEMENT Tag="req/node/<node_name>" Type="STRING"/>
    <ELEMENT Tag="req/parameters/<parameter_x>/@value" Type="INT"/>
    <ELEMENT Tag="req/parameters/<parameter_y>/@value" Type="STRING"/>
    <ELEMENT Tag="req/end_of_request" Type="BOOL"/>
  </XML>
</SEND>
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/parameters/<parameter_x>/@value" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_x>/@default" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_x>/@min" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_x>/@max" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@value" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@default" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@min" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@max" Type="REAL"/>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
```

(continues on next page)

(continued from previous page)

```
  </XML>
</RECEIVE>
```

The request is interpreted as a *get* request if all parameter's `value` attributes are empty. If any `value` attribute is non-empty, it is interpreted as *set* request of the non-empty parameters.

As an example, the current value of all parameters of `rc_stereomatching` can be queried using the XML telegram:

```
<req>
  <node>
    <rc_stereomatching></rc_stereomatching>
  </node>
  <parameters></parameters>
  <end_of_request></end_of_request>
</req>
```

This XML telegram can be sent out with Ethernet KRL using:

```
DECL EKI_STATUS RET
RET = EKI_Send("rc_stereomatching-parameters", "req")
```

The response from the EKI Bridge contains all parameters:

```
<res>
  <parameters>
    <acquisition_mode default="Continuous" max="" min="" value="Continuous"/>
    <quality default="High" max="" min="" value="High"/>
    <static_scene default="0" max="1" min="0" value="0"/>
    <seg default="200" max="4000" min="0" value="200"/>
    <smooth default="1" max="1" min="0" value="1"/>
    <fill default="3" max="4" min="0" value="3"/>
    <minconf default="0.5" max="1.0" min="0.5" value="0.5"/>
    <mindepth default="0.1" max="100.0" min="0.1" value="0.1"/>
    <maxdepth default="100.0" max="100.0" min="0.1" value="100.0"/>
    <maxdeptherr default="100.0" max="100.0" min="0.01" value="100.0"/>
  </parameters>
  <return_code message="" value="0"/>
</res>
```

The `quality` parameter of `rc_stereomatching` can be set to `Low` by the XML telegram:

```
<req>
    <node>
      <rc_stereomatching></rc_stereomatching>
    </node>
    <parameters>
      <quality value="Low"></quality>
    </parameters>
    <end_of_request></end_of_request>
</req>
```

This XML telegram can be sent out with Ethernet KRL using:

```
DECL EKI_STATUS RET
RET = EKI_SetString("rc_stereomatching-parameters", "req/parameters/quality/@value",
↪"Low")
RET = EKI_Send("rc_stereomatching-parameters", "req")
```

In this case, only the applied value of `quality` is returned by the EKI Bridge:

```
<res>
  <parameters>
    <quality default="High" max="" min="" value="Low"/>
  </parameters>
  <return_code message="" value="0"/>
</res>
```

8.5 Time synchronization

The *rc_visard* provides timestamps with all images and messages. To compare these with the time on the application host, the time needs to be properly synchronized.

This can be done either via the Network 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 time synchronization status can be queried via *REST-API* (Section 8.2) and seen on the *Web GUI*'s (Section 4.6) *System* tab.

Note: Depending on the reachability of NTP servers or PTP masters it might take up to several minutes until the time is synchronized.

8.5.1 NTP

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

By default the *rc_visard* tries to reach NTP servers from the NTP Pool Project, which will work if the *rc_visard* has access to the internet.

If the *rc_visard* is configured for *DHCP* (Section 4.5.2) (which is the default setting), it will also request NTP servers from the DHCP server and try to use those.

8.5.2 PTP

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

The *rc_visard* can be configured to act as a PTP slave via the standard *GigE Vision 2.0/GenICam interface* (Section 8.1) using the `GevIEEE1588` parameter.

At least one PTP master providing time has to be running in the network. On Linux the respective command for starting a PTP master on ethernet port `eth0` is, e.g., `sudo ptpd --masteronly --foreground -i eth0`.

While the *rc_visard* is synchronized with a PTP master (*rc_visard* in PTP status SLAVE), the NTP synchronization is paused.

9 Maintenance

Warning: The customer does not need to open the *rc_visard*'s housing to perform maintenance. Unauthorized opening will void the warranty.

9.1 Lens cleaning

Glass lenses with antireflective coating are used to reduce glare. Please take special care when cleaning the lenses. To clean them, use a soft lens-cleaning brush to remove dust or dirt particles. Then use a clean microfiber cloth that is designed to clean lenses, and gently wipe the lens using a circular motion to avoid scratches that may compromise the sensor's performance. For stubborn dirt, high purity isopropanol or a lens cleaning solution formulated for coated lenses (such as the Uvex Clear family of products) may be used.

9.2 Camera calibration

The cameras are calibrated during production. Under normal operating 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 4.6) *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 camera 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.

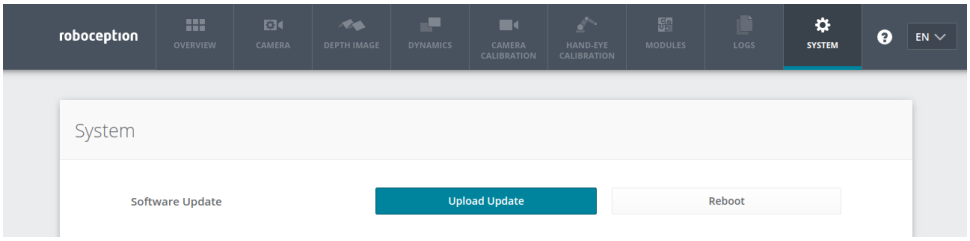


Fig. 9.1: Web GUI *System* tab

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.

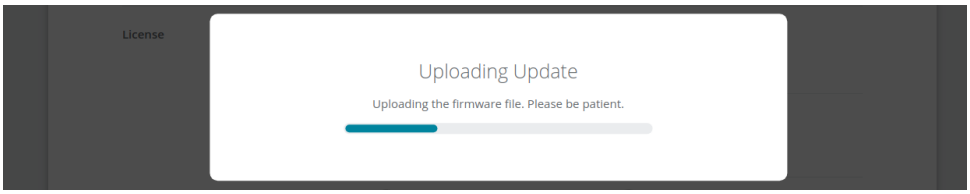


Fig. 9.2: Software update progress bar

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.

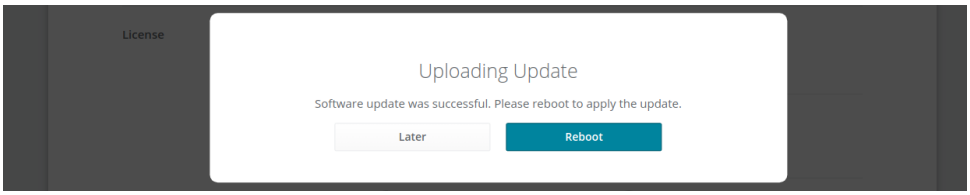


Fig. 9.3: Software update rebooting screen

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:** As with 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 4.6) *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 4.6) *System* tab. 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 4.6) *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.

Aside from 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*. For permanent installation, the customer is responsible for providing a suitable power supply. The connectivity kit consists of a:

- network cable with straight M12 plug to straight RJ45 connector in either 2 m, 5 m, or 10 m length,
- power adapter cable with straight M12 socket to DC barrel connector in 30 cm length,
- 24 V, 30 W wall power supply, or a 24 V, 60 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.

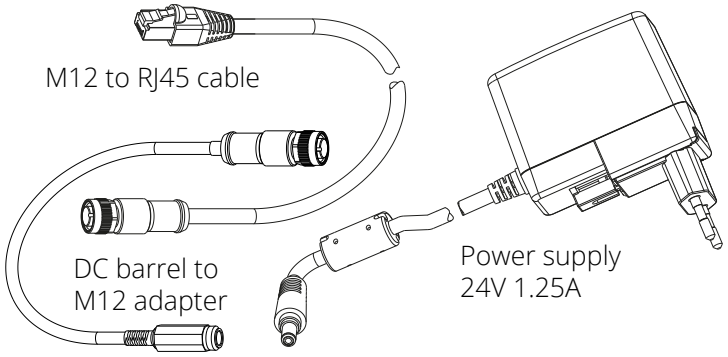


Fig. 10.1: The optional connectivity kit's components

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 B device and immune to light industrial and industrial environments. 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 3.6). 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 3.4).

- 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 4.5) 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 camera is in manual exposure mode, decrease the exposure time (see *Parameters*, Section 6.1.4), or
- switch to auto-exposure mode (see *Parameters*, Section 6.1.4).

The camera image is too dark

- If the camera is in manual exposure mode, increase the exposure time (see *Parameters*, Section 6.1.4), or
- switch to auto-exposure mode (see *Parameters*, Section 6.1.4).

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 6.1.4).

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 camera lenses are dirty and clean them if they are.
- 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 camera,
- decrease the motion speed of objects in the field of view of the camera, or
- decrease the exposure time of the camera (see *Parameters*, Section 6.1.4).

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 6.1.4).
- 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.
- 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.
- 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). Lower 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 6.1.4). 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).
- Increase the *Minimum Distance* (Section 6.2.4) as much as possible for the application.

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.
- 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).

The disparity values or the resulting depth values are too inaccurate

- Decrease the distance between the camera 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.

The disparity image is too smooth

- 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 8.1.2).

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., first A rotates around the Z axis, then B rotates around the Y axis, and then C rotates around the X axis. In this convention, the axes are the intrinsic, body-aligned axes which change during the rotation. Thus, A is the yaw angle, B the pitch angle and C the roll angle. 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., $q = \begin{pmatrix} x & y & z & w \end{pmatrix}^T$ with $||q|| = 1$. The corresponding rotation matrix and translation vector are defined by

$$
R = 2 \begin{pmatrix} \frac{1}{2} - y^2 - z^2 & xy - zw & xz + yw \\ xy + zw & \frac{1}{2} - x^2 - z^2 & yz - xw \\ xz - yw & yz + xw & \frac{1}{2} - x^2 - y^2 \end{pmatrix}, \qquad T = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}.
$$

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.

13.1.3 Conversion from ABC to quaternion

The conversion from the ABC Euler angles in degrees to a quaternion $q = \begin{pmatrix} x & y & z & w \end{pmatrix}^T$ can be done as follows.

$$\begin{aligned}
x &= \cos(A/2)\cos(B/2)\sin(C/2) - \sin(A/2)\sin(B/2)\cos(C/2) \\
y &= \cos(A/2)\sin(B/2)\cos(C/2) + \sin(A/2)\cos(B/2)\sin(C/2) \\
z &= \sin(A/2)\cos(B/2)\cos(C/2) - \cos(A/2)\sin(B/2)\sin(C/2) \\
w &= \cos(A/2)\cos(B/2)\cos(C/2) + \sin(A/2)\sin(B/2)\sin(C/2)
\end{aligned}$$

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.

13.1.4 Conversion from quaternion to ABC

The conversion from a quaternion $q = \begin{pmatrix} x & y & z & w \end{pmatrix}^T$ to the ABC Euler angles in degrees can be done as follows.

$$\begin{aligned}
A &= \text{atan2}(2(wz + xy), 1 - 2(y^2 + z^2))\frac{180}{\pi} \\
B &= \text{asin}(2(wy - zx))\frac{180}{\pi} \\
C &= \text{atan2}(2(wx + yz), 1 - 2(x^2 + y^2))\frac{180}{\pi}
\end{aligned}$$

HTTP Routing Table

/datastreams

/logs

/nodes

/system

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X