

Basler AG | October 2026

rc_visard 3D Stereo Sensor

ASSEMBLY AND OPERATING MANUAL

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 26.10.3 Oct 23, 2026

Applicable to *rc_visard* firmware 26.10.x

Distributor:
Basler AG
An der Strusbek 60-62
D-22926 Ahrensburg
Germany

Web: http://www.baslerweb.com
Email: info@baslerweb.com
Phone: +49 4102 463 0

Contents

1 Introduction **6**
- 1.1 Overview . . . 6
- 1.2 Warranty . . . 8
- 1.3 Applicable standards . . . 9
  - 1.3.1 Interfaces . . . 9
  - 1.3.2 Approvals . . . 9
  - 1.3.3 Standards . . . 9
- 1.4 Information on disposal . . . 10
- 1.5 Glossary . . . 12

2 Safety **14**
- 2.1 General warnings . . . 14
- 2.2 Intended use . . . 15

3 Hardware specification **16**
- 3.1 Scope of delivery . . . 16
- 3.2 Technical specification . . . 17
- 3.3 Environmental and operating conditions . . . 20
- 3.4 Power-supply specifications . . . 20
- 3.5 Wiring . . . 21
- 3.6 Mechanical interface . . . 23
- 3.7 Coordinate frames . . . 24

4 Installation **26**
- 4.1 Software license . . . 26
- 4.2 Power up . . . 26
- 4.3 Discovery of *rc_visard* devices . . . 27
  - 4.3.1 Resetting configuration . . . 27
- 4.4 Network configuration . . . 28
  - 4.4.1 Host name . . . 29
  - 4.4.2 Automatic configuration (factory default) . . . 29
  - 4.4.3 Manual configuration . . . 30

5 Software modules **31**
- 5.1 Camera module . . . 31
  - 5.1.1 Parameters, status values and services . . . 31
  - 5.1.2 Rectification . . . 39
  - 5.1.3 Viewing and downloading images . . . 39
- 5.2 Stereo matching module . . . 39
  - 5.2.1 Parameters . . . 39
  - 5.2.2 Status values . . . 46
  - 5.2.3 Services . . . 47
  - 5.2.4 Computing 3D data . . . 48
  - 5.2.5 Viewing and downloading images and point clouds . . . 50
- 5.3 Navigation modules . . . 50
  - 5.3.1 Sensor dynamics . . . 51

5.3.2 Visual odometry . . . 59
5.3.3 Stereo INS . . . 63
5.3.4 SLAM . . . 64
5.4 Detection & Measure modules . . . 71
5.4.1 Measure . . . 71
5.4.2 LoadCarrier . . . 78
5.4.3 TagDetect . . . 94
5.4.4 ItemPick . . . 107
5.4.5 BoxPick . . . 128
5.4.6 Pack . . . 161
5.4.7 SilhouetteMatch . . . 173
5.5 Configuration modules . . . 212
5.5.1 Hand-eye calibration . . . 213
5.5.2 CollisionCheck . . . 235
5.5.3 Camera calibration . . . 244
5.5.4 IO and Projector Control . . . 251
5.6 Database modules . . . 254
5.6.1 LoadCarrierDB . . . 255
5.6.2 RoiDB . . . 263
5.6.3 PrimitiveDB . . . 269
5.6.4 GripperDB . . . 273

6 Interfaces 287
6.1 Web GUI . . . 287
6.1.1 Accessing the Web GUI . . . 287
6.1.2 Exploring the Web GUI . . . 288
6.1.3 Web GUI access control . . . 289
6.1.4 Downloading camera images . . . 289
6.1.5 Downloading depth images and point clouds . . . 289
6.2 REST-API interface . . . 290
6.2.1 General API structure . . . 290
6.2.2 Migration from API version 1 . . . 292
6.2.3 Available resources and requests . . . 292
6.2.4 Data type definitions . . . 326
6.2.5 Swagger UI . . . 338
6.3 The rc_dynamics interface . . . 342
6.3.1 Starting/stopping dynamic-state estimation . . . 342
6.3.2 Configuring data streams . . . 342
6.3.3 Data-stream protocol . . . 343
6.3.4 rc_dynamics_api . . . 345
6.4 Generic Robot Interface . . . 345
6.4.1 Overview . . . 345
6.4.2 Job definition . . . 346
6.4.3 Hand-Eye Calibration . . . 355
6.4.4 GRI binary protocol specification . . . 355
6.4.5 Integration with a robot . . . 362
6.4.6 Job and HEC_config API . . . 362
6.5 OPC UA interface . . . 367
6.6 KUKA Ethernet KRL Interface . . . 367
6.6.1 Ethernet connection configuration . . . 368
6.6.2 Generic XML structure . . . 368
6.6.3 Services . . . 369
6.6.4 Parameters . . . 373
6.6.5 Migration to firmware version 22.01 . . . 375
6.6.6 Troubleshooting . . . 375
6.7 GigE Vision 2.0/GenICam image interface . . . 376
6.7.1 GigE Vision ports . . . 376
6.7.2 Important GenICam parameters . . . 376

6.7.3 Important standard GenICam features . . . 376
6.7.4 Custom GenICam features of the *rc_visard* . . . 380
6.7.5 Chunk data . . . 383
6.7.6 Provided image streams . . . 384
6.7.7 Image stream conversions . . . 385
6.8 gRPC image stream interface . . . 385
6.8.1 gRPC service definition . . . 386
6.9 Time synchronization . . . 389
6.9.1 NTP . . . 389
6.9.2 PTP . . . 390
6.9.3 Setting time manually . . . 390

7 Maintenance 391
7.1 Lens cleaning . . . 391
7.2 Camera calibration . . . 391
7.3 Creating and restoring backups of settings . . . 391
7.4 Updating the software license . . . 392
7.5 Downloading log files . . . 392
7.6 Updating the firmware . . . 392
7.7 Restoring the previous firmware version . . . 393
7.8 Rebooting the *rc_visard* . . . 394

8 Accessories 395
8.1 Connectivity . . . 395
8.2 Wiring . . . 395
8.2.1 Ethernet connections . . . 395
8.2.2 Power connections . . . 395
8.2.3 Power supplies . . . 396
8.3 Spare parts . . . 396

9 Troubleshooting 397
9.1 LED colors . . . 397
9.2 Hardware issues . . . 397
9.3 Connectivity issues . . . 398
9.4 Camera-image issues . . . 398
9.5 Depth/Disparity, error, and confidence image issues . . . 399
9.6 Dynamics issues . . . 400
9.7 GigE Vision/GenICam issues . . . 401

10 Contact 402
10.1 Support . . . 402
10.2 Downloads . . . 402
10.3 Address . . . 402

11 Appendix 404
11.1 Pose formats . . . 404
11.1.1 Rotation matrix and translation vector . . . 405
11.1.2 ABB pose format . . . 405
11.1.3 FANUC XYZ-WPR format . . . 405
11.1.4 Franka Pose Format . . . 406
11.1.5 Fruitcore HORST pose format . . . 408
11.1.6 Kawasaki XYZ-OAT format . . . 408
11.1.7 KUKA XYZ-ABC format . . . 409
11.1.8 Mitsubishi XYZ-ABC format . . . 410
11.1.9 Universal Robots pose format . . . 410
11.1.10 Yaskawa Pose Format . . . 411

HTTP Routing Table 413

Index **415**

1 Introduction

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 depth images, which can be used to compute 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 standardized interfaces, making it compatible with all major image processing libraries.

With optionally available software 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.

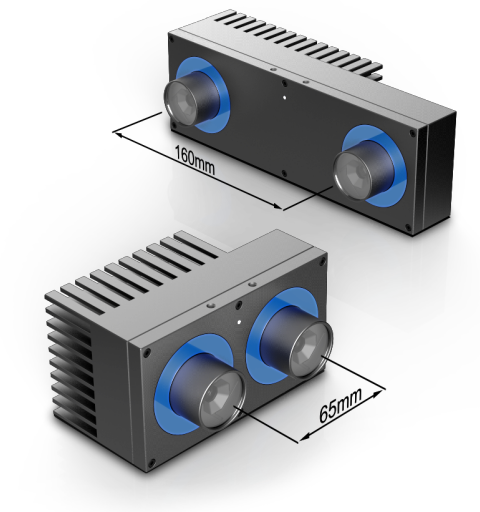


Fig. 1.1: Basler *rc_visard* 65 and *rc_visard* 160

The terms “sensor,” “*rc_visard* 65,” and “*rc_visard* 160” used throughout the manual all refer to the Basler *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 Basler *rc_visard* 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 Basler 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:

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

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:

EC Declaration of Conformity

NRTL certification by TÜV Süd

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.

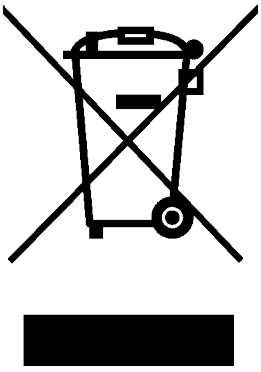
1.3.3 Standards

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 Information on disposal



1. **Disposal of Waste Electrical & Electronic Equipment**

   This symbol on the product(s) and / or accompanying documents means that used electrical and electronic products should not be mixed with general household waste. For proper treatment, recovery and recycling, please contact your supplier or the manufacturer. Disposing of this product correctly will help save valuable resources and prevent any potential negative effects on human health and the environment, which could otherwise arise from inappropriate waste handling.

2. **Removal of batteries**

   If the products contain batteries and accumulators that can be removed from the product without destruction, these must be removed before disposal and disposed of separately as batteries.

   The following batteries or accumulators are contained in the rc_visard: None

3. **Options for returning old equipment**

   Owners of old devices can return them to the manufacturer to ensure proper disposal.

   Please *contact support* (Section 10) about returning the device for disposal.

4. **Data protection**

   End users of Electrical & Electronic Equipment are responsible for deleting personal data on the waste equipment to be disposed of.

5. **WEEE registration number**

   Basler is registered under the registration number DE 83888045 at the stiftung elektro-altgeräte register, Nordostpark 72, 90411 Nuremberg, Germany, as a producer of electrical and/or electronic equipment.

6. **Collection and recovery quotas**

   According to the WEEE Directive, EU member states are obliged to collect data on waste electrical and electronic equipment and to transmit this data to the European Commission. Further information can be found on the German Ministry for the Environment website.

Information on Disposal outside the European Union

This symbol is valid only in the European Union. If you wish to discard this product please contact your local authorities or dealer and ask for the correct method of disposal.

1.5 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 components.

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 5.3.1.2) via the *rc_dynamics interface* (Section 6.3). 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_camera/parameters/fps`, which points to the `fps` run-time parameter of the stereo camera module.

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_camera/parameters/fps` where `<ip>` refers to the *rc_visard*'s *IP address*.

XYZ+quaternion Format to represent a pose. See *Rotation matrix and translation vector* (Section 11.1.1) for its definition.

XYZABC Format to represent a pose. See *KUKA XYZ-ABC format* (Section 11.1.7) 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 an industrial 3D vision sensor with processing unit intended for integration into automation and robotics applications. Its intended purpose is the acquisition and processing of image data and the provision of results (e.g. 3D data, detected poses) to a connected robot controller or host system.

The *rc_visard* is designed and intended to be operated in a trusted environment with a direct, point-to-point (1:1) connection between the device and the connected camera, robot controller, or host system. In this configuration, the devices are not intended to be connected to, or made accessible from, an open or untrusted network.

Several interfaces required for interoperability with robot controllers and industrial systems rely on established industry standards or vendor-specific protocols that do not support encryption or authentication. Enabling these interfaces outside a trusted, isolated network may expose the system to cybersecurity risks and is outside the intended use of the product.

The operator is responsible for ensuring that the operating environment matches these conditions, including physical access control to the device, network isolation, and secure configuration of the host system where applicable. Operating the product outside its intended use, in particular by exposing unsecured interfaces to an untrusted network, may compromise the security and reliability of the system and is not covered by the product's cybersecurity risk assessment.

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

The *rc_visard* may be used only within the scope of its technical specification. Any other use of the product is deemed unintended use. Basler 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 6.1).

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 8) 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 8) 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.

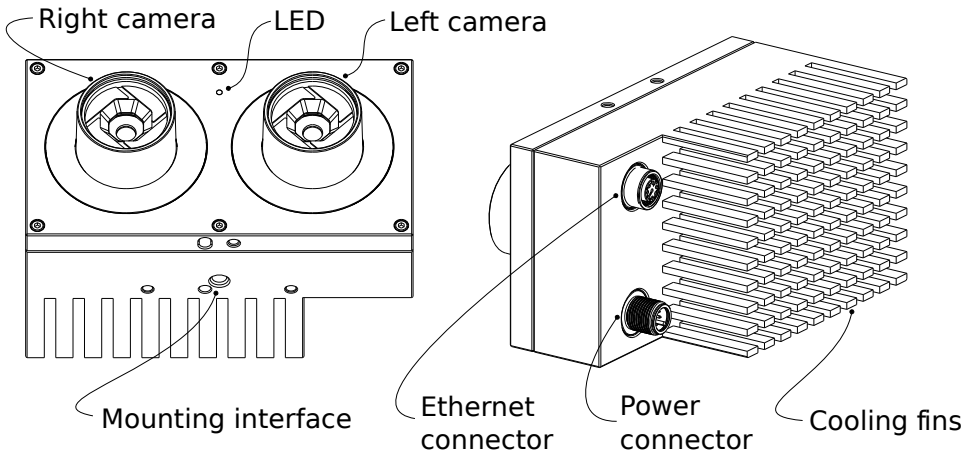


Fig. 3.1: Parts description

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

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

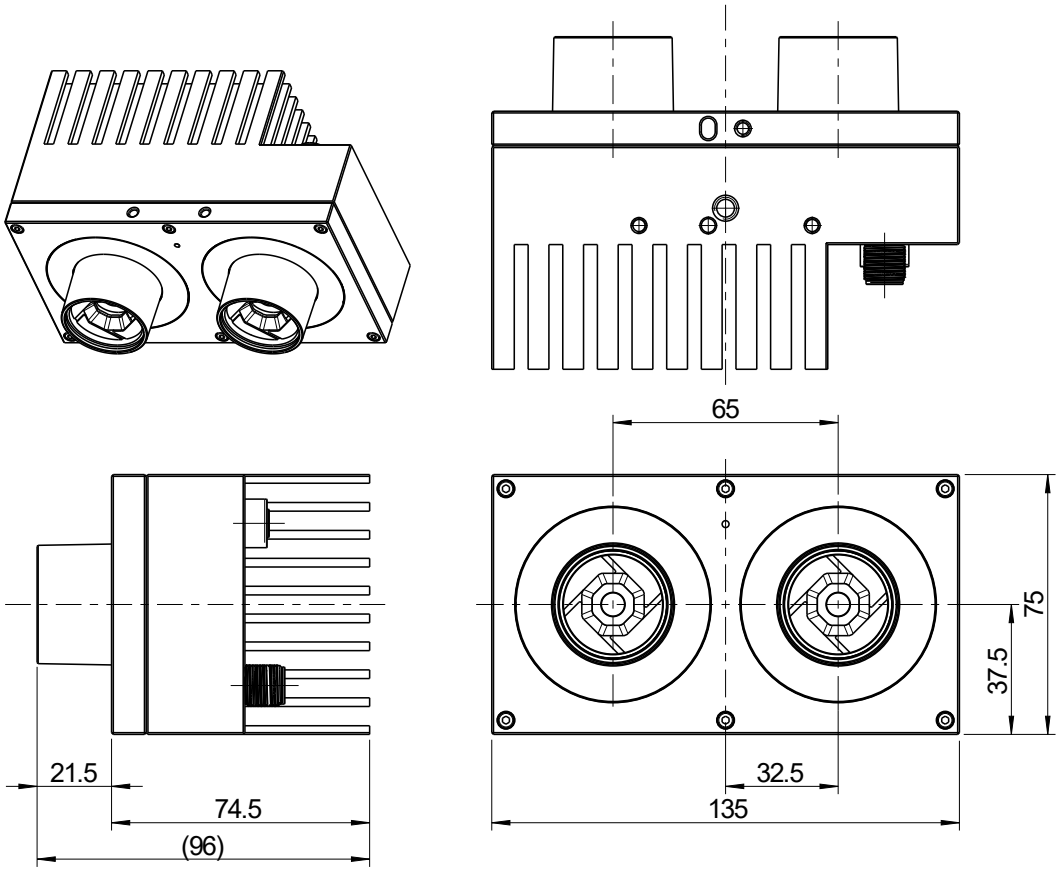


Fig. 3.2: Overall dimensions of the *rc_visard* 65

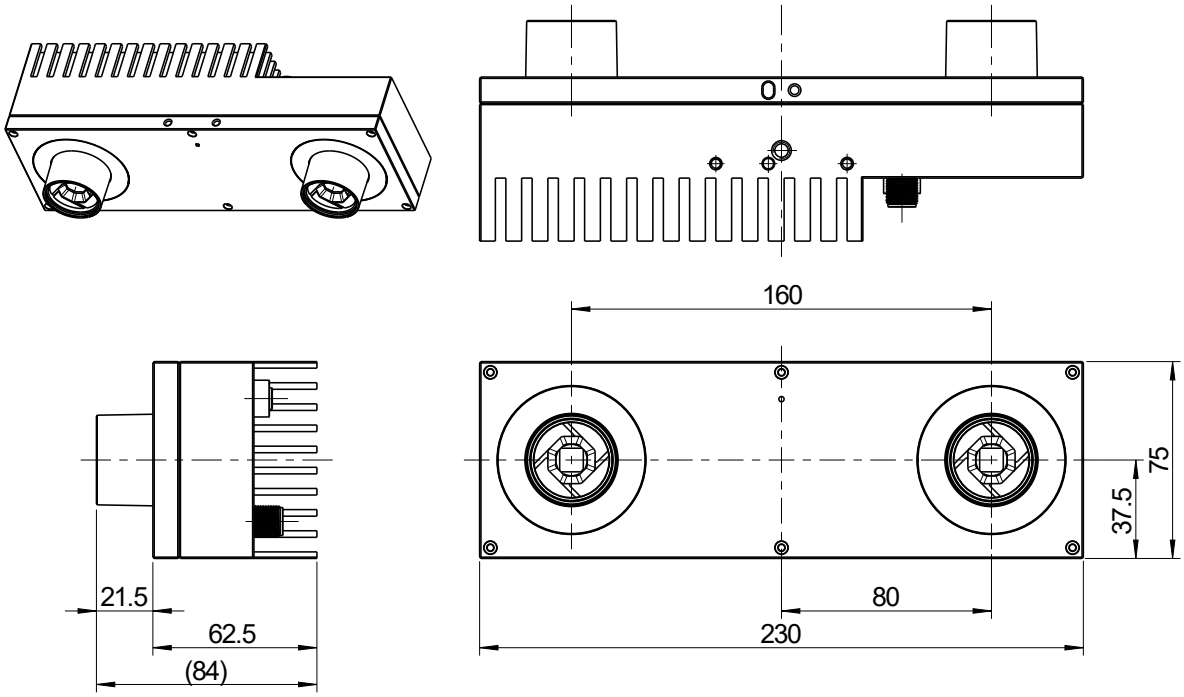


Fig. 3.3: Overall dimensions of the *rc_visard* 160

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
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 8) 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 center lines) are identical for both baseline variants. The location of both connectors on the *rc_visard* is shown in Fig. 3.4.

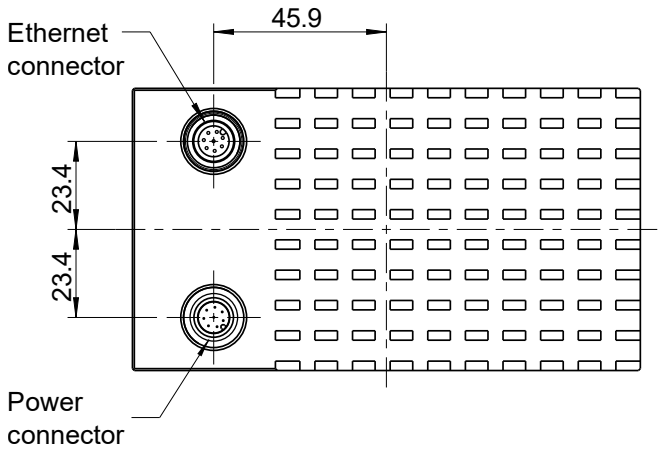


Fig. 3.4: Locations of the electrical connections for the *rc_visard*, with Ethernet on top and power on the bottom

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

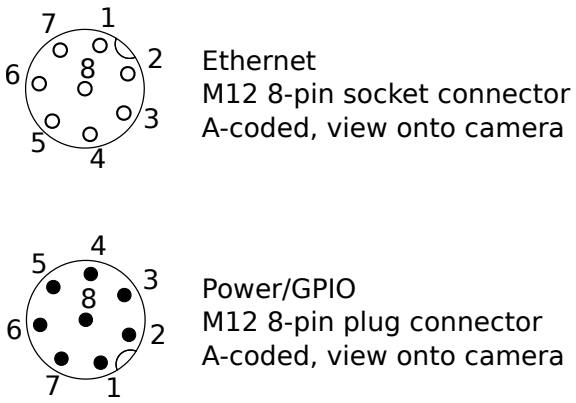


Fig. 3.5: Pin positions for power and Ethernet connector

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

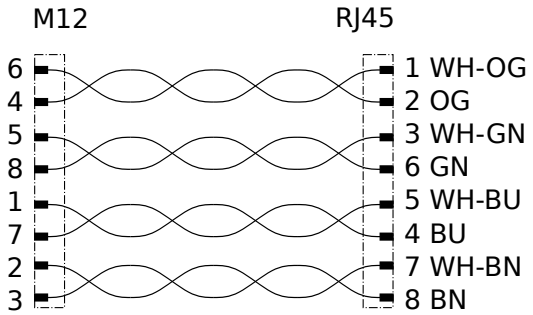


Fig. 3.6: Pin assignments for M12 to Ethernet cabling

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 IOControl module (*IO and Projector Control*, Section 5.5.4). 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.

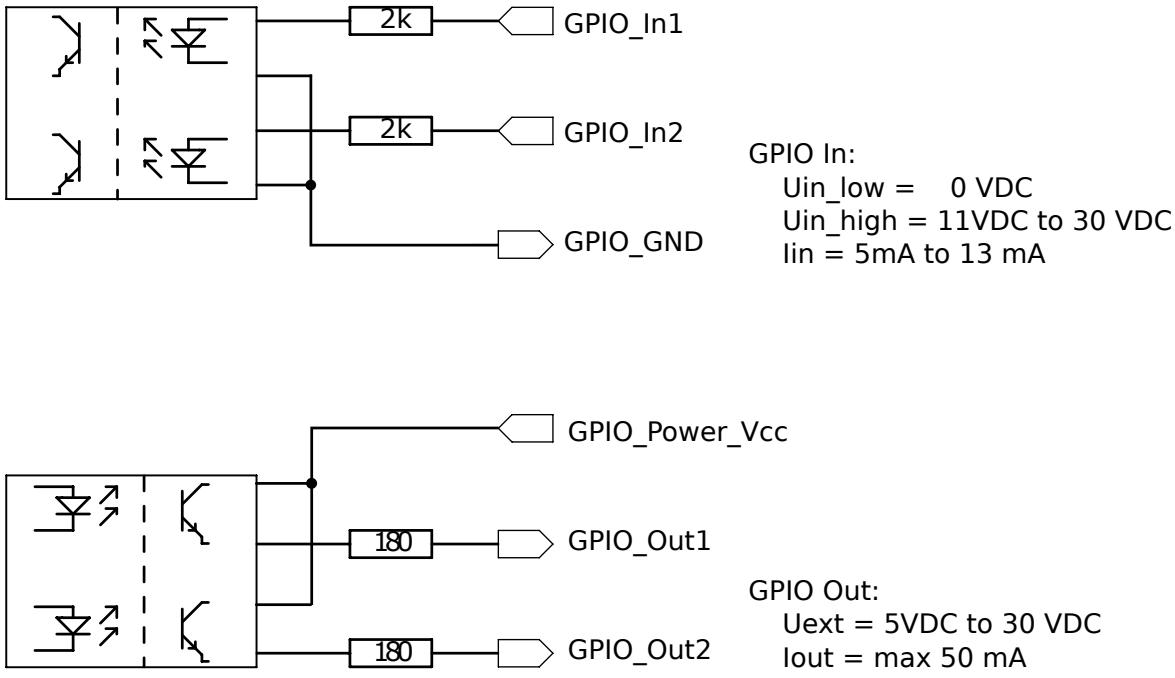


Fig. 3.7: GPIO circuitry and specifications – do not connect signals higher than 30 V

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.

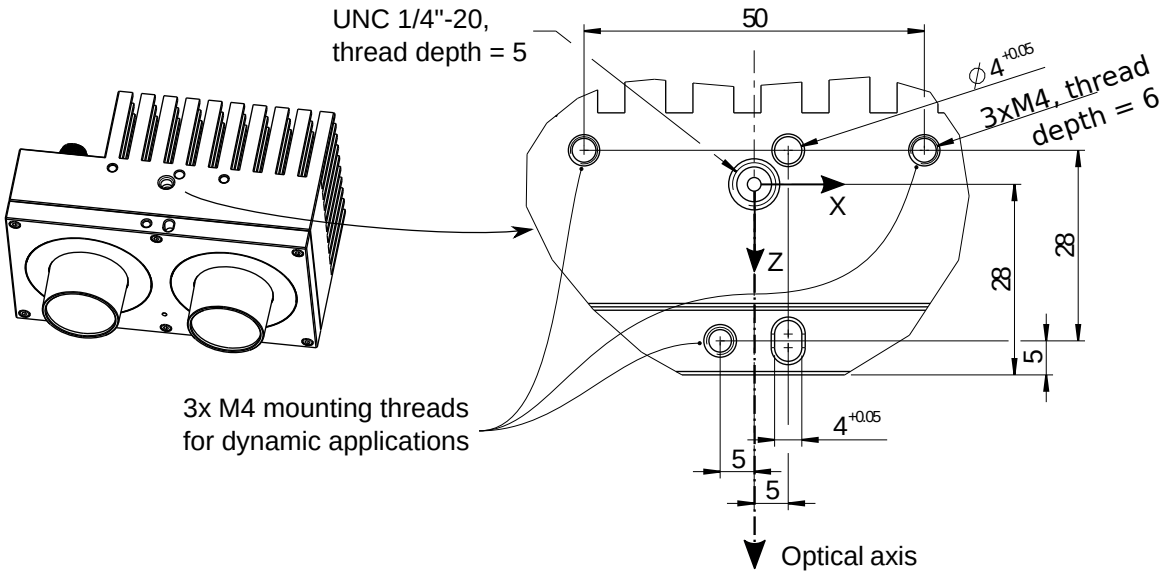


Fig. 3.8: Mounting-point for connecting the *rc_visard* to robots or other mountings

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* is shown in the next image.

The mounting-point frame for the *rc_visard* 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 and Fig. 3.10 show approximate offsets.

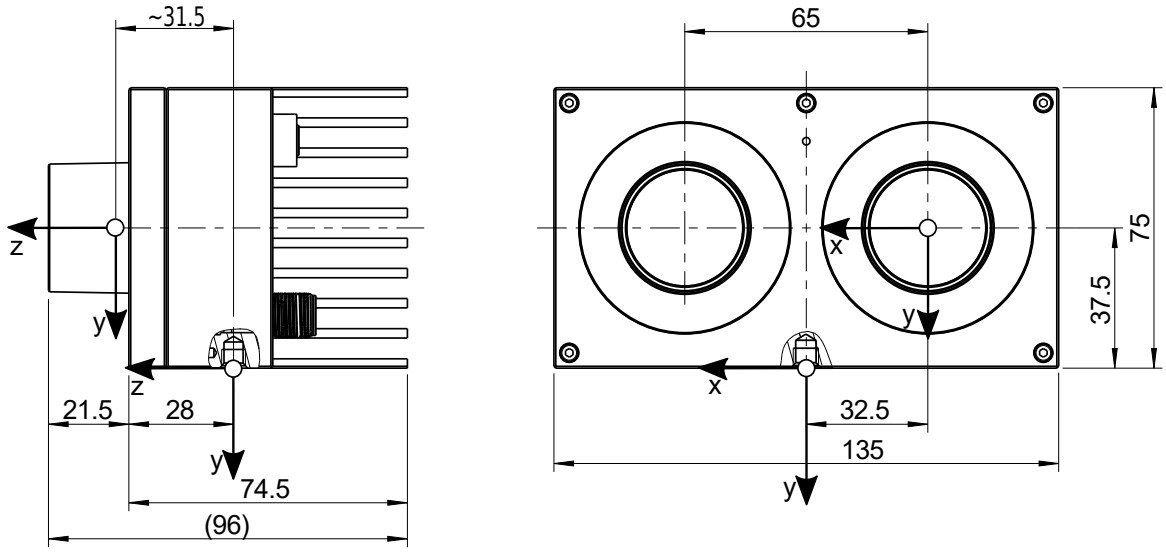


Fig. 3.9: Approximate location of sensor/camera coordinate frame (inside left lens) and mounting-point frame (at tripod thread) for the *rc_visard* 65

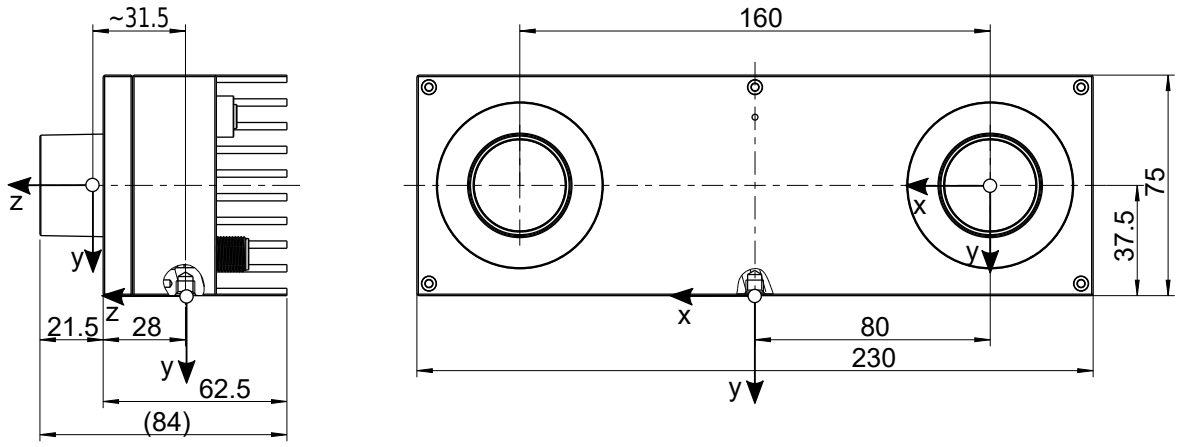


Fig. 3.10: Approximate locations of sensor/camera coordinate frame (inside left lens) and mounting-point frame (at tripod thread) for the *rc_visard* 160

Note: The correct offset between the sensor/camera frame and a robot coordinate frame can be calibrated through the *hand-eye-calibration procedure* (Section 5.5.1).

4 Installation

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

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.

4.1 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 7.4), e.g., for optionally available software modules.

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* → *Firmware & License* page of the *Web GUI* (Section 6.1).

4.2 Power up

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

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

If the network is not plugged in or the network is not properly configured, then the LED will flash red every 5 seconds. In this case, the device's network configuration should be verified. See *LED colors* (Section 9.1) for more information on the LED color codes.

4.3 Discovery of *rc_visard* devices

Basler *rc_visard* 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.

Basler offers the open-source tool `rcdiscover-gui`, which is available for Windows and Linux. The tool's Windows version consists of a single executable for Windows 7, 10 and 11, 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.

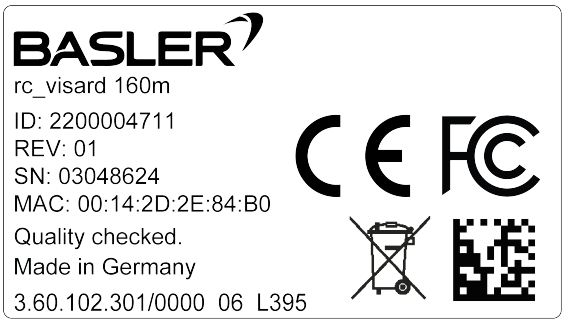


Fig. 4.1: Label on the *rc_visard* indicating model, serial number and MAC address

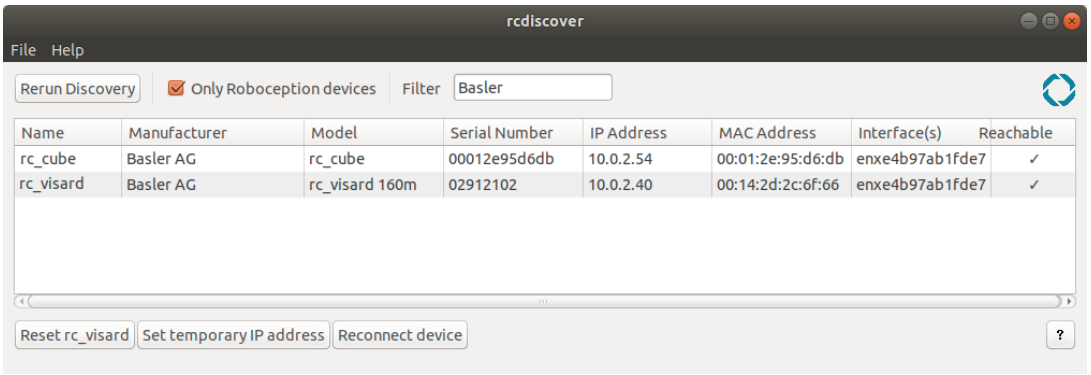


Fig. 4.2: `rcdiscover-gui` tool for finding connected GigE Vision® devices

After successful discovery, a double click on the device row opens the *Web GUI* (Section 6.1) of the device in the operating system's default web browser. Google Chrome or Mozilla Firefox are recommended as web browser.

4.3.1 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.

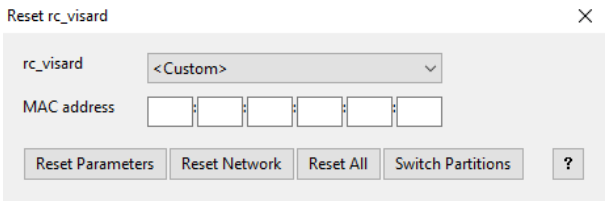


Fig. 4.3: Reset dialog of the `rcdiscover-gui` tool

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 6.1).
- *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 7.7).

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.

4.4 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

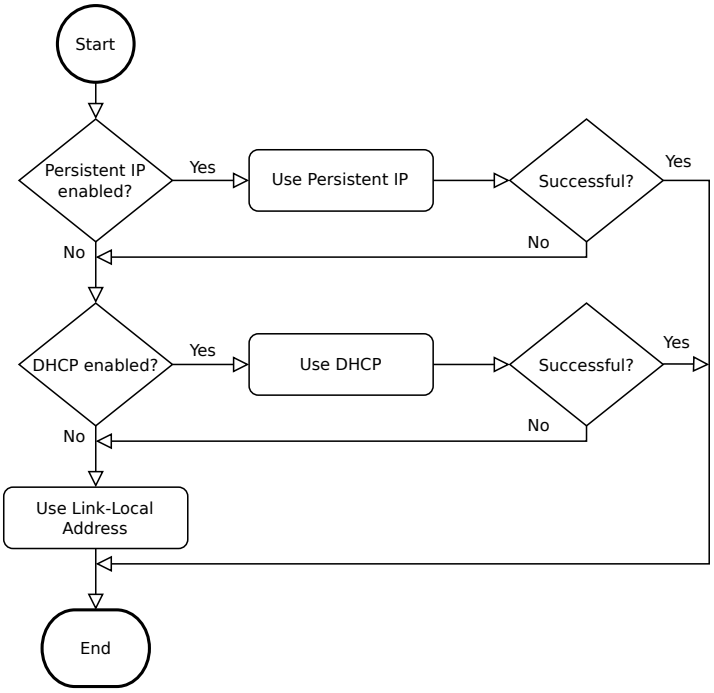


Fig. 4.4: *rc_visard*'s IP configuration method selection flowchart

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

- the *System* → *Network* page of the *rc_visard*'s Web GUI – if it is reachable in the local network already, see *Web GUI* (Section 6.1)
- any configuration tool compatible with *GigE Vision*® 2.0, or Basler'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 Basler's `rcdiscover-gui` tool, see *Discovery of rc_visard devices* (Section 4.3)

4.4.1 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>`.

4.4.2 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.

4.4.3 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 4.4.2).

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.

5 Software modules

The *rc_visard* comes with several software modules, each of which corresponds to a certain functionality and can be interfaced via its respective *node* in the *REST-API interface* (Section 6.2) or in the *Generic Robot Interface* (Section 6.4).

The *rc_visard*'s software modules can be divided into

- ***Camera module*** **(Section 5.1)** acquires images and performs planar rectification for using the camera as a measurement device. Images are provided both for further internal processing by other modules and for external use as *GenICam image streams*.
- ***Stereo matching module*** **(Section 5.2)** which provides 3D depth information such as disparity, error, and confidence images,
- ***Navigation modules*** **(Section 5.3)** which provide estimates of *rc_visard*'s current pose, velocity, and acceleration,
- ***Detection & Measure modules*** **(Section 5.4)** which provide a variety of detection functionalities, such as grasp point computation and object detection,
- ***Configuration modules*** **(Section 5.5)** which enable the user to perform calibrations and configure the *rc_visard* for specific applications.
- ***Database modules*** **(Section 5.6)** which enable the user to configure global data available to all other modules, such as load carriers, regions of interest and grippers.

5.1 Camera module

The camera module is a base module which is available on every *rc_visard* and is responsible for image acquisition and rectification. It provides various parameters, e.g. to control exposure and frame rate.

5.1.1 Parameters, status values and services

5.1.1.1 Parameters

The camera module is called `rc_camera` and is represented by the *Camera* page in the desired pipeline in the *Web GUI* (Section 6.1). The user can change the camera parameters there, or directly via the REST-API (*REST-API interface*, Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.1: The rc_camera module's run-time parameters

Name	Type	Min	Max	Default	Description
exp_auto	bool	false	true	true	Switching between auto and manual exposure (deprecated, please use exp_control instead)
exp_auto_average_max	float64	0.0	1.0	0.75	Maximum average intensity in Auto exposure mode
exp_auto_average_min	float64	0.0	1.0	0.25	Minimum average intensity in Auto exposure mode
exp_auto_mode	string	-	-	Normal	Auto-exposure mode: [Normal, Out1High, AdaptiveOut1]
exp_control	string	-	-	Auto	Exposure control mode: [Manual, Auto, HDR]
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 in Auto exposure mode
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	Exposure time in seconds in Manual exposure mode
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	Gain value in decibel if not in Auto exposure mode
gamma	float64	0.1	10.0	1.0	Gamma factor
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)

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 6.7).

Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's *Camera* page. 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.

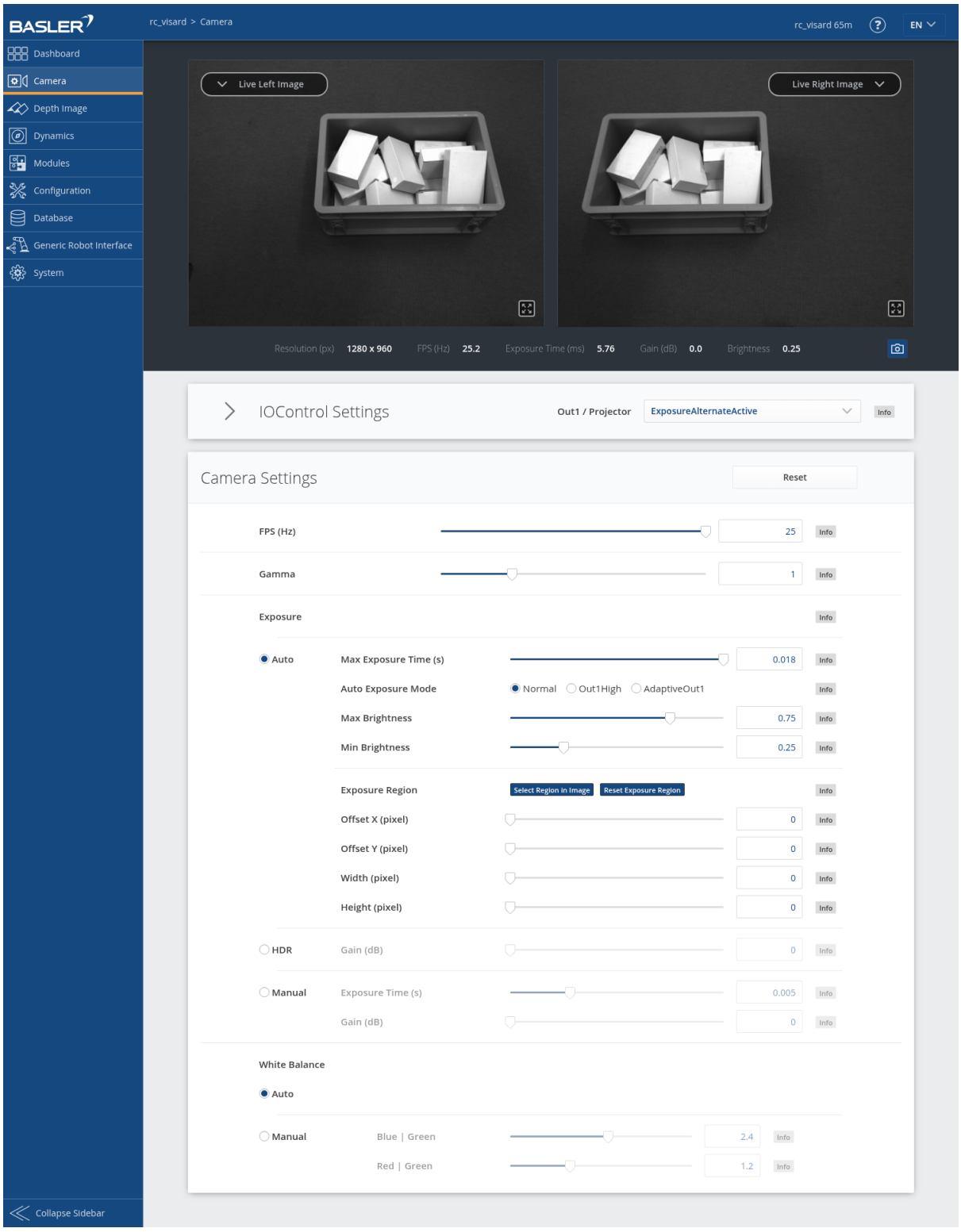


Fig. 5.1: The Web GUI's *Camera* page

fps (*FPS (Hz)*)

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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?fps=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?fps=<value>
```

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 the following figure.



Fig. 5.2: Images are internally always captured with 25 Hz. The `fps` parameter determines how many of them are sent as camera images via GigE Vision.

`gamma` (*Gamma*)

The gamma value determines the mapping of perceived light to the brightness of a pixel. A gamma value of 1 corresponds to a linear relationship. Lower gamma values let dark image parts appear brighter. A value around 0.5 corresponds to human vision.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?gamma=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?gamma=<value>
```

`exp_control` (Exposure *Auto*, *HDR* or *Manual*)

The exposure control mode can be set to *Auto*, *HDR* or *Manual*. This replaces the deprecated *exp_auto* parameter.

Auto: This is the default mode in which 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.

HDR: The HDR mode computes high-dynamic-range images by combining images with different exposure times to avoid under-exposed and over-exposed areas. This decreases the frame rate and is only suitable for static scenes.

Manual: In the manual exposure mode the exposure time and gain are kept fixed independent of the resulting image brightness.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?exp_control=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?exp_control=<value>
```

`exp_auto_mode` (*Auto Exposure Mode*)

The auto exposure mode can be set to *Normal*, *Out1High* or *AdaptiveOut1*. These modes are relevant when the *rc_visard* is used with an external light source or projector connected to the camera's GPIO Out1, which can be controlled by the IOControl module (*IO and Projector Control*, Section 5.5.4).

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.

Out1High: This exposure mode adapts the exposure time using only images with GPIO Out1 HIGH. Images where GPIO Out1 is LOW are not considered at all, which means, that the exposure time does not change when only images with Out1 LOW are acquired. This mode is recommended for using the `acquisition_mode` `SingleFrameOut1` in the stereo matching module as described in *Stereo Matching Parameters* (Section 5.2.1) and having an external projector connected to GPIO Out1, when changes in the brightness of the scene should only be considered when Out1 is HIGH. This is the case, for example, when a bright part of the robot moves through the field of view of the camera just before a detection is triggered, which should not affect the exposure time.

AdaptiveOut1: This exposure mode uses all camera images and tracks the exposure difference between images with GPIO Out1 LOW and HIGH. While the IOControl mode for GPIO Out1 is LOW, the images are under-exposed by this exposure difference to avoid over-exposure for when GPIO Out1 triggers an external projector. The resulting exposure difference is given as *Out1 Reduction* below the live images. This mode is recommended for using the `acquisition_mode` `SingleFrameOut1` in the stereo matching module as described in *Stereo Matching Parameters* (Section 5.2.1) and having an external projector connected to GPIO Out1, when changes in the brightness of the scene should be considered at all times. This is the case, for example, in applications where the external lighting changes.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?exp_auto_mode=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?exp_auto_mode=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?exp_max=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?exp_max=<value>
```

`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 precedence 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?<exp_auto_
↪average_max|exp_auto_average_min>=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?<exp_auto_average_max|exp_auto_
↪average_min>=<value>
```

`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`.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?<exp_offset_
↪x|exp_offset_y|exp_width|exp_height>=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?<exp_offset_x|exp_offset_y|exp_
↪width|exp_height>=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?exp_value=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?exp_value=<value>
```

`gain_value` (*Gain (dB)*)

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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?gain_value=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?gain_value=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?wb_auto=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?wb_auto=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/parameters?<wb_ratio_blue|wb_ratio_
↪red>=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/parameters?<wb_ratio_blue|wb_ratio_red>=<value>
```

5.1.1.2 Status values

This module reports the following status values:

Table 5.2: The `rc_camera` module's status values

Name	Description
`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
`device_trigger_sources`	Gives the available trigger sources
`exp`	Current 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`	Current frame rate of the camera images in Hertz. This value is shown in the Web GUI below the image preview as *FPS (Hz)*.
`gain`	Current gain factor in decibel. This value is shown in the Web GUI below the image preview as *Gain (dB)*.
`gamma`	Current gamma value.
`height`	Height of the camera image in pixels. This value is shown in the Web GUI below the image preview as the second part of *Resolution (px)*.
`last_timestamp_grabbed`	Timestamp of the last image acquired in case the camera is in trigger mode
`out1_reduction`	Fraction of reduction (0.0 - 1.0) of brightness for images with GPIO Out1=LOW in exp_auto_mode=AdaptiveOut1 or exp_auto_mode=Out1High. This value is shown in the Web GUI below the image preview as *Out1 Reduction (%)*.
`params_override_active`	1 if parameters are temporarily overwritten by a calibration process
`selfcalib_counter`	How often a correction has been performed by the self-calibration
`selfcalib_offset`	Current offset determined by the self-calibration
`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. This value is shown in the Web GUI below the image preview as the first part of *Resolution (px)*.

5.1.1.3 Services

The camera module offers the following services.

`reset_defaults`

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

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_camera/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_camera/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.1.2 Rectification

The model of a rectified camera is described with just one value, which is the focal length. 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.

5.1.3 Viewing and downloading images

The *rc_visard* provides the time-stamped rectified images over the GenICam interface (see *Provided image streams*, Section 6.7.6) or via the *gRPC image stream interface* (see Section 6.8).

Live streams of the images are provided with reduced quality in the *Web GUI* (Section 6.1).

The Web GUI also provides the possibility to download a snapshot of the current scene as a .tar.gz file as described in *Downloading camera images* (Section 6.1.4).

5.2 Stereo matching module

The stereo matching module is a base module which is available on every *rc_visard* and uses the rectified stereo-image pair to compute disparity, error, and confidence images.

To compute full resolution disparity, error and confidence images, an additional StereoPlus *license* (Section 7.4) is required. This license is included in every *rc_visard* purchased after 31.01.2019.

5.2.1 Parameters

The stereo matching module is called `rc_stereomatching` in the REST-API and it is represented by the *Depth Image* page in the *Web GUI* (Section 6.1). The user can change the stereo matching parameters there, or use the REST-API (*REST-API interface*, Section 6.2).

5.2.1.1 Parameter overview

This module offers the following run-time parameters:

Table 5.3: The `rc_stereomatching` module's run-time parameters

Name	Type	Min	Max	Default	Description
`acquisition_mode`	string	-	-	Continuous	Acquisition mode: [Continuous, SingleFrame, SingleFrameOut1]
`double_shot`	bool	false	true	false	Combination of disparity images from two subsequent stereo image pairs
`exposure_adapt_timeout`	float64	0.0	2.0	0.0	Maximum time in seconds to wait after triggering in SingleFrame modes until auto exposure has finished adjustments
`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
`method`	string	-	-	SGM	Stereo matching method, i.e. one of the names returned by get_methods call
`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	Quality: [Low, Medium, High, Full]. Full requires 'stereo_plus' license.
`roi_height`	int32	0	960	0	Height of region of interest. 0 for whole image.
`roi_offset_x`	int32	0	1280	0	First column of region of interest
`roi_offset_y`	int32	0	960	0	First row of region of interest
`roi_width`	int32	0	1280	0	Width of region of interest. 0 for whole image.
`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

These 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 6.7).

5.2.1.2 Description of run-time parameters

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

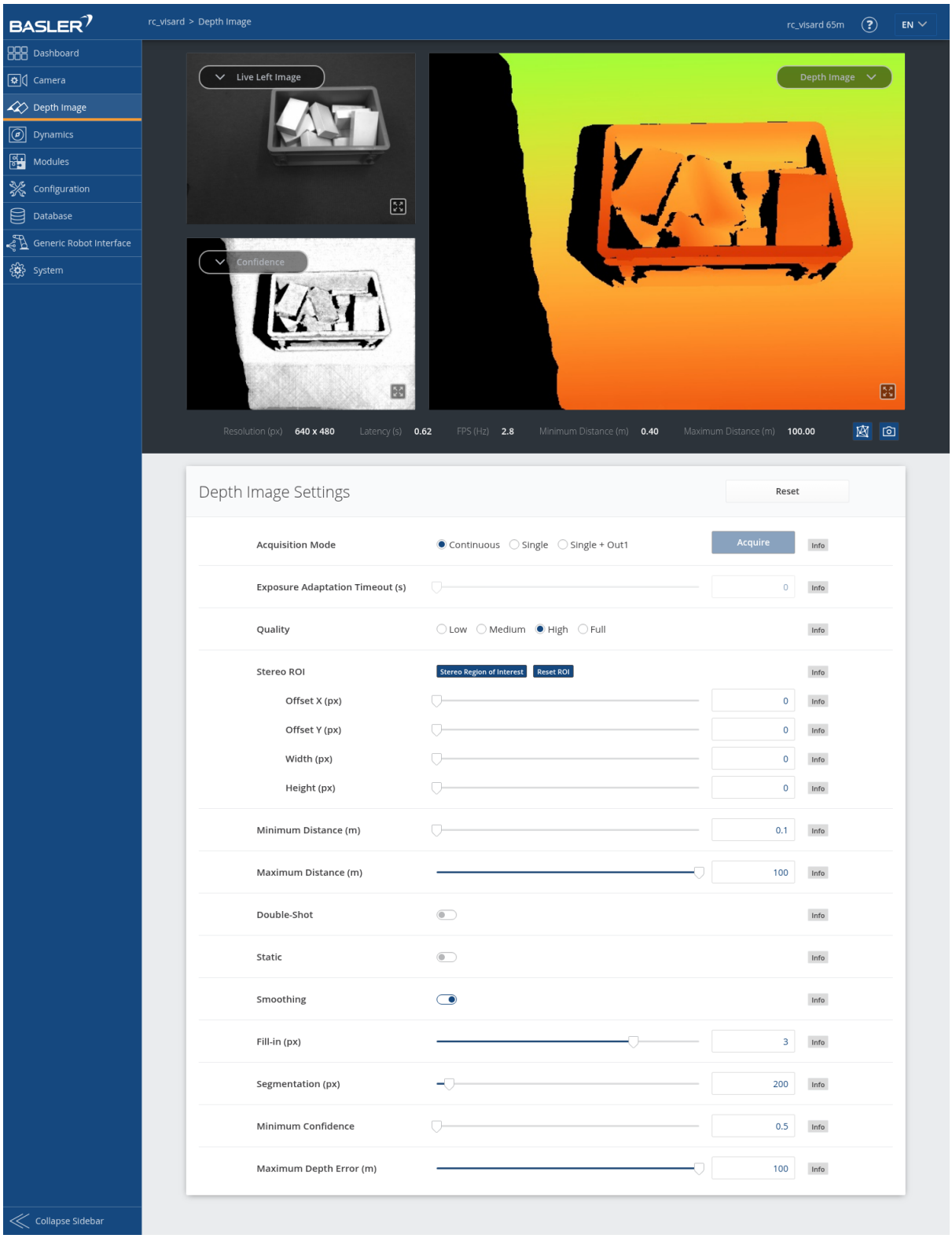


Fig. 5.3: The Web GUI's *Depth Image* page

`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 5.5.4). In this mode, `out1_mode` of the IOControl module 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*.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪acquisition_mode=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?acquisition_mode=<value>
```

`exposure_adapt_timeout` (*Exposure Adaptation Timeout*)

The exposure adaptation timeout gives the maximum time in seconds that the system will wait after triggering an image acquisition until auto exposure has found the optimal exposure time. This timeout is only used in `SingleFrame` (*Single*) or `SingleFrameOut1` (*Single + Out1*) acquisition mode with auto exposure active. This value should be increased in applications with changing lighting conditions, when images are under- oder overexposed and the resulting disparity images are too sparse. In these cases multiple images are acquired until the auto-exposure mode has adjusted or the timeout is reached, and only then the actual image acquisition is triggered.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪exposure_adapt_timeout=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?exposure_adapt_timeout=
↪<value>
```

`quality` (*Quality*)

Disparity images can be computed in different resolutions: `Full` (full image resolution), `High` (half of the full image resolution), `Medium` (quarter of the full image resolution) and `Low` (sixth of the full image resolution). Full resolution matching (`Full`) is only possible with a valid StereoPlus license. The lower the resolution, the higher the frame rate of the disparity image. 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. In case the projector is in `ExposureAlternateActive` mode, the frame rate of the images can be at most half of the camera frame rate.

A 25 Hz frame rate can be achieved only at the lowest resolution.

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.

Table 5.4: Depth image resolutions depending on the chosen quality

Quality	**Full**	**High**	**Medium**	**Low**
Resolution (pixel)	1280 x 960	640 x 480	320 x 240	214 x 160

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪quality=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?quality=<value>
```

`roi_offset_x`, `roi_offset_y`, `roi_width`, `roi_height` (*Stereo ROI*)

These values define a rectangular region in the left rectified image for limiting the area used for computing the depth values. This reduces the stereo matching run time and, thus, the latency. If either the width or height is 0, then the whole left and right images are used for stereo matching. By default, all values are 0.

The stereo ROI is visualized in the Web GUI by a white overlay of the unused image area in the left rectified image. It can be defined using the sliders or by selecting it in the image after pressing the button `Stereo Region of Interest`.

Via the REST-API, these parameters can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/services/parameters?<roi_
↪offset_x|roi_offset_y|roi_width|roi_height>=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?<roi_offset_x|roi_offset_
↪y|roi_width|roi_height>=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪mindepth=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?mindepth=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪maxdepth=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?maxdepth=<value>
```

`double_shot` (*Double-Shot*)

Enabling this option will lead to denser disparity images, but will increase processing time.

For scenes recorded with a projector in `Single + Out1` acquisition mode, or in continuous acquisition mode with the projector in `ExposureAlternateActive` mode, holes caused by reflections of the projector are filled with depth information computed from the images without projector pattern. In this case, the `double_shot` parameter must only be enabled if the scene does not change during the acquisition of the images.

For all other scenes, holes are filled with depth information computed from a downscaled version of the same image.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪double_shot=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?double_shot=<value>
```

`static_scene` (*Static*)

This option averages 8 consecutive camera images before matching. This reduces noise, which improves the stereo matching result. However, the latency increases significantly. 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪static_scene=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?static_scene=<value>
```

`smooth` (*Smoothing*)

This option activates advanced smoothing of disparity values. It is only available with a valid StereoPlus license.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪smooth=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?smooth=<value>
```

`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 preferred 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪fill=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?fill=<value>
```

`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 half 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?seg=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?seg=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪minconf=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?minconf=<value>
```

`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*).

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/parameters?
↪maxdeptherr=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/parameters?maxdeptherr=<value>
```

5.2.2 Status values

This module reports the following status values:

Table 5.5: The `rc_stereomatching` module'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 acquisition and publishing of disparity image. This value is shown in the Web GUI below the image preview as *Latency (s)*.
`width`	Current width of the disparity, error, and confidence images in pixels. This value is shown in the Web GUI below the image preview as the first number of *Resolution (px)*.
`height`	Current height of the disparity, error, and confidence images in pixels. This value is shown in the Web GUI below the image preview as the second number of *Resolution (px)*.
`mindepth`	Actual minimum working distance in meters. This value is shown in the Web GUI below the image preview as *Minimum Distance (m)*.
`maxdepth`	Actual maximum working distance in meters. This value is shown in the Web GUI below the image preview as *Maximum Distance (m)*.
`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
`reduced_depth_range`	Indicates whether the depth range is reduced due to computation resources

5.2.3 Services

The stereo matching module offers the following services.

5.2.3.1 `acquisition_trigger`

Signals the module to perform stereo matching of the next available images, if the parameter `acquisition_mode` is set to `SingleFrame` or `SingleFrameOut1`.

Details

An error is returned if the `acquisition_mode` is set to `Continuous`.

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/services/acquisition_
↪trigger
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/services/acquisition_trigger
```

Request

This service has no arguments.

Response

Possible return codes are shown below.

Table 5.6: 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

The definition for the response with corresponding datatypes is:

```
{
  "name": "acquisition_trigger",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.2.3.2 `reset_defaults`

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

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereomatching/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.2.4 Computing 3D data

The following sections describe how disparity, error and confidence images can be used to compute depth data and depth errors.

5.2.4.1 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:

$$\begin{aligned} P_x &= \frac{p_x \cdot t}{d} \\ P_y &= \frac{p_y \cdot t}{d} \\ P_z &= \frac{f \cdot t}{d}, \end{aligned} \tag{5.1}$$

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 6.7.4).

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 the equations 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.

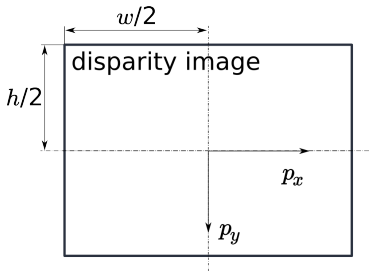


Fig. 5.4: The image coordinate frame's origin is defined to be at the image center – w is the image width and h is the image height.

The same equations, but with the corresponding GenICam parameters are given in *Image stream conversions* (Section 6.7.7).

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.

5.2.4.2 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.

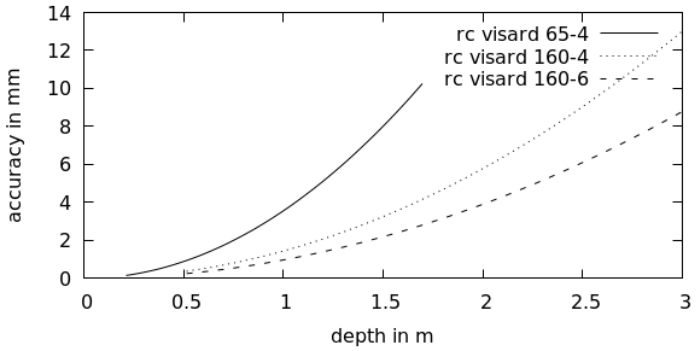
The disparity error d_{eps} (in pixels) can be converted to a depth error z_{eps} (in meters) using the focal length f (in pixels), the baseline t (in meters), and the disparity value d (in pixels) of the same pixel in the disparity image:

$$z_{eps} = \frac{d_{eps} \cdot f \cdot t}{d^2}. \tag{5.2}$$

Combining these equations 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 camera 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.



5.2.5 Viewing and downloading images and point clouds

The *rc_visard* provides time-stamped disparity, error, and confidence images over the GenICam interface (see *Provided image streams*, Section 6.7.6) or via the *gRPC image stream interface* (see Section 6.8).

Live streams of the images are provided with reduced quality on the *Depth Image* page of the *Web GUI* (Section 6.1).

The Web GUI also provides the possibility to download a snapshot of the current scene containing the depth, error and confidence images, as well as a point cloud in ply format as described in *Downloading depth images and point clouds* (Section 6.1.5).

5.3 Navigation modules

The *rc_visard*'s navigation modules contain:

- ***Sensor dynamics*** **(`rc_dynamics`, Section 5.3.1)** 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 module manages and fuses data from the following individual subcomponents:
  - ***Visual odometry*** **(`rc_stereovisodo`, Section 5.3.2)** 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 5.3.3)** 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.

- ***SLAM*** **(`rc_slam`, Section 5.3.4)** performs simultaneous localization and mapping for correcting accumulated poses. The *rc_visard*'s covered trajectory is offered via the *REST-API interface* (Section 6.2).

5.3.1 Sensor dynamics

The dynamics module is a base module which is available on every *rc_visard* and provides estimates of the sensor state. These include pose, linear velocity, linear acceleration, and rotational rates. The module 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 5.3.2).
- ***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 5.3.3).
- ***SLAM*** **(`rc_slam`)** performs simultaneous localization and mapping (*SLAM*) for correcting accumulated poses (Section 5.3.4).

Note: Using *Stereo matching module* (Section 5.2) in parallel to the dynamics module may lead to decreased localization accuracy. See *Visual odometry* (Section 5.3.2) for how to avoid this.

5.3.1.1 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.

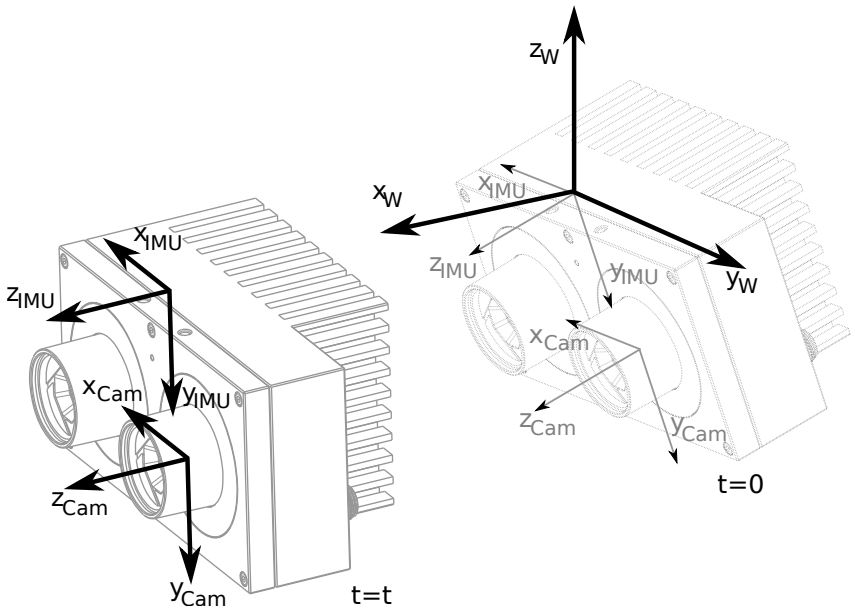


Fig. 5.5: Coordinate frames for state estimation. The IMU coordinate frame is inside the *rc_visard*'s housing. The *camera coordinate frame* (Section 3.7) is in the focal point of the left camera.

The transformation between the IMU coordinate frame and the camera/sensor frame is also estimated and provided in the *real-time dynamics stream* over the rc_dynamics interface (see *Interfaces*, Section 6).

Warning: The stereo INS module 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 module.

5.3.1.2 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 6.3):

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 module 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 5.3.1.1). 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 module 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 5.3.1.1) 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 module 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 $\mathbf{q} = (q_x, q_y, q_z, q_w)$ 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 module, 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 5.3.1.1). 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 module (see *SLAM*, Section 5.3.4) 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*.

5.3.1.3 Status values

This module reports the following status values:

Table 5.7: The `rc_dynamics` module's status values

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

5.3.1.4 Services

The sensor dynamics module 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 5.8.

Table 5.8: Possible states of the sensor dynamics module

State name	Description
IDLE	The module 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 module 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 occurred (either in stereo INS or SLAM)

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

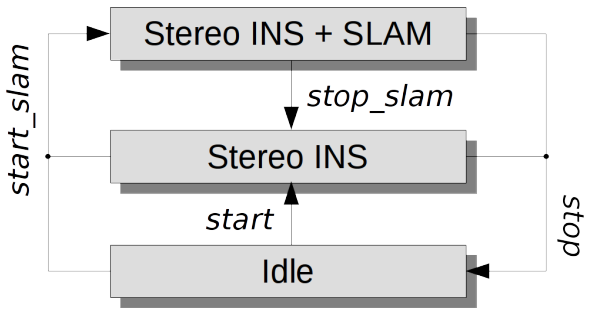


Fig. 5.6: Simplified state and transition diagram

These services shall respond quickly. Therefore, for services that cause a state transition the value of the returned `current_state` in general is the first new (intermediate) state that was transitioned to, not the final state. E.g., for the `start` command the returned `current_state` will be WAITING_FOR_INS, not

state RUNNING. If the transition does not take place within 0.1 seconds, the current state is returned. See Table 5.8 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 modules can also be started and stopped on the *Dynamics* page of the *Web GUI*.

`start`

Starts the stereo INS module.

Details

Transitions from state `IDLE` through `WAITING_FOR_INS` to `RUNNING`.

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_dynamics/services/start
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_dynamics/services/start
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "start",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

`start_slam`

Starts the SLAM and – if not yet started – the stereo INS module.

Details

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 can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_dynamics/services/start_slam
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_dynamics/services/start_slam
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "start_slam",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

stop

Stops the stereo INS and – if running – the SLAM modules.

Details

The trajectory estimate of the SLAM module will still be available. Transitions from state `RUNNING` or `RUNNING_WITH_SLAM` through `STOPPING` to `IDLE`.

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_dynamics/services/stop
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_dynamics/services/stop
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "stop",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

stop_slam

Stops the SLAM module. Stereo INS will continue to run.

Details

The trajectory estimate of the SLAM module will still be available. Transitions from state `RUNNING_WITH_SLAM` to `RUNNING`.

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_dynamics/services/stop_slam
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_dynamics/services/stop_slam
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "stop_slam",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

restart

Restarts to stereo INS. Equivalent to successive `stop` and `start`.

Details

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

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_dynamics/services/restart
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_dynamics/services/restart
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "restart",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

restart_slam

Restarts to SLAM mode. Equivalent to successive `stop` and `start_slam`.

Details

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 can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_dynamics/services/restart_slam
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_dynamics/services/restart_slam
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "restart_slam",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

`get_cam2imu_transform`

returns the transformation from camera to IMU coordinate frame.

Details

This is equivalent to the `cam2imu_transform` in the *Dynamics message* (Section 6.3.3).

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_dynamics/services/get_cam2imu_transform
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_dynamics/services/get_cam2imu_transform
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_cam2imu_transform",
  "response": {
    "name": "string",
    "parent": "string",
    "pose": {
      "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"
        }
      },
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      }
    },
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.3.2 Visual odometry

The visual odometry module is a base module which is available on every *rc_visard*.

Visual odometry is part of the sensor dynamics module. 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 module. 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 module* in parallel to the dynamics module may lead to a decreased frame rate of the visual odometry. In this case, we recommend to decrease the frame rate of the *Camera module* (effectively decreasing the frame rate of the depth image computation), to lower the computational load of stereo matching.

The visual odometry module'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 5.3.3).

5.3.2.1 Parameters

The visual odometry software module is called `rc_stereovisodo` and it is represented by the *Dynamics* page in the *Web GUI* (Section 6.1). The user can change the visual odometry parameters there, or use the REST-API (*REST-API interface*, Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.9: The `rc_stereovisodo` module'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* page.

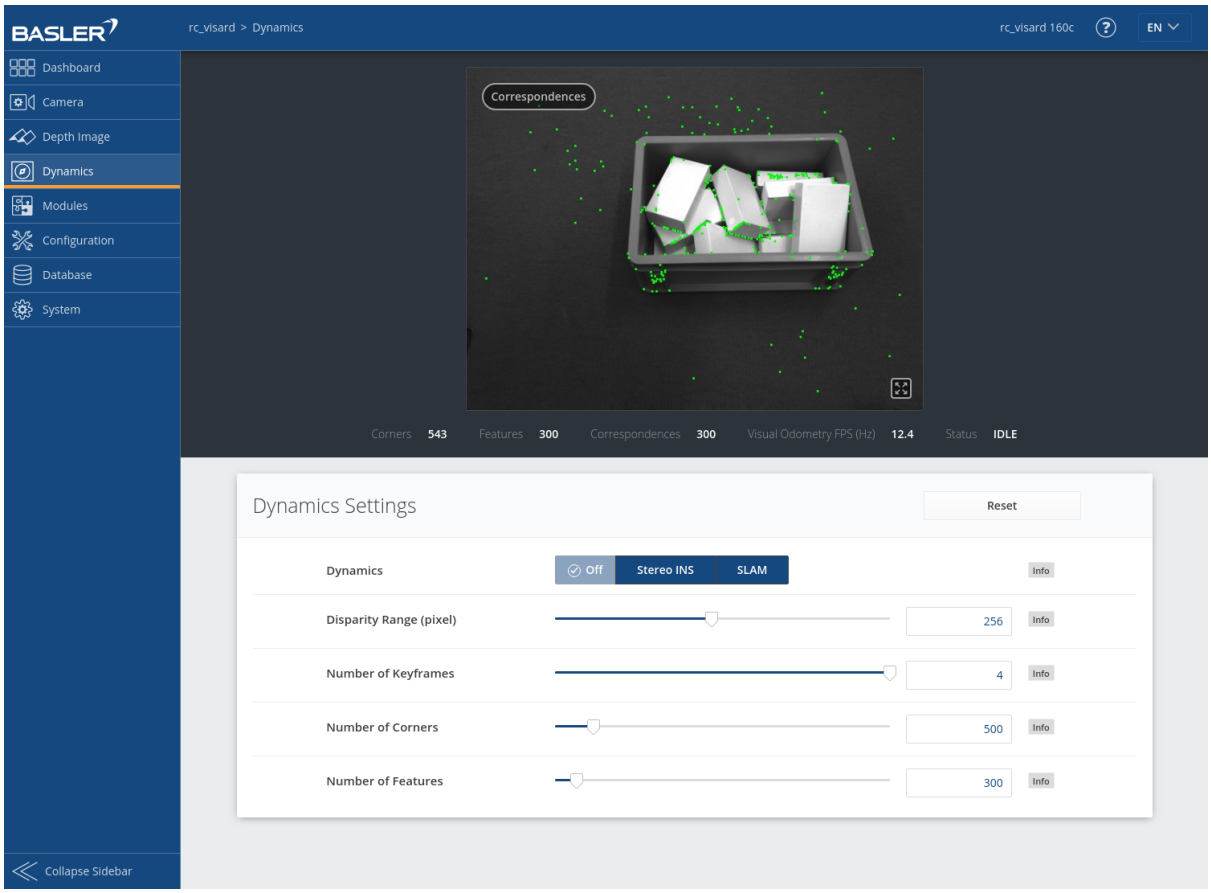


Fig. 5.7: The Web GUI's *Dynamics* page

The camera image shown on this page depicts image features as small green dots. The bold green dots are the features in the current image for which correspondences could be found in a previous keyframe. Green lines depict the motion of these features relative to the previous keyframe. This visualization should help to find a good set of parameters for visual odometry. The number of correspondences is

reported as a status parameter and is shown below the camera image on the *Web GUI*'s *Dynamics* page. For robust visual-odometry measurements, the parameters should be adjusted so that the resulting number of correspondences in the target environment is around at least 50 when the sensor is moving. The correspondence count will be larger when the *rc_visard* is static, and the number will change when the *rc_visard* moves through the environment. Short failures of the visual odometry are tolerated due to the fusion with IMU measurements. Longer failures should be avoided because they lead to large pose uncertainties and can lead to errors in the state estimation.

Each run-time parameter is represented by a row on the Web GUI's *Dynamics* page. 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:

`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 (half image resolution). 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereovisodo/parameters?disprange=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereovisodo/parameters?disprange=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereovisodo/parameters?nkey=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereovisodo/parameters?nkey=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereovisodo/parameters?ncorner=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereovisodo/parameters?ncorner=<value>
```

`nfeature` (*Number of Features*)

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

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereovisodo/parameters?nfeature=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereovisodo/parameters?nfeature=<value>
```

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 modules active on the *rc_visard*. The visual-odometry frame rate should be at least 10 Hz.

5.3.2.2 Status values

This module reports the following status values:

Table 5.10: The `rc_stereovisodo` module'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

5.3.2.3 Services

This module offers no start or stop services itself, because the *dynamics module* (Section 5.3.1) starts and stops it.

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

`reset_defaults`

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

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereovisodo/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_stereovisodo/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.3.3 Stereo INS

The stereo-vision-aided Inertial Navigation System (*INS*) module is a base module which is available on every *rc_visard* and is part of the sensor dynamics module. 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 module. In case no texture is visible for a longer period of time, the stereo INS module will stop instead of providing highly erroneous data.

5.3.3.1 Self-Calibration

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

5.3.3.2 Parameters

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

This module has no run-time parameters.

5.3.3.3 Status values

This module reports the following status values:

Table 5.11: The `rc_stereo_ins` module's status values

Name	Description
`freq`	Frequency of the stereo INS process in Hertz
`state`	String representing the internal state

5.3.4 SLAM

The SLAM module is an optional on-board module of the *rc_visard* and requires a separate SLAM *license* (Section 7.4) to be purchased. If a SLAM license is stored on the *rc_visard*, then the SLAM module is shown as *Available* on the *Web GUI*'s *License* section of the *System* page.

The SLAM module is part of the sensor dynamics module. 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 module can correct these pose errors by recognizing previously visited places.

The acronym SLAM stands for Simultaneous Localization and Mapping. The SLAM module creates a map consisting of the image features as used in the visual odometry module. 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*) into a common coordinate frame. The full trajectory can be requested from the SLAM module for this purpose.

5.3.4.1 Usage

The SLAM module can be activated at any time, either via the rc_dynamics interface (see the documentation of the respective *Services*, Section 5.3.1.4) or from the *Dynamics* page of the *Web GUI* (Section 6.1).

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

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

The full trajectory is available through the service `get_trajectory`, see *Services* (Section 5.3.4.5) below for details.

To store the feature map on the *rc_visard*, the SLAM module 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 module will therefore assume that the *rc_visard* is started in

the rough vicinity (a few meters) of the origin of the map. The origin of the map is where the Stereo-INS module was started when the map was recorded.

5.3.4.2 Memory limitations

In contrast to the other software modules running on the *rc_visard*, the SLAM module 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 module increase over time.

Given the memory limitations of the hardware, the SLAM module needs to reduce its own memory footprint when running continuously. When the available memory runs low, the SLAM module 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 module 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 module will be restarted. If the `autorecovery` parameter is set to true, the SLAM module 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 5.3.1.4).

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 module* 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.

5.3.4.3 Parameters

The SLAM module is called `rc_slam` in the REST-API. The user can change the SLAM parameters using the *REST-API interface* (Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.12: The `rc_slam` module'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

5.3.4.4 Status values

This module reports the following status values:

Table 5.13: The `rc_slam` module'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 5.14: Possible states of the `rc_slam` module

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

5.3.4.5 Services

Note: Activation and deactivation of the SLAM module is done via the service interface of rc_dynamics (see *Services*, Section 5.3.1.4).

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 module offers the following services.

`get_trajectory`

returns the trajectory.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_slam/services/get_trajectory
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_slam/services/get_trajectory
```

Request

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.

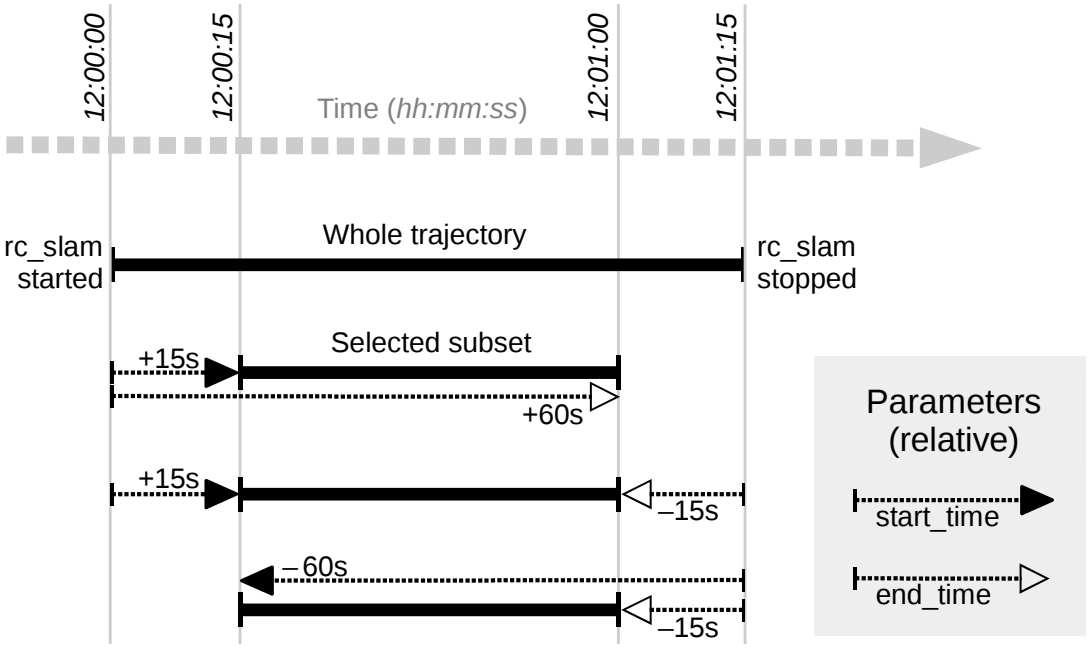


Fig. 5.8: Examples for combinations of relative start and end times for the `get_trajectory` service. All combinations shown select the same subset of the trajectory.

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 request arguments with corresponding datatypes is:

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

Response

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)

The definition for the response with corresponding datatypes is:


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


save_map

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

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_slam/services/save_map
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_slam/services/save_map
```

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

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "save_map",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

`load_map`

loads a previously saved map.

Details

This is only applicable when the SLAM module 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 can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_slam/services/load_map
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_slam/services/load_map
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "load_map",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

remove_map

removes the stored map from the persistent memory.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_slam/services/remove_map
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_slam/services/remove_map
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "remove_map",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

reset

clears the internal state of the SLAM module.

Details

This service is to be used after stopping the SLAM module using the rc_dynamics interface (see the respective *Services*, Section 5.3.1.4). The SLAM module 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 can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_slam/services/reset
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_slam/services/reset
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

5.4 Detection & Measure modules

The *rc_visard* offers software modules for different detection and measuring applications:

- ***Measure*** **(`rc_measure`, Section 5.4.1)** provides measure functionalities, such as depth measurements.
- ***LoadCarrier*** **(`rc_load_carrier`, Section 5.4.2)** allows detecting load carriers and their filling levels.
- ***TagDetect*** **(`rc_april_tag_detect` and `rc_qr_code_detect`, Section 5.4.3)** allows the detection of AprilTags and QR codes, as well as the estimation of their poses.
- ***ItemPick*** **(`rc_itempick`, Section 5.4.4)** provides an out-of-the-box perception solution for robotic pick-and-place applications of unknown objects.
- ***BoxPick*** **(`rc_boxpick`, Section 5.4.5)** provides an out-of-the-box perception solution for robotic pick-and-place applications of boxes or textured boxes.
- ***Pack*** **(`rc_pack`, Section 5.4.6)** computes place poses for objects inside a packing volume or load carrier.
- ***SilhouetteMatch*** **(`rc_silhouettematch`, Section 5.4.7)** provides an object detection solution for objects placed on a plane or stacked planar objects.

These modules are optional and can be activated by purchasing a separate *license* (Section 7.4).

5.4.1 Measure

5.4.1.1 Introduction

The Measure module allows measuring of depth values in a specific region of interest and estimating planes from the current sensor data.

The Measure module is a base module which is available on every *rc_visard*.

5.4.1.2 Measuring Depth

The Measure module provides functionality to measure depth values in the current scene in a 2D region of interest. Optionally, the region of interest can be subdivided into up to 100 cells, for which separate depth measurements are returned in addition to the overall depth measurements of the whole region.

A depth measurement consist of the average depth `mean_z`, the minimum depth `min_z` and the maximum depth `max_z`, each containing 3D coordinates. The coordinates of the `min_z` and `max_z` measurements correspond to the point in the cell or overall region with the minimum and maximum distance from the

camera, respectively. The `x` and `y` coordinates of the `mean_z` measurements define a point in the center of the cell or the overall region and the `z` coordinate is determined by the average of all depth value measurements (distances from the camera) in this region. Additionally, a `coverage` value is returned for each cell and the overall region, which is a number between 0 and 1 that reflects the fraction of valid depth values inside the respective region. A coverage value of 0 means that the cell is invalid and no depth value could be computed.

When the external `pose_frame` is used for the depth measurements, all 3D coordinates are computed as described above, but then transformed to the external frame. That means, the depth is always measured along the line of sight of the camera, independently of the chosen pose frame.

5.4.1.3 Estimating planes

The Measure module provides functionality to estimate a plane in the current scene, e.g. the surface of a conveyor belt or a table.

The plane is returned in Hesse normal form with a `normal` and a `distance` (see *Plane*, Section 5.6.3.2). Each plane estimation triggers a new image acquisition.

The plane can be estimated with two different methods (`plane_estimation_method`):

- `DEPTH`: The plane is estimated from the point cloud. If several planes are found, `plane_preference` selects whether the `FARTHEST` (default) or the `CLOSEST` plane is returned.
- `APRILTAG`: The plane is estimated from at least three AprilTags lying on a common plane. This method requires a valid license for AprilTag detection (see *TagDetect*, Section 5.4.3). The AprilTags are detected in images without projector pattern. If a projector is used, `out1_mode` of the IOControl module must be set to `Low` or `ExposureAlternateActive`.

Optionally, the plane estimation can be restricted to a 2D region of interest. An `offset` in meters can be given, which shifts the returned plane along the line of sight of the camera. A positive offset moves the plane towards the camera.

The plane estimation fails if no plane is found, if fewer than three AprilTags are visible or they do not lie on a common plane, or if the plane is tilted by more than 45° against the optical axis of the camera.

The plane can be stored in the *PrimitiveDB* (Section 5.6.3) with the `set_plane` service and then be used, for example, as reference plane for dimensioning in *BoxPick* (Section 5.4.5.2).

5.4.1.4 Interaction with other modules

Internally, the Measure module depends on, and interacts with other on-board modules as listed below.

Note: All changes and configuration updates to these modules will affect the performance of the Measure module.

Depth Images

The Measure module internally makes use of the following data:

- Disparity images from the *Stereo matching module* (`rc_stereomatching`, Section 5.2)

Hand-eye calibration

In case the camera has been calibrated to a robot, the Measure module can automatically provide points in the robot coordinate frame. For the Measure node's *Services* (Section 5.4.1.7), the frame of the output points can be controlled with the `pose_frame` argument.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). All points provided by the module are in the camera frame, and no prior knowledge about the pose of the camera in the environment is required. It is the user's responsibility to update the configured points if the camera frame moves (e.g. with a robot-mounted camera).
2. **External frame** (`external`). All points provided by the module are in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) 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.

5.4.1.5 Parameters

The Measure module is called `rc_measure` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Modules* → *Measure*.

Parameter overview

This module has no run-time parameters.

5.4.1.6 Status values

The Measure module reports the following status values:

Table 5.15: The `rc_measure` module's status values

Name	Description
`data_acquisition_time`	Time in seconds required to acquire depth image
`last_return_code_message`	Return code message of the last detection
`last_return_code_value`	Return code value of the last detection
`last_timestamp_processed`	The timestamp of the last processed depth image
`processing_time`	Processing time of the last measurement in seconds

5.4.1.7 Services

The user can explore and call the Measure module's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1) on the *Measure* page under *Modules*.

The Measure module offers the following services.

`measure_depth`

Computes the mean, minimum and maximum depth in a given region of interest, which can optionally be subdivided into cells.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_measure/services/measure_depth
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_measure/services/measure_depth
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.1.4).

Optional arguments:

`region_of_interest_2d_id` is the ID of the 2D region of interest (see *RoiDB*, Section 5.6.2) that will be used for the depth measurements.

`region_of_interest_2d` is an alternative on-the-fly definition of the region of interest for the depth measurements. This region of interest will be ignored if a `region_of_interest_2d_id` is given. The region of interest is always defined on the camera image with full resolution, where `offset_x` and `offset_y` are the pixel coordinates of the upper left corner of the rectangular region of interest, and `width` and `height` are the width and height of it in pixels. Default is a region of interest covering the whole image.

`cell_count` is the number of cells in x and y direction into which the region of interest is divided. If not given, a cell count of 0, 0 is assumed and only the overall values will be computed. The total cell count is computed as product of the x and y values must not exceed 100.

`data_acquisition_mode`: if set to `CAPTURE_NEW` (default), a new image dataset will be used for the measurement. If set to `USE_LAST`, the previous dataset will be used for the measurement.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.1.4).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "cell_count": {
      "x": "uint32",
      "y": "uint32"
    },
    "data_acquisition_mode": "string",
    "pose_frame": "string",
    "region_of_interest_2d": {
      "height": "uint32",
      "offset_x": "uint32",
      "offset_y": "uint32",
      "width": "uint32"
    },
    "region_of_interest_2d_id": "string",
    "robot_pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
```

(continues on next page)

(continued from previous page)

```
        "y": "float64",
        "z": "float64"
      }
    }
  }
}
```

Response

`cells` contains the depth measurements of all requested cells. The cells are always ordered from left to right and top to bottom in image coordinates.

`overall` contains the depth measurements of the full region of interest.

`coverage` contains the valid pixel ratio as described in *Measuring Depth* (Section 5.4.1.2).

`mean_z`, `min_z` and `max_z` contains the measurement coordinates as described in *Measuring Depth* (Section 5.4.1.2).

`region_of_interest_2d` returns the definition of the requested region of interest for the depth measurement.

`pose_frame` contains the pose frame of the depth measurement coordinates.

The definition for the response with corresponding datatypes is:

```
{
  "name": "measure_depth",
  "response": {
    "cell_count": {
      "x": "uint32",
      "y": "uint32"
    },
    "cells": [
      {
        "coverage": "float64",
        "max_z": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "mean_z": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "min_z": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      }
    ],
    "overall": {
      "coverage": "float64",
      "max_z": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "mean_z": {
        "x": "float64",
        "y": "float64",
```

(continues on next page)

(continued from previous page)


```
        "z": "float64"
      },
      "min_z": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "pose_frame": "string",
    "region_of_interest_2d": {
      "height": "uint32",
      "offset_x": "uint32",
      "offset_y": "uint32",
      "width": "uint32"
    },
    "return_code": {
      "message": "string",
      "value": "int16"
    },
    "timestamp": {
      "nsec": "int32",
      "sec": "int32"
    }
  }
}
```


compute_plane

Estimates a plane from the current sensor data as described in *Estimating planes* (Section 5.4.1.3) and returns it without storing it.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_measure/services/compute_plane
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_measure/services/compute_plane
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.1.4).

`plane_estimation_method`: method for estimating the plane, either `DEPTH` or `APRILTAG`.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.1.4).

Optional arguments:

`depth`: options for the `DEPTH` method. `plane_preference` selects whether the `FARTHEST` (default) or the `CLOSEST` plane is returned if several planes are found.

`region_of_interest_2d_id` is the ID of the 2D region of interest (see *RoiDB*, Section 5.6.2) that will be used for the plane estimation. Default is the whole image.

`offset`: shift of the returned plane in meters along the line of sight of the camera. A positive offset moves the plane towards the camera.

The definition for the request arguments with corresponding datatypes is:


```
{
  "args": {
    "depth": {
      "plane_preference": "string"
    },
    "offset": "float64",
    "plane_estimation_method": "string",
    "pose_frame": "string",
    "region_of_interest_2d_id": "string",
    "robot_pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    }
  }
}
```


Response

`plane` contains the estimated plane with its `normal`, `distance` and `pose_frame` (see *Plane*, Section 5.6.3.2).

`timestamp`: acquisition time of the data the plane was estimated from.

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

The definition for the response with corresponding datatypes is:


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

5.4.1.8 Return codes

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 5.16: Return codes of the Measure module's services

Code	Description
0	Success
-1	An invalid argument was provided
-4	Data acquisition took longer than allowed
-8	The request is not applicable, e.g. the `APRILTAG` method is not supported in the current pipeline
-11	Sensor not connected, not supported or not ready
-101	No plane could be estimated, e.g. no plane was found or the AprilTags do not lie on a common plane
-104	The estimated plane is tilted too strongly against the optical axis of the camera

5.4.2 LoadCarrier

5.4.2.1 Introduction

The LoadCarrier module allows the detection of load carriers, which is usually the first step when objects or grasp points inside a bin should be found. The models of the load carriers to be detected have to be defined in the *LoadCarrierDB* (Section 5.6.1) module.

The LoadCarrier module is an optional on-board module of the *rc_visard* and is licensed with any of the modules *ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5) or *SilhouetteMatch* (Section 5.4.7). Otherwise it requires a separate LoadCarrier *license* (Section 7.4) to be purchased.

5.4.2.2 Detection of load carriers

The load carrier detection algorithm detects load carriers that match a specific load carrier model, which must be defined in the *LoadCarrierDB* (Section 5.6.1) module. The load carrier model is referenced by its ID, which is passed to the load carrier detection. The detection of a load carrier is based on the detection of its rectangular rim. For this, it uses lines detected in the left camera image and the depth values of the load carrier rim. Thus, the rim should form a contrast to the background and the disparity image must be dense on the rim.

If multiple load carriers of the specified load carrier ID are visible in the scene, all of them will be detected and returned by the load carrier detection.

By default, when `assume_gravity_aligned` is true and gravity measurements are available, the algorithm searches for load carriers whose rim planes are perpendicular to the measured gravity vector. To detect tilted load carriers, `assume_gravity_aligned` must be set to false or the load carrier's approximate orientation must be specified as `pose` and the `pose_type` should be set to `ORIENTATION_PRIOR`.

Load carriers can be detected at a distance of up to 3 meters from the camera.

When a 3D region of interest (see *RoiDB*, Section 5.6.2) is used to limit the volume in which load carriers should be detected, only the load carriers' rims must be fully included in the region of interest.

The detection algorithm returns the poses of the load carriers' origins (see *Load carrier definition*, Section 5.6.1.2) in the desired pose frame.

The detection functionality also determines if the detected load carriers are `overfilled`, which means, that objects protrude from the plane defined by the load carrier's outer part of the rim.

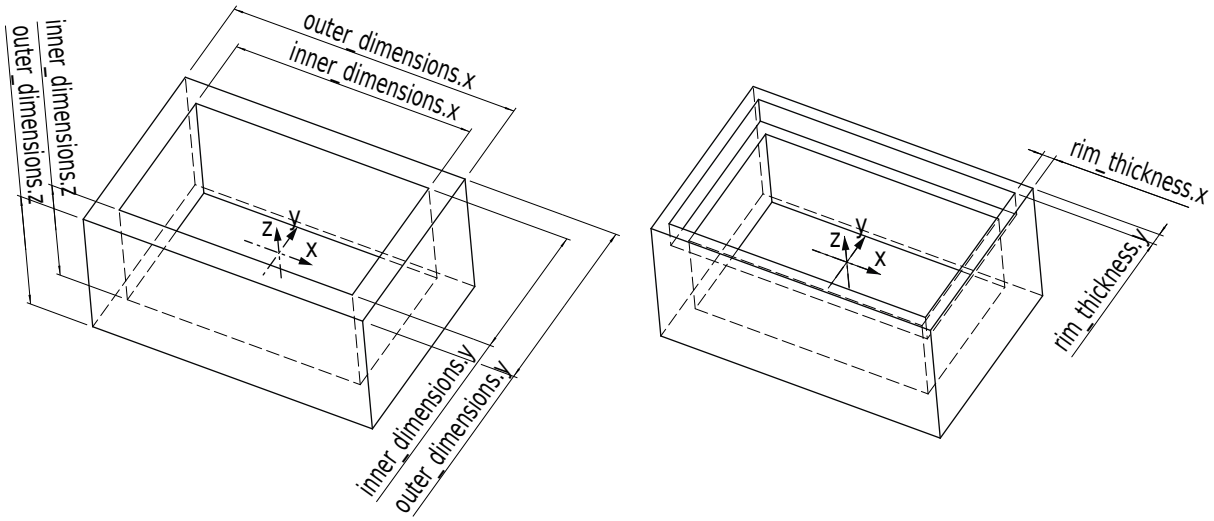


Fig. 5.9: Load carrier models and reference frames

5.4.2.3 Detection of filling level

The LoadCarrier module offers the `detect_filling_level` service to compute the filling level of all detected load carriers.

The load carriers are subdivided into a configurable number of cells in a 2D grid. The maximum number is 10x10 cells. 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 5.4.2.4). 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.

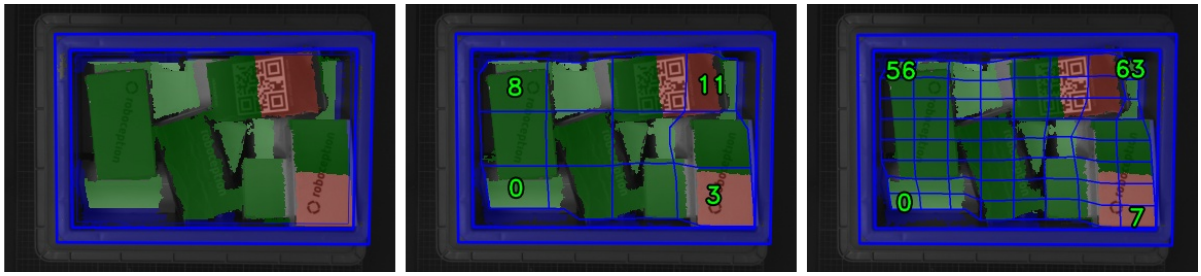


Fig. 5.10: Visualizations of the load carrier filling level: overall filling level without grid (left), 4x3 grid (center), 8x8 grid (right). The load carrier content is shown in a green gradient from white (on the load carrier floor) to dark green. The overfilled regions are visualized in red. Numbers indicate cell IDs.

5.4.2.4 Interaction with other modules

Internally, the LoadCarrier module depends on, and interacts with other on-board modules as listed below.

Note: All changes and configuration updates to these modules will affect the performance of the LoadCarrier module.

Camera and depth data

The LoadCarrier module makes internally use of the following data:

- Rectified images from the *Camera module* (`rc_camera`, Section 5.1);
- Disparity, error, and confidence images from the *Stereo matching module* (`rc_stereomatching`, Section 5.2)

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

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* module (`rc_iocontrol`, Section 5.5.4), it is recommended to connect the projector to GPIO Out 1 and set the stereo-camera module's acquisition mode to `SingleFrameOut1` (see *Stereo matching parameters*, Section 5.2.1), 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 5.5.4.1).

In either case, the *Auto Exposure Mode* `exp_auto_mode` should be set to `AdaptiveOut1` to optimize the exposure of both images.

No additional changes are required to use the LoadCarrier module in combination with a random dot projector.

Hand-eye calibration

In case the camera has been calibrated to a robot, the LoadCarrier module can automatically provide poses in the robot coordinate frame. For the LoadCarrier node's *Services* (Section 5.4.2.7), 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 module 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 module are in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) 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.

5.4.2.5 Parameters

The LoadCarrier module is called `rc_load_carrier` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Modules* → *LoadCarrier*. The user can explore and configure the LoadCarrier module's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.17: The `rc_load_carrier` module's run-time parameters

Name	Type	Min	Max	Default	Description
`assume_gravity_aligned`	bool	false	true	true	When true, only gravity-aligned load carriers are detected, if gravity measurement is available.
`crop_distance`	float64	0.0	0.05	0.005	Safety margin in meters by which the load carrier inner dimensions are reduced to define the region of interest for detection
`min_plausibility`	float64	0.0	0.99	0.8	Indicates how much of the plane surrounding the load carrier rim must be free to count as valid detection
`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 *LoadCarrier Settings* section of the Web GUI's *LoadCarrier* page under *Modules*. 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. The parameters are prefixed with `load_carrier_` when they are used outside the `rc_load_carrier` module from another detection module using the *REST-API interface* (Section 6.2).

`assume_gravity_aligned` (*Assume Gravity Aligned*)

If this parameter is set to true, then only load carriers without tilt will be detected. This can speed up the detection. If this parameter is set to false, tilted load carriers will also be detected.

This parameter is ignored for load carriers with an orientation prior.

`model_tolerance` (*Model Tolerance*)

indicates how much the estimated load carrier dimensions are allowed to differ from the load carrier model dimensions in meters.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_load_carrier/parameters?model_tolerance=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/parameters?model_tolerance=<value>
```

`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 (ref. Fig. 5.41).

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_load_carrier/parameters?crop_distance=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/parameters?crop_distance=<value>
```

`min_plausibility` (*Minimum Plausibility*):

The minimum plausibility defines how much of the plane around the load carrier rim must at least be free to count as valid detection. Increase the minimal plausibility to reject false positive detections and decrease the value in case a clearly visible load carrier cannot be detected.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_load_carrier/parameters?min_plausibility=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/parameters?min_plausibility=<value>
```

5.4.2.6 Status values

The LoadCarrier module reports the following status values:

Table 5.18: The `rc_load_carrier` module's status values

Name	Description
`data_acquisition_time`	Time in seconds required to acquire image pair
`last_return_code_message`	Return code message of the last detection
`last_return_code_value`	Return code value of the last detection
`last_timestamp_processed`	The timestamp of the last processed image pair
`load_carrier_detection_time`	Processing time of the last detection in seconds

5.4.2.7 Services

The user can explore and call the LoadCarrier module's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1) on the *LoadCarrier* page under *Modules*.

The LoadCarrier module offers the following services.

`detect_load_carriers`

Triggers a load carrier detection as described in *Detection of load carriers* (Section 5.4.2.2).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_load_carrier/services/detect_load_
↪carriers
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/detect_load_carriers
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.2.4).

`load_carrier_ids`: IDs of the load carriers which should be detected. Currently only one ID can be specified.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.2.4).

Optional arguments:

`region_of_interest_id`: ID of the 3D region of interest where to search for the load carriers.

`region_of_interest_2d_id`: ID of the 2D region of interest where to search for the load carriers.

Note: Only one type of region of interest can be set.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "load_carrier_ids": [
      "string"
    ],
    "pose_frame": "string",
    "region_of_interest_2d_id": "string",
    "region_of_interest_id": "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"
      }
    }
  }
}
```

Response

`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.

The definition for the response with corresponding datatypes is:

```
{
  "name": "detect_load_carriers",
  "response": {
    "load_carriers": [
      {
        "height_open_side": "float64",
        "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_ledge": {
          "x": "float64",
          "y": "float64"
        },
        "rim_step_height": "float64",
        "rim_thickness": {
          "x": "float64",
          "y": "float64"
        },
        "type": "string"
```

(continues on next page)

(continued from previous page)

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

detect_filling_level

Triggers a load carrier filling level detection as described in *Detection of filling level* (Section 5.4.2.3).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_load_carrier/services/detect_filling_
↪level
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/detect_filling_level
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.2.4).

`load_carrier_ids`: IDs of the load carriers which should be detected. Currently only one ID can be specified.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.2.4).

Optional arguments:

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

`region_of_interest_id`: ID of the 3D region of interest where to search for the load carriers.

`region_of_interest_2d_id`: ID of the 2D region of interest where to search for the load carriers.

Note: Only one type of region of interest can be set.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "filling_level_cell_count": {
      "x": "uint32",
```

(continues on next page)

(continued from previous page)

```
      "y": "uint32"
    },
    "load_carrier_ids": [
      "string"
    ],
    "pose_frame": "string",
    "region_of_interest_2d_id": "string",
    "region_of_interest_id": "string",
    "robot_pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    }
  }
}
```

Response

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

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

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

The definition for the response with corresponding datatypes is:

```
{
  "name": "detect_filling_level",
  "response": {
    "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"
            }
          }
```

(continues on next page)

(continued from previous page)


```
        ],
        "filling_level_cell_count": {
          "x": "uint32",
          "y": "uint32"
        },
        "height_open_side": "float64",
        "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",
            "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_ledge": {
          "x": "float64",
          "y": "float64"
        },
        "rim_step_height": "float64",
        "rim_thickness": {
          "x": "float64",
```


(continues on next page)

(continued from previous page)

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

detect_filling_level_as_image

Triggers the same load carrier filling level detection as with `detect_filling_level`, but returns the filling level values as images instead of list of cells. This leads to shorter response times, especially when the number of cells is large. The mean, minimum and maximum values of `level_free_in_meters` and the `coverage` are returned as base64-encoded glib-compressed images. Each pixel corresponds to a cell in the filling level grid, ordered by the cell index.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_load_carrier/services/detect_filling_
↪level_as_image
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/detect_filling_level_as_image
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.2.4).

`load_carrier_ids`: IDs of the load carriers which should be detected. Currently only one ID can be specified.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.2.4).

Optional arguments:

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

`region_of_interest_id`: ID of the 3D region of interest where to search for the load carriers.

`region_of_interest_2d_id`: ID of the 2D region of interest where to search for the load carriers.

Note: Only one type of region of interest can be set.

Response

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

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

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

The definition for the response with corresponding datatypes is:


```
{
  "name": "detect_filling_level_as_image",
  "response": {
    "load_carriers": [
      {
        "filling_level_cell_count": {
          "x": "uint32",
          "y": "uint32"
        },
        "filling_level_cell_size": {
          "x": "float64",
          "y": "float64"
        },
        "filling_level_images": {
          "coverage": "string",
          "level_free_in_meters_max": "string",
          "level_free_in_meters_mean": "string",
          "level_free_in_meters_min": "string"
        },
        "height_open_side": "float64",
        "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",
            "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",
```


(continues on next page)

(continued from previous page)


```
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "pose_frame": "string",
        "rim_ledge": {
          "x": "float64",
          "y": "float64"
        },
        "rim_step_height": "float64",
        "rim_thickness": {
          "x": "float64",
          "y": "float64"
        },
        "type": "string"
      }
    ],
    "return_code": {
      "message": "string",
      "value": "int16"
    },
    "timestamp": {
      "nsec": "int32",
      "sec": "int32"
    }
  }
}
```


reset_defaults

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

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_load_carrier/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

`set_load_carrier` (deprecated)

Persistently stores a load carrier on the *rc_visard*.

API version 2

This service is not available in API version 2. Use *set_load_carrier* (Section 5.6.1.5) in `rc_load_carrier_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/set_load_carrier
```

The definitions of the request and response are the same as described in *set_load_carrier* (Section 5.6.1.5) in `rc_load_carrier_db`.

`get_load_carriers` (deprecated)

Returns the configured load carriers with the requested `load_carrier_ids`.

API version 2

This service is not available in API version 2. Use *get_load_carriers* (Section 5.6.1.5) in `rc_load_carrier_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/get_load_carriers
```

The definitions of the request and response are the same as described in *get_load_carriers* (Section 5.6.1.5) in `rc_load_carrier_db`.

`delete_load_carriers` (deprecated)

Deletes the configured load carriers with the requested `load_carrier_ids`.

API version 2

This service is not available in API version 2. Use *delete_load_carriers* (Section 5.6.1.5) in `rc_load_carrier_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/delete_load_carriers
```

The definitions of the request and response are the same as described in *delete_load_carriers* (Section 5.6.1.5) in `rc_load_carrier_db`.

`set_region_of_interest` (deprecated)

Persistently stores a 3D region of interest on the *rc_visard*.

API version 2

This service is not available in API version 2. Use *set_region_of_interest* (Section 5.6.2.4) in `rc_roi_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/set_region_of_interest
```

The definitions of the request and response are the same as described in *set_region_of_interest* (Section 5.6.2.4) in `rc_roi_db`.

`get_regions_of_interest` (deprecated)

Returns the configured 3D regions of interest with the requested `region_of_interest_ids`.

API version 2

This service is not available in API version 2. Use *get_regions_of_interest* (Section 5.6.2.4) in `rc_roi_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/get_regions_of_interest
```

The definitions of the request and response are the same as described in *get_regions_of_interest* (Section 5.6.2.4) in `rc_roi_db`.

`delete_regions_of_interest` (deprecated)

Deletes the configured 3D regions of interest with the requested `region_of_interest_ids`.

API version 2

This service is not available in API version 2. Use *delete_regions_of_interest* (Section 5.6.2.4) in `rc_roi_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/delete_regions_of_interest
```

The definitions of the request and response are the same as described in *delete_regions_of_interest* (Section 5.6.2.4) in `rc_roi_db`.

`set_region_of_interest_2d` (deprecated)

Persistently stores a 2D region of interest on the *rc_visard*.

API version 2

This service is not available in API version 2. Use *set_region_of_interest_2d* (Section 5.6.2.4) in `rc_roi_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/set_region_of_interest_2d
```

The definitions of the request and response are the same as described in *set_region_of_interest_2d* (Section 5.6.2.4) in `rc_roi_db`.

`get_regions_of_interest_2d` (deprecated)

Returns the configured 2D regions of interest with the requested `region_of_interest_2d_ids`.

API version 2

This service is not available in API version 2. Use *get_regions_of_interest_2d* (Section 5.6.2.4) in `rc_roi_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/get_region_of_interest_2d
```

The definitions of the request and response are the same as described in *get_regions_of_interest_2d* (Section 5.6.2.4) in `rc_roi_db`.

`delete_regions_of_interest_2d` (deprecated)

Deletes the configured 2D regions of interest with the requested `region_of_interest_2d_ids`.

API version 2

This service is not available in API version 2. Use *delete_regions_of_interest_2d* (Section 5.6.2.4) in `rc_roi_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_load_carrier/services/delete_regions_of_interest_2d
```

The definitions of the request and response are the same as described in *delete_regions_of_interest_2d* (Section 5.6.2.4) in `rc_roi_db`.

5.4.2.8 Return codes

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 5.19: Return codes of the LoadCarrier module's services

Code	Description
0	Success
-1	An invalid argument was provided
-4	Data acquisition took longer than allowed
-10	New element could not be added as the maximum storage capacity of load carriers has been exceeded
-11	Sensor not connected, not supported or not ready
-12	Resource busy, e.g. when `trigger_dump` is called too frequently
-302	More than one load carrier provided to the `detect_load_carriers` or `detect_filling_level` services, but only one is supported
3	The detection timeout during load carrier detection has been reached. Consider reducing the model tolerance.
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 has no points inside
300	A valid `robot_pose` was provided as argument but it is not required

5.4.3 TagDetect

5.4.3.1 Introduction

The TagDetect modules are optional on-board modules of the *rc_visard* and require separate *licenses* (Section 7.4) to be purchased. The licenses are included in every *rc_visard* purchased after 01.07.2020.

The TagDetect modules run on board the *rc_visard* and allow the detection of 2D matrix codes and tags. Currently, there are TagDetect modules for *QR codes* and *AprilTags*. The modules, 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 5.4.3.2).
2. Estimation of the pose of each tag (see *Pose estimation*, Section 5.4.3.3).
3. Re-identification of previously seen tags (see *Tag re-identification*, Section 5.4.3.4).

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

QR code



Fig. 5.11: Sample QR code

QR codes are two-dimensional matrix 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 module is not guaranteed. The same holds for QR codes which contain characters that are not part of regular ASCII.

AprilTag

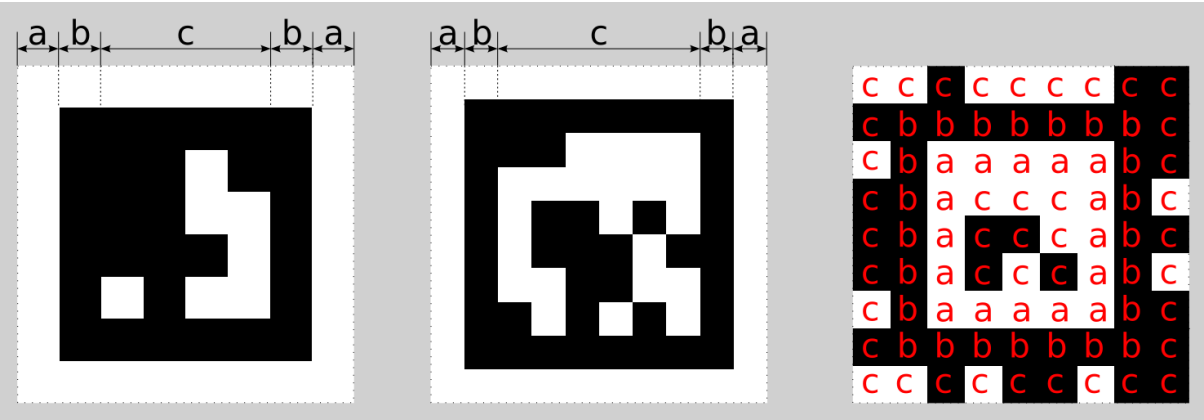


Fig. 5.12: A 16h5 tag (left), a 36h11 tag (center) and a 41h12 tag (right). AprilTags consist of a mandatory white (a) and black (b) border and a variable amount of data bits (c).

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. 5.12 shows how AprilTags are structured. They have a mandatory white and black border, each one module wide. The tag families 16h5, 25h9, 36h10 and 36h11 are surrounded by this border and carry a variable amount of data modules in the center. For tag family 41h12, the black and white border is shifted towards the inside and the data

modules are in the center and also at the border of the tags. Other than QR codes, AprilTags do not contain any user-defined information but are identified by a predefined *family* and *ID*. The tags in Fig. 5.12 for example are of family 16h5, 36h11 and 41h12 have id 0, 11 and 0, respectively. All supported families are shown in Table 5.20.

Table 5.20: AprilTag families

Family	Number of tag IDs	Recommended
16h5	30	-
25h9	35	o
36h10	2320	o
36h11	587	+
41h12	2115	+

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. 5.12), a 36h11 tag contains 36 (6x6) modules and a 41h12 tag contains 41 (3x3 inner + 4x8 outer) 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 5.20).

The advantage of fewer 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. The 41h12 family has its border shifted towards the inside, which gives it more data modules at a lower number of total modules compared to the 36h11 family.

For these reasons we recommend using the 41h12 and 36h11 families and highly discourage the use of the 16h5 family. The latter family 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 from the website https://github.com/AprilRobotics/apriltag-imgs. There, each family consists of multiple PNGs containing single tags. Each pixel in the PNGs corresponds to one AprilTag module. When printing the tags of the families 36h11, 36h10, 25h9 and 16h5 special care must be taken to also include the white border around the tag that is contained in the PNG (see (a) in Fig. 5.12). Moreover, all 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 border in AprilTags allow for more precise pose estimation.

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

5.4.3.2 Tag reading

The first step in the tag detection pipeline is reading the tags on the 2D image pair. This step takes most of the processing time and its precision is crucial for the precision of the resulting tag pose. To control the speed of this step, the `quality` parameter can be set by the user. It results in a downscaling of the image pair before reading the tags. `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 module* (Section 5.2).

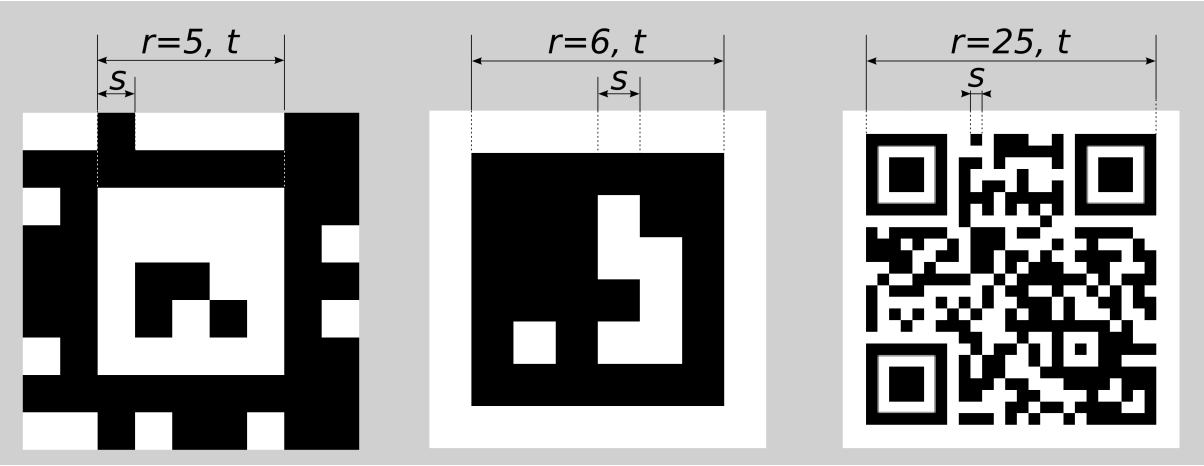


Fig. 5.13: Visualization of module size s, size of a tag in modules r, and size of a tag in meters t for AprilTags (left and center) and QR codes (right)

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 5.1.2) 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. 5.13 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 5.21: 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
41h12 (recommended)	5 modules	1.7 m

Table 5.22: 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

5.4.3.3 Pose estimation

For each detected tag, the pose of this tag in the camera coordinate frame is estimated.

A requirement for pose estimation is that a tag is fully visible in the left and right camera image.

The coordinate frame of the tag is aligned as shown below.

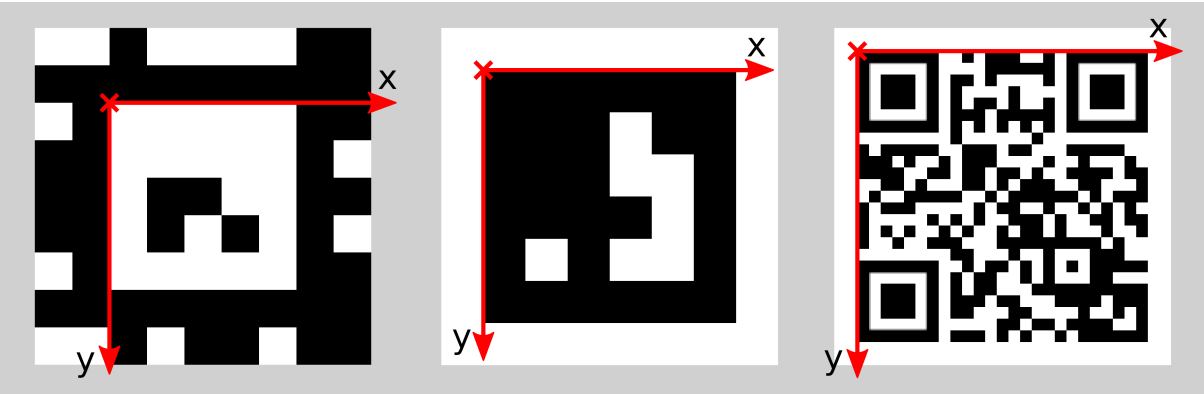


Fig. 5.14: Coordinate frames of AprilTags (left and center) and QR codes (right)

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 part inside the border defined by the transition from the black to the white border modules, hence ignoring the outermost white border for the tag families 16h5, 25h9, 36h10 and 36h11.

The user can also specify the approximate size ($\pm 10\%$) of tags. All tags not matching this size constraint 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.

Note: 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 modules try to detect such situations and filter out affected tags.

Below tables give approximate precisions of the estimated poses of AprilTags. 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`, that the camera’s viewing direction is parallel to the tag’s normal and that the images are well exposed and do not suffer from motion blur. 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 5.23: Approximate position precision for AprilTag detections with `High` quality in an ideal scenario for different base lines

Distance	*rc_visard* 65 - lateral	*rc_visard* 65 - z	*rc_visard* 160 - lateral	*rc_visard* 160 - z
0.5 m	0.05 mm	0.5 mm	0.05 mm	0.3 mm
1.0 m	0.15 mm	1.8 mm	0.15 mm	1.4 mm
2.0 m	1.5 mm	14.5 mm	0.5 mm	3.7 mm

Table 5.24: Approximate orientation precision for AprilTag detections with `High` quality in an ideal scenario for different tag sizes

Distance	60 x 60 mm	120 x 120 mm
0.5 m	0.2°	–
1.0 m	0.8°	0.3°
2.0 m	2.0°	0.8°
3.0 m	–	1.8°

5.4.3.4 Tag re-identification

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

For distinction of these tags, the TagDetect modules 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 5.3.1) 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. 5.15. 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.

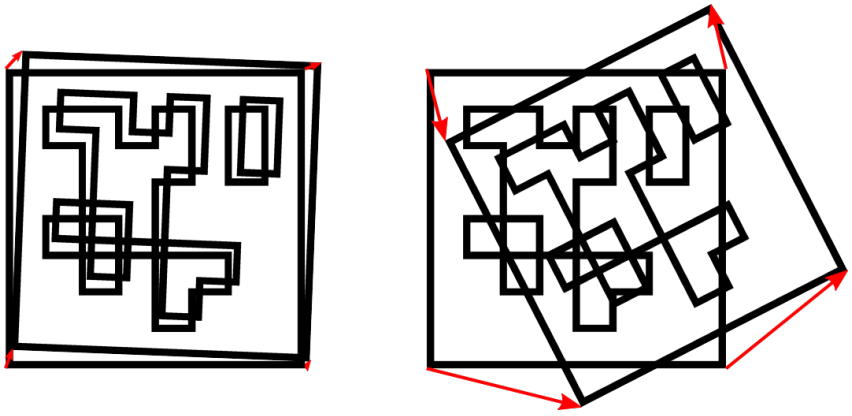


Fig. 5.15: Simplified visualization of tag re-identification. Euclidean distances between associated tag corners in 3D are compared (red arrows).

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`.

5.4.3.5 Hand-eye calibration

In case the camera has been calibrated to a robot, the TagDetect module can automatically provide poses in the robot coordinate frame. For the TagDetect node's *Services* (Section 5.4.3.8), 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 module are in the camera frame.
2. **External frame** (`external`). All poses provided by the module are in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) 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.

5.4.3.6 Parameters

There are two separate modules available 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 module names they share the same interface definition.

In addition to the *REST-API interface* (Section 6.2), the TagDetect modules provide pages on the Web GUI under *Modules* → *AprilTag* and *Modules* → *QR Code*, 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 module offers the following run-time parameters:

Table 5.25: The `rc_qr_code_detect` module'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: [Low, Medium, High]
`use_cached_images`	bool	false	true	false	Use most recently received image pair instead of waiting for a new pair

Via the REST-API, these parameters can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/<rc_qr_code_detect|rc_april_tag_detect>/
↪parameters/parameters?<parameter-name>=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/<rc_qr_code_detect|rc_april_tag_detect>/parameters?
↪<parameter-name>=<value>
```

5.4.3.7 Status values

The TagDetect modules reports the following status values:

Table 5.26: The `rc_qr_code_detect` and `rc_april_tag_detect` module's status values

Name	Description
`data_acquisition_time`	Time in seconds required to acquire image pair
`last_return_code_message`	Return code message of the last detection
`last_return_code_value`	Return code value of the last detection
`last_timestamp_processed`	The timestamp of the last processed image pair
`processing_time`	Processing time of the last detection in seconds
`state`	The current state of the node

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

Table 5.27: Possible states of the TagDetect modules

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

5.4.3.8 Services

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

The user can explore and call the `rc_qr_code_detect` and `rc_april_tag_detect` modules' services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1).

`detect`

Triggers a tag detection.

Details

Depending on the `use_cached_images` parameter, the module 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 module. 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 module before calling `detect`.

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 7.5).

A visualization of the latest detection is shown on the Web GUI tabs of the TagDetect modules. 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. Forward jumps do not have an effect on the TagDetect module. 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. This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/<rc_qr_code_detect|rc_april_tag_
↪detect>/services/detect
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/<rc_qr_code_detect|rc_april_tag_detect>/
↪services/detect
```

Request

Optional arguments:

`tags` is the list of tag IDs that the TagDetect module 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 module can only be triggered for AprilTags, and the QR code module 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 5.4.3.3), this information helps to resolve ambiguities in pose estimation that may arise in certain situations and can be used to filter out tags not fulfilling the given size constraint.

The `tags` list is OR-connected. All tags will be returned that match any of `id-size` pair elements in the `tags` list.

`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 5.4.3.5). The default is `camera`.

The definition for the request arguments with corresponding datatypes is:

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

Response

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

`tags` contains all detected tags.

`id` is the ID of the tag, similar to `id` in the request.

`instance_id` is the random unique identifier of the tag assigned by tag re-identification.

`pose` contains `position` and `orientation`. The orientation is in quaternion format.

`pose_frame` is set to the coordinate frame above pose refers to. It will either be "camera" or "external".

`size` will be set to the estimated tag size in meters.

`return_code` holds possible warnings or error codes.

The definition for the response with corresponding datatypes is:

```
{
  "name": "detect",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    },
    "tags": [
      {
        "id": "string",
        "instance_id": "string",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "pose_frame": "string",
```

(continues on next page)

(continued from previous page)


```
        "size": "float64",
        "timestamp": {
          "nsec": "int32",
          "sec": "int32"
        }
      }
    ],
    "timestamp": {
      "nsec": "int32",
      "sec": "int32"
    }
  }
}
```


start

Starts the module by transitioning from `IDLE` to `RUNNING`.

Details

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

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/<rc_qr_code_detect|rc_april_tag_detect>/
↪services/start
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/<rc_qr_code_detect|rc_april_tag_detect>/services/start
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:


```
{
  "name": "start",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```


stop

Stops the module by transitioning to `IDLE`.

Details

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

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/<rc_qr_code_detect|rc_april_tag_
↪detect>/services/stop
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/<rc_qr_code_detect|rc_april_tag_detect>/
↪services/stop
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "stop",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

restart

Restarts the module.

Details

If in `RUNNING` or `FATAL`, the module will be stopped and then started. If in `IDLE`, the module will be started.

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/<rc_qr_code_detect|rc_april_tag_
↪detect>/services/restart
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/<rc_qr_code_detect|rc_april_tag_detect>/
↪services/restart
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "restart",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

reset_defaults

Resets all parameters of the module to its default values, as listed in above table.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/<rc_qr_code_detect|rc_april_tag_detect>/
↪services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/<rc_qr_code_detect|rc_april_tag_detect>/services/reset_
↪defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.4.3.9 Return codes

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 return 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
-11	Sensor not connected, not supported or not ready
-12	Resource busy, e.g. when `trigger_dump` is called too frequently
-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 module was not in state `RUNNING`

5.4.4 ItemPick

5.4.4.1 Introduction

The ItemPick module provides an out-of-the-box perception solution for robotic pick-and-place applications. ItemPick targets the detection of flat surfaces on unknown objects for picking with a suction gripper.

In addition, the module offers:

- A dedicated page on the *rc_visard* *Web GUI* (Section 6.1) for easy setup, configuration, testing, and application tuning.
- The definition of regions of interest to select relevant volumes in the scene (see *RoiDB*, Section 5.6.2).
- A load carrier detection functionality for bin-picking applications (see *LoadCarrier*, Section 5.4.2), 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.
- Collision checking between the gripper and the load carrier and/or the point cloud.
- Support for static and robot-mounted cameras and optional integration with the *Hand-eye calibration* (Section 5.5.1) module, 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.
- Selection of a sorting strategy to sort the returned grasps.
- 3D visualization of the detection results with grasp points and gripper animations in the Web GUI.

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

The module is an optional on-board module of the *rc_visard* and requires a separate ItemPick *license* (Section 7.4) to be purchased.

5.4.4.2 Computation of grasps

The ItemPick module offers a service for computing grasps for suction grippers. The gripper is defined by its suction surface length and width.

The ItemPick module 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.

Optionally, further information can be given to the modules 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 (see *Load carrier compartments*, Section 5.6.1.3).
- The ID of the 3D region of interest where to search for the load carriers if a load carrier is set. Otherwise, the ID of the 3D 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. Details on collision checking are given below in *CollisionCheck* (Section 5.4.4.5).

A grasp provided by the ItemPick module represents the recommended pose of the TCP (Tool Center Point) of the suction gripper. The grasp `type` is always set to `SUCTION`.

For ItemPick with an `UNKNOWN` item model, 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. Since the x axis can have two possible directions, the one that better fits to the preferred TCP orientation (see *Setting the preferred orientation of the TCP*, Section 5.4.4.4) is selected. If the run-time parameter `allow_any_grasp_z_rotation` is set to true, the x axis will not be forced to be aligned with the maximum elongation of the graspable ellipse, but can have any rotation around the z axis. In this case, the returned grasp will have the orientation that best fits to the preferred TCP orientation and is collision free, if collision checking.

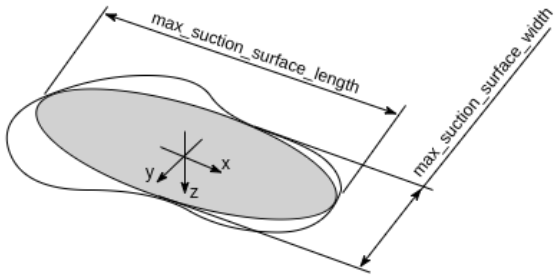


Fig. 5.16: Illustration of a suction grasp with coordinate system and ellipse representing the maximum suction surface

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. If the run-time parameter `allow_any_grasp_z_rotation` is set to true, `max_suction_surface_length` and `max_suction_surface_width` will be equal and correspond to the shortest axis of the largest graspable ellipse.

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.

The grasps are sorted according to the user-defined sorting strategy (see *Setting the sorting strategies*, Section 5.4.4.3).

5.4.4.3 Setting the sorting strategies

The grasps returned by the service calls are sorted according to a sorting strategy which can be chosen by the user. The following sorting strategies are available and can be set in the *Web GUI* (Section 6.1) or using the `set_sorting_strategies` service call:

- `gravity`: highest grasp points along the gravity direction are returned first,
- `surface_area`: grasp points with the largest surface area are returned first,
- `direction`: grasp points with the shortest distance along a defined direction `vector` in a given `pose_frame` are returned first.
- `distance_to_point`: grasp points with the shortest or farthest (if `farthest_first` is true) distance from a `point` in a given `pose_frame` are returned first.

If no sorting strategy is set or default sorting is chosen in the Web GUI, sorting is done based on a combination of `gravity` and `surface_area`.

5.4.4.4 Setting the preferred orientation of the TCP

The ItemPick module determines the reachability of grasp points based on the *preferred orientation* of the TCP. The preferred orientation can be set via the `set_preferred_orientation` service or on the ItemPick 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, the preferred orientation is used to select one grasp of several possible symmetries that is best reachable for the robot.

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. If no preferred orientation is set, the orientation of the left camera (see Coordinate frames) will be used as the preferred orientation of the TCP.

5.4.4.5 Interaction with other modules

Internally, the ItemPick module depends on, and interacts with other on-board modules as listed below.

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

Camera and depth data

The ItemPick module makes internally use of the following data:

- Rectified images from the *Camera module* (`rc_camera`, Section 5.1);
- Disparity, error, and confidence images from the *Stereo matching module* (`rc_stereomatching`, Section 5.2)

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

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* module (`rc_iocontrol`, Section 5.5.4), it is recommended to connect the projector to GPIO Out 1 and set the stereo-camera module's acquisition mode to `SingleFrameOut1` (see *Stereo matching parameters*, Section 5.2.1), 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 5.5.4.1).

In either case, the *Auto Exposure Mode* `exp_auto_mode` should be set to `AdaptiveOut1` to optimize the exposure of both images.

Hand-eye calibration

In case the camera has been calibrated to a robot, the ItemPick module can automatically provide poses in the robot coordinate frame. For the ItemPick node's *Services* (Section 5.4.4.8), 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 modules 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 modules are in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) 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.

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 sorting direction, sorting point or preferred orientation:

- If `pose_frame` is set to `external`, providing the robot pose is obligatory.
- If the sorting direction is defined in `external`, providing the robot pose is obligatory.
- If the distance-to-point sorting strategy is defined in `external`, providing the robot pose is obligatory.
- If the preferred orientation is defined in `external`, providing the robot pose is obligatory.
- In all other cases, providing the robot pose is optional.

LoadCarrier

The ItemPick module uses the load carrier detection functionality provided by the *LoadCarrier* module (`rc_load_carrier`, Section 5.4.2), with the run-time parameters specified for this module. However, only one load carrier will be returned and used in case multiple matching load carriers could be found in the scene. In case multiple load carriers of the same type are visible, a 3D region of interest should be set to ensure that always the same load carrier is used for the ItemPick module.

CollisionCheck

Collision checking can be easily enabled for grasp computation of the ItemPick module by passing the ID of the used gripper and optionally a pre-grasp offset to the `compute_grasps` or `compute_grasps_extended` service call. The gripper has to be defined in the GripperDB module (see *Setting a gripper*, Section 5.6.4.2) and details about collision checking are given in *Collision checking within other modules* (Section 5.5.2.2).

If collision checking is enabled, only grasps which are collision free will be returned. However, the visualization images on the ItemPick page 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 5.5.2.3).

5.4.4.6 Parameters

ItemPick is represented by the `rc_itempick` node in the REST-API and are reached in the *Web GUI* (Section 6.1) under *Modules* → *ItemPick*. The user can explore and configure the `rc_itempick` module's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* :(Section 6.2).

The user can explore and configure the `rc_itempick` module's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.28: The `rc_itempick` module's run-time parameters

Name	Type	Min	Max	Default	Description
`allow_any_grasp_pose`	bool	false	true	false	Whether the grasps are allowed to be placed anywhere on the objects where planar surfaces are detected
`allow_any_grasp_z_rotation`	bool	false	true	false	Whether the grasps are allowed to have arbitrary rotation instead being aligned with the major axis of the graspable ellipse
`check_collisions_with_point_cloud`	bool	false	true	false	Whether to check for collisions between gripper and the point cloud
`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.
`cluster_max_dimension`	float64	0.05	2.0	0.3	Maximum allowed diameter for a cluster in meters. Clusters with a diameter larger than this value are not used for grasp computation.
`clustering_discontinuity_factor`	float64	0.1	5.0	1.0	Factor used to discriminate depth discontinuities within a patch. The smaller this value, the more clusters will be split apart.
`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_patch_size`	int32	3	10	4	Size in pixels of the square patches the depth map is subdivided into during the first clustering step
`discard_cut_off_items`	bool	false	true	false	Whether items that are cut off by the image border, the stereo ROI, the missing stereo overlap or the 3D region of interest should be discarded
`grasp_filter_orientation_threshold`	float64	0.0	180.0	45.0	Maximum allowed orientation change between grasp and preferred orientation in degrees
`max_grasps`	int32	1	100	5	Maximum number of provided grasps

Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's ItemPick page. 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/parameters/parameters?max_
↪grasps=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?max_grasps=<value>
```

`cluster_max_dimension` (*Cluster Maximum Dimension*)

is the maximum allowed diameter for a cluster in meters. Clusters with a diameter larger than this value are not used for grasp computation.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/parameters/parameters?cluster_
↪max_dimension=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?cluster_max_dimension=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/parameters/parameters?cluster_
↪max_curvature=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?cluster_max_curvature=<value>
```

`clustering_patch_size` (*Patch Size*)

is the size of the square patches the depth map is subdivided into during the first clustering step in pixels.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/parameters/parameters?
↪clustering_patch_size=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?clustering_patch_size=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/parameters/parameters?
↪clustering_discontinuity_factor=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?clustering_discontinuity_factor=
↪<value>
```

`clustering_max_surface_rmse` (*Maximum Surface RMSE*)

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

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/parameters/parameters?
↪clustering_max_surface_rmse=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?clustering_max_surface_rmse=
↪<value>
```

`grasp_filter_orientation_threshold` (*Grasp Orientation Threshold*)

is the maximum deviation of the TCP's z axis at the grasp point from the z axis of the TCP's preferred orientation in degrees. Only grasp points which are within this threshold are returned. When set to zero, any deviations are valid.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/<0,1,2,3>/nodes/rc_itempick/parameters?grasp_filter_
↪orientation_threshold=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?grasp_filter_orientation_
↪threshold=<value>
```

`allow_any_grasp_z_rotation` (*Allow Any Grasp Z Rotation*)

If set to true, the returned grasps are no longer forced to have their x axes aligned with the maximum elongation of the graspable ellipse, but can have any rotation around the z axis. The returned `max_suction_surface_length` and `max_suction_surface_width` will be equal and correspond to the shortest diameter of the largest graspable ellipse. This parameter enables the robot to get more options for grasping objects, especially in scenes where collisions can occur. However, in case of `UNKNOWN` item models, since the grasp is no longer

aligned with the graspable ellipse, the correct orientation for placing the object must be determined by other means.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/<0,1,2,3>/nodes/rc_itempick/parameters?allow_any_
↪grasp_z_rotation=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?allow_any_grasp_z_rotation=
↪<value>
```

`check_collisions_with_point_cloud` (*Check Collisions with Point Cloud*)

This parameter is only used when collision checking is enabled by passing a gripper to the `compute_grasps` or `compute_grasps_extended` service call. If `check_collisions_with_point_cloud` is set to true, all grasp points will be checked for collisions between the gripper and a watertight version of the point cloud, and only grasp points at which the gripper would not collide with this point cloud will be returned.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/<0,1,2,3>/nodes/rc_itempick/parameters?check_
↪collisions_with_point_cloud=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/parameters?check_collisions_with_point_
↪cloud=<value>
```

5.4.4.7 Status values

The `rc_itempick` node reports the following status values:

Table 5.29: The `rc_itempick` node's status values

Name	Description
`data_acquisition_time`	Time in seconds required by the last active service to acquire images
`grasp_computation_time`	Processing time of the last grasp computation in seconds
`last_return_code_message`	Return code message of the last detection
`last_return_code_value`	Return code value of the last detection
`last_timestamp_processed`	The timestamp of the last processed dataset
`load_carrier_detection_time`	Processing time of the last load carrier detection in seconds
`processing_time`	Processing time of the last detection (including load carrier detection) in seconds
`state`	The current state of the rc_itempick node

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

Table 5.30: Possible states of the ItemPick module

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

5.4.4.8 Services

The user can explore and call the `rc_itempick` node's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1).

The ItemPick module offers the following services.

`compute_grasps`

Triggers the computation of grasping poses for a suction device as described in *Computation of grasps* (Section 5.4.4.2).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/compute_grasps
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/compute_grasps
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.4.5).

`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 5.4.4.5).

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 (see *Load carrier compartments*, Section 5.6.1.3).

`region_of_interest_id`: if `load_carrier_id` is set, ID of the 3D region of interest where to search for the load carriers. Otherwise, ID of the 3D region of interest where to compute grasps.

`item_models`: list of items to be detected. In case of ItemPick, currently only a single item model of type `UNKNOWN` with minimum and maximum dimensions is supported, with the minimum dimensions strictly smaller than the maximum dimensions.

`collision_detection`: see *Collision checking within other modules* (Section 5.5.2.2).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "collision_detection": {
      "gripper_id": "string",
      "pre_grasp_offset": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "dimensioning": {
      "plane_ids": [
        "string"
      ]
    },
    "item_models": [
      {
        "type": "string",
        "unknown": {
          "max_dimensions": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "min_dimensions": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        }
      }
    ],
    "load_carrier_compartment": {
      "box": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      }
    },
    "load_carrier_id": "string",
    "pose_frame": "string",
    "region_of_interest_id": "string",
    "robot_pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
```


(continues on next page)

(continued from previous page)

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

Response

`load_carriers`: list of detected load carriers.

`grasps`: sorted list of suction grasps.

`items`: sorted list of items corresponding to the returned grasps. In case of ItemPick, this list is always empty.

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

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

The definition for the response with corresponding datatypes is:

```
{
  "name": "compute_grasps",
  "response": {
    "grasps": [
      {
        "item_uuid": "string",
        "max_suction_surface_length": "float64",
        "max_suction_surface_width": "float64",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "pose_frame": "string",
        "quality": "float64",
        "tcp_id": "string",
        "timestamp": {
          "nsec": "int32",
          "sec": "int32"
        },
        "type": "string",
        "uuid": "string"
      }
    ],
    "items": [
      {
        "bounding_box": {
          "x": "float64",
          "y": "float64",
```

(continues on next page)

(continued from previous page)


```
          "z": "float64"
        },
        "grasp_uuids": [
          "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"
        },
        "type": "string",
        "uuid": "string"
      }
    ],
    "load_carriers": [
      {
        "height_open_side": "float64",
        "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_ledge": {
          "x": "float64",
          "y": "float64"
        },
        "rim_step_height": "float64",
```


(continues on next page)

(continued from previous page)

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

compute_grasps_extended

Triggers the computation of grasping poses for a suction device in the same way as `compute_grasps`, but returns the item information for each grasp directly instead of as a separate list. This allows for easier parsing when item information is required for the grasps.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/compute_grasps_extended
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/compute_grasps_extended
```

Request

See `compute_grasps` service.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "collision_detection": {
      "gripper_id": "string",
      "pre_grasp_offset": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "dimensioning": {
      "plane_ids": [
        "string"
      ]
    },
    "item_models": [
      {
        "type": "string",
        "unknown": {
```

(continues on next page)

(continued from previous page)


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


Response

`load_carriers`: list of detected load carriers.

`grasps`: sorted list of suction grasps. Each grasp also contains the item information, if available.

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

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

The definition for the response with corresponding datatypes is:


```
{
  "name": "compute_grasps_extended",
  "response": {
    "grasps": [
      {
        "item": {
          "bounding_box": {
            "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",
          "type": "string",
          "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",
        "tcp_id": "string",
        "timestamp": {
          "nsec": "int32",
          "sec": "int32"
        },
        "type": "string",
        "uuid": "string"
      }
    ],
    "load_carriers": [
      {
        "height_open_side": "float64",
        "id": "string",
        "inner_dimensions": {
```


(continues on next page)

(continued from previous page)


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


set_preferred_orientation

Persistently stores the preferred orientation of the TCP to compute the reachability of the grasps, which is used for filtering and the grasps returned by the `compute_grasps` and `compute_grasps_extended` service (see *Setting the preferred orientation of the TCP*, Section 5.4.4.4).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/set_preferred_
↪orientation
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/set_preferred_orientation
```

Request

The definition for the request arguments with corresponding datatypes is:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_preferred_orientation",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

`get_preferred_orientation`

Returns the preferred orientation of the TCP to compute the reachability of the grasps, which is used for filtering the grasps returned by the `compute_grasps` and `compute_grasps_extended` service (see *Setting the preferred orientation of the TCP*, Section 5.4.4.4).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/get_preferred_
↪orientation
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/get_preferred_orientation
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:


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


set_sorting_strategies

Persistently stores the sorting strategy for sorting the grasps (see *Setting the sorting strategies*, Section 5.4.4.3).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/set_sorting_strategies
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/set_sorting_strategies
```

Request

Only one strategy may have a `weight` greater than 0. If all `weight` values are set to 0, the module will use the default sorting strategy.

If the weight for `direction` is set, the `vector` must contain the direction vector and `pose_frame` must be either `camera` or `external`.

If the weight for `distance_to_point` is set, `point` must contain the sorting point and `pose_frame` must be either `camera` or `external`.

The definition for the request arguments with corresponding datatypes is:


```
{
  "args": {
    "direction": {
      "pose_frame": "string",
      "vector": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "weight": "float64"
    },
    "distance_to_point": {
      "farthest_first": "bool",
      "point": {
```


(continues on next page)

(continued from previous page)

```
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose_frame": "string",
      "weight": "float64"
    },
    "gravity": {
      "weight": "float64"
    },
    "surface_area": {
      "weight": "float64"
    }
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_sorting_strategies",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

`get_sorting_strategies`

Returns the sorting strategy for sorting the grasps (see *Setting the sorting strategies*, Section 5.4.4.3).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/get_sorting_strategies
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/get_sorting_strategies
```

Request

This service has no arguments.

Response

All `weight` values are 0 when the module uses the default sorting strategy.

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_sorting_strategies",
  "response": {
    "direction": {
      "pose_frame": "string",
```

(continues on next page)

(continued from previous page)

```
      "vector": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "weight": "float64"
    },
    "distance_to_point": {
      "farthest_first": "bool",
      "point": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose_frame": "string",
      "weight": "float64"
    },
    "gravity": {
      "weight": "float64"
    },
    "return_code": {
      "message": "string",
      "value": "int16"
    },
    "surface_area": {
      "weight": "float64"
    }
  }
}
```

start

Starts the module. If the command is accepted, the module moves to state `RUNNING`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/start
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/start
```

Request

This service has no arguments.

Response

The `current_state` value in the service response may differ from `RUNNING` if the state transition is still in process when the service returns.

The definition for the response with corresponding datatypes is:

```
{
  "name": "start",
  "response": {
    "accepted": "bool",
```

(continues on next page)

(continued from previous page)

```
    "current_state": "string"
  }
}
```

stop

Stops the module. If the command is accepted, the module moves to state `IDLE`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/stop
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/stop
```

Request

This service has no arguments.

Response

The `current_state` value in the service response may differ from `IDLE` if the state transition is still in process when the service returns.

The definition for the response with corresponding datatypes is:

```
{
  "name": "stop",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

reset_defaults

Resets all parameters of the module to its default values, as listed in above table. Also resets the sorting strategies and the preferred orientation.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_itempick/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_itempick/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.4.4.9 Return codes

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 5.31: Return codes of the ItemPick services

Code	Description
0	Success
-1	An invalid argument was provided
-3	An internal timeout occurred, e.g. during box detection if the given dimension range is too large
-4	Data acquisition took longer than allowed
-8	The template has been deleted during detection
-10	New element could not be added as the maximum storage capacity of load carriers, regions of interest or template has been exceeded
-11	Sensor not connected, not supported or not ready
-12	Resource busy, e.g. when `trigger_dump` is called too frequently
-200	Fatal internal error
-301	More than one item model provided to the `compute_grasps` or `compute_grasps_extended` service
10	The maximum storage capacity of load carriers, regions of interest or templates 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 were found in the scene
102	The detected load carrier is empty
103	All computed grasps are in collision
112	Rejected detections of one or more clusters, because min_cluster_coverage was not reached
115	Items without grasps were detected
300	A valid `robot_pose` was provided as argument but it is not required
999	Additional hints for application development

5.4.5 BoxPick

5.4.5.1 Introduction

The BoxPick module provides an out-of-the-box perception solution for robotic pick-and-place applications. It detects rectangular surfaces and determines their position, orientation and size for grasping.

With the +Match extension, BoxPick can be used to detect textured rectangles with consistent orientations, such as printed product packaging, labels, brochures or books.

In addition, the module offers:

- A dedicated page on the *rc_visard* *Web GUI* (Section 6.1) for easy setup, configuration, testing, and application tuning.
- The definition of regions of interest to select relevant volumes in the scene (see *RoiDB*, Section 5.6.2).
- A load carrier detection functionality for bin-picking applications (see *LoadCarrier*, Section 5.4.2), 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.
- Collision checking between the gripper and the load carrier and/or the point cloud.
- Support for static and robot-mounted cameras and optional integration with the *Hand-eye calibration* (Section 5.5.1) module, 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.
- Selection of a sorting strategy to sort the returned grasps.
- 3D visualization of the detection results with grasp points and gripper animations in the Web GUI.

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

The module is an optional on-board module of the *rc_visard* and requires a separate BoxPick *license* (Section 7.4) to be purchased. The +Match extension requires an extra license.

5.4.5.2 Detection of items

There are two different types of models for the rectangles to be detected by the BoxPick module.

Per default, BoxPick only supports `item_models` of `type` `RECTANGLE`. With the +Match extension, also item models of `type` `TEXTURED_BOX` can be detected. The detection of the different item model types is described below.

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

- 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 ID of a reference plane for measuring the dimensions of the detected items (see *Dimensioning*, Section 5.4.5.2).
- 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 returned `pose` of a detected `item` is the pose of the center of the detected rectangle in the desired reference frame (`pose_frame`), with its z axis pointing towards the camera and the x axis aligned with the long side of the item. This pose has a 180° rotation ambiguity around the z axis, which can be resolved by using the +Match extension with a `TEXTURED_BOX` item model. Each detected item includes a `uuid` (Universally Unique Identifier) and the `timestamp` of the oldest image that was used to detect it.

The items are sorted according to the chosen sorting strategy (see *Setting the sorting strategies*, Section 5.4.5.4).

Detection of items of type RECTANGLE

BoxPick supports 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.

The detection of the rectangles runs in several steps. First, the point cloud is segmented into preferably plane clusters. Then, straight line segments are detected in the 2D images and projected onto the corresponding clusters. The clusters and the detected lines are visualized in the "Intermediate Result" visualization on the Web GUI's *BoxPick* page. Finally, for each cluster, the set of rectangles best fitting to the detected line segments is extracted.

Detection of items of type TEXTURED_BOX (BoxPick+Match)

With the +Match extension, BoxPick additionally supports `item_models` of `type` `TEXTURED_BOX`. When this item model type is used, only one item model can be given for each request.

The `TEXTURED_BOX` item model type should be used to detect multiple rectangles that have the same texture, i.e. the same look or print, such as printed product packaging, labels, brochures or books. It is required that for all objects the texture is at the same position with respect to the object geometry. Furthermore, the texture should not be repetitive.

A `TEXTURED_BOX` item is defined by the item's exact `dimensions` `x`, `y` and `z` (only `z` is allowed to be 0) with a tolerance `dimensions_tolerance_m` that indicates, how much the detected dimensions are allowed to deviate from the given dimensions. By default, a tolerance of 0.01 m is assumed. Furthermore, a `template_id` must be given, which will be used to refer to the specified dimensions and the textures of the detected rectangles. Additionally, the maximum possible deformation of the items `max_deformation_m` can be given in meters (default 0.004 m), to account for rigid or more flexible objects.

If a `template_id` is used for the first time, BoxPick will run the detection of rectangles as for the item model type `RECTANGLE`, and use the given dimensions and tolerance to specify the dimensions range. If the z dimension is given in addition to x and y, rectangles with all possible combinations of the three dimensions will be detected. From the detected rectangles, so-called *views* are created, which contain the shape and the image intensity values of the rectangles, and are stored in a newly created template with the given `template_id`. The views are created iteratively: Starting from the detected rectangle with the highest score, a view is created and then used to detect more rectangles with the same texture. Then, all remaining clusters are used to detect further rectangles by the given dimensions range and again a view is created from the best rectangle and used for further detections. Each template can store up to 10 different views, for example corresponding to different types of the same product packaging. Each view will be assigned a unique ID (`view_uuid`) and all rectangle items with a matching texture will be assigned the same `view_uuid`. That also means that all items with the same `view_uuid` will have consistent orientations, because the orientation of each item is aligned with its texture. The views can be displayed, deleted and the orientation of each view can be set via the *Web GUI* (Section 6.1) by clicking on the template or its edit symbol in the template list. Each detected item contains a field `view_pose_set` indicating whether the orientation of the item's view was explicitly set or is still unset at its original random state, which has a 180° ambiguity. Additionally, a user-defined name can be set for each view, that is returned along with the `view_uuid` for all items and allows an easier identification of a specific view. The `type` of a returned item with a `view_uuid` will be `TEXTURED_RECTANGLE`.

If the template with the given `template_id` already exists, the existing views will be used to detect rectangles based on their texture. If additional rectangles are found with matching dimensions, but different texture, new views will be generated and added to the template. When the maximum number of views is reached, views that are matched only rarely will be deleted so that newly generated views can be added to the template and the template is kept up-to-date. To prevent a template from being updated, automatic view updating can be disabled and enabled for each template in the Web GUI by clicking on the template or the edit symbol in the template list. The dimension tolerance and the maximum deformation can also be changed there for each template. The maximum deformation determines the tolerance for the texture matching, representing possible shifts within the texture, e.g. caused by deformations of the

object surface. For rigid objects the `max_deformation_m` should be set to a low value in meters to ensure accurate matching.

The template's `dimensions` can only be specified when creating a new template. Once the template is generated, the dimensions cannot be changed and do not need to be given in the detect request. If the dimensions are still given in the request, they must match the existing dimensions in the template. However, the `dimensions_tolerance_m` and `max_deformation_m` can be set differently in every detect request and their values will also be updated in the stored template.

Dimensioning

BoxPick only detects the top surface of an item as a rectangle. To measure the full dimensions of the detected items, a reference plane on which the items are placed, e.g. a conveyor belt or a table, can be given with the `dimensioning` argument of the `detect_items`, `compute_grasps` and `compute_grasps_extended` services. `dimensioning.plane_ids` must contain exactly one ID of a plane that is stored in the *PrimitiveDB* (Section 5.6.3).

For each detected item, a box is computed from the four corners of the detected rectangle, which are measured down to the reference plane. The box is the bounding box of the rectangle corners that is aligned with the reference plane.

The returned item is of `type BOX` (or `TEXTURED_BOX` with the +Match extension) and contains its dimensions in the `box` field instead of the `rectangle` field. The `pose` of the item is the center of the box. The returned grasps are not affected by dimensioning.

Dimensioning is not possible for `TEXTURED_BOX` item models for which all three dimensions are given, because the dimensions of these items are already known.

Items that lie behind the reference plane are discarded, which is reported by the `return_code` value 115.

5.4.5.3 Computation of grasps

The BoxPick module offers a service for computing grasps for suction grippers. The gripper is defined by its suction surface length and width.

The grasps are computed on the detected rectangular `items` (see *Detection of items*, Section 5.4.5.2).

Optionally, further information can be given to the module 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 (see *Load carrier compartments*, Section 5.6.1.3).
- The ID of the 3D region of interest where to search for the load carriers if a load carrier is set. Otherwise, the ID of the 3D 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. Details on collision checking are given below in *CollisionCheck* (Section 5.4.5.6).
- The ID of a reference plane for measuring the dimensions of the detected items (see *Dimensioning*, Section 5.4.5.2).

A grasp provided by the BoxPick module 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. Since the x axis can have two possible directions, the one that better fits to the preferred TCP orientation (see *Setting the preferred orientation of the TCP*, Section 5.4.5.5) is selected. If the run-time parameter `allow_any_grasp_z_rotation` is set to true, the x axis will not be forced to be aligned with the maximum elongation of the graspable ellipse, but can have any rotation around the z axis. In this case, the

returned grasp will have the orientation that best fits to the preferred TCP orientation and is collision free, if collision checking is enabled.

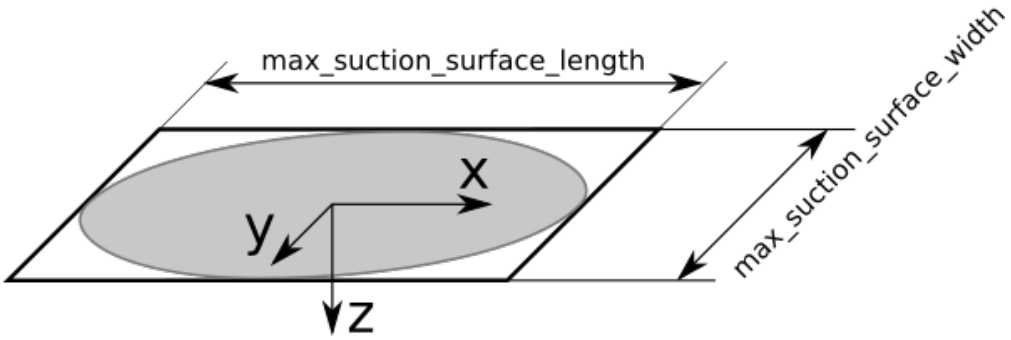


Fig. 5.17: Illustration of a suction grasp with coordinate system and ellipse representing the maximum suction surface

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. If the run-time parameter `allow_any_grasp_z_rotation` is set to true, `max_suction_surface_length` and `max_suction_surface_width` will be equal and correspond to the shortest axis of the largest graspable ellipse.

In the BoxPick module, the grasp position corresponds to the center of the detected rectangle. When BoxPick is called with item models of type `RECTANGLE`, the dimensions of the maximum suction surface available matches the estimated rectangle dimensions. In this case, detected rectangles with missing data or occlusions by other objects for more than 15% of their surface do not get an associated grasp.

When BoxPick is called with item models of type `TEXTURED_BOX`, grasps can also be computed on partly occluded boxes. The maximum suction surface available matches the free surface that is not occluded by other clusters.

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.

Grasp sorting is performed based on the selected sorting strategy (see *Setting the sorting strategies*, Section 5.4.5.4).

5.4.5.4 Setting the sorting strategies

The items and grasps returned by the service calls are sorted according to a sorting strategy which can be chosen by the user. The following sorting strategies are available and can be set in the *Web GUI* (Section 6.1) or using the `set_sorting_strategies` service call:

- `gravity`: highest items or grasp points along the gravity direction are returned first,
- `surface_area`: items or grasp points with the largest surface area are returned first,
- `direction`: items or grasp points with the shortest distance along a defined direction `vector` in a given `pose_frame` are returned first.
- `distance_to_point`: items or grasp points with the shortest or farthest (if `farthest_first` is true) distance from a `point` in a given `pose_frame` are returned first.

If no sorting strategy is set or default sorting is chosen in the Web GUI, sorting is done based on a combination of `gravity` and `surface_area`.

5.4.5.5 Setting the preferred orientation of the TCP

The BoxPick module determines the reachability of grasp points based on the *preferred orientation* of the TCP. The preferred orientation can be set via the `set_preferred_orientation` service or on the BoxPick 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, the preferred orientation is used to select one grasp of several possible symmetries that is best reachable for the robot.

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. If no preferred orientation is set, the orientation of the left camera (see Coordinate frames) will be used as the preferred orientation of the TCP.used.

5.4.5.6 Interaction with other modules

Internally, the BoxPick module depends on, and interacts with other on-board modules as listed below.

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

Camera and depth data

The BoxPick module makes internally use of the following data:

- Rectified images from the *Camera module* (`rc_camera`, Section 5.1)
- Disparity, error, and confidence images from the *Stereo matching module* (`rc_stereomatching`, Section 5.2)

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

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* module (`rc_iocontrol`, Section 5.5.4), it is recommended to connect the projector to GPIO Out 1 and set the stereo-camera module's acquisition mode to `SingleFrameOut1` (see *Stereo matching parameters*, Section 5.2.1), 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 5.5.4.1).

In either case, the *Auto Exposure Mode* `exp_auto_mode` should be set to `AdaptiveOut1` to optimize the exposure of both images.

Hand-eye calibration

In case the camera has been calibrated to a robot, the BoxPick module can automatically provide poses in the robot coordinate frame. For the BoxPick node's *Services* (Section 5.4.5.9), 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 modules 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 modules are in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) 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.

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 sorting direction, sorting point or preferred orientation:

- If `pose_frame` is set to `external`, providing the robot pose is obligatory.
- If the sorting direction is defined in `external`, providing the robot pose is obligatory.
- If the distance-to-point sorting strategy is defined in `external`, providing the robot pose is obligatory.
- If the preferred orientation is defined in `external`, providing the robot pose is obligatory.
- In all other cases, providing the robot pose is optional.

LoadCarrier

The BoxPick module uses the load carrier detection functionality provided by the *LoadCarrier* module (`rc_load_carrier`, Section 5.4.2), with the run-time parameters specified for this module. However, only one load carrier will be returned and used in case multiple matching load carriers could be found in the scene. In case multiple load carriers of the same type are visible, a 3D region of interest should be set to ensure that always the same load carrier is used for the BoxPick module.

The load carrier is used to filter false detections when BoxPick is triggered with an item model of type `TEXTURED_BOX` and all three dimensions x, y, z are given. In this case, 3D boxes are created internally by adding the missing dimensions to the detected rectangles and only detections corresponding to boxes which are fully inside the detected load carrier are returned.

CollisionCheck

Collision checking can be easily enabled for grasp computation of the BoxPick module by passing the ID of the used gripper and optionally a pre-grasp offset to the `compute_grasps` or `compute_grasps_extended` service call. The gripper has to be defined in the GripperDB module (see *Setting a gripper*, Section 5.6.4.2) and details about collision checking are given in *Collision checking within other modules* (Section 5.5.2.2).

If collision checking is enabled, only grasps which are collision free will be returned. However, the visualization images on the *BoxPick* page 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 5.5.2.3).

5.4.5.7 Parameters

The BoxPick module is called `rc_boxpick` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Modules* → *BoxPick*. The user can explore and configure the `rc_boxpick` module's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.32: The `rc_boxpick` module's run-time parameters

Name	Type	Min	Max	Default	Description
`allow_any_grasp_z_rotation`	bool	false	true	false	Whether the grasps are allowed to have arbitrary rotation instead being aligned with the major axis of the graspable ellipse
`allow_untextured_detections`	bool	false	true	false	Whether to return also untextured detections in case a textured box was given
`check_collisions_with_point_cloud`	bool	false	true	false	Whether to check for collisions between gripper and the point cloud
`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_discontinuity_factor`	float64	0.1	5.0	1.0	Factor used to discriminate depth discontinuities within a patch. The smaller this value, the more clusters will be split apart.
`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
`discard_cut_off_items`	bool	false	true	false	Whether items that are cut off by the image border, the stereo ROI, the missing stereo overlap or the 3D region of interest should be discarded
`grasp_filter_orientation_threshold`	float64	0.0	180.0	45.0	Maximum allowed orientation change between grasp and preferred orientation in degrees
`line_sensitivity`	float64	0.1	1.0	0.1	Sensitivity of the line detector
`manual_line_sensitivity`	bool	false	true	false	Indicates whether the user-defined line sensitivity should be used or the automatic one
`max_grasps`	int32	1	100	5	Maximum number of provided grasps
`min_cluster_coverage`	float64	0.0	0.99	0.0	Gives the minimal ratio of points per cluster that must be covered with detected items.
`mode`	string	-	-	Unconstrained	Mode of the rectangle detection: [Unconstrained, PackedGridLayout, PackedLayers]
`prefer_splits`	bool	false	true	false	Indicates whether rectangles are split into smaller ones when possible

Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's *BoxPick* page. 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?max_grasps=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?max_grasps=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?cluster_
↪max_curvature=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?cluster_max_curvature=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?clustering_
↪discontinuity_factor=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?clustering_discontinuity_factor=
↪<value>
```

`clustering_max_surface_rmse` (*Maximum Surface RMSE*)

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

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?clustering_
↪max_surface_rmse=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?clustering_max_surface_rmse=
↪<value>
```

`mode` (*Mode*)

determines the mode of the rectangle detection. Possible values are `Unconstrained`, `PackedGridLayout` and `PackedLayers`. In `PackedGridLayout` mode, rectangles of a cluster are detected in a dense grid pattern. In `PackedLayers` mode, boxes are assumed to form layers and box detection will start searching for items at the cluster corners. Use this mode in de-palletizing applications. In `Unconstrained` mode (default), rectangles are detected without posing any constraints on their relative locations or their positions in the segmented cluster. Fig. 5.18 illustrates the modes for different scenarios.

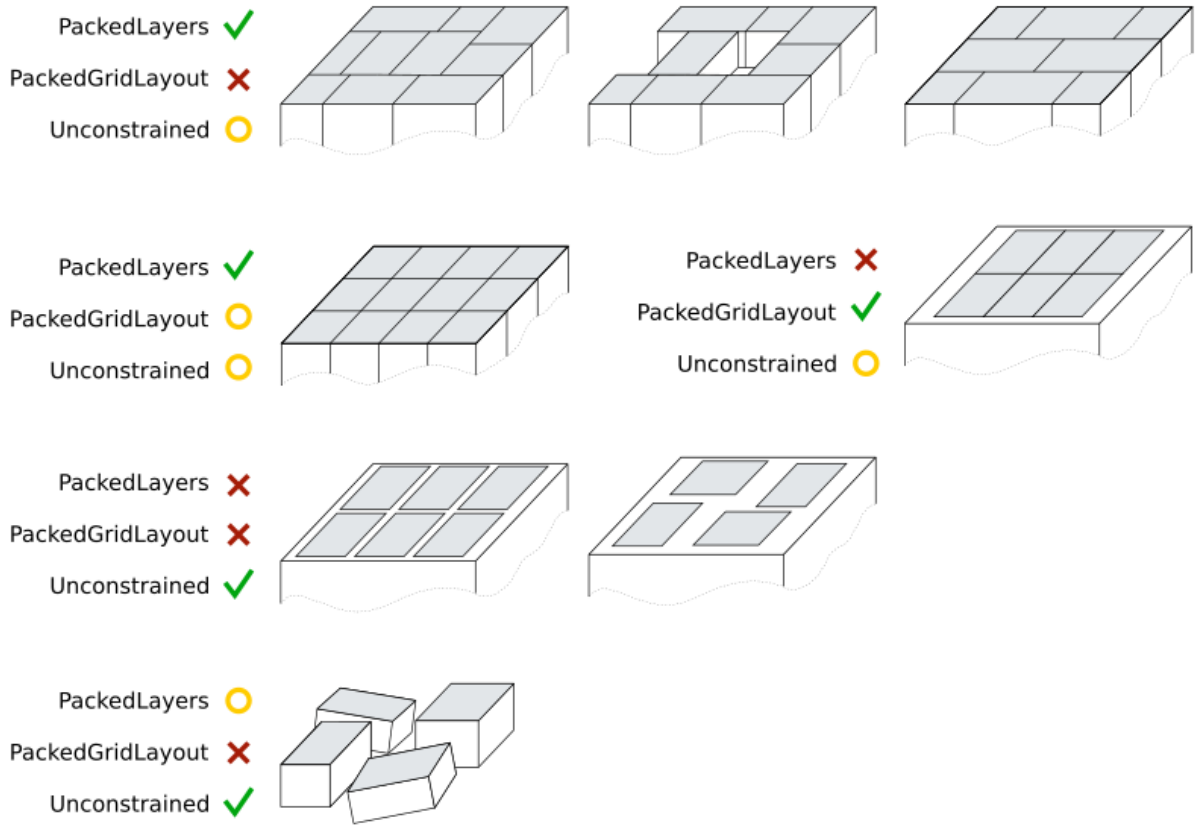


Fig. 5.18: Illustration of appropriate BoxPick modes for different scenes. Modes marked with yellow are applicable but not recommended for the corresponding scene. The gray areas indicate the rectangles to be detected.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?mode=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?mode=<value>
```

`manual_line_sensitivity` (*Manual Line Sensitivity*)

determines whether the user-defined line sensitivity should be used to extract the lines for rectangle detection. If this parameter is set to true, the user-defined `line_sensitivity` value will be used. If this parameter is set to false, automatic line sensitivity will be used. This parameter should be set to true when automatic line sensitivity does not give enough lines at the box boundaries so that boxes cannot be detected. The detected line segments are visualized in the "Intermediate Result" visualization on the Web GUI's *BoxPick* page.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?manual_
↪line_sensitivity=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?manual_line_sensitivity=<value>
```

`line_sensitivity` (*Line Sensitivity*)

determines the line sensitivity for extracting the lines for rectangle detection, if the parameter `manual_line_sensitivity` is set to true. Otherwise, the value of this parameter has no effect on the rectangle detection. Higher values give more line segments, but also increase the runtime of the box detection. This parameter should be increased when boxes cannot be detected because their boundary edges are not detected. The detected line segments are visualized in the "Intermediate Result" visualization on the Web GUI's *BoxPick* page.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?line_
↪sensitivity=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?line_sensitivity=<value>
```

`prefer_splits` (*Prefer Splits*)

determines whether rectangles should be split into smaller ones if the smaller ones also match the given item models. This parameter should be set to true for packed box layouts in which the given item models would also match a rectangle of the size of two adjoining boxes. If this parameter is set to false, the larger rectangles will be preferred in these cases.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?prefer_
↪splits=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?prefer_splits=<value>
```

`min_cluster_coverage` (*Minimum Cluster Coverage*)

determines which ratio of each segmented cluster must be covered with rectangle detections to consider the detections to be valid. If the minimum cluster coverage is not reached for a cluster, no rectangle detections will be returned for this cluster and a warning will be given. This parameter should be used to verify that all items on a layer in a de-palletizing scenario are detected.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?min_
↪cluster_coverage=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?min_cluster_coverage=<value>
```

`allow_untextured_detections` (Only for BoxPick+Match, *Allow Untextured Detections*)

enables returning all rectangles matching the given template dimensions, even when they cannot be matched to an existing view or when they do not have enough texture to create a new view from them. This parameter is only used when item models of type `TEXTURED_BOX` are detected. Disabling this parameter leads to faster detections when used with a template for which the automatic view updating is disabled.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?allow_
↪untextured_detections=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?allow_untextured_detections=
↪<value>
```

`discard_cut_off_items` (*Discard Cut-Off Items*)

determines, whether items that are cut off by the image border, the stereo region of interest (see *Stereo ROI Parameters*, Section 5.2.1), the missing stereo overlap between the left and right camera image, or the 3D region of interest should be discarded.

This parameter has no effect when the `UNKNOWN` item model type is used.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?discard_
↪cut_off_items=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?discard_cut_off_items=<value>
```

`grasp_filter_orientation_threshold` (*Grasp Orientation Threshold*)

is the maximum deviation of the TCP's z axis at the grasp point from the z axis of the TCP's preferred orientation in degrees. Only grasp points which are within this threshold are returned. When set to zero, any deviations are valid.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?grasp_
↪filter_orientation_threshold=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?grasp_filter_orientation_
↪threshold=<value>
```

`allow_any_grasp_z_rotation` (*Allow Any Grasp Z Rotation*)

If set to true, the returned grasps are no longer forced to have their x axes aligned with the maximum elongation of the graspable ellipse, but can have any rotation around the z axis. The returned `max_suction_surface_length` and `max_suction_surface_width` will be equal and correspond to the shortest diameter of the largest graspable ellipse. This parameter enables the robot to get more options for grasping objects, especially in scenes where collisions can occur. However, since the grasp is no longer aligned with the graspable ellipse, the correct orientation for placing the object must be determined from the corresponding `item`'s pose.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?allow_any_
↪grasp_z_rotation=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?allow_any_grasp_z_rotation=<value>
```

`check_collisions_with_point_cloud` (*Check Collisions with Point Cloud*)

This parameter is only used when collision checking is enabled by passing a gripper to the `compute_grasps` or `compute_grasps_extended` service call. If `check_collisions_with_point_cloud` is set to true, all grasp points will be checked for collisions between the gripper and a watertight version of the point cloud, and only grasp points at which the gripper would not collide with this point cloud will be returned.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/parameters/parameters?check_
↪collisions_with_point_cloud=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/parameters?check_collisions_with_point_cloud=
↪<value>
```

5.4.5.8 Status values

The `rc_boxpick` module reports the following status values:

Table 5.33: The `rc_boxpick` module's status values

Name	Description
`data_acquisition_time`	Time in seconds required by the last active service to acquire images
`grasp_computation_time`	Processing time of the last grasp computation in seconds
`last_return_code_message`	Return code message of the last detection
`last_return_code_value`	Return code value of the last detection
`last_timestamp_processed`	The timestamp of the last processed dataset
`load_carrier_detection_time`	Processing time of the last load carrier detection in seconds
`processing_time`	Processing time of the last detection (including load carrier detection) in seconds
`state`	The current state of the rc_boxpick node

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

Table 5.34: Possible states of the BoxPick module

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

5.4.5.9 Services

The user can explore and call the `rc_boxpick` module's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1).

The BoxPick module offers the following services.

`detect_items`

Triggers the detection of rectangles as described in *Detection of items* (Section 5.4.5.2).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/detect_items
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/detect_items
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.5.6).

`item_models`: list of item models to be detected. The type of the item model must be `RECTANGLE` or `TEXTURED_BOX`. For type `RECTANGLE`, `rectangle` must be filled, while for `TEXTURED_BOX`, `textured_box` must be filled. See *Detection of items* (Section 5.4.5.2) for a detailed description of the item model types.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.5.6).

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 (see *Load carrier compartments*, Section 5.6.1.3).

`region_of_interest_id`: if `load_carrier_id` is set, ID of the 3D region of interest where to search for the load carriers. Otherwise, ID of the 3D region of interest where to search for the items.

`dimensioning`: `plane_ids` contains the ID of the reference plane for measuring the dimensions of the detected items (see *Dimensioning*, Section 5.4.5.2).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "dimensioning": {
      "plane_ids": [
        "string"
      ]
    },
    "item_models": [
      {
        "rectangle": {
          "max_dimensions": {
            "x": "float64",
            "y": "float64"
          },
          "min_dimensions": {
            "x": "float64",
            "y": "float64"
          }
        },
        "textured_box": {
          "dimensions": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "dimensions_tolerance_m": "float64",
          "max_deformation_m": "float64",
          "template_id": "string"
        },
        "type": "string"
      }
    ],
    "load_carrier_compartment": {
      "box": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "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"
        }
      }
    },
    "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"
      }
    }
  }
}
```

Response

`load_carriers`: list of detected load carriers.

`items`: list of detected rectangles. If `dimensioning` is given, list of detected boxes.

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

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

The definition for the response with corresponding datatypes is:

```
{
  "name": "detect_items",
  "response": {
    "items": [
      {
        "box": {
          "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",
        "rectangle": {
```

(continues on next page)

(continued from previous page)


```
        "x": "float64",
        "y": "float64"
      },
      "template_id": "string",
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      },
      "type": "string",
      "uuid": "string",
      "view_name": "string",
      "view_pose_set": "bool",
      "view_uuid": "string"
    }
  ],
  "load_carriers": [
    {
      "height_open_side": "float64",
      "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_ledge": {
        "x": "float64",
        "y": "float64"
      },
      "rim_step_height": "float64",
      "rim_thickness": {
        "x": "float64",
        "y": "float64"
      },
      "type": "string"
    }
  ],
  "return_code": {
    "message": "string",
    "value": "int16"
  },
  "timestamp": {
```


(continues on next page)

(continued from previous page)

```
      "nsec": "int32",
      "sec": "int32"
    }
  }
}
```

compute_grasps

Triggers the detection of rectangles and the computation of grasping poses for the detected rectangles as described in *Computation of grasps* (Section 5.4.5.3).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/compute_grasps
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/compute_grasps
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.5.6).

`item_models`: list of item models to be detected. The type of the item model must be `RECTANGLE` or `TEXTURED_BOX`. For type `RECTANGLE`, `rectangle` must be filled, while for `TEXTURED_BOX`, `textured_box` must be filled. See *Detection of items* (Section 5.4.5.2) for a detailed description of the item model types.

`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 5.4.5.6).

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 (see *Load carrier compartments*, Section 5.6.1.3).

`region_of_interest_id`: if `load_carrier_id` is set, ID of the 3D region of interest where to search for the load carriers. Otherwise, ID of the 3D region of interest where to compute grasps.

`collision_detection`: see *Collision checking within other modules* (Section 5.5.2.2).

`dimensioning`: `plane_ids` contains the ID of the reference plane for measuring the dimensions of the detected items (see *Dimensioning*, Section 5.4.5.2).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "collision_detection": {
```

(continues on next page)

(continued from previous page)


```
      "gripper_id": "string",
      "pre_grasp_offset": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "dimensioning": {
      "plane_ids": [
        "string"
      ]
    },
    "item_models": [
      {
        "rectangle": {
          "max_dimensions": {
            "x": "float64",
            "y": "float64"
          },
          "min_dimensions": {
            "x": "float64",
            "y": "float64"
          }
        },
        "textured_box": {
          "dimensions": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "dimensions_tolerance_m": "float64",
          "max_deformation_m": "float64",
          "template_id": "string"
        },
        "type": "string"
      }
    ],
    "load_carrier_compartment": {
      "box": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      }
    },
    "load_carrier_id": "string",
    "pose_frame": "string",
    "region_of_interest_id": "string",
    "robot_pose": {
```


(continues on next page)

(continued from previous page)

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

Response

`load_carriers`: list of detected load carriers.

`grasps`: sorted list of suction grasps.

`items`: list of detected rectangles corresponding to the returned grasps. If `dimensioning` is given, list of detected boxes.

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

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

The definition for the response with corresponding datatypes is:

```
{
  "name": "compute_grasps",
  "response": {
    "grasps": [
      {
        "item_uuid": "string",
        "max_suction_surface_length": "float64",
        "max_suction_surface_width": "float64",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "pose_frame": "string",
        "quality": "float64",
        "tcp_id": "string",
        "timestamp": {
          "nsec": "int32",
          "sec": "int32"
        },
        "type": "string",
        "uuid": "string"
      }
```

(continues on next page)

(continued from previous page)


```
  ],
  "items": [
    {
      "box": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "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"
      },
      "template_id": "string",
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      },
      "type": "string",
      "uuid": "string",
      "view_name": "string",
      "view_pose_set": "bool",
      "view_uuid": "string"
    }
  ],
  "load_carriers": [
    {
      "height_open_side": "float64",
      "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",
```


(continues on next page)

(continued from previous page)


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


compute_grasps_extended

Triggers the detection of rectangles and the computation of grasping poses for for detected rectangles in the same way as `compute_grasps`, but returns the item information for each grasp directly instead of as a separate list. This allows for easier parsing when item information is required for the grasps.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/compute_grasps_extended
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/compute_grasps_extended
```

Request

See `compute_grasps` service.

The definition for the request arguments with corresponding datatypes is:


```
{
  "args": {
    "collision_detection": {
      "gripper_id": "string",
```


(continues on next page)

(continued from previous page)


```
      "pre_grasp_offset": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "dimensioning": {
      "plane_ids": [
        "string"
      ]
    },
    "item_models": [
      {
        "rectangle": {
          "max_dimensions": {
            "x": "float64",
            "y": "float64"
          },
          "min_dimensions": {
            "x": "float64",
            "y": "float64"
          }
        },
        "textured_box": {
          "dimensions": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "dimensions_tolerance_m": "float64",
          "max_deformation_m": "float64",
          "template_id": "string"
        },
        "type": "string"
      }
    ],
    "load_carrier_compartment": {
      "box": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      }
    },
    "load_carrier_id": "string",
    "pose_frame": "string",
    "region_of_interest_id": "string",
    "robot_pose": {
      "orientation": {
```


(continues on next page)

(continued from previous page)

```
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "suction_surface_length": "float64",
    "suction_surface_width": "float64"
  }
}
```

Response

`load_carriers`: list of detected load carriers.

`grasps`: sorted list of suction grasps. Each grasp also contains the item information of the corresponding detected rectangle.

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

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

The definition for the response with corresponding datatypes is:

```
{
  "name": "compute_grasps_extended",
  "response": {
    "grasps": [
      {
        "item": {
          "box": {
            "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",
          "rectangle": {
            "x": "float64",
            "y": "float64"
          },
          "template_id": "string",
          "type": "string",
          "uuid": "string",
          "view_name": "string",
          "view_pose_set": "bool",
```

(continues on next page)

(continued from previous page)


```
        "view_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",
      "tcp_id": "string",
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      },
      "type": "string",
      "uuid": "string"
    }
  ],
  "load_carriers": [
    {
      "height_open_side": "float64",
      "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_ledge": {
        "x": "float64",
        "y": "float64"
      },
```


(continues on next page)

(continued from previous page)


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


set_preferred_orientation

Persistently stores the preferred orientation of the TCP to compute the reachability of the grasps, which is used for filtering and the grasps returned by the `compute_grasps` or `compute_grasps_extended` service (see *Setting the preferred orientation of the TCP*, Section 5.4.5.5).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/set_preferred_
↪orientation
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/set_preferred_orientation
```

Request

The definition for the request arguments with corresponding datatypes is:


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


Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_preferred_orientation",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

get_preferred_orientation

Returns the preferred orientation of the TCP to compute the reachability of the grasps, which is used for filtering the grasps returned by the `compute_grasps` and `compute_grasps_extended` service (see *Setting the preferred orientation of the TCP*, Section 5.4.5.5).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/get_preferred_
↪orientation
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/get_preferred_orientation
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

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

set_sorting_strategies

Persistently stores the sorting strategy for sorting the items and grasps (see *Setting the sorting strategies*, Section 5.4.5.4).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/set_sorting_strategies
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/set_sorting_strategies
```

Request

Only one strategy may have a `weight` greater than 0. If all `weight` values are set to 0, the module will use the default sorting strategy.

If the weight for `direction` is set, the `vector` must contain the direction vector and `pose_frame` must be either `camera` or `external`.

If the weight for `distance_to_point` is set, `point` must contain the sorting point and `pose_frame` must be either `camera` or `external`.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "direction": {
      "pose_frame": "string",
      "vector": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "weight": "float64"
    },
    "distance_to_point": {
      "farthest_first": "bool",
      "point": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose_frame": "string",
      "weight": "float64"
    },
    "gravity": {
      "weight": "float64"
    },
    "surface_area": {
      "weight": "float64"
    }
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_sorting_strategies",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

get_sorting_strategies

Returns the sorting strategy for sorting the items and grasps (see *Setting the sorting strategies*, Section 5.4.5.4).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/get_sorting_strategies
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/get_sorting_strategies
```

Request

This service has no arguments.

Response

All `weight` values are 0 when the module uses the default sorting strategy.

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_sorting_strategies",
  "response": {
    "direction": {
      "pose_frame": "string",
      "vector": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "weight": "float64"
    },
    "distance_to_point": {
      "farthest_first": "bool",
      "point": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose_frame": "string",
      "weight": "float64"
    },
    "gravity": {
      "weight": "float64"
    },
    "return_code": {
      "message": "string",
      "value": "int16"
    },
    "surface_area": {
      "weight": "float64"
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

start

Starts the module. If the command is accepted, the module moves to state `RUNNING`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/start
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/start
```

Request

This service has no arguments.

Response

The `current_state` value in the service response may differ from `RUNNING` if the state transition is still in process when the service returns.

The definition for the response with corresponding datatypes is:

```
{
  "name": "start",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

stop

Stops the module. If the command is accepted, the module moves to state `IDLE`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/stop
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/stop
```

Request

This service has no arguments.

Response

The `current_state` value in the service response may differ from `IDLE` if the state transition is still in process when the service returns.

The definition for the response with corresponding datatypes is:

```
{
  "name": "stop",
  "response": {
    "accepted": "bool",
    "current_state": "string"
  }
}
```

reset_defaults

Resets all parameters of the module to its default values, as listed in above table. Also resets the sorting strategies and the preferred orientation.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_boxpick/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_boxpick/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.4.5.10 Return codes

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 5.35: Return codes of the BoxPick services

Code	Description
0	Success
-1	An invalid argument was provided
-3	An internal timeout occurred, e.g. during box detection if the given dimension range is too large
-4	Data acquisition took longer than allowed
-8	The template has been deleted during detection
-10	New element could not be added as the maximum storage capacity of load carriers, regions of interest or template has been exceeded
-11	Sensor not connected, not supported or not ready
-12	Resource busy, e.g. when `trigger_dump` is called too frequently
-200	Fatal internal error
-301	More than one item model provided to the `compute_grasps` or `compute_grasps_extended` service
10	The maximum storage capacity of load carriers, regions of interest or templates 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 were found in the scene
102	The detected load carrier is empty
103	All computed grasps are in collision
112	Rejected detections of one or more clusters, because min_cluster_coverage was not reached
300	A valid `robot_pose` was provided as argument but it is not required
999	Additional hints for application development

5.4.5.11 BoxPick Template API

BoxPick templates are only available with the +Match extension of BoxPick. For template upload, download, listing and removal, special REST-API endpoints are provided. Templates can also be uploaded, downloaded and removed via the Web GUI. The templates include the dimensions, the views and their poses, if set. Up to 100 templates can be stored persistently on the *rc_visard*.

`GET /templates/rc_boxpick`
Get list of all rc_boxpick templates.

Template request

```
GET /api/v2/templates/rc_boxpick HTTP/1.1
```

Template response

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

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

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Template* (Section 6.2.4)

GET /templates/rc_boxpick/{id}

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

Template request

```
GET /api/v2/templates/rc_boxpick/<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/ubjson application/octet-stream

Status Codes

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

Referenced Data Models

- *Template* (Section 6.2.4)

PUT /templates/rc_boxpick/{id}

Create or update a rc_boxpick template.

Template request

```
PUT /api/v2/templates/rc_boxpick/<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 application/ubjson

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 module.
- 404 Not Found – node or template not found
- 413 Request Entity Too Large – Template too large

Referenced Data Models

- *Template* (Section 6.2.4)

`DELETE /templates/rc_boxpick/{id}`
Remove a rc_boxpick template.

Template request

```
DELETE /api/v2/templates/rc_boxpick/<id> HTTP/1.1
Accept: application/json application/ubjson
```

Parameters

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

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

5.4.6 Pack

5.4.6.1 Introduction

The Pack module provides an out-of-the-box perception solution for robotic pick-and-place applications that need to place previously picked objects inside a packing volume, such as a box, bin or onto a pallet. Given the bounding box of the object to be placed and the grasp(s) that can be used to pick it up, Pack computes a pose at which the object can be placed, taking the current filling state of the packing volume into account. The place pose is chosen to be the lowest position that provides stable support for the object, thereby maximizing the residual space left after placing.

In addition, the module offers:

- A dedicated page on the *rc_visard Web GUI* (Section 6.1) for easy setup, configuration, testing, and application tuning.

- The definition of a region of interest to specify the relevant packing volume in the scene, or to restrict the search area for the load carrier (see *RoiDB*, Section 5.6.2).
- A load carrier detection functionality (see *LoadCarrier*, Section 5.4.2), to place objects inside a detected load carrier only.
- A configurable starting point to control where packing begins.
- Support for static and robot-mounted cameras and optional integration with the *Hand-eye calibration* (Section 5.5.1) module, to provide place poses in the user-configured external reference frame.
- 3D visualization of the packing volume and the computed place pose in the Web GUI.

The module is an optional on-board module of the *rc_visard* and requires a separate Pack *license* (Section 7.4) to be purchased.

5.4.6.2 Computation of place poses

The Pack module offers a service for computing a place pose for an object, regardless of whether it has already been picked or is yet to be picked. The object to be placed is described by its `bounding_box`. Optionally, the object's current `pose` and one or more `grasps` that can be used to pick it up can be given as well; this pose and these grasps can originate from another detection module, such as ItemPick or BoxPick. Alternatively, the `grasps` can be given relative to the center of the object's bounding box.

The module discretizes the packing volume in x and y direction according to the run-time parameter `resolution`. Starting from the corner of the packing volume closest to the configured *starting point*, it searches for the lowest position at which the object, padded by the run-time parameter `object_padding` as a safety margin, fits without overlapping already placed objects or the load carrier. This maximizes the residual space left after placing, keeping as many packing options open as possible for the next object to be placed.

Optionally, further information can be given to the module in a place pose computation request:

- The ID of the load carrier that defines the packing volume.
- The ID of the 3D region of interest that defines the packing volume, if no load carrier is set, or that restricts the search area for the load carrier.

The place pose definition is complemented by the `timestamp` of the oldest image that was used to compute it.

5.4.6.3 Setting the starting point

The Pack module determines where to begin filling the packing volume based on the *starting point*. The module compares the configured starting point against the corners of the packing volume, defined by the configured load carrier or region of interest, and begins filling from whichever corner is closest to it.

The starting point can be set via the `set_starting_point` service or on the Pack page in the Web GUI, and is persistently stored. It can be set in the camera coordinate frame or in the external coordinate frame, in case a hand-eye calibration is available.

5.4.6.4 Setting the preferred orientation

When placing a box, each place pose has a 180° rotation ambiguity around the z axis. Among the given `grasps`, the Pack module selects the ones whose orientations for placing the object are closest to the *preferred orientation*, thus resolving the 180° placing ambiguity. The preferred orientation can be set via the `set_preferred_orientation` service or on the Pack page in the Web GUI. Grasps that deviate from it by more than the run-time parameter `grasp_filter_orientation_threshold` are discarded.

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 place pose computation call.

If no preferred orientation is set, the orientation of the left camera (see Coordinate frames) will be used as the preferred orientation.

5.4.6.5 Interaction with other modules

Internally, the Pack module depends on, and interacts with other on-board modules as listed below.

Note: All changes and configuration updates to these modules will affect the performance of the Pack module.

Camera and depth data

The Pack module makes internally use of the following data:

- Rectified images from the *Camera module* (`rc_camera`, Section 5.1);
- Disparity, error, and confidence images from the *Stereo matching module* (`rc_stereomatching`, Section 5.2)

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

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* module (`rc_iocontrol`, Section 5.5.4), it is recommended to connect the projector to GPIO Out 1 and set the stereo-camera module's acquisition mode to `SingleFrameOut1` (see *Stereo matching parameters*, Section 5.2.1), 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 5.5.4.1).

In either case, the *Auto Exposure Mode* `exp_auto_mode` should be set to `AdaptiveOut1` to optimize the exposure of both images.

Hand-eye calibration

In case the camera has been calibrated to a robot, the Pack module can automatically provide poses in the robot coordinate frame. For the Pack node's *Services* (Section 5.4.6.8), 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 module 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 module are in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) 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.

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

- If `pose_frame` is set to `external`, providing the robot pose is obligatory.
- In all other cases, providing the robot pose is optional.

LoadCarrier

The Pack module uses the load carrier detection functionality provided by the *LoadCarrier* module (`rc_load_carrier`, Section 5.4.2), with the run-time parameters specified for this module, to determine the boundaries of the packing volume. However, only one load carrier will be used in case multiple matching load carriers could be found in the scene. In case multiple load carriers of the same type are visible, a 3D region of interest should be set to ensure that always the same load carrier is used for the Pack module.

5.4.6.6 Parameters

Pack is represented by the `rc_pack` node in the REST-API and are reached in the *Web GUI* (Section 6.1) under *Modules* → *Pack*. The user can explore and configure the `rc_pack` module's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.36: The `rc_pack` module's run-time parameters

Name	Type	Min	Max	Default	Description
`drop_height`	float64	0.0	0.5	0.0	height above target place position (in z direction of packing volume) to release object
`grasp_filter_orientation_threshold`	float64	0.0	180.0	45.0	Maximum allowed orientation change between grasp and preferred orientation in degrees
`object_padding`	float64	0.001	1.0	0.005	width in meters for padding the object dimensions in x and y direction as safety margin
`resolution`	float64	0.001	0.1	0.005	resolution for discretizing the packing volume in x and y direction

Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's Pack page. 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:

`resolution` (*Resolution*)

is the resolution used to discretize the packing volume in x and y direction when searching for a free placement position, in meters. The smaller this value, the finer the search grid, at the cost of additional computation time.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/parameters/parameters?resolution=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/parameters?resolution=<value>
```

`object_padding` (*Object Padding*)

is the width, in meters, used to pad the object's `bounding_box` in x and y direction as a safety margin when checking whether it fits at a candidate placement position.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/parameters/parameters?object_
↪padding=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/parameters?object_padding=<value>
```

`drop_height` (*Drop Height*)

defines the height, in meters, above the target place position, along the z axis of the packing volume, at which the gripper releases the object, e.g. to let it drop a short distance instead of being placed under full control all the way down to the surface. The place pose provided by the Pack module represents the recommended pose of the object at the moment of release.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/parameters/parameters?drop_height=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/parameters?drop_height=<value>
```

`grasp_filter_orientation_threshold` (*Grasp Orientation Threshold*)

is the maximum deviation of the TCP's z axis at the grasp point from the z axis of the TCP's preferred orientation (see *preferred orientation*) in degrees. Only grasp points which are within this threshold are returned. When set to zero, any deviations are valid.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/parameters/parameters?grasp_filter_
↪orientation_threshold=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/parameters?grasp_filter_orientation_threshold=
↪<value>
```

5.4.6.7 Status values

The `rc_pack` node reports the following status values:

Table 5.37: The `rc_pack` node's status values

Name	Description
`data_acquisition_time`	Time in seconds required by the last active service to acquire images
`last_return_code_message`	Return code message of the last detection
`last_return_code_value`	Return code value of the last detection
`last_timestamp_processed`	The timestamp of the last processed dataset
`load_carrier_detection_time`	Processing time of the last load carrier detection in seconds
`processing_time`	Processing time of the last detection (including load carrier detection) in seconds

5.4.6.8 Services

The user can explore and call the `rc_pack` node's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1).

The Pack module offers the following services.

`detect_place_pose`

Triggers the computation of a place pose as described in *Computation of place poses* (Section 5.4.6.2).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/services/detect_place_pose
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/services/detect_place_pose
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.6.5).

`objects`: the object to be placed, described by its `bounding_box` and, optionally, the `pose` at which it is positioned. Currently, only a single object is supported.

`grasps`: one or more grasps that were used, or can be used, to pick the object listed in `objects`, referenced via `object_id`. If no `pose_frame` is given, each grasp's `pose` is interpreted relative to the center of the object's bounding box. If a `pose_frame`

is given, the object also requires a `pose`. This allows detection results from other modules, such as ItemPick or BoxPick, to be reused for Pack.

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.6.5).

Optional arguments:

`load_carrier_id`: ID of the load carrier that defines the packing volume.

`region_of_interest_id`: ID of the 3D region of interest that defines the packing volume, if `load_carrier_id` is not set, or that restricts the area for detecting the load carrier.

At least one of `load_carrier_id` or `region_of_interest_id` must be provided.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "grasps": [
      {
        "gripper_id": "string",
        "id": "string",
        "object_id": "string",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "pose_frame": "string",
        "stroke_per_finger_approach_mm": "float64",
        "stroke_per_finger_grasp_mm": "float64",
        "tcp_id": "string"
      }
    ],
    "load_carrier_id": "string",
    "objects": [
      {
        "bounding_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"
```

(continues on next page)

(continued from previous page)

```
          }
        },
        "pose_frame": "string"
      }
    ],
    "pose_frame": "string",
    "region_of_interest_id": "string"
  }
}
```

Response

`pose`: the computed place pose.

`grasps`: the grasp(s) passed in the request that can be used to achieve the final placement.

`load_carriers`: the list containing the detected load carrier.

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

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

The definition for the response with corresponding datatypes is:

```
{
  "name": "detect_place_pose",
  "response": {
    "grasps": [
      {
        "collision_checked": "bool",
        "gripper_id": "string",
        "id": "string",
        "object_id": "string",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "pose_frame": "string",
        "timestamp": {
          "nsec": "int32",
          "sec": "int32"
        }
      }
    ],
    "load_carriers": [
      {
        "height_open_side": "float64",
        "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"
        },
        "overfilled": "bool",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "pose_frame": "string",
        "rim_ledge": {
          "x": "float64",
          "y": "float64"
        },
        "rim_step_height": "float64",
        "rim_thickness": {
          "x": "float64",
          "y": "float64"
        },
        "type": "string"
      }
    ],
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "pose_frame": "string",
    "return_code": {
      "message": "string",
      "value": "int16"
    },
    "timestamp": {
      "nsec": "int32",
      "sec": "int32"
    }
  }
}
```


get_starting_point

Returns the starting point used to select the corner of the packing volume from which filling begins (see *Setting the starting point*, Section 5.4.6.3).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/services/get_starting_point
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/services/get_starting_point
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_starting_point",
  "response": {
    "point": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "pose_frame": "string",
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

set_starting_point

Persistently stores the starting point used to select the corner of the packing volume from which filling begins (see *Setting the starting point*, Section 5.4.6.3).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/services/set_starting_point
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/services/set_starting_point
```

Request

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "point": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
```

(continues on next page)

(continued from previous page)

```
    "pose_frame": "string"
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_starting_point",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

get_preferred_orientation

Returns the preferred orientation used when selecting the grasp for placing the object (see *Setting the preferred orientation*, Section 5.4.6.4).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/services/get_preferred_orientation
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/services/get_preferred_orientation
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

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

set_preferred_orientation

Persistently stores the preferred orientation used when selecting the grasp for placing the object (see *Setting the preferred orientation*, Section 5.4.6.4).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/services/set_preferred_orientation
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/services/set_preferred_orientation
```

Request

The definition for the request arguments with corresponding datatypes is:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_preferred_orientation",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

reset_defaults

Resets all parameters of the module to its default values, as listed in above table. Also resets the starting point and the preferred orientation.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_pack/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_pack/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.4.6.9 Return codes

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 5.38: Return codes of the Pack services

Code	Description
0	Success
-1	An invalid argument was provided
-4	Data acquisition took longer than allowed
-11	Sensor not connected, not supported or not ready
-12	Resource busy, e.g. when `trigger_dump` is called too frequently
-200	Fatal internal error
100	The requested load carrier was not detected in the scene
115	No valid place pose was found within the packing volume
300	A valid `robot_pose` was provided as argument but it is not required
999	Additional hints for application development

5.4.7 SilhouetteMatch

5.4.7.1 Introduction

The SilhouetteMatch module is an optional on-board module of the *rc_visard* and requires a separate SilhouetteMatch *license* (Section 7.4) to be purchased.

The module detects objects by matching a predefined silhouette (“template”) to edges in the image.

The SilhouetteMatch module can detect objects in two different scenarios:

With calibrated base plane: The objects are placed on a common base plane, which must be calibrated before the detection, and the objects have significant edges on a common plane that is parallel to the base plane.

With object plane detection: The objects can be placed at different, previously unknown heights, if the objects have a planar surface and their outer contours are well visible in the images (e.g. stacked flat objects).

Templates for object detection can be created by uploading a DXF file and specifying the object height. The correct scale and unit of the contours are extracted from the DXF file. If no units are present in the DXF file, the user has to specify which units should be used. When the outer contour of the object in the DXF file is closed, a 3D collision model is created automatically by extruding the contour by the given object height. This model will then be used for collision checking and in 3D visualizations. Uploading a DXF file can be done in the Web GUI via the *+ Create a new Template* button in the *SilhouetteMatch Templates and Grasps* section on the *Modules* → *SilhouetteMatch* or *Database* → *Templates* pages.

Basler also offers a template generation service, where the user can upload CAD files or recorded data of the objects and request object templates for the SilhouetteMatch module.

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 module offers:

- A dedicated page on the *rc_visard* *Web GUI* (Section 6.1) for easy setup, configuration, testing, and application tuning.
- A *REST-API interface* (Section 6.2) and a *KUKA Ethernet KRL Interface* (Section 6.6).
- The definition of 2D regions of interest to select relevant parts of the camera image (see *Setting a region of interest*, Section 5.4.7.3).
- A load carrier detection functionality for bin-picking applications (see *LoadCarrier*, Section 5.4.2), to provide grasps for objects inside a bin only.
- Storing of up to 50 templates.
- The definition of up to 50 grasp points for each template via an interactive visualization in the Web GUI.
- Collision checking between the gripper and the load carrier, the calibrated base plane, other detected objects and/or the point cloud.
- Support for static and robot-mounted cameras and optional integration with the *Hand-eye calibration* (Section 5.5.1) module, to provide grasps in the user-configured external reference frame.
- Selection of a sorting strategy to sort the detected objects and returned grasps.
- 3D visualization of the detection results with grasp points and gripper animations in the Web GUI.

Suitable objects

The SilhouetteMatch module is intended for objects which have significant edges on a common plane that is parallel to the 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 detection works best for objects on a texture-free 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.

In case the objects are not placed on a common base plane or the base plane cannot be calibrated beforehand, the objects need to have a planar surface and their outer contour must be well visible in the left and right images. Furthermore, the template for these objects must have a closed outer contour.

Suitable scene

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

- The objects to be detected must be suitable for the SilhouetteMatch module 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.
- In case a calibrated base plane is used: The offset between the base plane normal and the camera's line of sight does not exceed 10 degrees.
- In case the object planes are detected automatically: The offset between the object's planar surface normal and the camera's line of sight does not exceed 25 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.

5.4.7.2 Base-plane calibration

In case all objects are placed on a common plane that is known beforehand, a base-plane calibration should be performed before triggering a detection. 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 5.4.3) is used to find AprilTags in the scene and fit a plane through them. At least three AprilTags must be placed on the base plane so that they are visible in the left and right camera images. The tags should be placed such that they are spanning a triangle that is as large as possible. The larger the triangle, the more accurate is the resulting base-plane estimate. Use this method if the base plane is untextured and no external random dot projector is available. This calibration mode is available via the *REST-API interface* (Section 6.2) and the *rc_visard* Web GUI.

Stereo based base-plane calibration

The 3D point cloud computed by the stereo matching module 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 6.2) 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 6.2) and not the *rc_visard* Web GUI.

5.4.7.3 Setting a region of interest

If objects are to be detected only in part of the camera's field of view, a 2D region of interest (ROI) can be set accordingly as described in *Region of interest* (Section 5.6.2.2).

5.4.7.4 Setting of grasp points

To use SilhouetteMatch directly in a robot application, up to 50 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. 5.19.

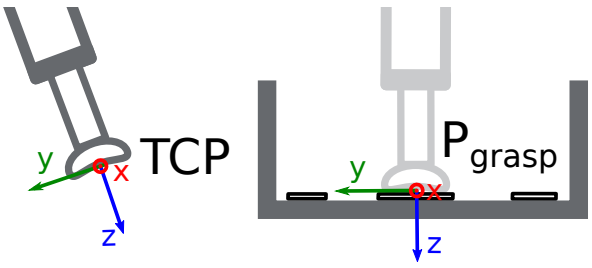


Fig. 5.19: Definition of grasp points with respect to the robot's TCP

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 6.2), or by using the interactive visualization in the Web GUI. Furthermore, a `priority` (spanning -2 for very low to 2 for very high) can be assigned to a grasp. Priorities can facilitate robot applications and shorten response times when the run-time parameter `only_highest_priority_grasps` is set to true. In this case collision checking concludes when grasps with the highest possible priority have been found. Finally,

different grasps can be associated with different grippers by specifying a `gripper_id`. These individual grippers are then used for collision checking of the corresponding grasps instead of the gripper defined in the `detect_object` or `detect_object_extended` request. If no `gripper_id` is given, the gripper defined in the `detect_object` or `detect_object_extended` request will be used for collision checking.

If a `gripper_id` is given for a grasp and the corresponding gripper has elements of `function_type FINGER`, each grasp can also specify values for `stroke_per_finger_approach_mm` and `stroke_per_finger_grasp_mm`. These values give the amount of translation of a finger in millimeters by which the finger element and all its child elements are moved from the `zero_pose` towards the `pose` of the finger element. The `stroke_per_finger_approach_mm` value is applied during collision checking when the grasp is approached. The `stroke_per_finger_grasp_mm` is not used for collision checking, but holds information about the gripper opening while grasping, so that this field implicitly defines the finger's motion direction for performing the grasp. If neither `stroke_per_finger_approach_mm` nor `stroke_per_finger_grasp_mm` are given, the gripper will be used for collision checking with the fingers in their default poses.

When a grasp is defined on a symmetric object, all grasps symmetric to the defined one will automatically be considered in the SilhouetteMatch module's `detect_object` and `detect_object_extended` service calls. Symmetric grasps for a given grasp point can be retrieved using the `get_symmetric_grasps` service call and visualized in the Web GUI.

Users can also define replications of grasps around a custom axis. These replications spawn multiple grasps and free users from setting too many grasps manually. The replication origin is defined as a coordinate frame in the object's coordinate frame and the x axis of the replication origin frame corresponds to the replication axis. The grasp is replicated by rotating it around this x axis starting from its original pose. The replication is done in steps of size `step_x_deg` degrees. The range is defined by the minimal and maximal boundaries `min_x_deg` and `max_x_deg`. The minimal (maximal) boundary must be a non-positive (non-negative) number up to (minus) 180 degrees.

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 in the Web GUI in any pipeline under *Modules* → *SilhouetteMatch* by clicking on *+ Add a new Template* in the *Templates and Grasps* section, or in *Database* → *Templates* in the *SilhouetteMatch Templates and Grasps* section. Once the template upload is complete, a dialog with a 3D visualization of the object template is shown for adding or editing grasp points. The same dialog appears when editing an existing template. If the template contains a collision model or a visualization model, this 3D model is visualized as well.

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 right corner of the visualization and then clicking on the desired point on the object template visualization. If the 3D model is displayed, the grasps will be attached to the surface of the 3D model. Otherwise, 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 disabled (default), the grasp can be translated and rotated freely in all three dimensions by clicking on *Move Grasp* in the visualization menu and then dragging the grasp along the appropriate axis to the desired position. The orientation of the grasp can also be changed by rotating the axis with the mouse. In case *Snap to surface* is enabled in the visualization, the grasp can only be moved along the model surface.

Users can also specify a grasp priority by changing the *Priority* slider. A dedicated gripper can be selected in the *Gripper* drop down field.

By activating the *Replication* check box, users can replicate the grasp around a custom axis. The replication axis and the resulting replicated grasps are visualized. The position and orientation of the replication axis relative to the object coordinate frame can be adjusted interactively by clicking on *Move Replication Axis* in the visualization menu and then dragging the axis to the desired position and orientation. The grasps are replicated within the specified rotation range at the selected rotation step size. Users can cycle through a visualization of the replicated grasps by dragging the bar below *Cycle through n replicated grasps* in the *View Options* section of the visualization menu. If a gripper is selected for the grasp or a gripper has been chosen in the visualization menu, the gripper is also shown at the currently selected grasp.

If the object template has symmetries, the grasps which are symmetric to the defined grasps can be displayed along with their replications (if defined) by enabling ... *symmetries* in the *View Options* section of the visualization menu. The user can also cycle through a visualization of the symmetric grasps by dragging the bar below *Cycle through n symmetric grasps*.

Setting grasp points via the REST-API

Grasp points can be set via the *REST-API interface* (Section 6.2) using the `set_grasp` or `set_all_grasps` services (see *Internal services*, Section 5.4.7.12). 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. A dedicated gripper can be specified through setting the `gripper_id` field. The `priority` is specified by an integer value, ranging from -2 for very low, to 2 for very high with a step size of 1. The replication origin is defined as a transformation in the object's coordinate frame and the x axis of the transformation corresponds to the replication axis. The replication range is controlled by the `min_x_deg` and `max_x_deg` fields and the step size `step_x_deg`.

5.4.7.5 Setting the preferred orientation of the TCP

The SilhouetteMatch module determines the reachability of grasp points based on the *preferred orientation* of the 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, the preferred orientation can be used to sort the reachable grasps by setting the corresponding sorting strategy.

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. If no preferred orientation is set, the orientation of the left camera (see Coordinate frames) will be used as the preferred orientation of the TCP.

5.4.7.6 Setting the sorting strategies

The objects and grasps returned by the `detect_object` and `detect_object_extended` service call are sorted according to a sorting strategy which can be chosen by the user. The following sorting strategies are available and can be set in the *Web GUI* (Section 6.1) or using the `set_sorting_strategies` service call:

- `preferred_orientation`: matches and grasp points with minimal rotation difference of a chosen `axis` (or all axes, when `axis` is empty) with respect to the preferred TCP orientation are returned first,
- `direction`: objects and grasp points with the shortest distance along a defined direction `vector` in a given `pose_frame` are returned first.

- `distance_to_point`: objects and grasp points with the shortest or farthest (if `farthest_first` is true) distance from a `point` in a given `pose_frame` are returned first.

If no sorting strategy is set or default sorting is chosen in the Web GUI, sorting is done based on a combination of `preferred_orientation` and the minimal distance from the camera along the z axis of the preferred orientation of the TCP.

5.4.7.7 Detection of objects

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

- 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 5.4.7.8).

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

- A flag `object_plane_detection` determining whether the surface plane of the objects should be used for the detection instead of the calibrated base plane.
- An offset, in case the calibrated base plane should be used but the objects are not lying on this 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. It must not be set if `object_plane_detection` is true.
- 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. Details on collision checking are given below in *CollisionCheck* (Section 5.4.7.8).

In case the `object_plane_detection` flag is not true, 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.

When `object_plane_detection` is set to true, a base-plane calibration is not required and an existing base-plane calibration will be ignored. During detection, the scene is clustered into planar surfaces and template matching is performed on each plane whose tilt with respect to the camera's line of sight is less than 25° and whose size is large enough to contain the selected template. When a match is found, its position and orientation are refined using the image edges and the point cloud inside the template's outer contour. For this, it is required that the outer contour of the template is closed and that the object's surface is planar.

On the Web GUI the detection can be tested in the *Try Out* section of the SilhouetteMatch page. Different image streams can be selected to show intermediate results and the final matches.

> The **"Template"** image stream shows the template to be matched in green with the defined grasp points in green (see *Setting of grasp points*, Section 5.4.7.4). The template is warped to the size and tilt matching objects on the calibrated base plane or, in case `object_plane_detection` was used, the highest segmented plane, would have. The corresponding plane is shown in dark blue.
>
> The **"Intermediate Result"** image stream shows the edges of the left image that were used to search for matches in light blue. The chosen region of interest is shown as bold petrol rectangle. A shaded blue area on the left visualizes the region of the left camera image which does not overlap with the right image, and in which no objects can be detected. If

`object_plane_detection` was used, this image stream also shows the detected planar clusters in the scene. Clusters that were not used for matching, because they were too small or too tilted, are visualized with a stripe pattern.

The **"Intermediate Result Right"** image stream shows the edges of the right image that were used to search for matches in light blue. The chosen region of interest is shown as bold petrol rectangle. A shaded blue area on the right visualizes the region of the right camera image which does not overlap with the left image, and in which no objects can be detected.

The poses of the object origins in the chosen coordinate frame are returned as results in a list of `instances`. In case the calibrated base plane was used for the detection (`object_plane_detection` not set or false), the orientations of the detected objects are aligned with the normal of the base plane. Otherwise, the orientations of the detected objects are aligned with the normal of a plane fitted to the object points in the 3D point cloud.

If the chosen template also has grasp points attached, a list of `grasps` for all objects is returned in addition to the list of detected objects. The grasp poses are given in the desired coordinate frame and the grasps are sorted according to the selected sorting strategy (see *Setting the sorting strategies*, Section 5.4.7.6). There are references between the detected object instances and the grasps via their `uuids`.

In case the templates have a continuous rotational symmetry (e.g. cylindrical objects), all returned object poses will have the same orientation. Furthermore, all grasps symmetric to each grasp point on an object are checked for reachability and collisions, and only the best one according to the given sorting strategy is returned.

For objects with a discrete symmetry (e.g. prismatic objects), all collision-free symmetries of each grasp point which are reachable according to the given preferred TCP orientation are returned, ordered by the given sorting strategy.

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

5.4.7.8 Interaction with other modules

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

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

Camera and depth data

The SilhouetteMatch module makes internally use of the rectified images from the *Camera module* (`rc_camera`, Section 5.1). Thus, the exposure time should be set properly to achieve the optimal performance of the module.

For base-plane calibration in stereo mode, for load carrier detection, for automatic object plane detection and for collision checking with the point cloud, the disparity images from the *Stereo matching module* (`rc_stereomatching`, Section 5.2) are used.

For detecting objects with a calibrated base plane, without load carrier and without collision checking with the point cloud, the stereo-matching module should not be run in parallel to the SilhouetteMatch module, because the detection runtime increases.

For best results it is recommended to enable *smoothing* (Section 5.2.1.2) for *Stereo matching module*.

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* module (`rc_iocontrol`, Section 5.5.4), the projector should be used for the stereo-based base-plane calibration, for automatic object plane detection and for collision checking with the point cloud.

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 is recommended to connect the projector to GPIO Out 1 and set the stereo-camera module's acquisition mode to `SingleFrameOut1` (see *Stereo matching parameters*, Section 5.2.1), 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 5.5.4.1).

In either case, the *Auto Exposure Mode* `exp_auto_mode` should be set to `AdaptiveOut1` to optimize the exposure of both images.

Hand-eye calibration

In case the camera has been calibrated to a robot, the SilhouetteMatch module can automatically provide poses in the robot coordinate frame. For the SilhouetteMatch node's *Services* (Section 5.4.7.11), 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 module are in the camera frame.
2. **External frame** (`external`). All poses and plane coordinates provided to and by the module are in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) 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`, the definition of the preferred TCP orientation and the sorting direction or sorting point:

- 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 the sorting direction is defined in `external`, providing the robot pose is obligatory.
- If the distance-to-point sorting strategy is defined in `external`, providing the robot pose is obligatory.
- In all other cases, 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`, `detect_object` or `detect_object_extended` 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.

LoadCarrier

The SilhouetteMatch module uses the load carrier detection functionality provided by the *LoadCarrier* module (`rc_load_carrier`, Section 5.4.2), with the run-time parameters specified for this module. However, only one load carrier will be returned and used in case multiple matching load carriers could be found in the scene. In case multiple load carriers of the same type are visible, a region of interest should be set to ensure that always the same load carrier is used for the SilhouetteMatch module.

CollisionCheck

Collision checking can be easily enabled for grasp computation of the SilhouetteMatch module by passing a `collision_detection` argument to the `detect_object` service call. It contains the ID of the used gripper and optionally a pre-grasp offset. The gripper has to be defined in the GripperDB module (see *Setting a gripper*, Section 5.6.4.2) and details about collision checking are given in *Collision checking within other modules* (Section 5.5.2.2).

Alternatively, grasp points can be assigned individual gripper IDs, and collision checking can be enabled for all grasp points with gripper IDs by enabling the run-time parameter `check_collisions`.

In addition to collision checking between the gripper and the detected load carrier, collisions between the gripper and the calibrated base plane will be checked, if the run-time parameter `check_collisions_with_base_plane` is true. If the selected SilhouetteMatch template contains a collision model and the run-time parameter `check_collisions_with_matches` is true, also collisions between the gripper and all other detected objects (not limited to `max_number_of_detected_objects`) will be checked. The object on which the grasp point to be checked is located, is excluded from the collision check.

If the run-time parameter `check_collisions_with_point_cloud` is true, also collisions between the gripper and a watertight version of the point cloud are checked. If this feature is used with suctions grippers, it should be ensured that the TCP is defined to be outside the gripper geometry, or that the grasp points are defined above the object surface. Otherwise every grasp will result in a collision between the gripper and the point cloud.

If the run-time parameter `check_collisions_during_retraction` is true and a load carrier and a pre-grasp offset are given, each grasp point will be checked for collisions between the object in the gripper and the load carrier walls during retraction. This collision check is performed along the full linear trajectory from the grasp point back to the pre-grasp position.

If collision checking is enabled, only grasps which are collision free or could not be checked for collisions (e.g. because no gripper was given) will be returned. The visualization images on the *SilhouetteMatch* page of the Web GUI shows collision-free grasps in green, unchecked grasps in yellow and colliding grasp points in red. The detected objects which are considered in the collision check are also visualized with their edges in green.

The CollisionCheck module's run-time parameters affect the collision detection as described in *CollisionCheck Parameters* (Section 5.5.2.3).

5.4.7.9 Parameters

The SilhouetteMatch software module is called `rc_silhouettematch` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Modules* → *SilhouetteMatch*. The user can explore and configure the

`rc_silhouettematch` module's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.39: The `rc_silhouettematch` module's run-time parameters

Name	Type	Min	Max	Default	Description
`check_collisions`	bool	false	true	false	Whether to check for collisions when a gripper is defined for a grasp
`check_collisions_during_retraction`	bool	false	true	false	Whether to check for collisions between the object in the gripper and the load carrier during retraction
`check_collisions_with_base_plane`	bool	false	true	true	Whether to check for collisions between gripper and base plane
`check_collisions_with_matches`	bool	false	true	true	Whether to check for collisions between gripper and detected matches
`check_collisions_with_point_cloud`	bool	false	true	false	Whether to check for collisions between gripper and the point cloud
`edge_sensitivity`	float64	0.1	1.0	0.7	Sensitivity of the edge detector
`match_max_distance`	float64	0.1	10.0	3.0	Maximum allowed distance in pixels between the template and the detected edges in the image
`match_percentile`	float64	0.7	1.0	0.8	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
`max_object_overlap`	float64	0.0	1.0	0.05	Maximum fraction of object surface that is allowed to be overlapped by other segmented objects
`only_highest_priority_grasps`	bool	false	true	false	Whether to return only the highest priority level grasps
`point_cloud_enhancement`	string	-	-	Off	Type of enhancement of the point cloud using the base plane: [Off, ReplaceBright]
`quality`	string	-	-	High	Quality: [Low, Medium, High]

Description of run-time parameters

Each run-time parameter is represented by a row on the Web GUI's SilhouetteMatch page. 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 matching best to the given sorting strategy are returned.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/parameters/parameters?
↪max_number_of_detected_objects=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?max_number_of_detected_
↪objects=<value>
```

`quality` (*Quality*)

Object detection can be performed on images with different resolutions: `High` (full image resolution), `Medium` (half image resolution) and `Low` (quarter image resolution). The lower the resolution, the lower the detection time, but the fewer details of the objects are visible.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/parameters/parameters?
↪quality=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?quality=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/parameters/parameters?
↪match_max_distance=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?match_max_distance=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/parameters/parameters?
↪match_percentile=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?match_percentile=<value>
```

`edge_sensitivity` (*Edge Sensitivity*)

This parameter influences how many edges are detected in the left and right camera images. The higher this number, the more edges are found in the intensity images. 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. It must be ensured that the edges of the objects to be detected are detected in both, the left and the right camera images.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/parameters/parameters?
↪edge_sensitivity=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?edge_sensitivity=<value>
```

`only_highest_priority_grasps` (*Only Highest Priority Grasps*)

If set to true, only grasps with the highest priority will be returned. If collision checking is enabled, only the collision-free grasps among the group of grasps with the highest priority are returned. This can save computation time and reduce the number of grasps to be parsed on the application side.

Without collision checking, only grasps of highest priority are returned.

API version 2

```
PUT http://<host>/api/v2/pipelines/<0,1,2,3>/nodes/rc_silhouettematch/parameters?only_
↪highest_priority_grasps=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?only_highest_priority_
↪grasps=<value>
```

`check_collisions` (*Check Collisions*)

If this parameter is enabled, collision checking will be performed for all grasps which have a gripper ID assigned, even when no default gripper is given in the `detect_object` service call. If a load carrier is used, the collision check will always be performed between the gripper and the load carrier. Collision checking with the point cloud and other matches is only performed when the corresponding runtime parameters are enabled.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/<0,1,2,3>/nodes/rc_silhouettematch/parameters?check_
↪collisions=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?check_collisions=<value>
```

`check_collisions_with_base_plane` (*Check Collisions with Base Plane*)

This parameter is only used when collision checking is enabled by passing a gripper to the `detect_object` service call or by enabling the `check_collisions` runtime parameter. If `check_collisions_with_base_plane` is set to true, all grasp points will be checked for collisions between the gripper and the calibrated base plane, and only grasp points at which the gripper would not collide with the base plane will be returned.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/parameters/parameters?
↪check_collisions_with_base_plane=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?check_collisions_with_
↪base_plane=<value>
```

`check_collisions_with_matches` (*Check Collisions with Matches*)

This parameter is only used when collision checking is enabled by passing a gripper to the `detect_object` service call or by enabling the `check_collisions` runtime parameter. If `check_collisions_with_matches` is set to true, all grasp points will be checked for collisions between the gripper and all other detected objects (not limited to `max_number_of_detected_objects`), and only grasp points at which the gripper would not collide with any other detected object will be returned.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/parameters/parameters?
↪check_collisions_with_matches=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?check_collisions_with_
↪matches=<value>
```

`check_collisions_with_point_cloud` (*Check Collisions with Point Cloud*)

This parameter is only used when collision checking is enabled by passing a gripper to the `detect_object` service call or by enabling the `check_collisions` runtime parameter. If `check_collisions_with_point_cloud` set to true, all grasp points will be checked for collisions between the gripper and a watertight version of the point cloud, and only grasp points at which the gripper would not collide with this point cloud will be returned.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/<0,1,2,3>/nodes/rc_silhouettematch/parameters?check_
↪collisions_with_point_cloud=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?check_collisions_with_
↪point_cloud=<value>
```

`point_cloud_enhancement` (*Enhance with Base Plane*)

This parameter is only considered when `check_collisions_with_point_cloud` is true and the object detection was triggered without `object_plane_detection`. By default, `point_cloud_enhancement` is set to `Off` (*Off*). If `point_cloud_enhancement` is set to `ReplaceBright` (*Replace Bright Image Pixels*), the calibrated base plane will be used to enhance the point cloud that is used for collision checking. For this, the depth values of all bright image pixels inside the image or, if set, the 2D region of interest will be set to the depth of the calibrated base plane. This parameter should be used when dark objects are placed on an untextured bright background, e.g. on a light table.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/<0,1,2,3>/nodes/rc_silhouettematch/parameters?point_
↪cloud_enhancement=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?point_cloud_enhancement=
↪<value>
```

`check_collisions_during_retraction` (*Check Collisions during Retraction*)

This parameter is only used when collision checking is enabled by passing a gripper to the `detect_object` service call or by enabling the `check_collisions` runtime parameter. When `check_collisions_during_retraction` is enabled and a load carrier and a pre-grasp offset are given, each grasp point will be checked for collisions between the object in the gripper and the load carrier walls during retraction. This collision checking is performed along the full linear trajectory from the grasp point back to the pre-grasp position. Only collision-free grasp points will be returned.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/<0,1,2,3>/nodes/rc_silhouettematch/parameters?check_
↪collisions_during_retraction=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/parameters?check_collisions_during_
↪retraction=<value>
```

5.4.7.10 Status values

This module reports the following status values:

Table 5.40: The `rc_silhouettematch` module's status values

Name	Description
`data_acquisition_time`	Time in seconds required by the last active service to acquire images
`last_return_code_message`	Return code message of the last detection
`last_return_code_value`	Return code value of the last detection
`last_timestamp_processed`	The timestamp of the last processed dataset
`load_carrier_detection_time`	Processing time of the last load carrier detection in seconds
`processing_time`	Processing time of the last detection (including load carrier detection) in seconds

5.4.7.11 Services

The user can explore and call the `rc_silhouettematch` module's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1).

The SilhouetteMatch module offers the following services.

`detect_object`

Triggers an object detection as described in *Detection of objects* (Section 5.4.7.7) and returns the pose of all found object instances.

Details

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

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/detect_object
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/detect_object
```

Request

Required arguments:

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

`pose_frame`: see *Hand-eye calibration* (Section 5.4.7.8).

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.7.8).

Optional arguments:

`object_plane_detection`: false if the objects are placed on a calibrated base plane, true if the objects' surfaces are planar and the base plane is unknown or the objects are located on multiple different planes, e.g. stacks.

`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 5.5.2.2).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "collision_detection": {
      "gripper_id": "string",
      "pre_grasp_offset": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "load_carrier_id": "string",
    "object_plane_detection": "bool",
    "object_segmentation_model": "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"
      }
    }
  }
}
```


Response

The maximum number of returned instances can be controlled with the `max_number_of_detected_objects` parameter.

`object_id`: ID of the detected template.

`instances`: list of detected object instances, ordered according to the chosen sorting strategy.

`grasps`: list of grasps on the detected objects, ordered according to the chosen sorting strategy. The `instance_uuid` gives the reference to the detected object in `instances` this grasp belongs to. The list of returned grasps will be trimmed to the 100 best grasps if more reachable grasps are found. Each grasp contains a flag `collision_checked` and a `gripper_id` (see *Collision checking within other modules*, Section 5.5.2.2).

`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.

The definition for the response with corresponding datatypes is:


```
{
  "name": "detect_object",
  "response": {
    "grasps": [
      {
```


(continues on next page)

(continued from previous page)


```
      "collision_checked": "bool",
      "gripper_id": "string",
      "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",
      "priority": "int8",
      "stroke_per_finger_approach_mm": "float64",
      "stroke_per_finger_grasp_mm": "float64",
      "tcp_id": "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": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "pose_frame": "string",
      "timestamp": {
        "nsec": "int32",
        "sec": "int32"
      },
      "uuid": "string"
    }
  ],
  "load_carriers": [
    {
      "height_open_side": "float64",
      "id": "string",
```


(continues on next page)

(continued from previous page)


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


detect_object_extended

Triggers an object detection in the same way as `detect_object`, but returns the instance information for each grasp directly instead of as a separate list. This allows for easier parsing, e.g. when the instance pose for each grasp is required for placing the object.

Details

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

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/detect_object_
↪extended
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/detect_object_extended
```

Request

See `detect_object` service.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "collision_detection": {
      "gripper_id": "string",
      "pre_grasp_offset": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "load_carrier_id": "string",
    "object_plane_detection": "bool",
    "object_segmentation_model": "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"
      }
    }
  }
}
```

Response

The maximum number of returned instances can be controlled with the `max_number_of_detected_objects` parameter.

`object_id`: ID of the detected template.

`grasps`: list of grasps on the detected objects, ordered according to the chosen sorting strategy. Each grasp contains an `instance` field with information about the detected object, e.g. its `pose`. The list of returned grasps will be trimmed to the 100 best grasps if more reachable grasps are found. Each grasp contains a flag `collision_checked` and a `gripper_id` (see *Collision checking within other modules*, Section 5.5.2.2).

`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.

The definition for the response with corresponding datatypes is:


```
{
  "name": "detect_object_extended",
  "response": {
    "grasps": [
      {
        "collision_checked": "bool",
        "gripper_id": "string",
        "id": "string",
        "instance": {
          "object_id": "string",
          "pose": {
            "orientation": {
              "w": "float64",
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "position": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            }
          },
          "pose_frame": "string",
          "uuid": "string"
        },
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "pose_frame": "string",
        "priority": "int8",
        "stroke_per_finger_approach_mm": "float64",
        "stroke_per_finger_grasp_mm": "float64",
        "tcp_id": "string",
        "timestamp": {
          "nsec": "int32",
          "sec": "int32"
        },
        "uuid": "string"
      }
    ],
    "load_carriers": [
      {
        "height_open_side": "float64",
        "id": "string",
        "inner_dimensions": {
          "x": "float64",
```


(continues on next page)

(continued from previous page)


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


`calibrate_base_plane`

Triggers the calibration of the base plane, as described in *Base-plane calibration* (Section 5.4.7.2).

Details

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

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

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/calibrate_base_
↪plane
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/calibrate_base_plane
```

Request

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 5.4.7.8).

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 5.4.7.8).

`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`.

The definition for the request arguments with corresponding datatypes is:

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

(continues on next page)

(continued from previous page)

```
      "plane_preference": "string"
    }
  }
}
```

Response

`plane`: calibrated base plane.

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

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

The definition for the response with corresponding datatypes is:

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

`get_base_plane_calibration`

Returns the configured base-plane calibration.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/get_base_plane_
↪calibration
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/get_base_plane_calibration
```

Request

Required arguments:

`pose_frame`: see *Hand-eye calibration* (Section 5.4.7.8).

Potentially required arguments:

`robot_pose`: see *Hand-eye calibration* (Section 5.4.7.8).

The definition for the request arguments with corresponding datatypes is:


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


Response

The definition for the response with corresponding datatypes is:


```
{
  "name": "get_base_plane_calibration",
  "response": {
    "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.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/delete_base_
↪plane_calibration
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/delete_base_plane_
↪calibration
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "delete_base_plane_calibration",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

set_preferred_orientation

Persistently stores the preferred orientation of the TCP to compute the reachability of the grasps, which is used for filtering and, optionally, sorting the grasps returned by the `detect_object` and `detect_object_extended` service (see *Setting the preferred orientation of the TCP*, Section 5.4.7.5).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/set_preferred_
↪orientation
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/set_preferred_orientation
```

Request

The definition for the request arguments with corresponding datatypes is:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_preferred_orientation",
  "response": {
    "return_code": {
      "message": "string",
```

(continues on next page)

(continued from previous page)

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

get_preferred_orientation

Returns the preferred orientation of the TCP to compute the reachability of the grasps, which is used for filtering and, optionally, sorting the grasps returned by the `detect_object` and `detect_object_extended` service (see *Setting the preferred orientation of the TCP*, Section 5.4.7.5).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/get_preferred_
↪orientation
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/get_preferred_orientation
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

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

set_sorting_strategies

Persistently stores the sorting strategy for sorting the grasps and detected objects returned by the `detect_object` and `detect_object_extended` service (see *Detection of objects*, Section 5.4.7.7).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/set_sorting_
↪strategies
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/set_sorting_strategies
```

Request

Only one strategy may have a `weight` greater than 0. If all `weight` values are set to 0, the module will use the default sorting strategy.

If the weight for `direction` is set, the `vector` must contain the direction vector and `pose_frame` must be either `camera` or `external`.

If the weight for `distance_to_point` is set, `point` must contain the sorting point and `pose_frame` must be either `camera` or `external`.

If the weight for `preferred_orientation` is set, the `axis` can be set to `x`, `y` or `z` to consider only rotational differences between the respective axes. If `axis` is empty, the full orientation difference will be used for sorting.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "direction": {
      "pose_frame": "string",
      "vector": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "weight": "float64"
    },
    "distance_to_point": {
      "farthest_first": "bool",
      "point": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose_frame": "string",
      "weight": "float64"
    },
    "preferred_orientation": {
      "axis": "string",
      "weight": "float64"
    }
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_sorting_strategies",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

get_sorting_strategies

Returns the sorting strategy for sorting the grasps and detected objects returned by the `detect_object` and `detect_object_extended` service (see *Detection of objects*, Section 5.4.7.7).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/get_sorting_
↪strategies
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/get_sorting_strategies
```

Request

This service has no arguments.

Response

All `weight` values are 0 when the module uses the default sorting strategy.

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_sorting_strategies",
  "response": {
    "direction": {
      "pose_frame": "string",
      "vector": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "weight": "float64"
    },
    "distance_to_point": {
      "farthest_first": "bool",
      "point": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "pose_frame": "string",
      "weight": "float64"
    },
    "preferred_orientation": {
      "axis": "string",
      "weight": "float64"
    },
    "return_code": {
      "message": "string",
      "value": "int16"
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

reset_defaults

Resets all parameters of the module to its default values, as listed in above table. Also resets preferred orientation and sorting strategies. The reset does not apply to templates and base-plane calibration.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.4.7.12 Internal services

The following services for configuring grasps can change in future without notice. Setting, retrieving and deleting grasps is recommend to be done via the Web GUI.

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.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/set_grasp
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/set_grasp
```

Request

Details for the definition of the `grasp` type are given in *Setting of grasp points* (Section 5.4.7.4).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "grasp": {
      "gripper_id": "string",
      "id": "string",
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "priority": "int8",
      "replication": {
        "max_x_deg": "float64",
        "min_x_deg": "float64",
        "origin": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "step_x_deg": "float64"
      },
      "stroke_per_finger_approach_mm": "float64",
      "stroke_per_finger_grasp_mm": "float64",
      "tcp_id": "string",
      "template_id": "string"
    }
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_grasp",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

set_all_grasps

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

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/set_all_grasps
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/set_all_grasps
```

Request

Details for the definition of the `grasp` type are given in *Setting of grasp points* (Section 5.4.7.4).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "grasps": [
      {
        "gripper_id": "string",
        "id": "string",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "priority": "int8",
        "replication": {
          "max_x_deg": "float64",
          "min_x_deg": "float64",
          "origin": {
            "orientation": {
              "w": "float64",
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "position": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            }
```

(continues on next page)

(continued from previous page)

```
          },
          "step_x_deg": "float64"
        },
        "stroke_per_finger_approach_mm": "float64",
        "stroke_per_finger_grasp_mm": "float64",
        "tcp_id": "string",
        "template_id": "string"
      }
    ],
    "template_id": "string"
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_all_grasps",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

get_grasps

Returns all configured grasps which have the requested `grasp_ids` and belong to the requested `template_ids`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/get_grasps
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/get_grasps
```

Request

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:

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

Response

The definition for the response with corresponding datatypes is:


```
{
  "name": "get_grasps",
  "response": {
    "grasps": [
      {
        "gripper_id": "string",
        "id": "string",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "priority": "int8",
        "replication": {
          "max_x_deg": "float64",
          "min_x_deg": "float64",
          "origin": {
            "orientation": {
              "w": "float64",
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "position": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            }
          },
          "step_x_deg": "float64"
        },
        "stroke_per_finger_approach_mm": "float64",
        "stroke_per_finger_grasp_mm": "float64",
        "tcp_id": "string",
        "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`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/delete_grasps
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/delete_grasps
```

Request

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:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "delete_grasps",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

get_symmetric_grasps

Returns all grasps that are symmetric to the given grasp.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_silhouettematch/services/get_symmetric_
↪grasps
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_silhouettematch/services/get_symmetric_grasps
```

Request

Details for the definition of the `grasp` type are given in *Setting of grasp points* (Section 5.4.7.4).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "grasp": {
      "pose": {
        "orientation": {
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "replication": {
        "max_x_deg": "float64",
        "min_x_deg": "float64",
        "origin": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "step_x_deg": "float64"
      },
      "template_id": "string"
    }
  }
}
```


Response

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.

Details for the definition of the `grasp` type are given in *Setting of grasp points* (Section 5.4.7.4).

The definition for the response with corresponding datatypes is:


```
{
  "name": "get_symmetric_grasps",
  "response": {
    "grasps": [
      {
        "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"
          }
        },
        "replication": {
          "max_x_deg": "float64",
          "min_x_deg": "float64",
          "origin": {
            "orientation": {
              "w": "float64",
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "position": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            }
          },
          "step_x_deg": "float64"
        },
        "template_id": "string"
      }
    ],
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```


5.4.7.13 Return codes

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 5.41: Return codes of the SilhouetteMatch module 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 allowed.
-7	Data could not be read or written to persistent storage.
-8	Module 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.
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.
109	The plane for object detection does not fit to the load carrier, e.g. objects are below the load carrier floor.
111	The detection result's pose could not be refined with the point cloud because the template's outer contour is not closed.
113	No gripper was found for collision checking.
114	Collision checking during retraction was skipped, e.g. because no load carrier or no pre-grasp offset were given.
151	The object template has a continuous symmetry.
999	Additional hints for application development

5.4.7.14 Template API

For template upload, download, listing and removal, special REST-API endpoints are provided. Templates can also be uploaded, downloaded and removed via the Web GUI. The templates include the grasp points, if grasp points have been configured. Up to 50 templates can be stored persistently on the *rc_visard*.

`GET /templates/rc_silhouettematch`

Get list of all rc_silhouettematch templates.

Template request

```
GET /api/v2/templates/rc_silhouettematch HTTP/1.1
```

Template response

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

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

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Template* (Section 6.2.4)

`GET /templates/rc_silhouettematch/{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/v2/templates/rc_silhouettematch/<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/ubjson application/octet-stream

Status Codes

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

Referenced Data Models

- *Template* (Section 6.2.4)

`PUT /templates/rc_silhouettematch/{id}`

Create or update a rc_silhouettematch template.

Template request

```
PUT /api/v2/templates/rc_silhouettematch/<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 or dxf file *(required)*
- **object_height** – object height in meters, required when uploading dxf *(optional)*
- **units** – Units for dxf file if not included in dxf (one of `mm`, `cm`, `m`, `in`, `ft`) *(optional)*

Request Headers

- Accept – multipart/form-data application/json

Response Headers

- Content-Type – application/json application/ubjson

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 module.
- 404 Not Found – node or template not found
- 413 Request Entity Too Large – Template too large

Referenced Data Models

- *Template* (Section 6.2.4)

DELETE /templates/rc_silhouettematch/{id}

Remove a rc_silhouettematch template.

Template request

```
DELETE /api/v2/templates/rc_silhouettematch/<id> HTTP/1.1
Accept: application/json application/ubjson
```

Parameters

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

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

5.5 Configuration modules

The *rc_visard* provides several configuration modules which enable the user to configure the *rc_visard* for specific applications.

The configuration modules are:

- ***Hand-eye calibration*** **(`rc_hand_eye_calibration`, Section 5.5.1)** enables the user to calibrate the camera with respect to a robot, either via the Web GUI or the REST-API.
- ***CollisionCheck*** **(`rc_collision_check`, Section 5.5.2)** provides an easy way to check if a gripper is in collision.
- ***Camera calibration*** **(`rc_stereocalib`, Section 5.5.3)** enables the user to check and perform camera calibration via the *WEB GUI* (Section 6.1).
- ***IO and Projector Control*** **(`rc_iocontrol`, Section 5.5.4)** provides control over the camera's general purpose inputs and outputs with special modes for controlling an external random dot projector.

5.5.1 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* module. It is a base module which is available on every *rc_visard*.

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 module.

The *Calibration routine* (Section 5.5.1.3) itself is an easy-to-use multi-step procedure using a calibration grid which can be obtained from Basler.

5.5.1.1 Calibration interfaces

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

1. All services and parameters of this module required to conduct the hand-eye calibration **programmatically** are exposed by the *rc_visard*'s *REST-API interface* (Section 6.2). The respective node name of this module is `rc_hand_eye_calibration` and the respective service calls are documented *Services* (Section 5.5.1.5).

   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 module.

2. For use cases where robot poses cannot be passed programmatically to the *rc_visard*'s hand-eye calibration module, the *Web GUI*'s *Hand-Eye Calibration* page under *Configuration*

   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 hand-held device.

5.5.1.2 Camera mounting

As illustrated in Fig. 5.20 and Fig. 5.22, 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 5.5.1.3) 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. 5.20.

> **Warning:** It is extremely important that the calibration grid does not move during step 2 of the *Calibration routine* (Section 5.5.1.3). 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 5.5.1.3) is a pose $\mathbf{T}^{\text{robot}}_{\text{camera}}$ describing the (previously unknown) relative positional and rotational transformation from the *camera* frame into the user-selected *robot* frame such that

$$\mathbf{p}_{\text{robot}} = \mathbf{R}^{\text{robot}}_{\text{camera}} \cdot \mathbf{p}_{\text{camera}} + \mathbf{t}^{\text{robot}}_{\text{camera}} \,, \tag{5.3}$$

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 $\mathbf{R}^{\text{robot}}_{\text{camera}}$ as well as $\mathbf{t}^{\text{robot}}_{\text{camera}}$ are the corresponding 3×3 rotation matrix and 3×1 translation vector of the pose $\mathbf{T}^{\text{robot}}_{\text{camera}}$, respectively. In practice, 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 11.1).

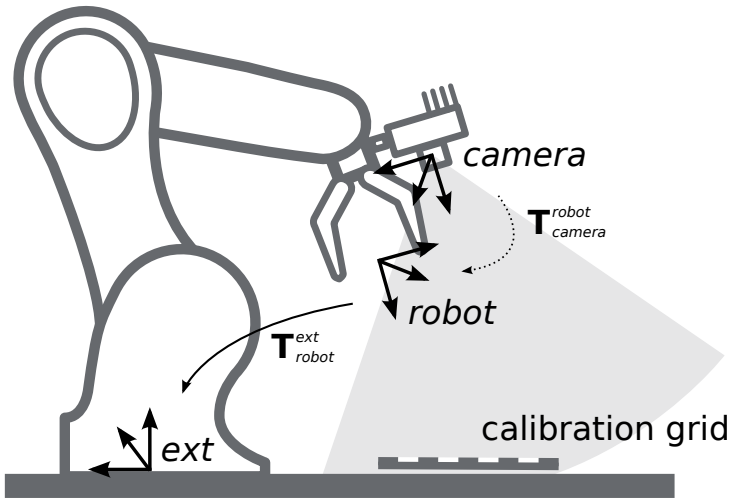


Fig. 5.20: Important frames and transformations for calibrating a camera that is mounted on a general robot. The camera is mounted with a fixed relative position to a user-defined *robot* frame (e.g., flange or TCP). It is important that the pose $\mathbf{T}^{\text{ext}}_{\text{robot}}$ of this *robot* 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 $\mathbf{T}^{\text{robot}}_{\text{camera}}$, 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. 5.21 illustrates the situation.

For the *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).

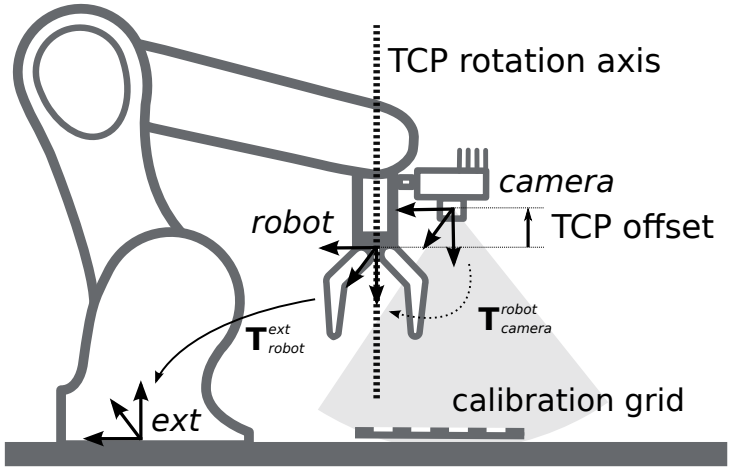


Fig. 5.21: In case of a 4DOF robot, the TCP rotation axis and the offset from the TCP to the camera coordinate system along the TCP rotation axis must be provided. In the illustrated case, this offset is negative.

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. 5.22 and Fig. 5.23.

Note: The hand-eye calibration module 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. 5.23.

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 5.5.1.3).

In this use case, the result of the calibration (step 3 of the *Calibration routine*, Section 5.5.1.3) is the pose $\mathbf{T}^{\text{ext}}_{\text{camera}}$ describing the (previously unknown) relative positional and rotational transformation between the *camera* frame and the user-selected external reference frame *ext* such that

$$\mathbf{p}_{\text{ext}} = \mathbf{R}^{\text{ext}}_{\text{camera}} \cdot \mathbf{p}_{\text{camera}} + \mathbf{t}^{\text{ext}}_{\text{camera}} \,, \tag{5.4}$$

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 practice, 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 11.1).

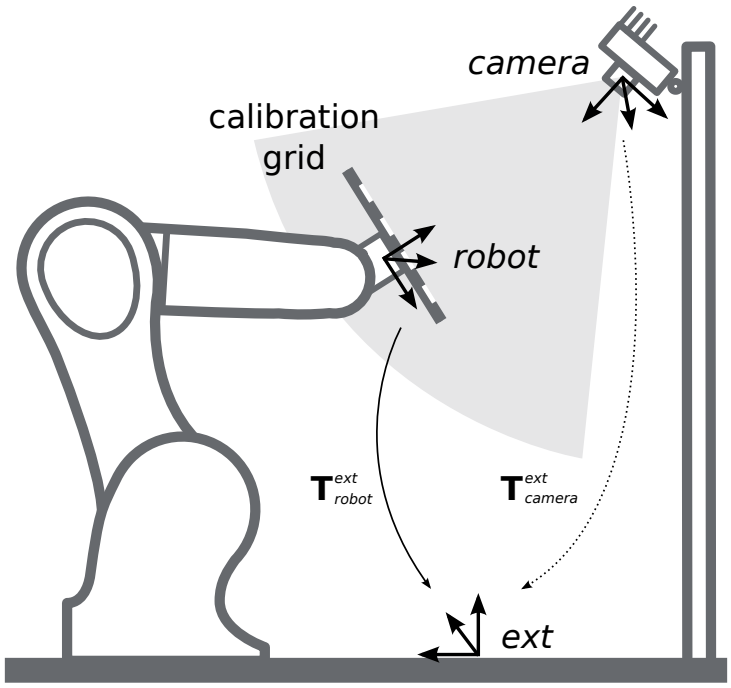


Fig. 5.22: Important frames and transformations for calibrating a statically mounted camera: The latter is mounted with a fixed position relative to a user-defined external reference frame *ext* (e.g., the world coordinate frame or the robot's mounting point). It is important that the pose $\mathbf{T}^{ext}_{robot}$ 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 $\mathbf{T}^{ext}_{camera}$, i.e., the pose of the *camera* frame in the user-defined external reference frame *ext*.

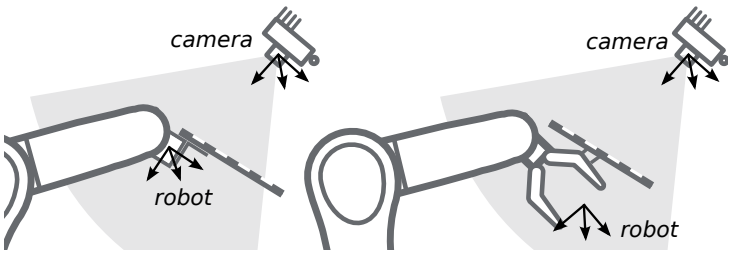


Fig. 5.23: Alternate mounting options for attaching the calibration grid to the robot

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. 5.24 illustrates the situation.

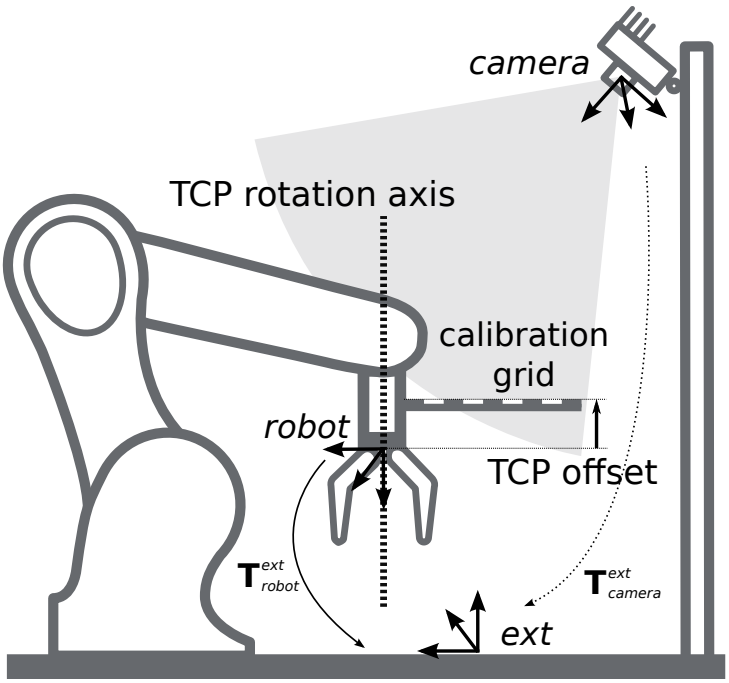


Fig. 5.24: In case of a 4DOF robot, the TCP rotation axis and the offset from the TCP to the visible surface of the grid along the TCP rotation axis must be provided. In the illustrated case, this offset is negative.

5.5.1.3 Calibration routine

The hand-eye calibration can be performed manually using the *Web GUI* (Section 6.1) or programmatically via the *REST-API interface* (Section 6.2). The general calibration routine will be described by following the steps of the hand-eye calibration wizard provided on the Web GUI. This wizard can be found in the *rc_visard*'s Web GUI under *Configuration* → *Hand-Eye Calibration*. References to the corresponding REST-API calls are provided at the appropriate places.

Step 1: Hand-Eye Calibration Status

The starting page of the hand-eye calibration wizard shows the current status of the hand-eye calibration. If a hand-eye calibration is saved on the *rc_visard*, the calibration transformation is displayed here (see Fig. 5.25).

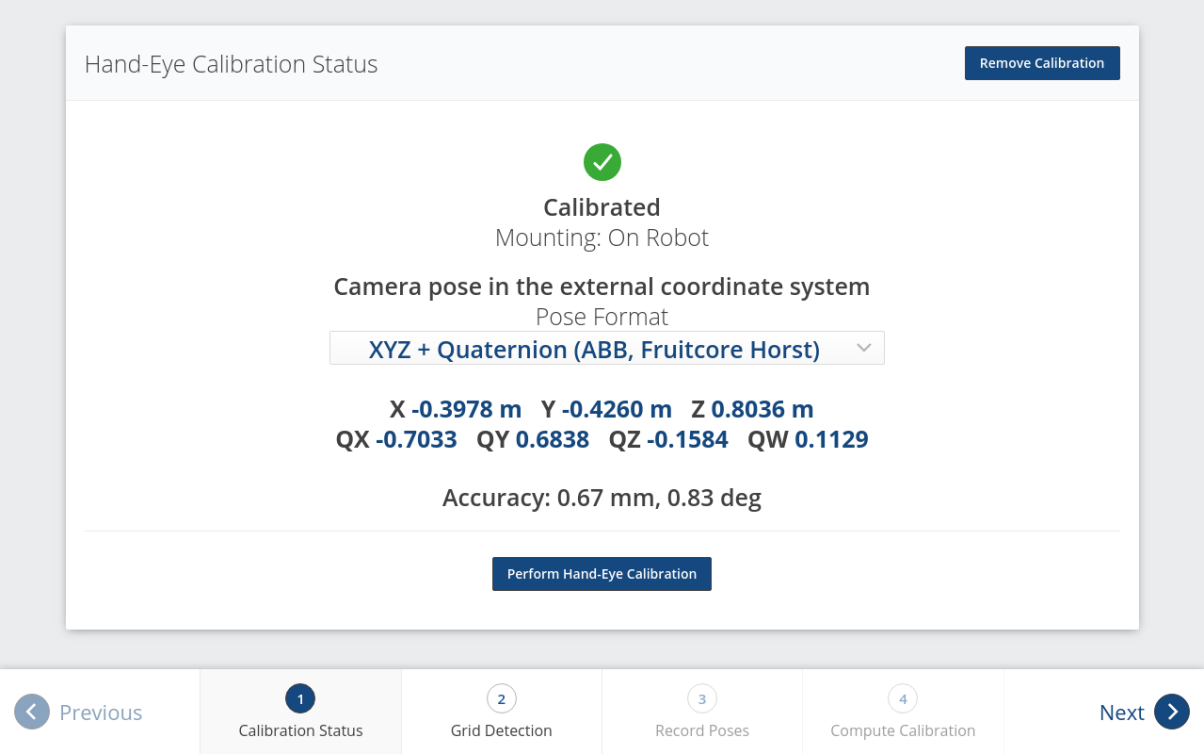


Fig. 5.25: Current status of the hand-eye calibration in case a hand-eye calibration is saved

To query the hand-eye calibration status programmatically, the module's REST-API offers the `get_calibration` service call (see *Services*, Section 5.5.1.5). An existing hand-eye calibration can be removed by pressing *Remove Calibration* or using `remove_calibration` in the REST-API (see *Services*, Section 5.5.1.5).

To start a new hand-eye calibration, click on *Perform Hand-Eye Calibration* or *Next*.

Step 2: Checking Grid Detection

To achieve good calibration results, the images should be well exposed so that the calibration grid can be detected accurately and reliably. In this step, the grid detection can be checked and the camera settings can be adjusted if necessary. In case parts of the calibration grid are overexposed, the respective squares of the calibration grid will be highlighted in red. A successful grid detection is visualized by green check marks on every square of the calibration grid and a thick green border around the grid as shown in Fig. 5.26.

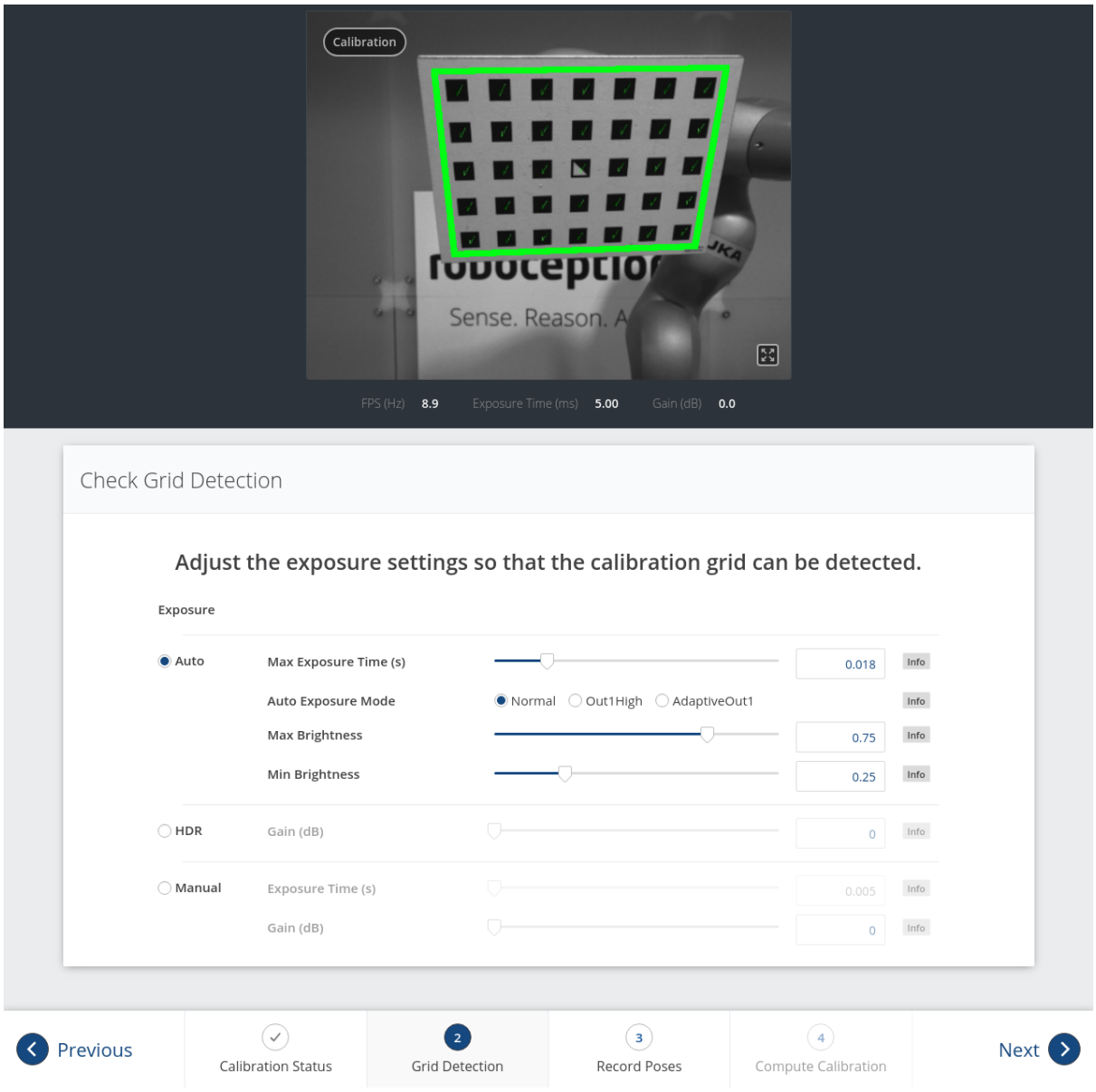


Fig. 5.26: Checking the calibration grid detection

Step 3: Record Poses

In this step, the user records images of the calibration grid at several different robot poses. These poses must each ensure that the calibration grid is completely visible in the left camera image. Furthermore, the robot poses need to be selected properly to achieve a variety of different perspectives for the camera to perceive the calibration grid. Fig. 5.27 shows a schematic recommendation of four different grid positions which should be recorded from a close and a far point of view, resulting in eight images for the calibration.

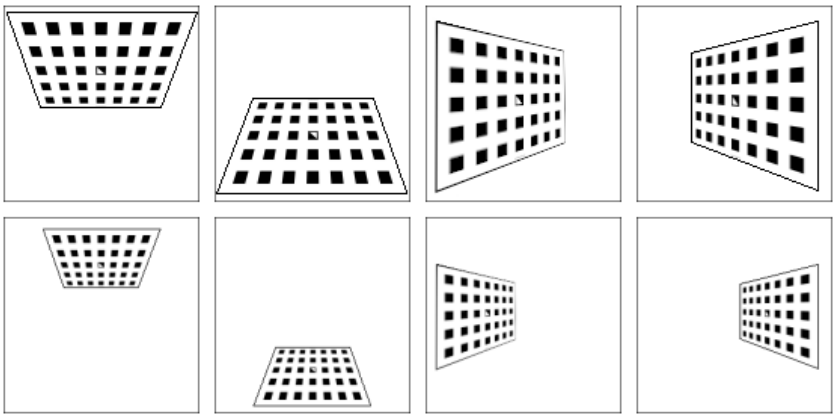


Fig. 5.27: Recommended views on the calibration grid during the calibration procedure. In case of a 4DOF robot, other views have to be chosen, which should be as different as possible.

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 pose only slightly before recording a new calibration pose, may lead to inaccurate estimation of the desired calibration transformation.

After the robot reaches each calibration pose, the corresponding pose $\mathbf{T}^{\text{ext}}_{\text{robot}}$ of the user-defined *robot* frame in the user-defined external reference frame *ext* needs to be reported to the hand-eye calibration module. For this purpose, the module 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).

In the Web GUI, the user can choose between many different pose formats for providing the calibration poses (see *Pose formats*, Section 11.1). When calibrating using the REST-API, the poses are always given in *XYZ+quaternion*. The Web GUI offers eight slots (*Close View 1*, *Close View 2*, etc.) for the user to fill manually with robot poses. Next to each slot, a figure suggests a respective dedicated viewpoint on the grid. For each slot, the robot should be operated to achieve the suggested view.

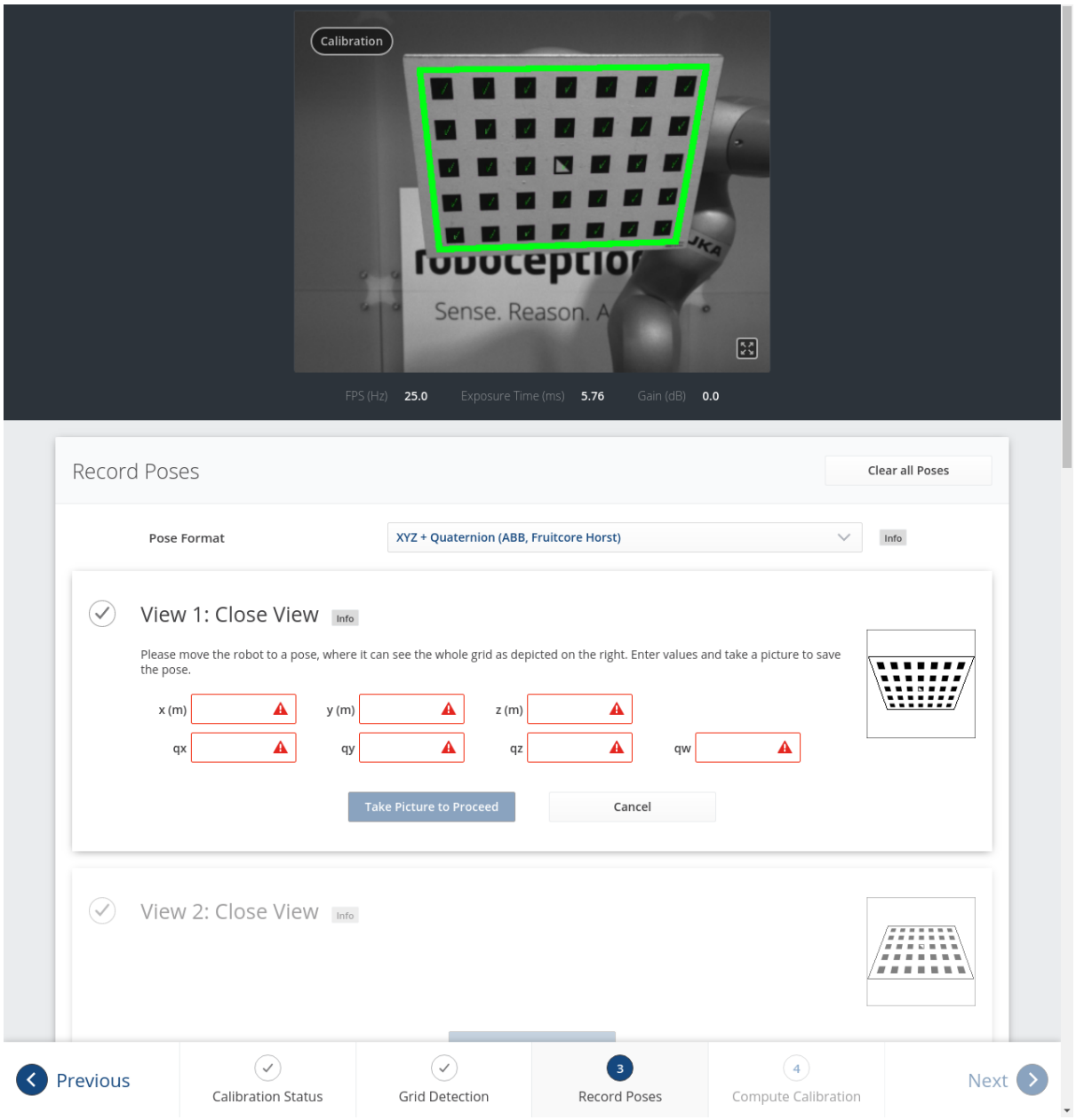


Fig. 5.28: Filling the first slot in the hand-eye calibration process for a statically mounted camera

To record a calibration pose, click on *Set Pose* for the respective slot and enter the *robot* frame's pose into the respective text fields. The pose is then stored with the corresponding camera image by clicking the *Take Picture to Proceed* button. This will save the calibration pose in the respective slot.

To transmit the poses programmatically, the module's REST-API offers the `set_pose` service call (see *Services*, Section 5.5.1.5).

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.

The Web GUI displays the currently saved poses (only with slot numbers from 0 to 7) with their camera

images and also allows to delete them by clicking *Delete Pose* to remove a single pose, or clicking *Clear all Poses* to remove all poses. In the REST-API the currently stored poses can be retrieved via `get_poses` and removed via `delete_poses` for single poses or `reset_calibration` for removing all poses (see *Services*, Section 5.5.1.5).

When at least four poses are set, the user can continue to the computation of the calibration result by pressing *Next*.

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

Step 4: Compute Calibration

Before computing the calibration result, the user has to provide the correct calibration parameters. These include the exact calibration grid dimensions and the sensor mounting type. 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. For the REST-API, the respective parameters are listed in *Parameters* (Section 5.5.1.4).

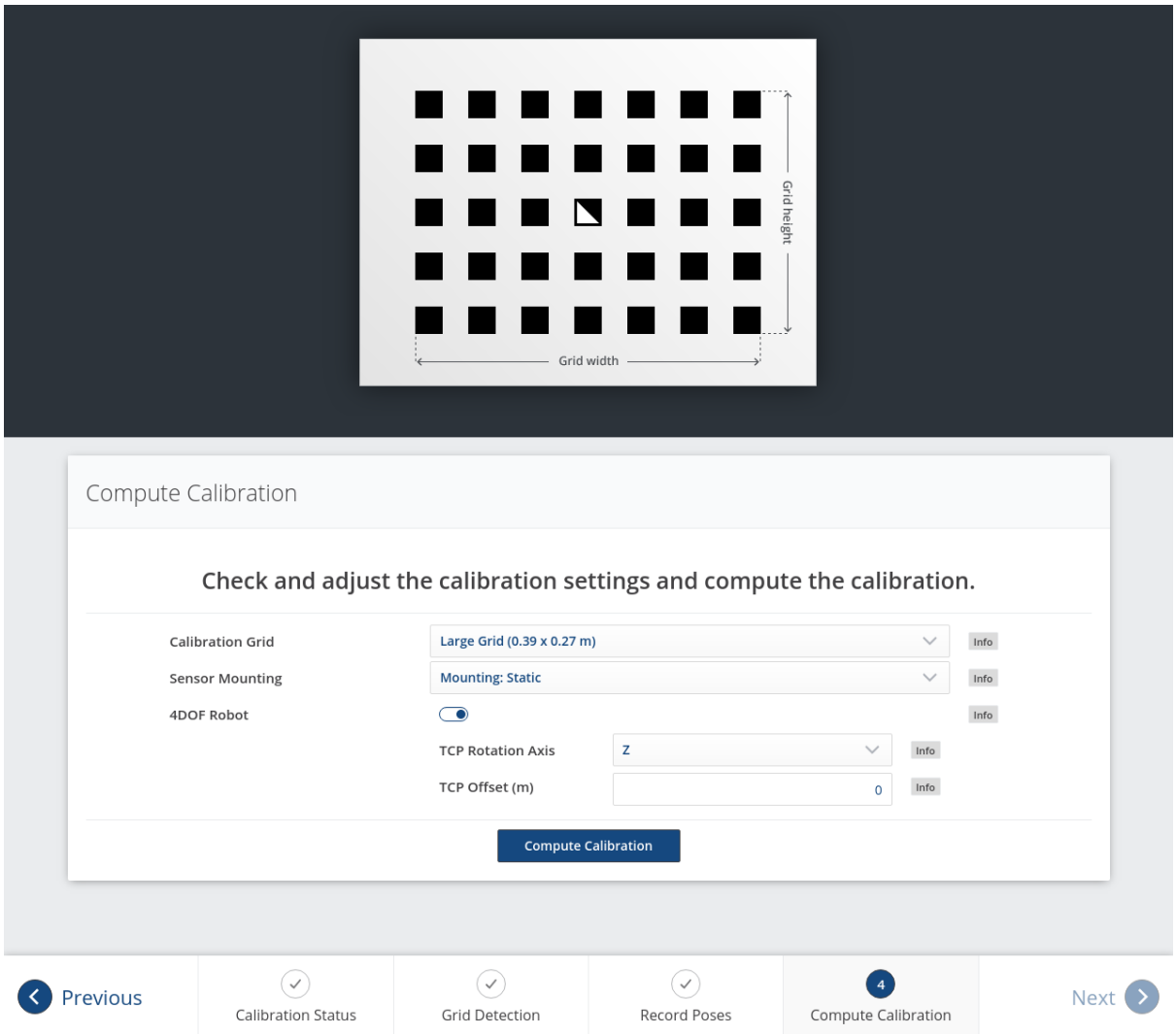


Fig. 5.29: Defining hand-eye calibration parameters and computing the calibration result via the *rc_visard*'s Web GUI

When the parameters are correct, the desired calibration transformation can be computed from the collected poses and camera images by clicking *Compute Calibration*. The REST-API offers this functionality via the `calibrate` service call (see *Services*, Section 5.5.1.5).

Depending on the way the camera is mounted, the calibration result contains 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 5.5.1.2).

To enable users to judge the quality of the resulting calibration transformation, the translational and rotational calibration errors are reported, which are computed from the variance of the calibration result.

If the calibration error is not acceptable, the user can change the calibration parameters and recompute the result, or return to step 3 of the calibration procedure and add more poses or update poses.

To save the calibration result, press *Save Calibration* or use the REST-API `save_calibration` service call (see *Services*, Section 5.5.1.5).

5.5.1.4 Parameters

The hand-eye calibration module is called `rc_hand_eye_calibration` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Configuration* → *Hand-Eye Calibration*. The user can change the calibration parameters there or use the *REST-API interface* (Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.42: The `rc_hand_eye_calibration` module'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
`tag_ids`	string	-	-	-	Optional, comma separated list of AprilTag IDs that will be calibrated too
`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` (*Width*)

Width of the calibration grid in meters. The width should be given with a very high accuracy, preferably with sub-millimeter accuracy.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/parameters?
↪grid_width=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/parameters?grid_width=<value>
```

`grid_height` (*Height*)

Height of the calibration grid in meters. The height should be given with a very high accuracy, preferably with sub-millimeter accuracy.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/parameters?
↪grid_height=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/parameters?grid_height=<value>
```

`robot_mounted` (*Sensor 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/parameters?
↪robot_mounted=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/parameters?robot_mounted=<value>
```

`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).

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/parameters?
↪tcp_offset=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/parameters?tcp_offset=<value>
```

`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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/parameters?
↪tcp_rotation_axis=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/parameters?tcp_rotation_axis=
↪<value>
```

5.5.1.5 Services

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

`get_calibration`

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

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/get_
↪calibration
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/get_calibration
```

Request

This service has no arguments.

Response

The field `error` gives the calibration error in pixels which is computed from the translational error `translation_error_meter` and the rotational error `rotation_error_degree`. This value is only given for compatibility with older versions. The translational and rotational errors should be preferred.

Table 5.43: Return codes of the `get_calibration` service call

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

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_calibration",
  "response": {
    "error": "float64",
    "message": "string",
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "robot_mounted": "bool",
    "rotation_error_degree": "float64",
    "status": "int32",
    "success": "bool",
    "tags": [
      {
        "id": "string",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "size": "float64"
      }
    ],
    "translation_error_meter": "float64"
  }
}
```


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 call will also delete all the stored calibration poses and corresponding camera images in the `slots`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/remove_
↪calibration
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/remove_calibration
```

Request

This service has no arguments.

Response

Table 5.44: 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

The definition for the response with corresponding datatypes is:

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

set_pose

allows to provide a robot pose as calibration pose to the hand-eye calibration routine and records the current image of the calibration grid.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/set_pose
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/set_pose
```

Request

The `slot` argument is used to assign unique numbers to the different calibration poses. The range for `slot` is from 0 to 15. 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:

```
{
  "args": {
    "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"
      }
    },
    "slot": "uint32"
  }
}
```

Response

Table 5.45: 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
13	false	invalid slot number

The field `overexposed` indicates if parts of the calibration grid were overexposed in this image.

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_pose",
  "response": {
    "message": "string",
    "overexposed": "bool",
    "status": "int32",
    "success": "bool"
  }
}
```

get_poses

returns the robot poses that are currently stored for the hand-eye calibration routine.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/get_poses
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/get_poses
```

Request

This service has no arguments.

Response

Table 5.46: Return codes of the `get_poses` service call

status	success	Description
0	true	stored poses are returned
1	true	no calibration pose available

The field `overexposed` indicates if parts of the calibration grid were overexposed in this image.

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_poses",
  "response": {
    "message": "string",
    "poses": [
      {
        "overexposed": "bool",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "slot": "uint32",
        "tag_ids": [
          "string"
        ]
      }
    ],
    "status": "int32",
    "success": "bool"
  }
}
```

delete_poses

deletes the calibration poses and corresponding images with the specified `slots`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/delete_
↪poses
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/delete_poses
```

Request

The `slots` argument specifies which calibration poses should be deleted. If no slots are provided, nothing will be deleted.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "slots": [
      "uint32"
    ]
  }
}
```

Response

Table 5.47: Return codes of the `delete_poses` service call

status	success	Description
0	true	poses successfully deleted
1	true	no slots given

The definition for the response with corresponding datatypes is:

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

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.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/reset_
↪calibration
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/reset_calibration
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_calibration",
  "response": {
```

(continues on next page)

(continued from previous page)

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

calibrate

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

Details

`save_calibration` must be called to make the calibration available for other modules via the `get_calibration` service call and to store it persistently.

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

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/calibrate
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/calibrate
```

Request

This service has no arguments.

Response

The field `error` gives the calibration error in pixels which is computed from the translational error `translation_error_meter` and the rotational error `rotation_error_degree`. This value is only given for compatibility with older versions. The translational and rotational errors should be preferred.

Table 5.48: 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 not distinct enough from each other

The definition for the response with corresponding datatypes is:

```
{
  "name": "calibrate",
  "response": {
    "error": "float64",
```

(continues on next page)

(continued from previous page)


```
    "message": "string",
    "pose": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "robot_mounted": "bool",
    "rotation_error_degree": "float64",
    "status": "int32",
    "success": "bool",
    "tags": [
      {
        "id": "string",
        "pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "size": "float64"
      }
    ],
    "translation_error_meter": "float64"
  }
}
```


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 call will also delete all the stored calibration poses and corresponding camera images in the `slots`.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/save_
↪calibration
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/save_calibration
```

Request

This service has no arguments.

Response

Table 5.49: 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

The definition for the response with corresponding datatypes is:

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

`set_calibration`

sets the hand-eye calibration transformation with arguments of this call.

Details

The calibration transformation is expected in the same format as returned by the `calibrate` and `get_calibration` calls. The given calibration information is also stored persistently on the sensor by internally calling `save_calibration`.

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/set_
↪calibration
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/set_calibration
```

Request

The definition for the request arguments with corresponding datatypes is:

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

Response

Table 5.50: 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

The definition for the response with corresponding datatypes is:

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

reset_defaults

restores and applies the default values for this module'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.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_hand_eye_calibration/services/reset_
↪defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_hand_eye_calibration/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.5.2 CollisionCheck

5.5.2.1 Introduction

The CollisionCheck module is an optional on-board module of the *rc_visard* and is licensed with any of the modules *ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5) or *SilhouetteMatch* (Section 5.4.7). Otherwise it requires a separate CollisionCheck *license* (Section 7.4) to be purchased.

The module provides an easy way to check if a gripper is in collision with a load carrier, the point cloud (only in combination with *SilhouetteMatch* (Section 5.4.7)), or other detected objects (only in combination with *SilhouetteMatch* (Section 5.4.7)). It is integrated with the *ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5) and *SilhouetteMatch* (Section 5.4.7) modules, but can be used as standalone product. The models of the grippers for collision checking have to be defined in the *GripperDB* (Section 5.6.4) module.

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. Only if `check_collisions_with_point_cloud` is enabled in the respective detection module, collisions between the gripper and a watertight version of the point cloud will be checked. Only in combination with *SilhouetteMatch* (Section 5.4.7), and only in case the selected template contains a collision geometry and `check_collisions_with_matches` is enabled in the respective detection module, also collisions between the gripper and other *detected* objects will be checked. Collisions with objects that cannot be detected will not be checked.

Table 5.51: Specifications of the CollisionCheck module

Collision checking with	detected load carrier, detected objects (only *SilhouetteMatch* (Section 5.4.7)), baseplane (only *SilhouetteMatch*, Section 5.4.7), point cloud (only *SilhouetteMatch* (Section 5.4.7))
Collision checking available in	*ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5), *SilhouetteMatch* (Section 5.4.7)

5.5.2.2 Collision checking

Stand-alone collision checking

The `check_collisions` service call triggers collision checking between the chosen gripper and the provided load carriers for each of the provided grasps. Checking collisions with other objects or the point cloud is not possible with the stand-alone `check_collisions` service. 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. 5.30). If the pre-grasp offset is defined, the grasp will be detected as colliding if the gripper is in collision at any point during motion from the pre-grasp position to the grasp position (assuming a linear movement).

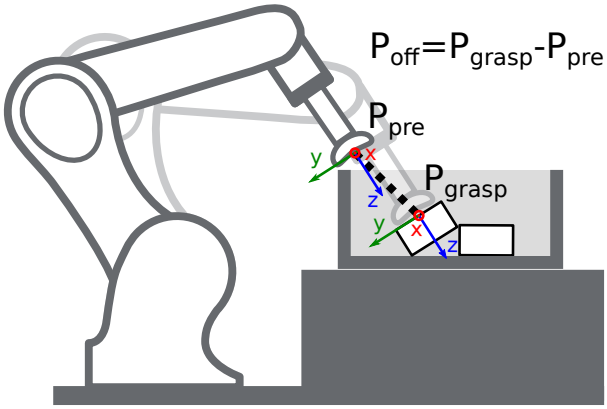


Fig. 5.30: Illustration of the pre-grasp offset parameter for collision checking. In this case, the pre-grasp position as well as the grasp position are collision free. However, the trajectory between these poses would have collisions. Thus, this grasp pose would be marked as colliding.

Collision checking within other modules

Collision checking is integrated in the following modules' services:

- *ItemPick* (Section 5.4.4): `compute_grasps` (see *compute_grasps*, Section 5.4.4.8)
- *BoxPick* (Section 5.4.5): `compute_grasps` (see *compute_grasps*, Section 5.4.5.9)
- *SilhouetteMatch* (Section 5.4.7): `detect_object` (see *detect_object*, Section 5.4.7.11)

Each of these services can take a `collision_detection` argument consisting of the `gripper_id` of the default gripper and the `pre_grasp_offset` as described in the previous section *Stand-alone collision checking* (Section 5.5.2.2). The default gripper given by the `gripper_id` argument is only used for grasp points which do not have an individual gripper ID assigned. When the `collision_detection` argument is given, these services only return the grasps at which the gripper is not in collision or which could not be checked for collisions. When a load carrier ID is provided to these services, collision checking will always be performed between the gripper and the load carrier. Additional collision check features can be enabled depending on the module.

Only for *SilhouetteMatch* (Section 5.4.7), and only in case the selected template contains a collision geometry and `check_collisions_with_matches` is enabled in the respective detection module, grasp points at which the gripper would be in collision with other *detected* objects are also rejected. The object on which the grasp point to be checked is located, is excluded from the collision check.

When a gripper is defined for a grasp point in the object template for *SilhouetteMatch* (Section 5.4.7), then this gripper will be used for collision checking at that specific grasp point instead of the default gripper

defined in the `collision_detection` argument of the `detect_object` service (see *Setting of grasp points*, Section 5.4.7.4). The grasps returned by the `detect_object` service contain a flag `collision_checked`, indicating whether the grasp was checked for collisions, and the field `gripper_id`. If `collision_checked` is true, the returned `gripper_id` contains the ID of the gripper that was used for the collision check. That is the ID of the gripper defined for that specific grasp, or, if empty, the gripper that was given in the `collision_detection` argument of the request. If `collision_checked` is false, the returned `gripper_id` is the gripper ID that was defined for that grasp.

In *SilhouetteMatch*, Section 5.4.7, collisions between the gripper and the base plane can be checked, if `check_collisions_with_base_plane` is enabled in SilhouetteMatch.

Collisions between the gripper and a watertight version of the point cloud can be checked if `check_collisions_with_point_cloud` is enabled in the respective module.

Warning: Collisions are checked only between the load carrier and the gripper, not the robot itself, the flange or other objects. Only if `check_collisions_with_point_cloud` is enabled, collisions between the gripper and a watertight version of the point cloud are checked. Only in combination with *SilhouetteMatch* (Section 5.4.7), and only in case the selected template contains a collision geometry and `check_collisions_with_matches` is enabled in the respective detection module, also collisions between the gripper and other *detected* objects are checked. Collisions with objects that cannot be detected will not be checked.

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

5.5.2.3 Parameters

The CollisionCheck module is called `rc_collision_check` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Configuration* → *CollisionCheck*. The user can explore and configure the `rc_collision_check` module's run-time parameters, e.g. for development and testing, using the Web GUI or the *REST-API interface* (Section 6.2).

Parameter overview

This module offers the following run-time parameters:

Table 5.52: The `rc_collision_check` module's run-time parameters

Name	Type	Min	Max	Default	Description
`check_bottom`	bool	false	true	true	Whether to enable collision checking with the bottom of the load carrier
`check_flange`	bool	false	true	true	Whether all grasps with the flange inside the load carrier should be marked as colliding
`collision_dist`	float64	0.0	0.1	0.01	Minimum distance in meters between any element of the gripper and the load carrier or the base plane (only SilhouetteMatch) for a collision-free grasp
`pointcloud_watertight`	bool	false	true	true	Whether to use a watertight disparity image for collision check with the point cloud

Description of run-time parameters

Each run-time parameter is represented by a row in the Web GUI's *Settings* section under *Configuration* → *CollisionCheck*. 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 the load carrier and/or the base plane (only SilhouetteMatch) for a grasp to be considered collision free.

Note: The collision distance is not applied when checking collisions between the gripper and the point cloud, or the gripper and other detected objects. It is not applied when checking if the flange is inside the load carrier (`check_flange`), either.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_collision_check/parameters?collision_
↪dist=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_collision_check/parameters?collision_dist=<value>
```

`check_flange` (*Check Flange*)

Performs an additional safety check as described in *Robot flange radius* (Section 5.6.4.2). If this parameter is set, all grasps in which any part of the robot's flange is inside the load carrier are marked as colliding.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_collision_check/parameters?check_flange=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_collision_check/parameters?check_flange=<value>
```

`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).

The load carrier bottom will always be excluded for the collision check between the object in the gripper and the load carrier during retraction in combination with *ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5) when `check_collisions_during_retraction` is enabled.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_collision_check/parameters?check_bottom=
↪<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_collision_check/parameters?check_bottom=<value>
```

`pointcloud_watertight` (*Watertight Point Cloud*)

When this option is enabled the point cloud will be made watertight for collision checking. In a watertight point cloud, holes from the disparity image will be interpolated by valid measurements of neighboring pixels, so that the resulting point cloud has no holes. This leads to conservative collision checking results.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_collision_check/parameters?pointcloud_
↪watertight=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_collision_check/parameters?pointcloud_watertight=
↪<value>
```

5.5.2.4 Status values

The `rc_collision_check` module reports the following status values:

Table 5.53: The `rc_collision_check` module status values

Name	Description
`last_evaluated_grasps`	Number of evaluated grasps
`last_collision_free_grasps`	Number of collision-free grasps
`collision_check_time`	Collision checking runtime

5.5.2.5 Services

The user can explore and call the `rc_collision_check` module's services, e.g. for development and testing, using *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1).

The CollisionCheck module offers the following services.

`reset_defaults`

Resets all parameters of the module to its default values, as listed in above table.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_collision_check/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_collision_check/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

`check_collisions` (deprecated)

Triggers a collision check between a gripper and a load carrier.

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_collision_check/services/check_collisions
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_collision_check/services/check_collisions
```

Request

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 5.4.2.2). 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).

The definition for the request arguments with corresponding datatypes is:

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


Response

`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.

The definition for the response with corresponding datatypes is:


```
{
  "name": "check_collisions",
  "response": {
    "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"
    }
  }
}
```


`set_gripper` (deprecated)

Persistently stores a gripper on the *rc_visard*.

API version 2

This service is not available in API version 2. Use *set_gripper* (Section 5.6.4.3) in `rc_gripper_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_collision_check/services/set_gripper
```

The definitions of the request and response are the same as described in *set_gripper* (Section 5.6.4.3) in `rc_gripper_db`.

`get_grippers` (deprecated)

Returns the configured grippers with the requested `gripper_ids`.

API version 2

This service is not available in API version 2. Use *get_grippers* (Section 5.6.4.3) in `rc_gripper_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_collision_check/services/get_grippers
```

The definitions of the request and response are the same as described in *get_grippers* (Section 5.6.4.3) in `rc_gripper_db`.

`delete_grippers` (deprecated)

Deletes the configured grippers with the requested `gripper_ids`.

API version 2

This service is not available in API version 2. Use *delete_grippers* (Section 5.6.4.3) in `rc_gripper_db` instead.

API version 1 (deprecated)

This service can be called as follows.

```
PUT http://<host>/api/v1/nodes/rc_collision_check/services/delete_grippers
```

The definitions of the request and response are the same as described in *delete_grippers* (Section 5.6.4.3) in `rc_gripper_db`.

5.5.2.6 Return codes

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 5.54: 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

5.5.3 Camera calibration

The camera calibration module is a base module which is available on every *rc_visard*.

To use the 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 *Rectification*, Section 5.1.2), which is the basis for all other image processing modules.

The *rc_visard* is calibrated at production time. Nevertheless, checking calibration and recalibration might be necessary if the *rc_visard* was exposed to strong mechanical impact.

The camera calibration module is responsible for checking calibration and calibrating.

5.5.3.1 Self-calibration

The camera calibration module 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 module's log file (see *Downloading log files*, Section 7.5) 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, recalibrating the camera manually is recommended if the update counter is not 0. The Web GUI displays a warning when self calibration has occurred.

5.5.3.2 Calibration process

Manual calibration can be done through the *Web GUI* (Section 6.1) under *Configuration* → *Camera Calibration*. This page provides a wizard to guide the user through the calibration process.

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

During calibration, the calibration grid must be detected in different poses. When holding the calibration grid, make sure that all black squares of the grid are completely visible and not occluded in both camera images. 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. Some of the squares not being detected, or being detected only briefly might indicate bad lighting conditions, or a damaged grid. Squares in overexposed parts of the calibration grid are highlighted in red. In this case, the lighting conditions or exposure setting must be adjusted. A thick green border around the calibration grid indicates that it was detected correctly in both camera images.

Calibration settings

The quality of camera calibration heavily depends on the quality of the calibration grid. Calibration grids can be obtained from Basler.

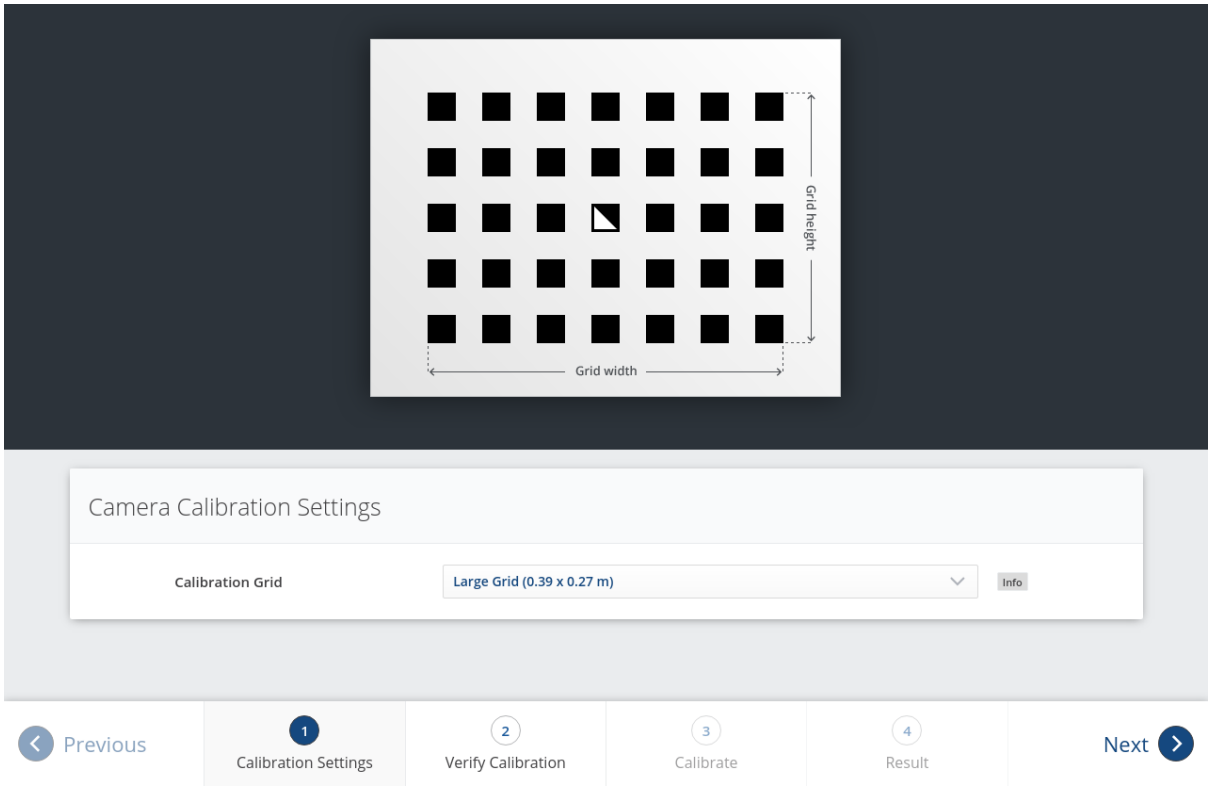


Fig. 5.31: Calibration settings

In the first step, the calibration grid must be specified. The *Next* button proceeds to the next step.

Verify calibration

In the next 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. When the grid is detected, the calibration error is automatically computed and the result is displayed on the screen.

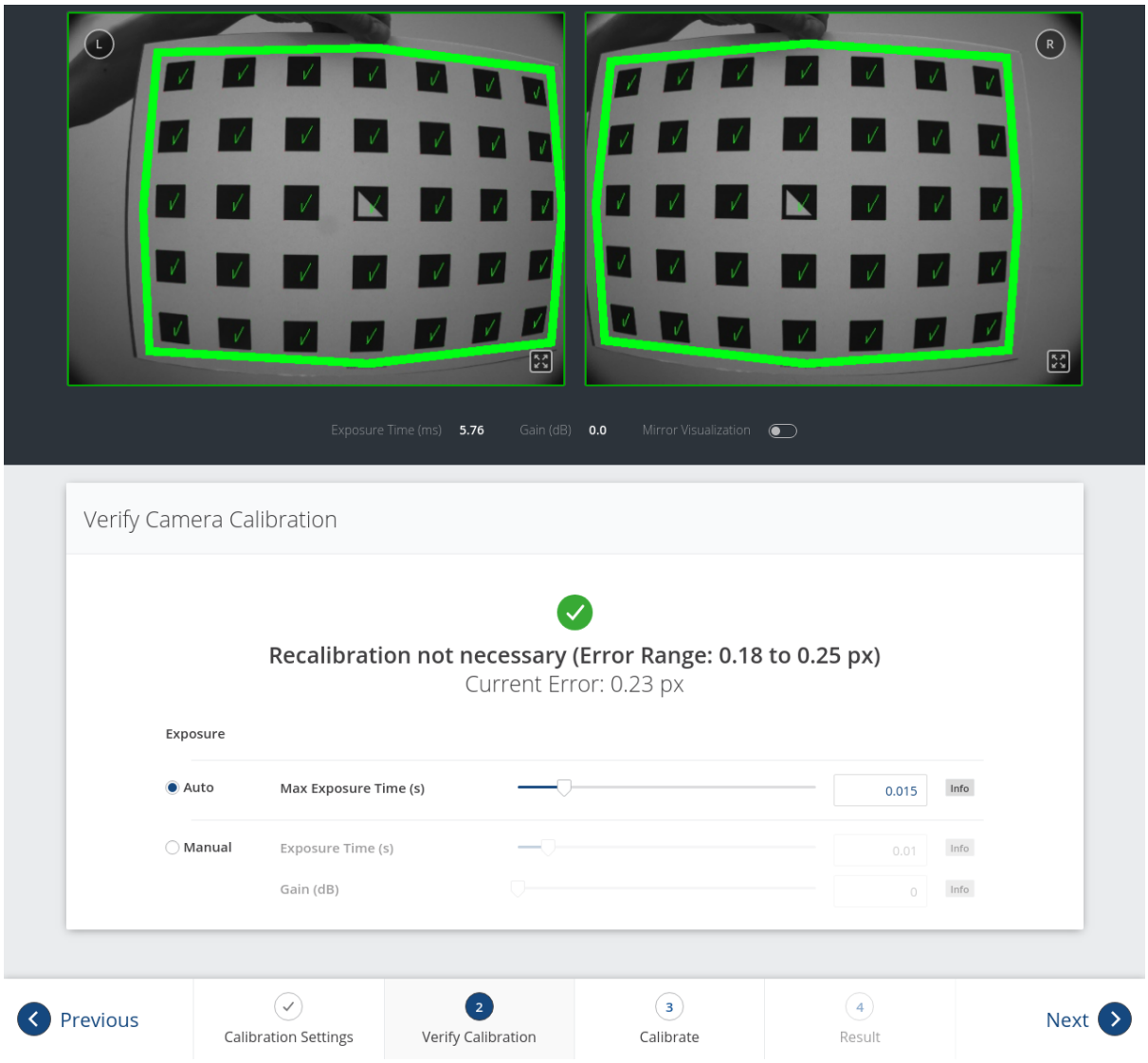


Fig. 5.32: Verification of calibration

Note: To compute a meaningful calibration error, the grid should be held as close 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. For this reason, the minimal and maximal calibration error during verification are shown in addition to the calibration error at the current grid position.

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!

Calibrate

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 motion blur should be avoided. Thus, the maximum auto-exposure time should be as short as possible, but still allow a good exposure. The current exposure time is displayed below the camera images as shown in Fig. 5.34.

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, monocalibration is skipped by default during a recalibration, but can be performed by clicking *Perform Monocalibration* in the *Calibrate* tab. This should only be done if the result of the stereo calibration is not satisfactory.

Stereo calibration

During stereo calibration, both cameras are calibrated to each other to find their relative rotation and translation.

The camera images can also be displayed mirrored to simplify the correct positioning of the calibration grid.

First, the grid should be held as close as possible to the camera and very still. It must be fully visible in both images and the cameras should look perpendicularly onto the grid. If the grid is not perpendicular to the line of sight of the cameras, this will be indicated by small green arrows pointing to the expected positions of the grid corners (see Fig. 5.33).

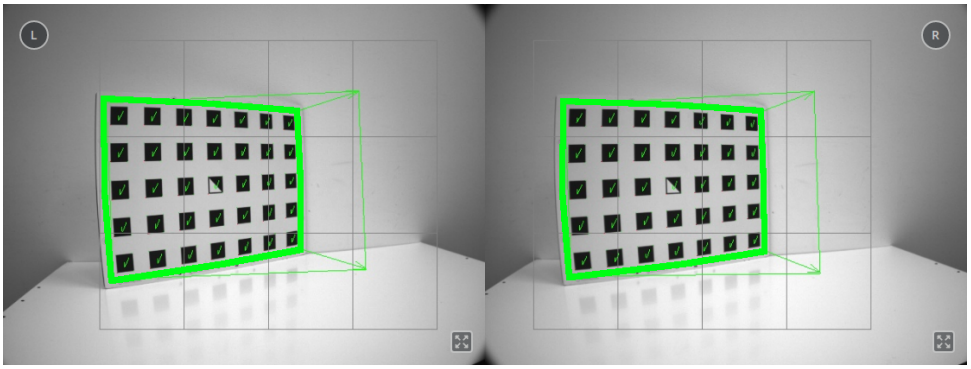


Fig. 5.33: Arrows indicating that the grid is not perpendicular to the camera's line of sight during stereo calibration

The grid must be kept very still for detection. If motion blur occurs, the grid will not be detected. All grid cells that are drawn onto the image have to be covered by the calibration grid. This is visualized by filling the covered cells in green (see Fig. 5.34).

For the *rc_visard* all cells can be covered at once by holding the grid close enough.

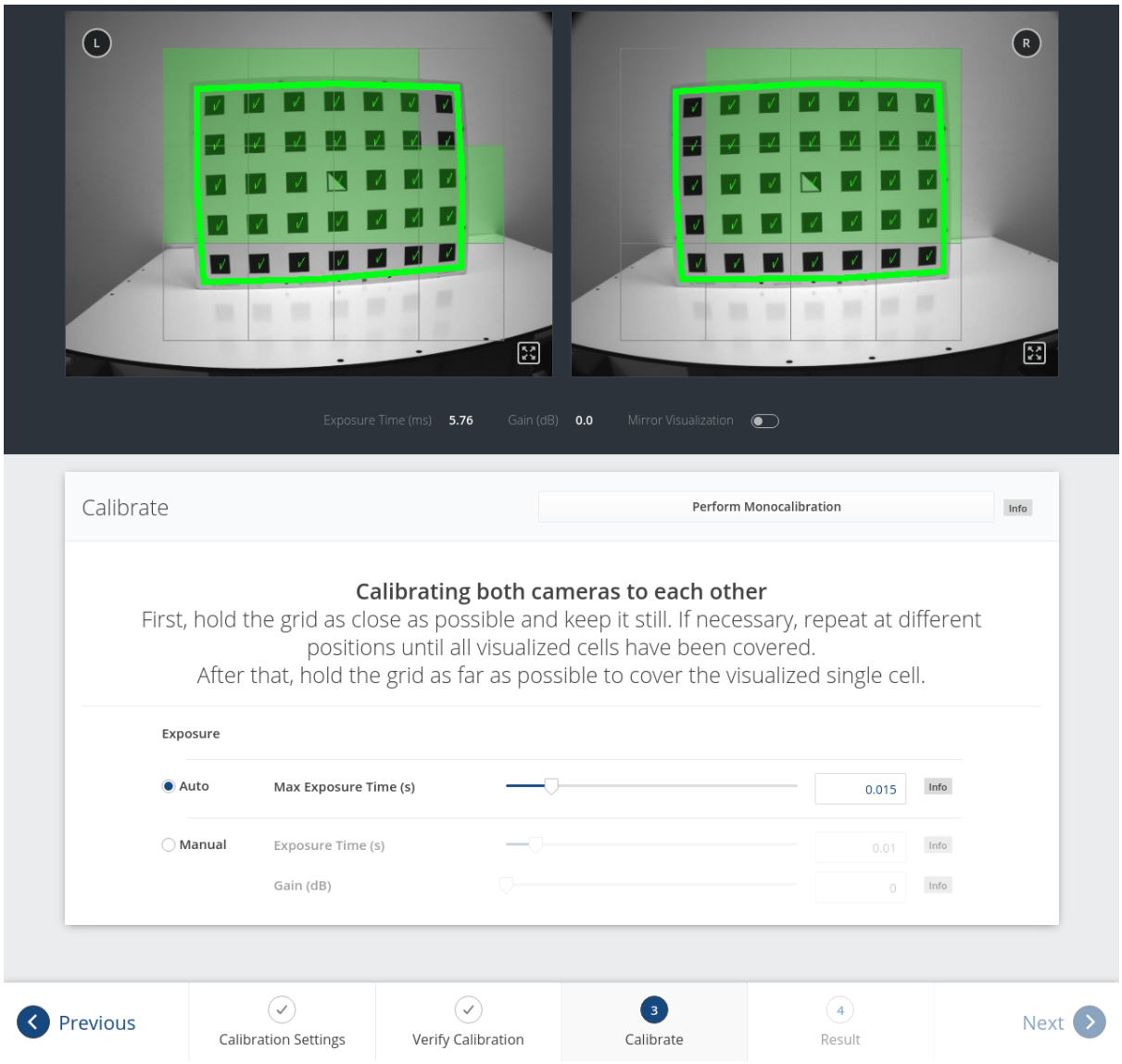


Fig. 5.34: Stereo calibration: Hold the grid as close as possible to fill all visualized cells

Note: If the check marks on the calibration grid all vanish, then either the camera does not look perpendicularly onto the grid, or the grid is too far away from the camera.

Once all grid cells are covered, they disappear and a single far cell is visualized. Now, the grid should be held as far as possible from the cameras, so that the small cell is covered. Arrows will indicate if the grid is still too close to the camera. When the grid is successfully detected at the far pose, the cell is filled in green and the result can be computed (see Fig. 5.35).

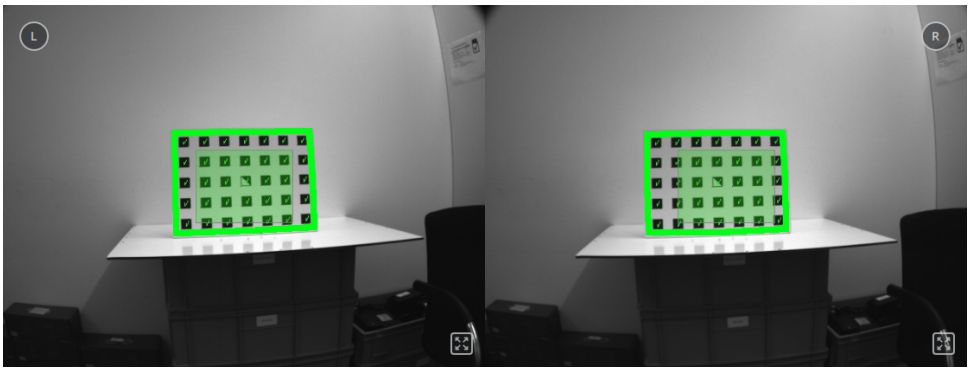


Fig. 5.35: Holding the grid far away during stereo calibration

If stereo calibration yields an unsatisfactory calibration error, then calibration should be repeated with monocalibration (see next Section *Monocalibration*).

Monocalibration

Monocalibration is the intrinsic calibration of each camera individually. Since the intrinsic calibration normally does not get corrupted, the monocalibration should only be performed if the result of stereo calibration is not satisfactory.

Click *Perform Monocalibration* in the *Calibrate* tab to start monocalibration.

For monocalibration, 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. 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. 5.36.

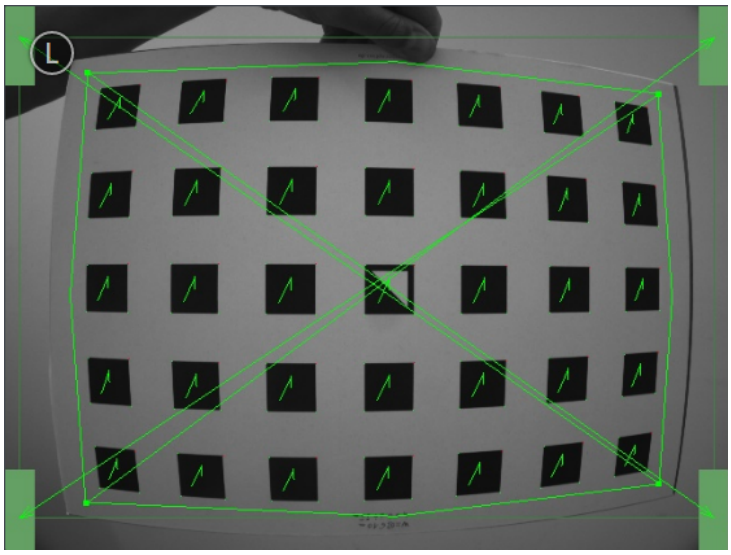


Fig. 5.36: Wrongly holding the grid upside down leads to crossed green lines.

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.

The monocalibration process involves five poses for each camera as shown in Fig. 5.37.

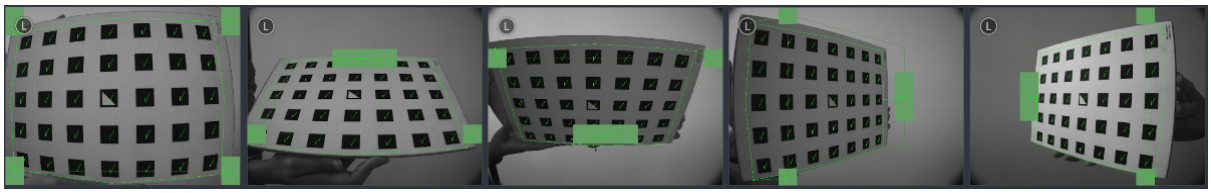


Fig. 5.37: Poses required for monocamera calibration

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.

Continue with the guidelines given in the previous Section *Stereo calibration*.

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 device.

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.

5.5.3.3 Parameters

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

Note: The camera calibration module'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.

5.5.3.4 Services

Note: The camera calibration module'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.

5.5.4 IO and Projector Control

The IOControl module is a base module which is available on every *rc_visard*.

The IOControl module allows reading the status of the general purpose digital inputs and controlling the digital general purpose outputs (GPIOs) of the camera. 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 module is the control of an external light source or a projector, which is connected to one of the camera'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 module 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.

5.5.4.1 Parameters

The IOControl module is called `rc_iocontrol` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Configuration* → *IOControl*.

The user can change the parameters via the Web GUI, the *REST-API interface* (Section 6.2), or via GigE Vision using the DigitalIOControl parameters `LineSelector` and `LineSource` (*Category: DigitalIOControl*, Section 6.7.3.4).

Parameter overview

This module offers the following run-time parameters:

Table 5.55: The `rc_iocontrol` module's run-time parameters

Name	Type	Min	Max	Default	Description
`out1_mode`	string	-	-	Low	Out1 mode: [Low, High, ExposureActive, ExposureAlternateActive]
`out2_mode`	string	-	-	Low	Out2 mode: [Low, High, ExposureActive, ExposureAlternateActive]

Description of run-time parameters

`out1_mode` and `out2_mode` (*Out1 / Projector* and *Out2*)

The output modes for GPIO Out 1 and Out 2 can be set individually:

`Low` sets the output 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.

Via the REST-API, this parameter can be set as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_iocontrol/parameters/parameters?<out1_
↪mode|out2_mode>=<value>
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_iocontrol/parameters?<out1_mode|out2_mode>=<value>
```

Fig. 5.38 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.

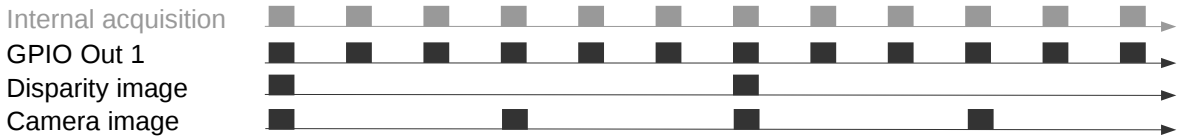


Fig. 5.38: Example of using the `ExposureActive` mode for GPIO Out 1 with a user-defined frame rate of 8 Hz. The internal image acquisition is always 25 Hz. GPIO Out 1 is HIGH for the exposure time of every image. A disparity image is computed for camera images that are sent out via GigE Vision according to the user-defined frame rate.

The mode `ExposureAlternateActive` is meant to be used when an external random dot projector is connected to the camera's GPIO Out 1. When setting Out 1 to `ExposureAlternateActive`, the *stereo matching* (Section 5.2) module 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. All modules which make use of the intensity image, like *TagDetect* (Section 5.4.3) and *ItemPick* (Section 5.4.4), use the intensity images with GPIO Out 1 being LOW, i.e. projector is off. Fig. 5.39 shows an example.

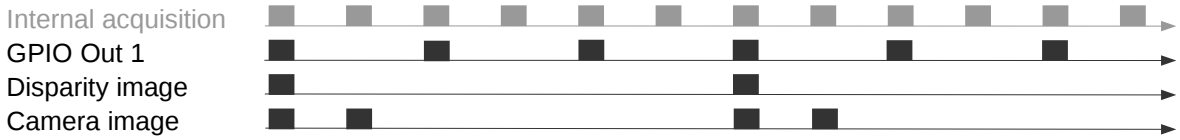


Fig. 5.39: Example of using the `ExposureAlternateActive` mode for GPIO Out 1 with a user-defined frame rate of 8 Hz. The internal image acquisition is always 25 Hz. GPIO Out 1 is HIGH for the exposure time of every second image. A disparity image is computed for images where Out 1 is HIGH and that are 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 6.7.3.4).

5.5.4.2 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 module offers the following services.

`get_io_values`

Retrieves the current state of the camera's general purpose inputs and outputs (GPIOs).

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_iocontrol/services/get_io_values
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_iocontrol/services/get_io_values
```

Request

This service has no arguments.

Response

The returned `timestamp` is the time of measurement.

`input_mask` and `output_mask` are bit masks defining which bits are used for input and output values, respectively.

`values` holds the values of the bits corresponding to input and output as given by the `input_mask` and `output_mask`.

`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

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_io_values",
  "response": {
    "input_mask": "uint32",
    "inverter_mask": "uint32",
    "output_mask": "uint32",
    "ratio_mask": "uint32",
    "return_code": {
      "message": "string",
      "value": "int16"
    },
    "timestamp": {
      "nsec": "int32",
```

(continues on next page)

(continued from previous page)

```
      "sec": "int32"
    },
    "values": "uint32"
  }
}
```

reset_defaults

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

Details

This service can be called as follows.

API version 2

```
PUT http://<host>/api/v2/pipelines/0/nodes/rc_iocontrol/services/reset_defaults
```

API version 1 (deprecated)

```
PUT http://<host>/api/v1/nodes/rc_iocontrol/services/reset_defaults
```

Request

This service has no arguments.

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "reset_defaults",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.6 Database modules

The *rc_visard* provides several database modules which enable the user to configure global data which is used in many detection modules, such as load carriers and regions of interest. Via the *REST-API interface* (Section 6.2) the database modules are only available in API version 2.

The database modules are:

- ***LoadCarrierDB*** **(`rc_load_carrier_db`, Section 5.6.1)** allows setting, retrieving and deleting load carriers.
- ***RoiDB*** **(`rc_roi_db` , Section 5.6.2)** allows setting, retrieving and deleting 2D and 3D regions of interest.
- ***PrimitiveDB*** **(`rc_primitive_db`, Section 5.6.3)** allows setting, retrieving and deleting geometric primitives such as planes.
- ***GripperDB*** **(`rc_gripper_db`, Section 5.6.4)** allows setting, retrieving and deleting grippers for collision checking.

5.6.1 LoadCarrierDB

5.6.1.1 Introduction

The LoadCarrierDB module (Load carrier database module) allows the global definition of load carriers, which can then be used in many detection modules. The specified load carriers are available for all modules supporting load carriers on the *rc_visard*.

The LoadCarrierDB module is a base module which is available on every *rc_visard*.

Table 5.56: Specifications of the LoadCarrierDB module

Supported load carrier types	4-sided or 3-sided
Supported rim types	solid rim, stepped rim or ledged rim
Min. load carrier dimensions	0.1 m x 0.1 m x 0.05 m
Max. load carrier dimensions	5 m x 5 m x 5 m
Max. number of load carriers	50
Load carriers available in	*ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5) and *SilhouetteMatch* (Section 5.4.7)
Supported pose types	no pose, orientation prior, exact pose
Supported reference frames	camera, external

5.6.1.2 Load carrier definition

A load carrier (bin) is a container with four walls, a floor and a rectangular rim, which can contain objects. It can be used to limit the volume in which to search for objects or grasp points.

A load carrier is defined by its `outer_dimensions` and `inner_dimensions`. The maximum `outer_dimensions` are 5.0 meters in every dimension.

The origin of the load carrier reference frame is in the center of the load carrier's *outer* box and its z axis is perpendicular to the load carrier's floor pointing outwards (see Fig. 5.40).

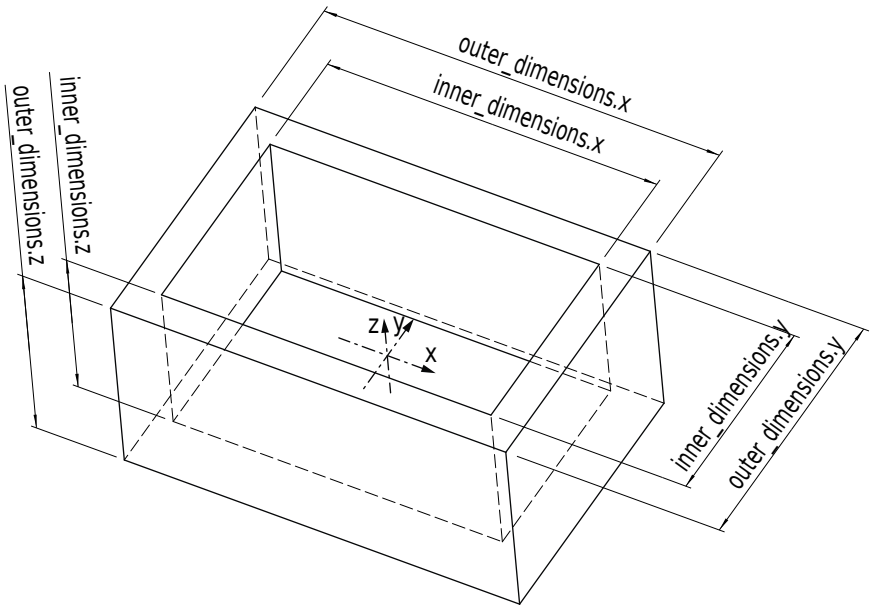


Fig. 5.40: Load carrier with reference frame and inner and outer dimensions

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

The inner volume of the load carrier is defined by its inner dimensions, but includes a region of 10 cm height above the load carrier, so that also items protruding from the load carrier are considered for detection or grasp computation. Furthermore, an additional `crop_distance` is subtracted from the inner volume in every dimension, which acts as a safety margin and can be configured as run-time parameter in the LoadCarrier module (see *Parameters*, Section 5.4.2.5). Fig. 5.41 visualizes the inner volume of a load carrier. Only points which are inside this volume are considered for detections.

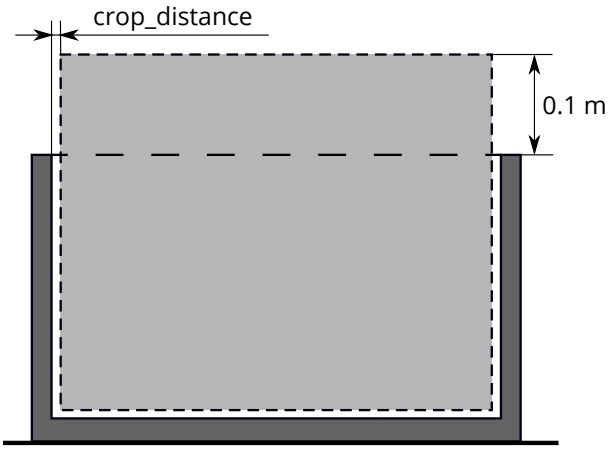


Fig. 5.41: Visualization of the inner volume of a load carrier. Only points which are inside this volume are considered for detections.

Since the load carrier detection is based on the detection of the load carrier's rim, the rim geometry must be specified if it cannot be determined from the difference between outer and inner dimensions. A load carrier with a stepped rim can be defined by setting a `rim_thickness`. The rim thickness gives the thickness of the outer part of the rim in the x and y direction. When a rim thickness is given, an optional `rim_step_height` can also be specified, which gives the height of the step between the outer and the inner part of the rim. When the step height is given, it will also be considered during collision checking (see *CollisionCheck*, Section 5.5.2). Examples of load carriers with stepped rims are shown in Fig. 5.42 A, B. In addition to the `rim_thickness` and `rim_step_height` the `rim_ledge` can be specified for defining load carriers whose inner rim protrudes into the interior of the load carrier, such as pallet cages. The `rim_ledge` gives the thickness of the inner part of the rim in the x and y direction. An example of a load carrier with a ledged rim is shown in Fig. 5.42 C.

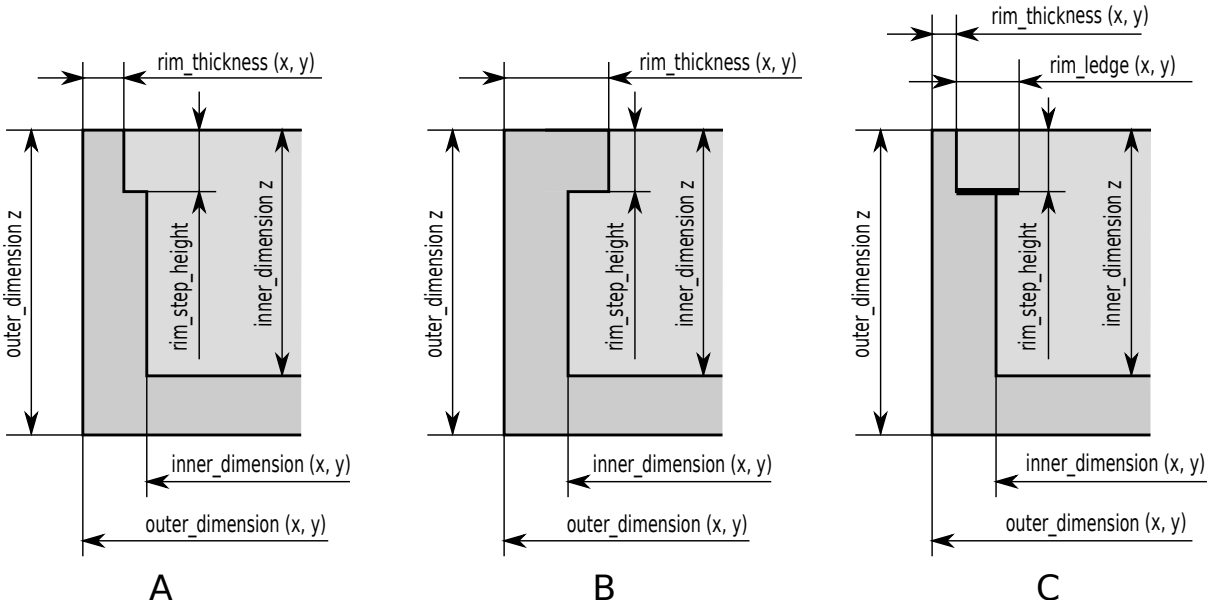


Fig. 5.42: Examples of load carriers with stepped rim (A, B) or ledged rim (C)

The different rim types are applicable to both, standard 4-sided and 3-sided load carriers. For a 3-sided load carrier, the `type` must be `THREE_SIDED`. If the type is set to `STANDARD` or left empty, a 4-sided load carrier is specified. A 3-sided load carrier has one side that is lower than the other three sides. This `height_open_side` is measured from the outer bottom of the load carrier. The open side is at the negative y-axis of the load carrier's coordinate system. Examples of the two load carrier types are given in Fig. 5.43. The height of the lower side is only considered during collision checking and not required for the detection of the load carrier.

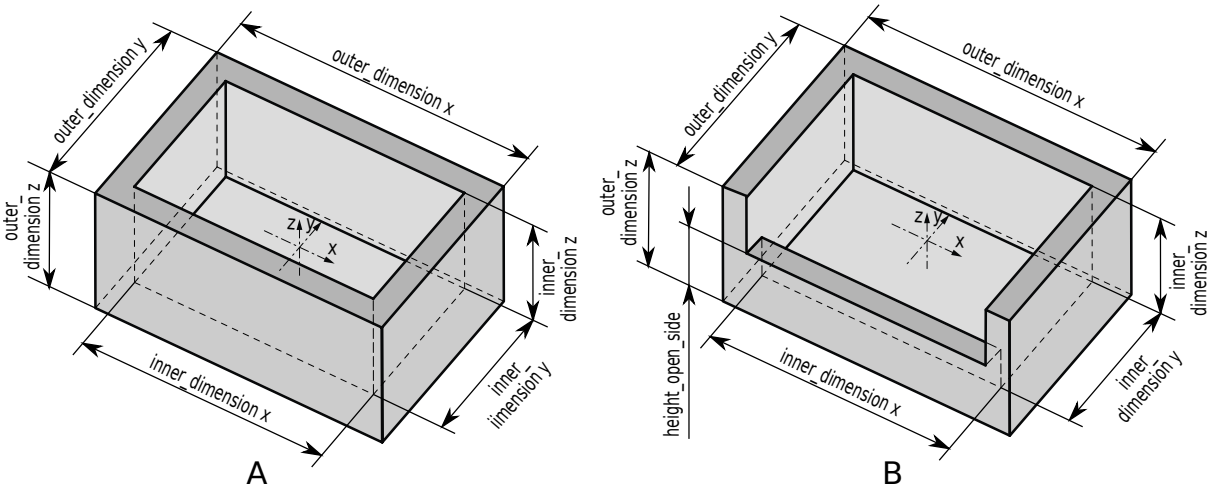


Fig. 5.43: Examples of a standard 4-sided load carrier (A) and a 3-sided load carrier (B)

A load carrier can be specified with a full 3D `pose` consisting of a `position` and an `orientation` quaternion, given in a `pose_frame`. Based on the given `pose_type` this pose is either used as an orientation prior (`pose_type` is `ORIENTATION_PRIOR` or empty), or as the exact pose of the load carrier (`pose_type` is `EXACT_POSE`).

In case the pose serves as orientation prior, the detected load carrier pose is guaranteed to have the minimum rotation with respect to the load carrier's prior pose. This pose type is useful for detecting tilted load carriers and for resolving the orientation ambiguity in the x and y direction caused by the symmetry of the load carrier model.

In case the pose type is set to `EXACT_POSE`, no load carrier detection will be performed on the scene data, but the given pose will be used in exactly the same way as if the load carrier is detected at that pose. This pose type is especially useful in cases where load carriers do not change their positions and/or are hard to detect (e.g. because their rim is too thin or the material is too shiny).

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 6.2) or in the *rc_visard* Web GUI.

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

5.6.1.3 Load carrier compartments

Some detection modules can make use of a `load_carrier_compartment` to further limit the volume for the detection, for example *ItemPick's compute_grasps service* (see 5.4.4.8). A load carrier compartment is a box whose `pose` is defined as the transformation from the load carrier reference frame to the compartment reference frame, which is located in the center of the compartment box (see Fig. 5.44). The load carrier compartment is defined for each detection call separately and is not part of the load carrier definition in the LoadCarrierDB module.

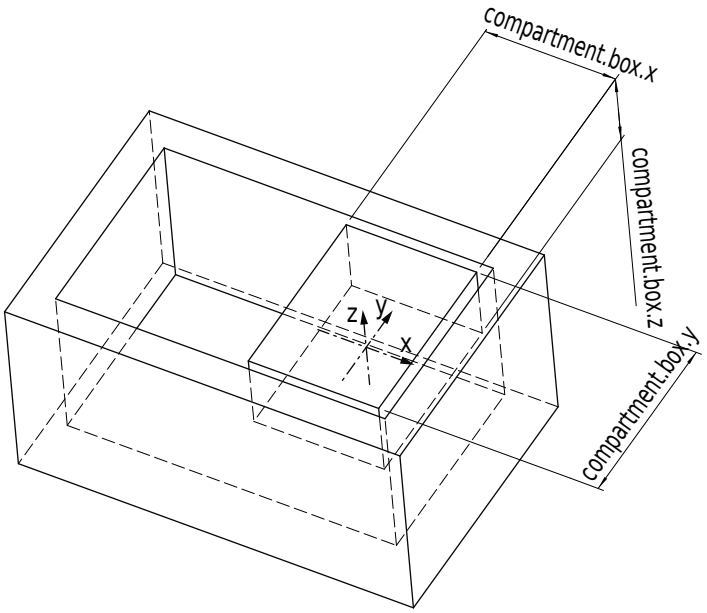


Fig. 5.44: Sample compartment inside a load carrier. The coordinate frame shown in the image is the reference frame of the compartment.

The compartment volume is intersected with the load carrier inner volume to compute the volume for the detection. If this intersection should also contain the 10 cm region above the load carrier, the height of the compartment box must be increased accordingly.

5.6.1.4 Interaction with other modules

Internally, the LoadCarrierDB module depends on, and interacts with other on-board modules as listed below.

Hand-eye calibration

In case the camera has been calibrated to a robot, the load carrier's exact pose or orientation prior can be provided in the robot coordinate frame by setting the corresponding `pose_frame` argument to `external`.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). The load carrier pose or orientation prior is provided 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`). The load carrier pose or orientation prior is provided in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) to retrieve the sensor mounting (static or robot mounted) and the hand-eye transformation.

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.

5.6.1.5 Services

The LoadCarrierDB module is called `rc_load_carrier_db` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Database* → *Load Carriers*. The user can explore and call the LoadCarrierDB module's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the Web GUI.

The LoadCarrierDB module offers 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.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_load_carrier_db/services/set_load_carrier
```

Request

Details for the definition of the `load_carrier` type are given in *Load carrier definition* (Section 5.6.1.2).

The field `type` is optional and accepts `STANDARD` and `THREE_SIDED`.

The field `pose_type` is optional and accepts `NO_POSE`, `EXACT_POSE` and `ORIENTATION_PRIOR`.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "load_carrier": {
      "height_open_side": "float64",
      "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",
      "pose_type": "string",
      "rim_ledge": {
        "x": "float64",
        "y": "float64"
      },
      "rim_step_height": "float64",
      "rim_thickness": {
        "x": "float64",
        "y": "float64"
      },
      "type": "string"
    }
  }
}
```


Response

The definition for the response with corresponding datatypes is:


```
{
  "name": "set_load_carrier",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```


get_load_carriers

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

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_load_carrier_db/services/get_load_carriers
```

Request

The definition for the request arguments with corresponding datatypes is:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_load_carriers",
  "response": {
    "load_carriers": [
      {
        "height_open_side": "float64",
        "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",
        "pose_type": "string",
        "rim_ledge": {
          "x": "float64",
          "y": "float64"
        },
        "rim_step_height": "float64",
        "rim_thickness": {
          "x": "float64",
          "y": "float64"
        },
        "type": "string"
      }
    ],
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
```

(continues on next page)

(continued from previous page)

```
}
```

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`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_load_carrier_db/services/delete_load_carriers
```

Request

The definition for the request arguments with corresponding datatypes is:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "delete_load_carriers",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.6.1.6 Return codes

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 5.57: Return codes of the LoadCarrierDB module's services

Code	Description
0	Success
-1	An invalid argument was provided
-10	New element could not be added as the maximum storage capacity of load carriers has been exceeded
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`

5.6.2 RoiDB

5.6.2.1 Introduction

The RoiDB module (region of interest database module) allows the global definition of 2D and 3D regions of interest, which can then be used in many detection modules. The ROIs are available for all modules supporting 2D or 3D ROIs on the *rc_visard*.

The RoiDB module is a base module which is available on every *rc_visard*.

3D ROIs can be used in *ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5). 2D ROIs can be used in *SilhouetteMatch* (Section 5.4.7), and *LoadCarrier* (Section 5.4.2).

Table 5.58: Specifications of the RoiDB module

Supported ROI types	2D, 3D
Supported ROI geometries	2D ROI: rectangle, 3D ROI: box, sphere
Max. number of ROIs	2D: 100, 3D: 100
ROIs available in	2D: *SilhouetteMatch* (Section 5.4.7), *LoadCarrier* (Section 5.4.2), 3D: *ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5)
Supported reference frames	camera, external

5.6.2.2 Region of interest

A region of interest (ROI) defines a volume in space (3D region of interest, `region_of_interest`), or a rectangular region in the left camera image (2D region of interest, `region_of_interest_2d`) 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.

3D 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 3D region of interest `pose` in the `camera` or the `external` coordinate system. `External` can only be chosen if a *Hand-eye calibration* (Section 5.5.1) is available. When the sensor 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.

A 2D ROI is defined as a rectangular part of the left camera image, and can be set via the *REST-API interface* (Section 6.2) or the *rc_visard Web GUI* (Section 6.1) on the page *Regions of Interest* under *Database*. The Web GUI offers an easy-to-use selection tool. Each ROI must have a unique name to address a specific 2D ROI.

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

The *rc_visard* can persistently store up to 100 different 3D regions of interest and the same number of 2D regions of interest. 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 6.2) of RoiDB module, or in the *rc_visard Web GUI* (Section 6.1) on the page *Regions of Interest* under *Database*.

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

5.6.2.3 Interaction with other modules

Internally, the RoiDB module depends on, and interacts with other on-board modules as listed below.

Hand-eye calibration

In case the camera has been calibrated to a robot, the pose of a 3D ROI can be provided in the robot coordinate frame by setting the corresponding `pose_frame` argument.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). The ROI pose is provided 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`). The ROI pose is provided in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) to retrieve the sensor mounting (static or robot mounted) and the hand-eye transformation.

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.

5.6.2.4 Services

The RoiDB module is called `rc_roi_db` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Database* → *Regions of Interest*. The user can explore and call the RoiDB module's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the Web GUI.

The RoiDB module offers the following services.

`set_region_of_interest`

Persistently stores a 3D region of interest on the *rc_visard*. All configured 3D regions of interest are persistent over firmware updates and rollbacks.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_roi_db/services/set_region_of_interest
```

Request

Details for the definition of the `region_of_interest` type are given in *Region of interest* (Section 5.6.2.2).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "region_of_interest": {
      "box": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
```

(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"
        }
      },
      "pose_frame": "string",
      "sphere": {
        "radius": "float64"
      },
      "type": "string"
    }
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_region_of_interest",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

set_region_of_interest_2d

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

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_roi_db/services/set_region_of_interest_2d
```

Request

Details for the definition of the `region_of_interest_2d` type are given in *Region of interest* (Section 5.6.2.2).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "region_of_interest_2d": {
      "height": "uint32",
      "id": "string",
      "offset_x": "uint32",
```

(continues on next page)

(continued from previous page)

```
      "offset_y": "uint32",
      "width": "uint32"
    }
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_region_of_interest_2d",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

get_regions_of_interest

Returns the configured 3D regions of interest with the requested `region_of_interest_ids`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_roi_db/services/get_regions_of_interest
```

Request

If no `region_of_interest_ids` are provided, all configured 3D regions of interest are returned.

The definition for the request arguments with corresponding datatypes is:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_regions_of_interest",
  "response": {
    "regions_of_interest": [
      {
        "box": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "id": "string",
        "pose": {
```

(continues on next page)

(continued from previous page)


```
          "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"
    }
  }
}
```


get_regions_of_interest_2d

Returns the configured 2D regions of interest with the requested `region_of_interest_2d_ids`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_roi_db/services/get_regions_of_interest_2d
```

Request

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:


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


Response

The definition for the response with corresponding datatypes is:


```
{
  "name": "get_regions_of_interest_2d",
  "response": {
    "regions_of_interest": [
      {
```


(continues on next page)

(continued from previous page)

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

delete_regions_of_interest

Deletes the configured 3D regions of interest with the requested `region_of_interest_ids`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_roi_db/services/delete_regions_of_interest
```

Request

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:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "delete_regions_of_interest",
  "response": {
    "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`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_roi_db/services/delete_regions_of_interest_2d
```

Request

All 2D regions of interest to be deleted must be explicitly stated in `region_of_interest_2d_ids`.

The definition for the request arguments with corresponding datatypes is:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "delete_regions_of_interest_2d",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.6.2.5 Return codes

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 5.59: Return codes of the RoiDB module's 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` or `set_region_of_interest_2d`

5.6.3 PrimitiveDB

5.6.3.1 Introduction

The PrimitiveDB module (geometric primitive database module) allows the global definition of geometric primitives, which can then be used in detection modules. Currently, only planes are supported.

A plane can, for example, serve as reference plane for measuring the dimensions of detected items in *BoxPick* (Section 5.4.5.2).

The PrimitiveDB module is a base module which is available on every *rc_visard*.

Table 5.60: Specifications of the PrimitiveDB module

Supported primitives	plane
Max. number of planes	100
Planes used for	Dimensioning of detected items
Supported reference frames	camera, external

5.6.3.2 Plane

A plane is defined in Hesse normal form

$$\mathbf{n} \cdot \mathbf{x} + d = 0,$$

where $\mathbf{n}$ is the normal of the plane (`normal`) and d is the signed distance of the plane from the origin of the reference frame (`distance`) in meters. All points $\mathbf{x}$ that fulfill this equation lie on the plane. The normal is stored as a unit vector. If a normal with a different length is given, the normal and the distance are scaled accordingly, which describes the same plane.

In the REST-API, a plane is defined by the following values:

- `id`: Unique name of the plane
- `normal`: normal of the plane with the components `x`, `y` and `z`
- `distance`: signed distance of the plane from the origin of the reference frame in meters
- `pose_frame`: reference frame of the plane, either `camera` or `external`

The PrimitiveDB module only stores planes and does not estimate them. A plane can be estimated from the current sensor data with the `compute_plane` service of the Measure module (see *Estimating planes*, Section 5.4.1.3) and then stored with `set_plane`.

The *rc_visard* can persistently store up to 100 different planes. The configuration of planes is normally performed offline, during the set up of the desired application. This can be done via the *REST-API interface* (Section 6.2) of the PrimitiveDB module, or in the *rc_visard Web GUI* (Section 6.1) on the page *Planes* under *Database*.

Note: The configured planes are persistent even over firmware updates and rollbacks.

5.6.3.3 Interaction with other modules

Internally, the PrimitiveDB module depends on, and interacts with other on-board modules as listed below.

Hand-eye calibration

In case the camera has been calibrated to a robot, a plane can be provided in the robot coordinate frame by setting the corresponding `pose_frame` argument.

Two different `pose_frame` values can be chosen:

1. **Camera frame** (`camera`). The plane is provided in the camera frame, and no prior knowledge about the pose of the camera in the environment is required. This means that the configured planes move with the camera. It is the user's responsibility to update the configured planes if the camera frame moves (e.g. with a robot-mounted camera).

2. **External frame** (`external`). The plane is provided in the external frame, configured by the user during the hand-eye calibration process. The module relies on the on-board *Hand-eye calibration module* (Section 5.5.1) to retrieve the sensor mounting (static or robot mounted) and the hand-eye transformation.

The plane is not transformed when it is stored, i.e. it must already be given in the desired `pose_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.

5.6.3.4 Services

The PrimitiveDB module is called `rc_primitive_db` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Database* → *Planes*. The user can explore and call the PrimitiveDB module's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the Web GUI.

The PrimitiveDB module offers the following services.

`set_plane`

Persistently stores a plane on the *rc_visard*. An existing plane with the same `id` is overwritten. All configured planes are persistent over firmware updates and rollbacks.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_primitive_db/services/set_plane
```

Request

Details for the definition of a plane are given in *Plane* (Section 5.6.3.2).

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "distance": "float64",
    "id": "string",
    "normal": {
      "x": "float64",
      "y": "float64",
      "z": "float64"
    },
    "pose_frame": "string"
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_plane",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

get_planes

Returns the configured planes with the requested `plane_ids`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_primitive_db/services/get_planes
```

Request

If no `plane_ids` are provided, all configured planes are returned.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "plane_ids": [
      "string"
    ]
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_planes",
  "response": {
    "planes": [
      {
        "distance": "float64",
        "id": "string",
        "normal": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "pose_frame": "string"
      }
    ],
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

delete_planes

Deletes the configured planes with the requested `plane_ids`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_primitive_db/services/delete_planes
```

Request

All planes to be deleted must be explicitly stated in `plane_ids`.

The definition for the request arguments with corresponding datatypes is:

```
{
  "args": {
    "plane_ids": [
      "string"
    ]
  }
}
```

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "delete_planes",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.6.3.5 Return codes

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 5.61: Return codes of the PrimitiveDB module's services

Code	Description
0	Success
-1	An invalid argument was provided
-10	New element could not be added as the maximum storage capacity of planes has been exceeded
10	The maximum storage capacity of planes has been reached
11	An existing plane was overwritten by the call to `set_plane`

5.6.4 GripperDB

5.6.4.1 Introduction

The GripperDB module (gripper database module) is an optional on-board module of the *rc_visard* and is licensed with any of the modules *ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5) or *SilhouetteMatch* (Section 5.4.7). Otherwise it requires a separate CollisionCheck *license* (Section 7.4) to be purchased.

The module provides services to set, retrieve and delete grippers which can then be used for checking collisions with a load carrier or other detected objects (only in combination with *SilhouetteMatch* (Section 5.4.7)). The specified grippers are available for all modules supporting collision checking on the *rc_visard*.

Table 5.62: Specifications of the GripperDB module

Max. number of grippers	50
Supported gripper element geometries	Box, Cylinder, CAD Element
Max. number of elements per gripper	15
Collision checking available in	*ItemPick* (Section 5.4.4) and *BoxPick* (Section 5.4.5), *SilhouetteMatch* (Section 5.4.7)

5.6.4.2 Setting a gripper

The gripper is a collision geometry used to determine whether the gripper is in collision with the load carrier, the point cloud or other objects in the scene when performing the grasp. The gripper consists of up to 15 elements connected to each.

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`.
- `CAD`, with the id `cad.id` of the chosen CAD element.

Each gripper element can be assigned one of the following `function_type` values:

- `NONE`: default, same as empty. This element has no special function and will be considered during collision checking as modelled.
- `FINGER`: This element is a movable finger or jaw and has a `zero_pose` in addition to its default `pose`. It can move linearly from the zero pose towards the default pose by the stroke defined for each grasp.
- `SUCTION_CUP`: This element is a deformable suction cup and will hence be ignored during collision checking. It serves purely for visualization.

Additionally, for each gripper the flange radius, and information about the Tool Center Points (TCPs) have to be defined. A gripper must have at least one TCP but can also have multiple TCPs. The TCPs must be defined in the order of priority for grasping. Each TCP requires an ID `id`, the pose `pose_parent` in the coordinate frame of the parent element defined in `parent_id`. Additionally, each element can define the `min_suction_dimensions` can determine the minimum required dimensions of the suction surface in corresponding to the axes of the TCP. When checking for collisions, all TCPs will be checked in the given order and the ID of the first TCP that is collision free and whose `min_suction_dimensions` requirements are fulfilled by the grasp's available suction surface, will be returned in the `tcp_id` field of the grasp.

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 6.2) or the *rc_visard Web GUI* (Section 6.1).

Robot flange radius

Collisions are checked only with the gripper, 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. 5.45). 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 CollisionCheck module's run-time parameter `check_flange` as described in *Parameter overview* (Section 5.5.2.3).

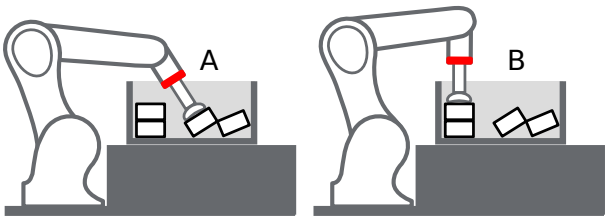


Fig. 5.45: Case A would be marked as collision only if `check_flange` is true, because the robot's flange (red) is inside the load carrier. Case B is collision free independent of `check_flange`.

Uploading gripper CAD elements

A gripper can consist of boxes, cylinders and CAD elements. While boxes and cylinders can be parameterized when the gripper is created, the CAD elements must be uploaded beforehand to be available during gripper creation. A CAD element can be uploaded via the *REST-API interface* (Section 6.2) as described in Section *CAD element API* (Section 5.6.4.5) or via the *rc_visard Web GUI* (Section 6.1). Supported file formats are STEP (*.stp, *.step), STL (*.stl), OBJ (*.obj) and PLY (*.ply). The maximum file size to be uploaded is limited to 10 MB. The files are internally converted to PLY and, if necessary, simplified. The CAD elements can be referenced during gripper creation by their ID.

Creating a gripper via the REST-API or the Web GUI

When creating a gripper via the *REST-API interface* (Section 6.2) or the *Web GUI* (Section 6.1), 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. The coordinate frame of a `CYLINDER` or `BOX` 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. 5.46).

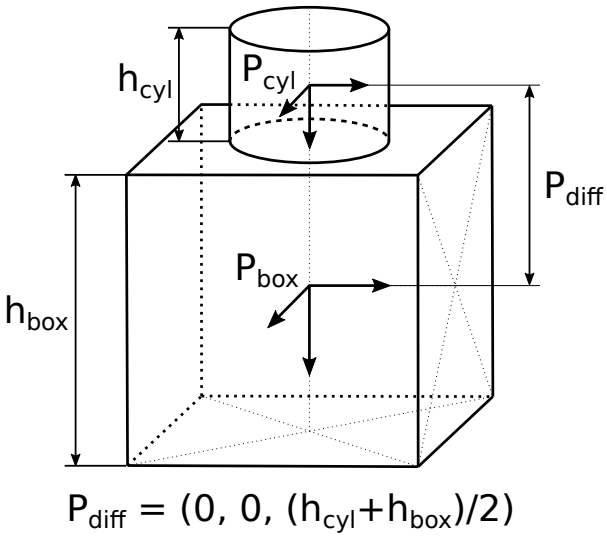


Fig. 5.46: Reference frames for gripper creation via the REST-API and the Web GUI

In case a CAD element is used, the element's origin is defined in the CAD data and is not necessarily located in the center of the element's bounding box.

It is recommended to create a gripper via the Web GUI, because it provides a 3D visualization of the gripper geometry and also allows to automatically attach the child element to the bottom of its parent

element, when the corresponding option for this element is activated. In this case, the elements also stay attached when any of their sizes change. Automatic attachment of CAD elements uses the element's bounding box as reference. Automatic attachment is only possible when the child element is not rotated around the x or y axis with respect to its parent.

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. 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.

Calculated TCP position

After gripper creation via the `set_gripper` service call, the TCP positions in the flange coordinate system are calculated and returned as `pose_flange` in the `tcps` element list. It is important to check if this value is the same as the robot's true TCP position. When creating a gripper in the Web GUI the current TCP position is always displayed in the 3D gripper visualization.

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 GripperDB module corresponds to reality.

5.6.4.3 Services

The GripperDB module is called `rc_gripper_db` in the REST-API and is represented in the *Web GUI* (Section 6.1) under *Database* → *Grippers*. The user can explore and call the GripperDB module's services, e.g. for development and testing, using the *REST-API interface* (Section 6.2) or the Web GUI.

The GripperDB module offers the following services.

`set_gripper`

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

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_gripper_db/services/set_gripper
```

Request

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, or of type 'CAD' with the corresponding `id` in the `cad` field. The pose of each element must be given in the coordinate frame of the parent element (see *Setting a gripper* (Section 5.6.4.2) 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. Each element can have a `function_type` which is either `NONE`, `FINGER` or `SUCTION_CUP`. Elements of type `FINGER` additionally need a `zero_pose` whose orientation has to be the same as in the element's `pose`. `SUCTION_CUP` elements cannot have child elements.

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

`id`: unique name of the gripper

`tcps`: list of TCPs for the gripper. Each element must have an `id` and the pose `pose_parent` of the TCP with respect to the coordinate frame of the element specified in `parent_id`. Optionally, the `min_suction_dimensions` can determine the minimum required dimensions of the suction surface in `x`, `y` and `z`, corresponding to the axes of the TCP. The order of the list determines the priority of the TCPs.

`tcp_parent_id`: deprecated, use the `parent_id` in `tcps` instead. ID of the element on which the TCP is defined.

`tcp_pose_parent`: deprecated, use the `pose_parent` in `tcps` instead. The pose of the TCP with respect to the coordinate frame of the element specified in `tcp_parent_id`.

The definition for the request arguments with corresponding datatypes is:


```
{
  "args": {
    "elements": [
      {
        "box": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "cad": {
          "id": "string"
        },
        "cylinder": {
          "height": "float64",
          "radius": "float64"
        },
        "finger_zero_pose": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "function_type": "string",
        "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_parent": {
      "orientation": {
        "w": "float64",
        "x": "float64",
        "y": "float64",
        "z": "float64"
      },
      "position": {
        "x": "float64",
        "y": "float64",
        "z": "float64"
      }
    },
    "tcps": [
      {
        "id": "string",
        "min_suction_dimensions": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "parent_id": "string",
        "pose_parent": {
          "orientation": {
            "w": "float64",
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        }
      }
    ]
  }
}
```

Response

`gripper`: returns the gripper as defined in the request with an additional field `pose_flange` in `tcps`. 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.

The definition for the response with corresponding datatypes is:

```
{
  "name": "set_gripper",
  "response": {
    "gripper": {
      "elements": [
        {
          "box": {
```

(continues on next page)

(continued from previous page)


```
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "cad": {
            "id": "string"
          },
          "cylinder": {
            "height": "float64",
            "radius": "float64"
          },
          "finger_zero_pose": {
            "orientation": {
              "w": "float64",
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "position": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            }
          },
          "function_type": "string",
          "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": {
```


(continues on next page)

(continued from previous page)


```
          "w": "float64",
          "x": "float64",
          "y": "float64",
          "z": "float64"
        },
        "position": {
          "x": "float64",
          "y": "float64",
          "z": "float64"
        }
      },
      "tcps": [
        {
          "id": "string",
          "min_suction_dimensions": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "parent_id": "string",
          "pose_flange": {
            "orientation": {
              "w": "float64",
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "position": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            }
          },
          "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"
    }
  }
}
```


get_grippers

Returns the configured grippers with the requested `gripper_ids`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_gripper_db/services/get_grippers
```

Request

If no `gripper_ids` are provided, all configured grippers are returned.

The definition for the request arguments with corresponding datatypes is:

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

Response

`elements`: list of geometric elements for the gripper

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

`id`: unique name of the gripper

`tcps`: list of TCPs for the gripper. `pose_flange` contains the pose of the TCP with respect to the robot flange.

`tcp_parent_id`: ID of the element on which the first TCP is defined (only for backwards compatibility, use `tcps` instead).

`tcp_pose_parent`: The pose of the first TCP with respect to the coordinate frame of the element specified in `tcp_parent_id` (only for backwards compatibility, use `tcps` instead).

`tcp_pose_flange`: The pose of the first TCP with respect to the robot flange (only for backwards compatibility, use `tcps` instead).

The definition for the response with corresponding datatypes is:

```
{
  "name": "get_grippers",
  "response": {
    "grippers": [
      {
        "elements": [
          {
            "box": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "cad": {
              "id": "string"
            },
            "cylinder": {
              "height": "float64",
              "radius": "float64"
            },
            "finger_zero_pose": {
              "orientation": {
                "w": "float64",
```

(continues on next page)

(continued from previous page)


```
            "x": "float64",
            "y": "float64",
            "z": "float64"
          },
          "position": {
            "x": "float64",
            "y": "float64",
            "z": "float64"
          }
        },
        "function_type": "string",
        "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"
      }
    },
    "tcps": [
      {
        "id": "string",
```


(continues on next page)

(continued from previous page)


```
            "min_suction_dimensions": {
              "x": "float64",
              "y": "float64",
              "z": "float64"
            },
            "parent_id": "string",
            "pose_flange": {
              "orientation": {
                "w": "float64",
                "x": "float64",
                "y": "float64",
                "z": "float64"
              },
              "position": {
                "x": "float64",
                "y": "float64",
                "z": "float64"
              }
            },
            "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"
      }
    }
}
```


delete_grippers

Deletes the configured grippers with the requested `gripper_ids`.

Details

This service can be called as follows.

```
PUT http://<host>/api/v2/nodes/rc_gripper_db/services/delete_grippers
```

Request

All grippers to be deleted must be explicitly stated in `gripper_ids`.

The definition for the request arguments with corresponding datatypes is:

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

Response

The definition for the response with corresponding datatypes is:

```
{
  "name": "delete_grippers",
  "response": {
    "return_code": {
      "message": "string",
      "value": "int16"
    }
  }
}
```

5.6.4.4 Return codes

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 5.63: Return codes of the GripperDB 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

5.6.4.5 CAD element API

For gripper CAD element upload, download, listing and removal, special REST-API endpoints are provided. CAD elements can also be uploaded, downloaded and removed via the Web GUI. Up to 50 CAD elements can be stored persistently on the *rc_visard*.

The maximum file size to be uploaded is limited to 10 MB.

`GET /cad/gripper_elements`
Get list of all CAD gripper elements.

Template request

```
GET /api/v2/cad/gripper_elements HTTP/1.1
```

Template response

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

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

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *GripperElement* (Section 6.2.4)

`GET /cad/gripper_elements/{id}`

Get a CAD gripper element. If the requested content-type is application/octet-stream, the gripper element is returned as file.

Template request

```
GET /api/v2/cad/gripper_elements/<id> HTTP/1.1
```

Template response

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

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

Parameters

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

Response Headers

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

Status Codes

- 200 OK – successful operation *(returns GripperElement)*
- 404 Not Found – element not found

Referenced Data Models

- *GripperElement* (Section 6.2.4)

`PUT /cad/gripper_elements/{id}`

Create or update a CAD gripper element.

Template request

```
PUT /api/v2/cad/gripper_elements/<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 element *(required)*

Form Parameters

- **file** – CAD file *(required)*

Request Headers

- Accept – multipart/form-data application/json

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation *(returns GripperElement)*
- 400 Bad Request – CAD is not valid or max number of elements reached
- 404 Not Found – element not found
- 413 Request Entity Too Large – File too large

Referenced Data Models

- *GripperElement* (Section 6.2.4)

DELETE /cad/gripper_elements/{id}

Remove a CAD gripper element.

Template request

```
DELETE /api/v2/cad/gripper_elements/<id> HTTP/1.1
Accept: application/json application/ubjson
```

Parameters

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

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation
- 404 Not Found – element not found

6 Interfaces

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

- *Web GUI* (Section 6.1)

  Easy-to-use graphical interface to configure the *rc_visard*, do calibrations, view live images, do service calls, visualize results, etc.

- *REST-API interface* (Section 6.2)

  API to configure the *rc_visard*, query status information, do service calls, etc.

- *rc_dynamics streams* (Section 6.3)

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

- *Generic Robot Interface* (Section 6.4)

  TCP socket communication interface for configuring the *rc_visard* and for service calls.

- *OPC UA interface* (Section 6.5)

  OPC UA interface for configuring the *rc_visard* and for service calls.

- *KUKA Ethernet KRL Interface* (Section 6.6)

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

- *GigE Vision 2.0/GenICam* (Section 6.7)

  Images and camera related configuration.

- *gRPC image stream interface* (Section 6.8)

  Stream synchronized image sets via gRPC.

- *Time synchronization* (Section 6.9)

  Time synchronization between the *rc_visard* and the application host.

6.1 Web GUI

The *rc_visard*'s Web GUI can be used to test, calibrate, and configure the device.

6.1.1 Accessing the Web GUI

The Web GUI 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 4.3).

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`.

6.1.2 Exploring the Web GUI

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

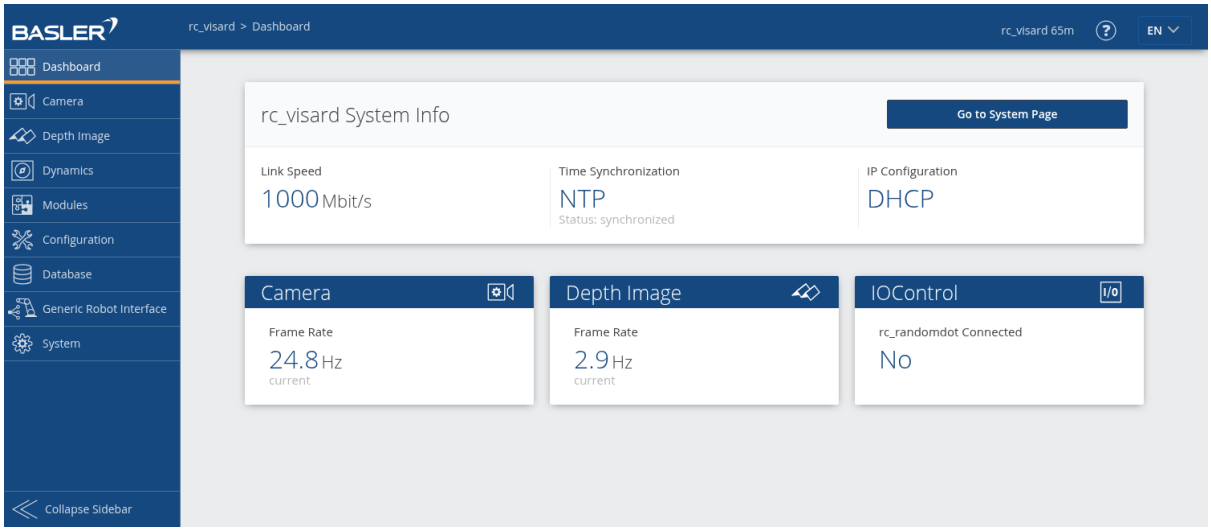


Fig. 6.1: Dashboard page of the *rc_visard*'s Web GUI

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

Camera shows a live stream of the rectified camera images and allows changing camera parameters. See *Camera module* (Section 5.1) for more information.

Depth Image shows a live stream of the left rectified, disparity, and confidence images. The page contains various settings for depth-image computation and filtering. See *Stereo matching module* (Section 5.2) 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 5.3.2.1) for more information.

Modules gives access to the detection modules of the *rc_visard* (see *Detection & Measure modules*, Section 5.4).

Configuration gives access to the configuration modules of the *rc_visard* (see *Configuration modules*, Section 5.5).

Database gives access to the database modules of the *rc_visard* (see *Database modules*, Section 5.6).

Generic Robot Interface shows the jobs and hand-eye calibration configurations defined for the Generic Robot Interface.

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

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

6.1.3 Web GUI access control

The Web GUI has a simple mechanism to lock the UI against casual and accidental changes.

When enabling Web GUI access control via the *System* page, you will be asked to set a password. Now the Web GUI is in a locked mode indicated by the lock symbol in the top bar. All pages, camera streams, parameters and detections can be inspected as usual, but changes are not possible.

To temporarily unlock the Web GUI and make changes, click the lock symbol and enter the password. While enabling or disabling Web GUI access control affects anyone accessing this *rc_visard*, the unlocked state is only valid for the browser where it was unlocked and indicated by the open lock symbol. It is automatically locked again after 10 minutes of inactivity.

Web GUI access control can also be disabled again on the *System* page after providing the current password.

Warning: This is not a security feature! It only locks the Web GUI and not the REST-API. It is meant to prevent accidental and casual changes e.g. via a connected screen.

Note: In case the password is lost, this can be disabled via the REST-API *delete ui_lock* (Section 6.2.3.3).

6.1.4 Downloading camera images

The Web GUI provides an easy way 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* page. This snapshot contains:

- the rectified 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 pipeline_status.json file containing information about all 3D-camera, detection and configuration nodes running on the *rc_visard*,
- a backup.json file containing the settings of the *rc_visard* including grippers, load carriers and regions of interest,
- a system_info.json file containing system information about the *rc_visard*.

The filenames contain the timestamps.

6.1.5 Downloading depth images and point clouds

The Web GUI provides an easy way to download the depth data of the current scene as a .tar.gz file by clicking on the camera icon below the image live streams on the *Depth Image* page. This snapshot contains:

- the rectified left and right camera images in full resolution as .png files,
- an image parameter file corresponding to the left image containing the camera matrix, image dimensions, exposure time, gain value and the stereo baseline,

- the disparity, error and confidence images in the resolution corresponding to the currently chosen quality as .png files,
- a disparity parameter file corresponding to the disparity image containing the camera matrix, image dimensions, exposure time, gain value and the stereo baseline, and information about the disparity values (i.e. invalid values, scale, offset),
- the current IMU readings as imu.csv file,
- a pipeline_status.json file containing information about all 3D-camera, detection and configuration nodes running on the *rc_visard*,
- a backup.json file containing the settings of the *rc_visard* including grippers, load carriers and regions of interest,
- a system_info.json file containing system information about the *rc_visard*.

The filenames contain the timestamps.

When clicking on the mesh icon below the image live streams on the *Depth Image* page, a snapshot is downloaded which additionally includes a mesh of the point cloud in the current depth quality (resolution) as .ply file.

Note: Downloading a depth snapshot will trigger an acquisition in the same way as clicking on the "Acquire" button on the *Depth Image* page of the Web GUI, and, thus, might affect running applications.

6.2 REST-API interface

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 6.1), 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 modules;
- 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.

6.2.1 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 4.4). 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 6.2.5).

For actual HTTP requests, the **current API version is appended** to the entry point of the API, i.e., `http://<host>/api/v2`.

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 6.2.3) 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/v2/pipelines/0/nodes/rc_stereomatching'
```

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

```
curl -X GET 'http://<host>/api/v2/pipelines/0/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/v2/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/v2/pipelines/0/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

```

(continues on next page)

(continued from previous page)


```
{
    "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 6.2.3) and to each *software module's* (Section 5) parameters and services.

6.2.2 Migration from API version 1

API version 1 has become deprecated with the 22.01 firmware release of the *rc_visard*. The following changes were introduced in API version 2.

- All 3D-camera, navigation, detection and configuration modules which were located under `/nodes` in API version 1 are now under `/pipelines/0/nodes/`, e.g. `/pipelines/0/nodes/rc_camera`.
- Configuring load carriers, grippers and regions of interest is now only possible in the global database modules, which are located under `/nodes`, e.g. `/nodes/rc_load_carrier_db`. The corresponding services in the detection modules have been removed or deprecated.
- Templates can now be accessed under `/templates`, e.g. `/templates/rc_silhouettematch`.

6.2.3 Available resources and requests

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

- **`/nodes`** Access the *rc_visard*'s global *Database modules* (Section 5.6) with their run-time status, parameters, and offered services, for storing data used in multiple modules, such as load carriers, grippers and regions of interest.
- **`/pipelines/0/nodes`** Access the *rc_visard*'s 3D-camera, navigation, detection and configuration *software modules* (Section 5) with their run-time status, parameters, and offered services.
- **`/pipelines`** Access to the status and configuration of the camera pipelines. There is always only one camera pipeline with number 0.
- **`/templates`** Access the object templates on the *rc_visard*.
- **`/cad`** Access the cad elements, e.g. for grippers, on the *rc_visard*.
- **`/datastreams`** Access and manage data streams of the *rc_visard*'s *rc_dynamics interface* (Section 6.3).
- **`/system`** Access the system state, set network configuration,

  and manage licenses as well as firmware updates.
- **`/logs`** Access the log files on the *rc_visard*.
- **`/generic_robot_interface`** Access the job and hec_configs for the Generic Robot Interface on the *rc_visard*.

6.2.3.1 Nodes, parameters, and services

Nodes represent the *rc_visard*'s *software modules* (Section 5), each bundling a certain algorithmic functionality. All available global REST-API database nodes can be listed with their service calls and parameters using

```
curl -X GET http://<host>/api/v2/nodes
```

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

```
curl -X GET http://<host>/api/v2/nodes/rc_load_carrier_db
```

All available 3D camera, navigation, detection and configuration REST-API nodes can be listed with their service calls and parameters using

```
curl -X GET http://<host>/api/v2/pipelines/0/nodes
```

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

```
curl -X GET http://<host>/api/v2/pipelines/0/nodes/rc_camera
```

Status: During run-time, each node offers information about its current status. This includes not only the current **processing status** of the module (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_camera` values can be retrieved using

```
curl -X GET http://<host>/api/v2/pipelines/0/nodes/rc_camera/status
```

Note: The returned **status values** are specific to individual nodes and are documented in the respective *software module* (Section 5).

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/v2/pipelines/0/nodes/rc_stereomatching/parameters
```

Its `quality` parameter could be set to `Full` using

```
curl -X PUT http://<host>/api/v2/pipelines/0/nodes/rc_stereomatching/parameters?quality=Full
```

or equivalently

```
curl -X PUT --header 'Content-Type: application/json' -d '{ "value": "Full" }' http://<host>
↪/api/v2/pipelines/0/nodes/rc_stereomatching/parameters/quality
```

Note: Run-time parameters are specific to individual nodes and are documented in the respective *software module* (Section 5).

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 6.1).

Note: Some parameters exposed via the *rc_visard*'s REST-API are also available from the *GigE Vision 2.0/GenICam image interface* (Section 6.7). 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 a service to restore the default values for all of its parameters.

Services: Most nodes also offer services that can be called via REST-API, e.g., to restore parameters as discussed above, or to start and stop nodes. As an example, the *services of the hand-eye calibration module* (Section 5.5.1.5) could be listed using

```
curl -X GET http://<host>/api/v2/pipelines/0/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 6.2.4). As an example, the stereo matching module can be triggered to do an acquisition by:

```
curl -X PUT --header 'Content-Type: application/json' -d '{ "args": {} }' http://<host>/api/
↪v2/pipelines/0/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 module* (Section 5).

The following list includes all REST-API requests regarding the global database nodes' status, parameters, and services calls:

`GET /nodes`
Get list of all available global nodes.

Template request

```
GET /api/v2/nodes HTTP/1.1
```

Sample response

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

[
  {
    "name": "rc_roi_db",
    "parameters": [],
    "services": [
      "set_region_of_interest",
      "get_regions_of_interest",
      "delete_regions_of_interest",
      "set_region_of_interest_2d",
      "get_regions_of_interest_2d",
      "delete_regions_of_interest_2d"
    ],
    "status": "running"
  },
  {
    "name": "rc_load_carrier_db",
    "parameters": [],
    "services": [
      "set_load_carrier",
      "get_load_carriers",
      "delete_load_carriers"
    ],
    "status": "running"
```

(continues on next page)

(continued from previous page)

```
  },
  {
    "name": "rc_gripper_db",
    "parameters": [],
    "services": [
      "set_gripper",
      "get_grippers",
      "delete_grippers"
    ],
    "status": "running"
  }
]
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *NodeInfo* (Section 6.2.4)

`GET /nodes/{node}`

Get info on a single global node.

Template request

```
GET /api/v2/nodes/<node> HTTP/1.1
```

Sample response

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

{
  "name": "rc_roi_db",
  "parameters": [],
  "services": [
    "set_region_of_interest",
    "get_regions_of_interest",
    "delete_regions_of_interest",
    "set_region_of_interest_2d",
    "get_regions_of_interest_2d",
    "delete_regions_of_interest_2d"
  ],
  "status": "running"
}
```

Parameters

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

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *NodeInfo* (Section 6.2.4)

GET /nodes/{node}/services

Get descriptions of all services a global node offers.

Template request

```
GET /api/v2/nodes/<node>/services HTTP/1.1
```

Template response

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

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

Parameters

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

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Service* (Section 6.2.4)

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

Get description of a global node's specific service.

Template request

```
GET /api/v2/nodes/<node>/services/<service> HTTP/1.1
```

Template response

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

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

Parameters

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

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

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Service* (Section 6.2.4)

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/v2/nodes/<node>/services/<service> HTTP/1.1
Accept: application/json application/ubjson

{}
```

Template response

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

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

Parameters

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

Request JSON Object

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

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Service* (Section 6.2.4)

GET /nodes/{node}/status
Get status of a global node.

Template request

```
GET /api/v2/nodes/<node>/status HTTP/1.1
```

Sample response

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

{
  "status": "running",
  "timestamp": 1503075030.2335997,
  "values": []
}
```

Parameters

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

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *NodeStatus* (Section 6.2.4)

The following list includes all REST-API requests regarding the 3D camera, navigation, detection and configuration nodes' status, parameters, and services calls:

GET /pipelines/{pipeline}/nodes
Get list of all available nodes.

Template request

```
GET /api/v2/pipelines/<pipeline>/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": [
      "reset_defaults",
      "change_state"
    ],
    "status": "idle"
  },
```

(continues on next page)

(continued from previous page)


```
  {
    "name": "rc_camera",
    "parameters": [
      "fps",
      "exp_auto",
      "exp_value",
      "exp_max"
    ],
    "services": [
      "reset_defaults"
    ],
    "status": "running"
  },
  {
    "name": "rc_hand_eye_calibration",
    "parameters": [
      "grid_width",
      "grid_height",
      "robot_mounted"
    ],
    "services": [
      "reset_defaults",
      "set_pose",
      "reset",
      "save",
      "calibrate",
      "get_calibration"
    ],
    "status": "idle"
  },
  {
    "name": "rc_stereo_ins",
    "parameters": [],
    "services": [],
    "status": "idle"
  },
  {
    "name": "rc_stereomatching",
    "parameters": [
      "quality",
      "seg",
      "fill",
      "minconf",
      "mindepth",
      "maxdepth",
      "maxdeptherr"
    ],
    "services": [
      "reset_defaults"
    ],
    "status": "running"
  }
]
```


Parameters

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *NodeInfo* (Section 6.2.4)

GET /pipelines/{pipeline}/nodes/{node}
Get info on a single node.

Template request

```
GET /api/v2/pipelines/<pipeline>/nodes/<node> HTTP/1.1
```

Sample response

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

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

Parameters

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*
- **node** (*string*) – name of the node *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *NodeInfo* (Section 6.2.4)

GET /pipelines/{pipeline}/nodes/{node}/parameters
Get parameters of a node.

Template request

```
GET /api/v2/pipelines/<pipeline>/nodes/<node>/parameters?name=<name> HTTP/1.1
```

Sample response

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

[
  {
```

(continues on next page)

(continued from previous page)


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


Parameters

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*
- **node** (*string*) – name of the node *(required)*

Query Parameters

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

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Parameter* (Section 6.2.4)

PUT /pipelines/{pipeline}/nodes/{node}/parameters

Update multiple parameters.

Template request

```
PUT /api/v2/pipelines/<pipeline>/nodes/<node>/parameters HTTP/1.1
Accept: application/json application/ubjson

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

(continues on next page)

(continued from previous page)

```
  }
]
```

Sample response

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

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

Parameters

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*
- **node** (*string*) – name of the node *(required)*

Request JSON Array of Objects

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

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

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 module.
- 404 Not Found – node not found

Referenced Data Models

- *Parameter* (Section 6.2.4)
- *ParameterNameValue* (Section 6.2.4)

`GET /pipelines/{pipeline}/nodes/{node}/parameters/{param}`
Get a specific parameter of a node.

Template request

```
GET /api/v2/pipelines/<pipeline>/nodes/<node>/parameters/<param> HTTP/1.1
```

Sample response

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

{
  "default": 25,
  "description": "Frames per second in Hertz",
  "max": 25,
  "min": 1,
  "name": "fps",
  "type": "float64",
  "value": 10
}
```

Parameters

- **`pipeline`** (*`string`*) – name of the pipeline (one of `0`) *(required)*
- **`node`** (*`string`*) – name of the node *(required)*
- **`param`** (*`string`*) – name of the parameter *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Parameter* (Section 6.2.4)

`PUT /pipelines/{pipeline}/nodes/{node}/parameters/{param}`
Update a specific parameter of a node.

Template request

```
PUT /api/v2/pipelines/<pipeline>/nodes/<node>/parameters/<param> HTTP/1.1
Accept: application/json application/ubjson

{
  "value": {}
}
```

Sample response

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

{
  "default": 25,
  "description": "Frames per second in Hertz",
  "max": 25,
  "min": 1,
  "name": "fps",
  "type": "float64",
  "value": 10
}
```

Parameters

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*
- **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 application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

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 module.
- 404 Not Found – node or parameter not found

Referenced Data Models

- *ParameterValue* (Section 6.2.4)
- *Parameter* (Section 6.2.4)

GET /pipelines/{pipeline}/nodes/{node}/services

Get descriptions of all services a node offers.

Template request

```
GET /api/v2/pipelines/<pipeline>/nodes/<node>/services HTTP/1.1
```

Sample response

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

[
  {
    "args": {},
    "description": "Restarts the module.",
```

(continues on next page)

(continued from previous page)

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

Parameters

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*
- **node** (*string*) – name of the node *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Service* (Section 6.2.4)

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

Get description of a node's specific service.

Template request

```
GET /api/v2/pipelines/<pipeline>/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",
```

(continues on next page)

(continued from previous page)


```
        "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

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*
- **node** (*string*) – name of the node *(required)*
- **service** (*string*) – name of the service *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Service* (Section 6.2.4)

PUT /pipelines/{pipeline}/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/v2/pipelines/<pipeline>/nodes/<node>/services/<service> HTTP/1.1
Accept: application/json application/ubjson

{}
```

Sample response


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

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


(continues on next page)

(continued from previous page)

```
  }
}
```

Parameters

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*
- **node** (*string*) – name of the node *(required)*
- **service** (*string*) – name of the service *(required)*

Request JSON Object

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

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *Service* (Section 6.2.4)

GET /pipelines/{pipeline}/nodes/{node}/status

Get status of a node.

Template request

```
GET /api/v2/pipelines/<pipeline>/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

- **pipeline** (*string*) – name of the pipeline (one of `0`) *(required)*
- **node** (*string*) – name of the node *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *NodeStatus* (Section 6.2.4)

6.2.3.2 Datastreams

The following resources and requests allow access to and configuration of the *rc_dynamics interface* data streams (Section 6.3). 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/v2/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": [
```

(continues on next page)

(continued from previous page)


```
      "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 application/ubjson

Status Codes

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

Referenced Data Models

- *Stream* (Section 6.2.4)

`GET /datastreams/{stream}`

Get datastream configuration.

Template request

```
GET /api/v2/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 application/ubjson

Status Codes

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

Referenced Data Models

- *Stream* (Section 6.2.4)

`PUT /datastreams/{stream}`

Update a datastream configuration.

Template request

```
PUT /api/v2/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 application/ubjson

Status Codes

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

Referenced Data Models

- *Stream* (Section 6.2.4)

`DELETE /datastreams/{stream}`

Delete a destination from the datastream configuration.

Template request

```
DELETE /api/v2/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 application/ubjson

Status Codes

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

Referenced Data Models

- *Stream* (Section 6.2.4)

6.2.3.3 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 module-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/v2/logs HTTP/1.1
```

Sample response

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

[
  {
```

(continues on next page)

(continued from previous page)


```
    "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 application/ubjson

Status Codes

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

Referenced Data Models

- *LogInfo* (Section 6.2.4)

`GET /logs/{log}`

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

Template request

```
GET /api/v2/logs/<log>?format=<format>&limit=<limit> HTTP/1.1
```

Sample response


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

{
  "date": 1503060035.2115774,
  "log": [
    {
      "component": "rc_stereo_ins",
      "level": "INFO",
      "message": "Running rc_stereo_ins version 2.4.0",
      "timestamp": 1503060034.083
    },
    {
      "component": "rc_stereo_ins",
      "level": "INFO",
      "message": "Starting up communication interfaces",
      "timestamp": 1503060034.085
    },
    {
      "component": "rc_stereo_ins",
      "level": "INFO",
```


(continues on next page)

(continued from previous page)


```
      "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 6.2.4)

`GET /system`

Get system information on sensor.

Template request

```
GET /api/v2/system HTTP/1.1
```

Sample response

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

{
  "dns": {
    "dns_servers": [
      "10.0.0.1",
      "1.1.1.1"
    ],
    "manual_dns_servers": [
      "1.1.1.1"
    ]
  },
  "firmware": {
    "active_image": {
      "image_version": "1.1.0"
    },
    "fallback_booted": true,
    "inactive_image": {
      "image_version": "1.0.0"
    },
    "next_boot_image": "active_image"
  },
  "hostname": "rc-visard-02873515",
  "link_speed": 1000,
  "mac": "00:14:2D:2B:D8:AB",
  "ntp": {
    "enabled": true,
    "manual_ntp_servers": [
      "10.0.0.1"
    ],
    "offset": -3.2666e-05,
    "selected_ntp_servers": [
      "10.0.0.1"
    ],
    "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 application/ubjson

Status Codes

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

Referenced Data Models

- *SysInfo* (Section 6.2.4)

GET /system/backup

Get backup.

Template request

```
GET /api/v2/system/backup?pipelines=<pipelines>&load_carriers=<load_carriers>&regions_of_
↪interest=<regions_of_interest>&grippers=<grippers> HTTP/1.1
```

Query Parameters

- **pipelines** (*boolean*) – backup pipelines with node settings, i.e. parameters and preferred_orientation (default: `True`) *(optional)*
- **load_carriers** (*boolean*) – backup load_carriers (default: `True`) *(optional)*
- **regions_of_interest** (*boolean*) – backup regions_of_interest (default: `True`) *(optional)*
- **grippers** (*boolean*) – backup grippers (default: `True`) *(optional)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

POST /system/backup

Restore backup.

Template request

```
POST /api/v2/system/backup HTTP/1.1
Accept: application/json application/ubjson

{}
```

Sample response

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

{
  "return_code": {
    "message": "backup restored",
    "value": 0
  },
  "warnings": []
}
```

Request JSON Object

- **backup** (*object*) – backup data as json object *(required)*

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

GET /system/dns

Get DNS settings.

Template request

```
GET /api/v2/system/dns HTTP/1.1
```

Sample response

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

{
  "dns": {
    "dns_servers": [
      "10.0.0.1",
      "1.1.1.1"
    ],
    "manual_dns_servers": [
      "1.1.1.1"
    ]
  }
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *DNS* (Section 6.2.4)

PUT /system/dns

Set manual DNS servers.

Template request

```
PUT /api/v2/system/dns HTTP/1.1
Accept: application/json application/ubjson

{}
```

Sample response

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

{
  "dns": {
    "dns_servers": [
```

(continues on next page)

(continued from previous page)

```
      "10.0.0.1",
      "1.1.1.1"
    ],
    "manual_dns_servers": [
      "1.1.1.1"
    ]
  }
}
```

Request JSON Object

- **manual_dns_servers** (*ManualDNSServers*) – Manual DNS servers *(required)*

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation *(returns DNS)*
- 400 Bad Request – invalid/missing arguments

Referenced Data Models

- *DNS* (Section 6.2.4)
- *ManualDNSServers* (Section 6.2.4)

GET /system/license

Get information about licenses installed on sensor.

Template request

```
GET /api/v2/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 application/ubjson

Status Codes

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

Referenced Data Models

- *LicenseInfo* (Section 6.2.4)

POST /system/license
Update license on sensor with a license file.

Template request

```
POST /api/v2/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/max_power_test
Get last max power test result.

Template request

```
GET /api/v2/system/max_power_test HTTP/1.1
```

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – successful operation

POST /system/max_power_test
Run max power test. Fully load GPU (and CPU) to consume max power for 10 seconds to test the power supply. WARNING: The system might not return a response due to immediate reboot if the power supply is insufficient.

Template request

```
POST /api/v2/system/max_power_test?nocpu=<nocpu> HTTP/1.1
```

Query Parameters

- **nocpu** (*boolean*) – Don't run CPU workers and only load the GPU. *(optional)*

Response Headers

- Content-Type – application/json

Status Codes

- 200 OK – Test finished. See return_code for result.
- 400 Bad Request – Test already running.

GET /system/network
Get current network configuration.

Template request

```
GET /api/v2/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 application/ubjson

Status Codes

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

Referenced Data Models

- *NetworkInfo* (Section 6.2.4)

`GET /system/network/settings`
Get current network settings.

Template request

```
GET /api/v2/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 application/ubjson

Status Codes

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

Referenced Data Models

- *NetworkSettings* (Section 6.2.4)

PUT /system/network/settings
Set current network settings.

Template request

```
PUT /api/v2/system/network/settings HTTP/1.1
Accept: application/json application/ubjson

{}
```

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 application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

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 6.2.4)

GET /system/ntp
Get NTP settings.

Template request

```
GET /api/v2/system/ntp HTTP/1.1
```

Sample response

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

{
  "ntp": {
    "enabled": true,
    "manual_ntp_servers": [
      "10.0.0.1"
    ],
```

(continues on next page)

(continued from previous page)

```
    "offset": -3.2666e-05,
    "selected_ntp_servers": [
      "10.0.0.1"
    ],
    "synchronized": true
  }
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *NTP* (Section 6.2.4)

`PUT /system/ntp`

Set manual NTP servers.

Template request

```
PUT /api/v2/system/ntp HTTP/1.1
Accept: application/json application/ubjson

{}
```

Sample response

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

{
  "ntp": {
    "enabled": true,
    "manual_ntp_servers": [
      "10.0.0.1"
    ],
    "offset": -3.2666e-05,
    "selected_ntp_servers": [
      "10.0.0.1"
    ],
    "synchronized": true
  }
}
```

Request JSON Object

- **`manual_ntp_servers`** (*`ManualNTPServers`*) – Manual NTP servers *(required)*

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

- 400 Bad Request – invalid/missing arguments

Referenced Data Models

- *NTP* (Section 6.2.4)
- *ManualNTPServers* (Section 6.2.4)

PUT /system/reboot

Reboot the sensor.

Template request

```
PUT /api/v2/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/v2/system/rollback HTTP/1.1
```

Sample response

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

{
  "active_image": {
    "image_version": "1.1.0"
  },
  "fallback_booted": false,
  "inactive_image": {
    "image_version": "1.0.0"
  },
  "next_boot_image": "active_image"
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *FirmwareInfo* (Section 6.2.4)

PUT /system/rollback

Rollback to previous firmware version (inactive system image).

Template request

```
PUT /api/v2/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/time
Get system time in UTC as string with format “YYYY-MM-DD hh:mm:ss”

Template request

```
GET /api/v2/system/time HTTP/1.1
```

Sample response

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

{
  "utc": "2023-10-05 08:35:26"
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

PUT /system/time
Set system time in UTC as string with format “YYYY-MM-DD hh:mm:ss”

Template request

```
PUT /api/v2/system/time?utc=<utc> HTTP/1.1
```

Sample response

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

{
  "utc": "2023-10-05 08:35:26"
}
```

Query Parameters

- **utc** (*string*) – Time in UTC as string with format “YYYY-MM-DD hh:mm:ss” *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation
- 400 Bad Request – invalid/missing arguments
- 403 Forbidden – Changing time forbidden because time is synchronized via NTP or PTP.

GET /system/ui_lock
Get UI lock status.

Template request

```
GET /api/v2/system/ui_lock HTTP/1.1
```

Sample response

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

{
  "enabled": false
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *UILock* (Section 6.2.4)

`DELETE /system/ui_lock`

Remove UI lock.

Template request

```
DELETE /api/v2/system/ui_lock HTTP/1.1
```

Sample response

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

{
  "enabled": false,
  "valid": false
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

`POST /system/ui_lock`

Verify or set UI lock.

Template request

```
POST /api/v2/system/ui_lock?hash=<hash>&set=<set> HTTP/1.1
```

Sample response

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

{
  "enabled": true,
  "valid": true
}
```

Query Parameters

- **hash** (*string*) – hash of the UI lock password *(required)*
- **set** (*boolean*) – set new hash instead of veryfing *(optional)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

GET /system/update

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

Template request

```
GET /api/v2/system/update HTTP/1.1
```

Sample response

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

{
  "active_image": {
    "image_version": "1.1.0"
  },
  "fallback_booted": false,
  "inactive_image": {
    "image_version": "1.0.0"
  },
  "next_boot_image": "active_image"
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

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

Referenced Data Models

- *FirmwareInfo* (Section 6.2.4)

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/v2/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

6.2.4 Data type definitions

The REST-API defines the following data models, which are used to access or modify *the available resources* (Section 6.2.3) either as required attributes/parameters of the requests or as return types.

DNS: DNS settings.

An object of type DNS has the following properties:

- **dns_servers** (array of string)
- **manual_dns_servers** (array of string)

Template object

```
{
  "dns_servers": [
    "string",
    "string"
  ],
  "manual_dns_servers": [
    "string",
    "string"
  ]
}
```

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

- *GET /system/dns*
- *PUT /system/dns*

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 occurred
- **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*

GripperElement: CAD gripper element

An object of type GripperElement has the following properties:

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

Template object

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

GripperElement objects are used in the following requests:

- *GET /cad/gripper_elements*
- *GET /cad/gripper_elements/{id}*
- *PUT /cad/gripper_elements/{id}*

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 module 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 modules. The respective flag is true if the module is unlocked with the currently applied software license.

An object of type LicenseComponents has the following properties:

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

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

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 modules.

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) - module 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*

ManualDNSServers: List of manual DNS servers.

An object of type ManualDNSServers has the following properties:

- **manual_dns_servers** (array of string)

Template object

```
{
  "manual_dns_servers": [
    "string",
    "string"
  ]
}
```

ManualDNSServers objects are used in the following requests:

- *PUT /system/dns*

ManualNTPServers: List of manual NTP servers.

An object of type ManualNTPServers has the following properties:

- **manual_ntp_servers** (array of string)

Template object

```
{
  "manual_ntp_servers": [
    "string",
    "string"
  ]
}
```

ManualNTPServers objects are used in the following requests:

- *PUT /system/ntp*

NTP: Status of the NTP time sync.

An object of type NTP has the following properties:

- **enabled** (boolean) - NTP is enabled
- **manual_ntp_servers** (array of string)
- **offset** (string) - time sync offset reported by NTP
- **selected_ntp_servers** (array of string)
- **synchronized** (boolean) - synchronized with NTP server

Template object

```
{
  "enabled": false,
  "manual_ntp_servers": [
    "string",
    "string"
  ],
  "offset": "string",
  "selected_ntp_servers": [
    "string",
    "string"
  ],
  "synchronized": false
}
```

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

- *GET /system/ntp*
- *PUT /system/ntp*

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,
  "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`, `idle`, `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}*
- *GET /pipelines/{pipeline}/nodes*
- *GET /pipelines/{pipeline}/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`, `idle`, `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*
- *GET /pipelines/{pipeline}/nodes/{node}/status*

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 /pipelines/{pipeline}/nodes/{node}/parameters*
- *PUT /pipelines/{pipeline}/nodes/{node}/parameters*
- *GET /pipelines/{pipeline}/nodes/{node}/parameters/{param}*
- *PUT /pipelines/{pipeline}/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 /pipelines/{pipeline}/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 /pipelines/{pipeline}/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}*
- *GET /pipelines/{pipeline}/nodes/{node}/services*
- *GET /pipelines/{pipeline}/nodes/{node}/services/{service}*
- *PUT /pipelines/{pipeline}/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: Representation 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:

- **dns** (*DNS*) - see description of *DNS*
- **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** (*NTP*) - see description of *NTP*
- **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
- **ui_lock** (*UILock*) - see description of *UILock*
- **uptime** (float) - system uptime in seconds

Template object


```
{
  "dns": {
    "dns_servers": [
      "string",
      "string"
    ],
    "manual_dns_servers": [
      "string",
      "string"
    ]
  },
  "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": {
```


(continues on next page)

(continued from previous page)


```
      "dhcp_enabled": false,
      "persistent_default_gateway": "string",
      "persistent_ip_address": "string",
      "persistent_ip_enabled": false,
      "persistent_subnet_mask": "string"
    },
    "subnet_mask": "string"
  },
  "ntp": {
    "enabled": false,
    "manual_ntp_servers": [
      "string",
      "string"
    ],
    "offset": "string",
    "selected_ntp_servers": [
      "string",
      "string"
    ],
    "synchronized": false
  },
  "ptp_status": {
    "master_ip": "string",
    "offset": 0,
    "offset_dev": 0,
    "offset_mean": 0,
    "state": "string"
  },
  "ready": false,
  "serial": "string",
  "time": 0,
  "ui_lock": {
    "enabled": false
  },
  "uptime": 0
}
```


SysInfo objects are used in the following requests:

- *GET /system*

Template: Detection 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 /templates/rc_boxpick*
- *GET /templates/rc_boxpick/{id}*
- *PUT /templates/rc_boxpick/{id}*
- *GET /templates/rc_silhouettematch*
- *GET /templates/rc_silhouettematch/{id}*
- *PUT /templates/rc_silhouettematch/{id}*

UILock: UI lock status.

An object of type UILock has the following properties:

- **enabled** (boolean)

Template object

```
{
  "enabled": false
}
```

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

- *GET /system/ui_lock*

6.2.5 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 6.2.3) 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.

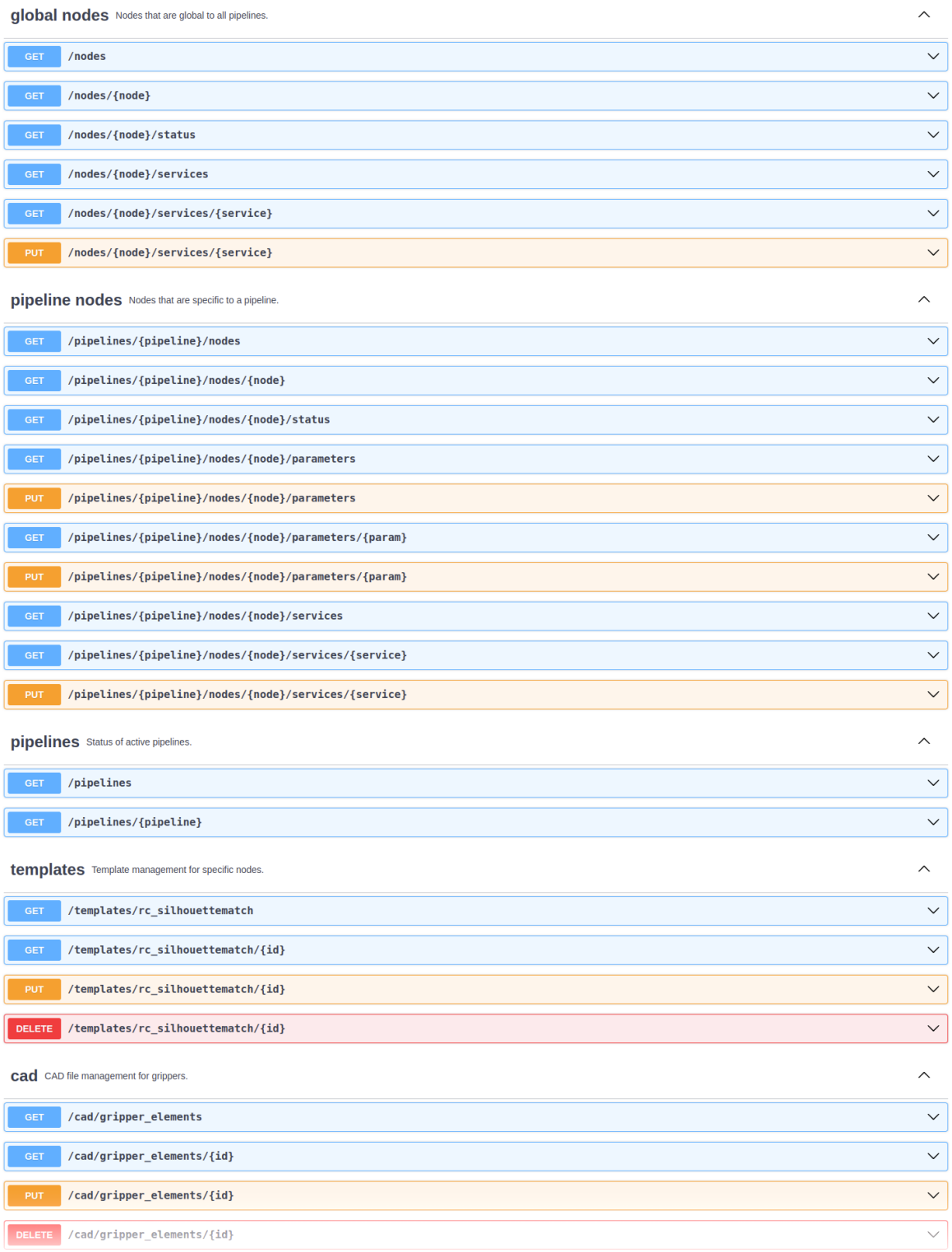


Fig. 6.2: Initial view of the *rc_visard*'s Swagger UI with its resources and requests

Using this interface, available resources and requests can be explored by clicking on them to uncollapse or recollapse them. The following figure shows an example of how to get a node's current status by filling in the necessary parameters (`pipeline` number and `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.

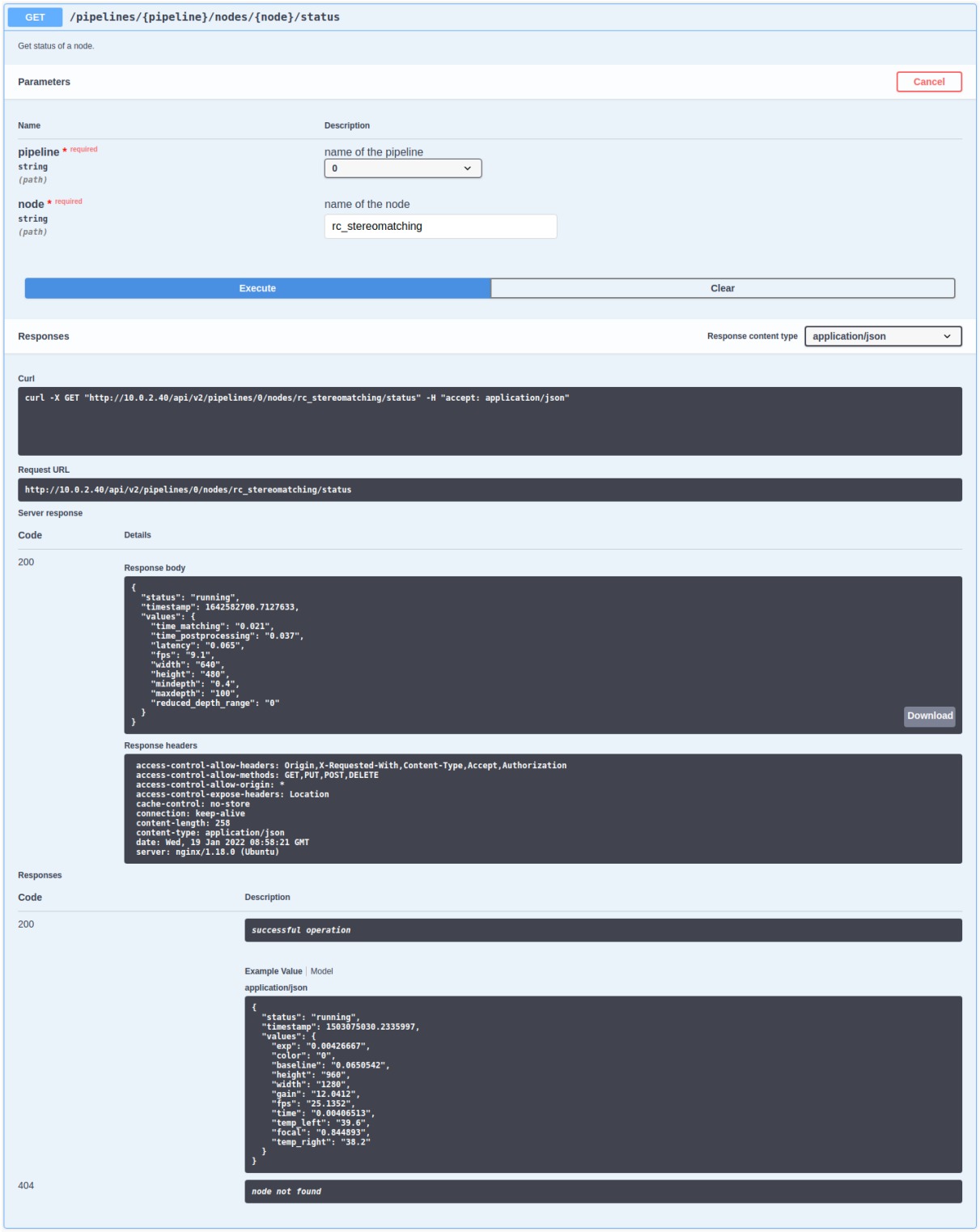


Fig. 6.3: Result of requesting the `rc_stereomatching` node's status

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.

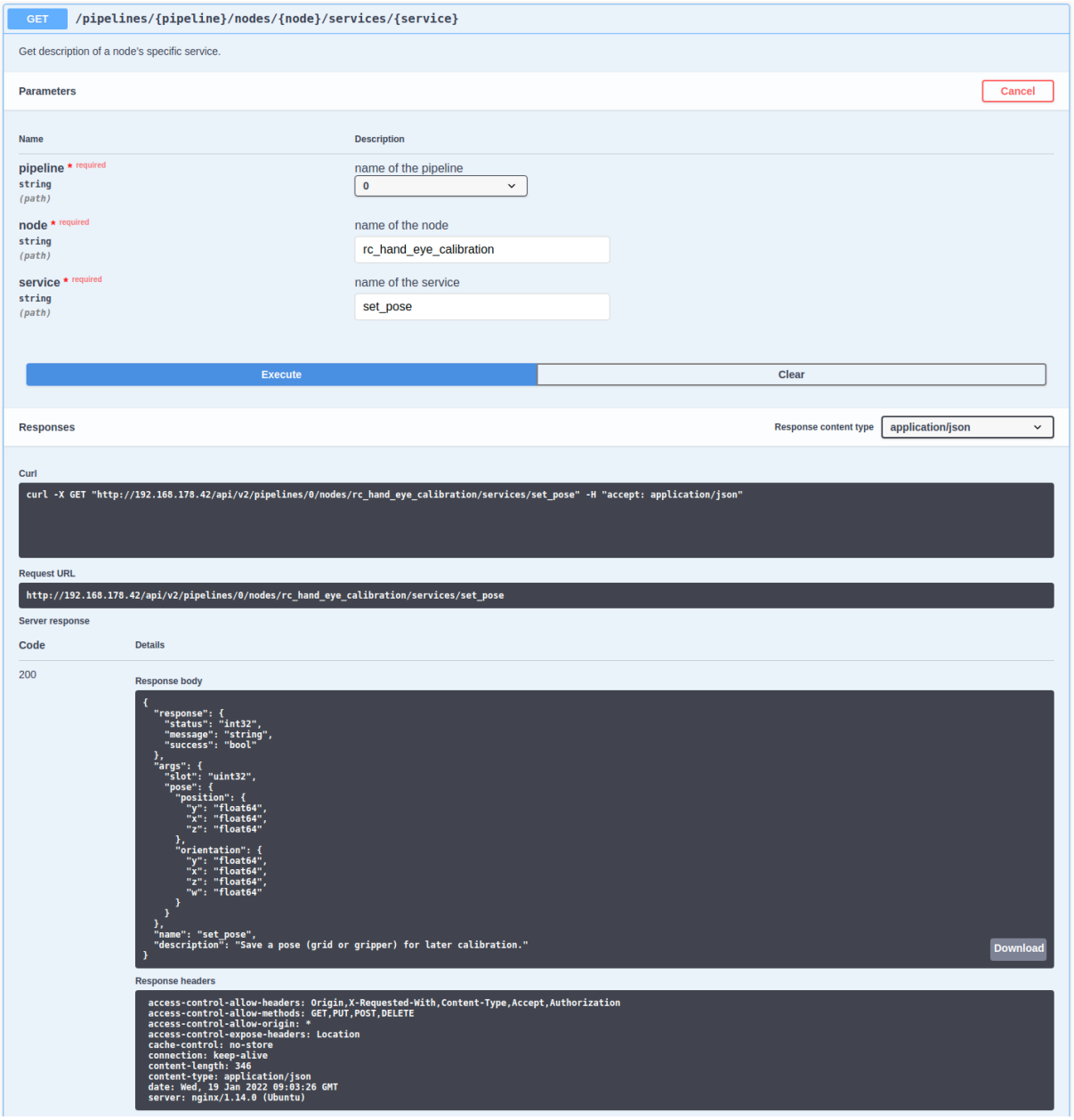


Fig. 6.4: The result of the `GET` request on the `set_pose` service shows the required arguments for this service call.

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

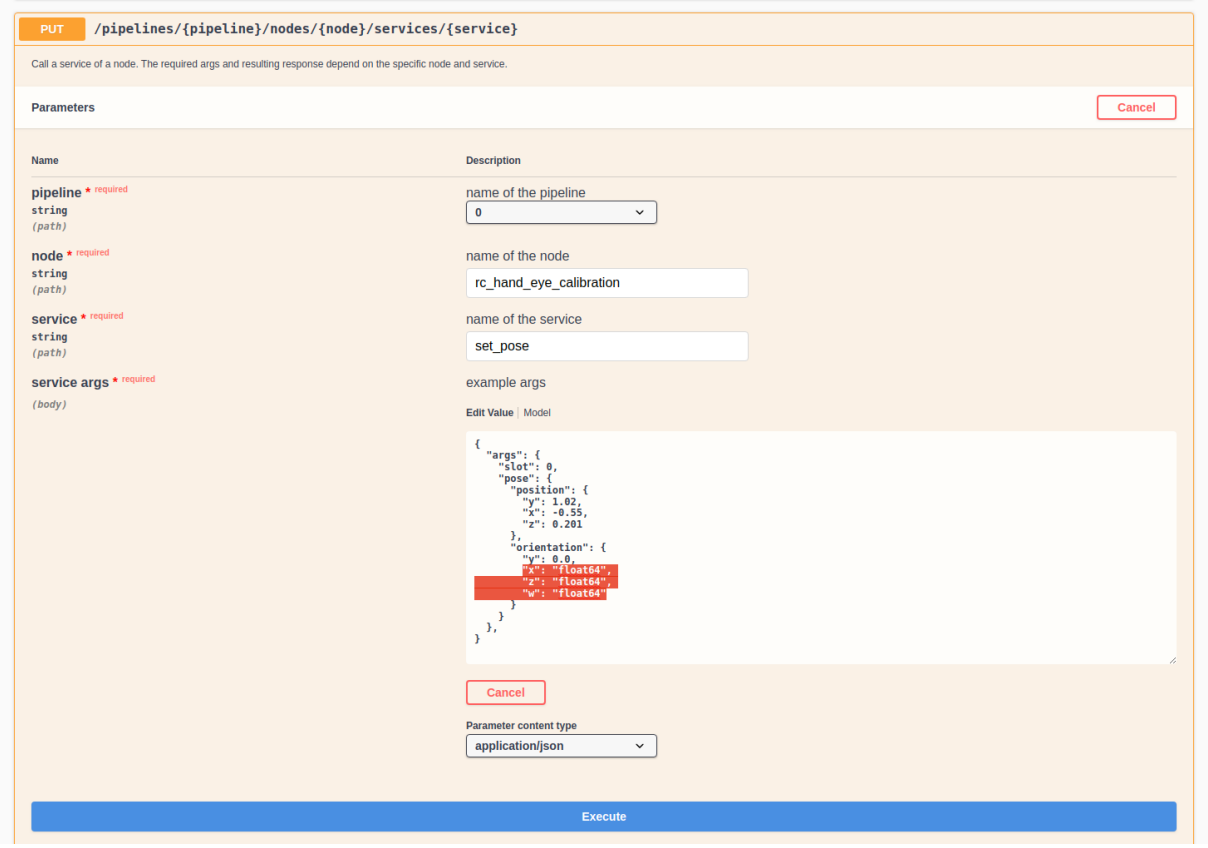


Fig. 6.5: Filling in the arguments of the `set_pose` service request

6.3 The rc_dynamics interface

The rc_dynamics interface offers continuous, real-time data-stream access to *rc_visard*'s several *dynamic state estimates* (Section 5.3.1.2) 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 6.3.3) used is agnostic w.r.t. operating system and programming language.

6.3.1 Starting/stopping dynamic-state estimation

The *rc_visard*'s dynamic-state estimates are only available if the respective module, i.e., the *sensor dynamics module* (Section 5.3.1), is turned on. This can be done either in the Web GUI - a respective switch is offered on the *Dynamics* page - or via the REST-API by using the module'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://<host>/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.

6.3.2 Configuring data streams

Available data streams, i.e., dynamic-state estimates, can be listed and configured by the *rc_visard*'s *REST-API* (Section 6.2.3.2), 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 5.3.1.2).

Table 6.1: Available data streams via the rc_dynamics interface

Name	Protocol	Protobuf	Description
dynamics	UDP	*Dynamics*	Dynamics of sensor (pose, velocity, acceleration) from INS or SLAM (best effort depending on availability) at realtime frequency (IMU rate)
dynamics_ins	UDP	*Dynamics*	Dynamics of sensor (pose, velocity, acceleration) from stereo INS at realtime frequency (IMU rate)
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://<host>/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 6.3.3), 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://<host>/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.

6.3.3 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 5.3.1.2) 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 5.3.1.2) 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 5.3.1.2) is serialized using the `Imu` message type:

```
message Imu
{
  optional Time timestamp                    = 1; // Time when the data was␣
↪captured
  optional Vector3d linear_acceleration      = 2; // Linear acceleration in m/
↪s^2 measured by the IMU
  optional Vector3d angular_velocity         = 3; // Angular velocity in rad/
↪s measured by the IMU
}
```

- The nested types `PoseStamped`, `Pose`, `Time`, `Quaternion`, and `Vector3D` are defined as follows:

```
message PoseStamped
{
  optional Time timestamp  = 1; // Time when the data was captured
  optional Pose pose       = 2;
}
```

```
message Pose
{
  optional Vector3d position      = 1; // Position in meters
  optional Quaternion orientation = 2; // Orientation as unit quaternion
  repeated double covariance      = 3 [packed=true]; // Row-major␣
↪representation of the 6x6 covariance matrix (x, y, z, rotation about X axis,␣
↪rotation about Y axis, rotation about Z axis)
}
```

```
message Time
{
  /// \brief Seconds
  optional int64 sec = 1;

  /// \brief Nanoseconds
  optional int32 nsec = 2;
}
```

```
message Quaternion
{
  optional double x = 2;
  optional double y = 3;
  optional double z = 4;
  optional double w = 5;
}
```

```
message Vector3d
{
  optional double x = 1;
  optional double y = 2;
  optional double z = 3;
}
```

6.3.4 rc_dynamics_api

The open-source `rc_dynamics_api` package provides a simple, convenient C++ wrapper to request and parse rc_dynamics streams.

6.4 Generic Robot Interface

The Generic Robot Interface (GRI) is an integration layer that bridges the *REST-API v2* (Section 6.2) and provides a standardized way to communicate with the software modules using simple TCP socket communication on port 7100. It can be activated via a separate *license* (Section 7.4).

6.4.1 Overview

The GRI is used in two phases. In the configuration phase, the user creates *jobs* and stores them under unique numeric IDs. A job contains everything needed for one operation: the module and service to

call, the arguments to use, and the data to return to the robot. Jobs, and optionally *hand-eye calibration configurations*, are prepared once and are defined via the Web GUI or the REST-API (see *Job types*).

At run time, the robot triggers these jobs by their ID over a TCP socket on port 7100, exchanging fixed-size binary messages (see *GRI binary protocol specification*). The GRI internally manages the REST-API communication and returns the selected pose results in a format that can be chosen specifically for the robot. A typical job trigger proceeds as follows:

1. The robot sends a trigger *action* (`TRIGGER_JOB_SYNC` or `TRIGGER_JOB_ASYNC`, see *Actions*) together with the job ID, optionally providing run-time values in the request `data` fields.
2. The GRI looks up the job and translates it into the corresponding REST-API call(s) to the selected module.
3. The results of the module, such as grasp poses, are cached by the GRI, and the first pose is returned to the robot in the chosen *pose format*.
4. The robot retrieves any further results one at a time with `GET_NEXT_POSE` and `GET_RELATED_POSE`.

A job definition therefore serves as a reusable template that the robot triggers by its ID at run time. The following section describes how to define jobs; the binary protocol and the details of robot integration are described further below.

6.4.2 Job definition

Jobs are pre-configured tasks that can be triggered by the robot application. Each job has a unique ID and contains all the necessary information for a specific operation, e.g. computing grasps for bin picking or changing run-time parameters of a module. Once configured, the robot can execute these jobs using simple socket commands and, if applicable, receive the returned poses.

The jobs can be defined via the Web GUI or via the REST-API (see *Job and HEC_config API*).

In the Web GUI, a pipeline service job is created with *Create Job* in the *Generic Robot Interface* panel of the *Try Out* section of the respective module, using the settings currently configured there. *Load Job* loads a saved job back into the *Try Out* section for editing. When saving the job, its additional return values, dynamic arguments and mapping tables (see *Returning poses and extra data*) can be defined and edited with *Configure Extra Data*. Parameter setting jobs can be created and edited on the Web GUI's *Generic Robot Interface* page.

6.4.2.1 Job types

The Generic Robot Interface supports three types of jobs, selected via the `job_type` field:

- `CALL_PIPELINE_SERVICE` – call a service on a specific camera pipeline (e.g. to detect objects or compute grasps) and return pose data to the robot.
- `CALL_GLOBAL_SERVICE` – call a service that is not tied to a pipeline (e.g. database services for regions of interest, load carriers or grippers). Returns a pose only if explicitly configured to.
- `SET_PIPELINE_PARAMETERS` – set run-time parameters on a pipeline node. Returns no pose.

The fields `selected_return`, `pose_path`, `additional_return` and `dynamic_args` control *which* pose a job returns and how the robot can influence the call. They are described in detail in *Returning poses and extra data* and demonstrated on complete jobs in *Worked examples*.

Pipeline service job (`CALL_PIPELINE_SERVICE`)

This job calls a service on a specific camera pipeline, e.g. to detect objects or compute grasps, and returns pose data (e.g. grasp poses) to the robot.

Field	Presence	Description
job_type	required	the job type CALL_PIPELINE_SERVICE
name	required	descriptive name to distinguish jobs
pipeline	required	the camera pipeline to use for the job (e.g. "0")
node	required	the REST-API name of the pipeline node (e.g. rc_itempick)
service	required	the REST-API name of the service to call (e.g. compute_grasps)
args	required	the REST-API JSON arguments passed to the service
selected_return	required	the REST-API response field holding the poses to return (e.g. grasps or matches); see *Choosing the returned pose: selected_return and pose_path*
pose_path	optional	the field inside the selected_return object that holds the pose (defaults to pose); see *Choosing the returned pose: selected_return and pose_path*
additional_return	optional	extra response fields to return in data_4 through data_10; see *additional_return (module to robot)*
dynamic_args	optional	fill args from the robot request fields data_1 through data_4; see *dynamic_args (robot to module)*

A minimal pipeline service job definition is:

```
{
   "args": {
      "pose_frame": "external",
      "suction_surface_length": 0.02,
      "suction_surface_width": 0.02
   },
   "job_type": "CALL_PIPELINE_SERVICE",
   "name": "Compute Grasps",
   "node": "rc_itempick",
   "pipeline": "0",
   "selected_return": "grasps",
   "service": "compute_grasps"
}
```

For pipeline service jobs, the GRI adds the robot pose sent with the trigger request to the service arguments as robot_pose (position in meters, orientation as quaternion). A robot_pose given in args is overwritten. Global service jobs do not receive the robot pose.

The available values for selected_return depend on the chosen node and can be e.g. grasps or matches. Refer to the service definitions of the corresponding module for details about node, service, args and selected_return.

Global service job (CALL_GLOBAL_SERVICE)

This job calls a service that is not tied to a specific pipeline, e.g. database services for setting or retrieving regions of interest, load carriers or grippers. By default, global service jobs do not return any pose; they return a pose only if selected_return (and, typically, pose_path) is configured.

Field	Presence	Description
job_type	required	the job type CALL_GLOBAL_SERVICE
name	required	descriptive name to distinguish jobs
node	required	the REST-API name of the global node (e.g. rc_load_carrier_db)
service	required	the REST-API name of the service to call
args	required	the REST-API JSON arguments passed to the service
selected_return	optional	the REST-API response field holding the pose(s) to return (e.g. to retrieve gripper or TCP definitions); without it the job returns no pose; see *Choosing the returned pose: selected_return and pose_path*
pose_path	optional	the field inside the selected_return object that holds the pose (defaults to pose); see *Choosing the returned pose: selected_return and pose_path*
additional_return	optional	extra response fields to return in data_4 through data_10; see *additional_return (module to robot)*
dynamic_args	optional	fill args from the robot request fields data_1 through data_4; see *dynamic_args (robot to module)*

The field pipeline must not be set for global service jobs.

A minimal global service job definition is:

```
{
   "args": {
      "region_of_interest_2d": {
         "id": "2d_roi",
         "width": 526,
         "height": 501,
         "offset_x": 558,
         "offset_y": 307
      }
   },
   "job_type": "CALL_GLOBAL_SERVICE",
   "name": "Set 2D ROI",
   "node": "rc_roi_db",
   "service": "set_region_of_interest_2d"
}
```

Refer to the service definitions of the corresponding module for details about node, service and args.

Parameter setting job (SET_PIPELINE_PARAMETERS)

This job sets run-time parameters on a pipeline node, e.g. for adjusting camera or detection module settings. Parameter setting jobs do not return any pose.

Field	Presence	Description
job_type	required	the job type SET_PIPELINE_PARAMETERS
name	required	descriptive name to distinguish jobs
pipeline	required	the camera pipeline to use for the job (e.g. "0")
node	required	the REST-API name of the pipeline node (e.g. rc_stereomatching)
parameters	required	the run-time parameters to set as key-value pairs

A sample parameter setting job definition is:

```
{
   "job_type": "SET_PIPELINE_PARAMETERS",
```

(continues on next page)

(continued from previous page)

```
   "name": "Set Stereo Parameters",
   "node": "rc_stereomatching",
   "parameters": {
      "maxdepth": 2,
      "quality": "High"
   },
   "pipeline": "0"
}
```

When a parameter setting job is triggered, `data_1` of the response is `0` if all parameters were set and `-1` otherwise; in the latter case `error_code` is `API_RESPONSE_ERROR`. Unlike for the other job types, `data_1` is not a node's `return_code`. `GET_JOB_STATUS` reports `data_1` as `0` for these jobs.

Refer to the run-time parameter definitions of the corresponding module for details about `node` and `parameters`.

6.4.2.2 Primary and related objects

Detection results often come in two linked kinds of objects. The *primary objects* are the objects named by the job's `selected_return`, e.g. `grasps`. The *related objects* are the objects that a primary object corresponds to, e.g. the `items` or `matches` a grasp was computed for.

The robot iterates over the primary objects one at a time with `GET_NEXT_POSE`. After each primary object, it can request the corresponding related object with `GET_RELATED_POSE` (see *Actions*). The link between the two is established by the REST-API response itself: for example, each `grasp` carries an `item_uuid` that matches the `uuid` of exactly one `item`.

A primary object *grasp* therefore has exactly one related object, an *item* or a *match*. Conversely, a primary object *match* can have several related *grasps*. Because the relation is resolved per primary object, two grasps that belong to the same item each return that item on `GET_RELATED_POSE`; the GRI does not filter out these repeated related objects.

6.4.2.3 Returning poses and extra data

Each result that a job delivers carries one pose in the response's pose fields (`pos_x` through `rot_4`). In addition, Protocol V1 leaves several request and response fields unused, and the job configuration uses them to exchange additional information between the robot and the REST-API:

- `selected_return` and `pose_path` choose *which* pose is returned.
- `additional_return` returns extra values from the REST-API response in the response fields `data_4` through `data_10` (module to robot, see *Response body definition*).
- `dynamic_args` injects robot-provided values from the request fields `data_1` through `data_4` into the `args` of the REST-API call (robot to module, see *Request body definition*).
- `maps` translates between the integers the robot exchanges and the string values the REST-API uses.

`additional_return`, `dynamic_args` and `maps` can also be defined and edited in the Web GUI with *Configure Extra Data* when saving a job.

All of these locate values inside the REST-API response or request through a `json_path`, which is explained first.

Locating values with `json_path`

A `json_path` addresses a single value inside the REST-API response (or, for `dynamic_args`, inside the request arguments). Two forms exist:

- **Object-relative** (no leading dot, e.g. `quality`): resolved *inside* the primary or related object that is currently being returned by `GET_NEXT_POSE` / `GET_RELATED_POSE`.
- **Root-relative** (leading dot, e.g. `.timestamp.sec`): resolved from the root of the entire REST-API response, which is useful for global metadata or for a parent property.

To descend into a list, use bracket notation with a zero-based index, e.g. `item_models[0].type`.

The following annotated response shows how each form resolves, assuming the job returns `grasps` as primary objects and `items` as related objects:

```
{
   "timestamp": {
      "sec": 1719900000
   },
   "grasps": [
      {
         "quality": 0.85,
         "item_uuid": "abc-123"
      }
   ],
   "items": [
      {
         "uuid": "abc-123",
         "bounding_box": { "x": 0.15 }
      }
   ]
}
```

- `quality` (object-relative, `primary` scope) resolves to `0.85`.
- `bounding_box.x` (object-relative, `related` scope) resolves to `0.15`.
- `.timestamp.sec` (root-relative) resolves to `1719900000`.

Note: Always use bracket notation (`item_models[0]`) to access a list element. Writing `item_models.0` makes the GRI search for a dictionary key literally named `"0"` instead of the first list element.

Because `GET_NEXT_POSE` and `GET_RELATED_POSE` return one object at a time, an object-relative `json_path` is written for a *single* object: use `quality`, not `grasps[0].quality`.

Choosing the returned pose: `selected_return` and `pose_path`

`selected_return` names the response field holding the primary objects to iterate over, e.g. `grasps`, `matches`, `items` or `load_carriers`. It can also index into a list nested deeper in the response using bracket notation, for example to iterate over the TCPs of a specific gripper:

```
{
   "selected_return": "grippers[0].tcps",
   "pose_path": "pose_flange"
}
```

`pose_path` tells the GRI where, *inside* each selected object, to find the pose (an object with `position` and `orientation`) to return to the robot. It follows the same path rules as `json_path` and defaults to `pose`, which is why it is normally omitted: all built-in pipeline return types (`grasps`, `matches`, `items`, `load_carriers`) expose their pose in a field named `pose`.

Set `pose_path` explicitly when the selected object has no `pose` field, or when it exposes more than one candidate pose and a specific one has to be chosen. This is the case in the gripper example above, where each TCP has separate `pose_flange` and `pose_parent` fields instead of a single `pose`. The same value is applied to both the primary and the related object, so it only needs to differ from the default

when the relevant field is not named `pose`. If the resolved value is missing, or is not an object with both a `position` and an `orientation`, the job is rejected as misconfigured (`MISCONFIGURED_JOB`).

`additional_return` (module to robot)

`additional_return` maps extra values from the REST-API response into the response fields `data_4` through `data_10`. Its mappings are grouped by the object they apply to:

- `primary`: applied to the primary object returned by `GET_NEXT_POSE` (the object type named by `selected_return`, e.g. `grasps`).
- `related`: applied to the corresponding related object returned by `GET_RELATED_POSE` (see *Primary and related objects*).

Each mapping is an object with the following fields:

- `target_field`: the response field to write to (`data_4` through `data_10`)
- `json_path`: where to read the value in the REST-API response (see *Locating values with json_path*)
- `type`: how to convert the value into the `int32` response field:
  - `int`: written as an integer (the default if `type` is omitted); a value with a fractional part is rejected with `TYPE_CONVERSION_ERROR`
  - `float`: multiplied by `1,000,000` (analogous to the pose scaling in *Pose formats*)
  - `bool`: `1` or `0`
  - `map`: translated to an integer via a `maps` lookup table (see *Mapping tables (maps)*)

If a `json_path` does not resolve to a value, that field is simply skipped (`MISSING_ADDITIONAL_DATA` warning, see *Error codes and semantics*). A failed `map` lookup or type conversion instead aborts the whole response with an error (`UNMAPPED_VALUE_ERROR` / `TYPE_CONVERSION_ERROR`).

If a job has no `selected_return`, there are no primary objects to iterate over. `primary` mappings are then applied directly to the `TRIGGER_JOB_SYNC` response, and their `json_path` values are resolved from the root of the REST-API response. This allows returning values from global service jobs that do not return a pose.

`dynamic_args` (robot to module)

`dynamic_args` maps values from the robot request into the `args` of the REST-API service call, letting the robot adjust detection arguments at trigger time without reconfiguring the job. It applies to `CALL_PIPELINE_SERVICE` and `CALL_GLOBAL_SERVICE` jobs; `SET_PIPELINE_PARAMETERS` jobs do not use it. It is a list of entries, each with:

- `source_field`: the request field to read from (`data_1` through `data_4`, see *Request body definition*)
- `json_path`: the target key path within the REST-API `args` (same path rules as *Locating values with json_path*)
- `type`: how to convert the value before injecting it:
  - `int`: injected as-is (the default if `type` is omitted)
  - `float`: divided by `1,000,000`
  - `bool`: interpreted as `1` / `0`
  - `map`: the integer sent by the robot is looked up in a `maps` table to inject the corresponding value

The values are applied on top of the static `args` when the job is triggered, so a `json_path` pointing at an existing argument overwrites it. A failed `map` lookup or type conversion is rejected with an error (`UNMAPPED_VALUE_ERROR` / `TYPE_CONVERSION_ERROR`, see *Error codes and semantics*).

Mapping tables (maps)

Some arguments and return values are strings (such as IDs) rather than numbers. maps translates between the integers the robot exchanges and these string values on the GRI server. The same table is used in both directions – dynamic_args looks up *robot ID to value*, additional_return looks up *value to robot ID* – so each table must be a 1:1 (bijective) mapping. A dynamic_args or additional_return entry of type map selects the table to use via a map field:

```
{
   "dynamic_args": [
      { "source_field": "data_2", "json_path": "template_id", "type": "map", "map": "models" }
   ],
   "maps": {
      "models": {
         "1": "part_A_large",
         "2": "part_B_small"
      }
   }
}
```

If the robot sends data_2 = 2, the template ID "part_B_small" is injected into the REST-API call.

6.4.2.4 Worked examples

The following complete job definitions combine the options above. For each one, the configuration is shown first, followed by what the robot sends or receives at run time.

Example 1: pipeline service job with additional return

This ItemPick job returns grasp poses and, alongside each pose, extra information about the grasp (primary scope) and about the item the grasp was computed for (related scope).

```
{
   "name": "ItemPickAIJob",
   "job_type": "CALL_PIPELINE_SERVICE",
   "pipeline": "0",
   "node": "rc_itempick",
   "service": "compute_grasps",
   "selected_return": "grasps",
   "args": {
      "collision_detection": {
         "gripper_id": "MultiSuctionGripper",
         "pre_grasp_offset": { "x": 0, "y": 0, "z": -0.1 }
      },
      "item_models": [ { "type": "BAG" } ],
      "pose_frame": "external",
      "region_of_interest_id": "ROI",
      "suction_surface_length": 0.05,
      "suction_surface_width": 0.05
   },
   "additional_return": {
      "primary": [
         { "json_path": "quality", "target_field": "data_4", "type": "float" },
         { "json_path": "tcp_id",  "target_field": "data_5", "type": "map", "map": "gripper_tcps"␣
↪}
      ],
      "related": [
         { "json_path": "bounding_box.x", "target_field": "data_6", "type": "float" },
         { "json_path": "bounding_box.y", "target_field": "data_7", "type": "float" },
```

(continues on next page)

(continued from previous page)

```
        { "json_path": "bounding_box.z", "target_field": "data_8", "type": "float" }
      ]
   },
   "maps": {
      "gripper_tcps": { "1": "TCP1", "2": "TCP2", "3": "TCP3" }
   }
}
```

- `selected_return` is `grasps`, so each `GET_NEXT_POSE` returns one grasp pose.
- `additional_return.primary` reads two values from that grasp: its `quality` into `data_4` (as a float) and its `tcp_id` into `data_5` (translated to an integer through the `gripper_tcps` table).
- `additional_return.related` reads the bounding box of the associated item into `data_6` through `data_8`, available after `GET_RELATED_POSE`.

At run time, for a grasp with `quality = 0.85` and `tcp_id = "TCP2"` on an item with bounding box `(0.15, 0.10, 0.05)` m:

- `GET_NEXT_POSE` returns the grasp pose, `data_4 = 850000` (`0.85 x 1,000,000`) and `data_5 = 2` (the key mapped to `"TCP2"` in `gripper_tcps`).
- `GET_RELATED_POSE` returns the item pose, `data_6 = 150000`, `data_7 = 100000` and `data_8 = 50000` (each bounding-box dimension x `1,000,000`).

Example 2: global service job returning a pose

Global service jobs return no pose by default. This job sets `selected_return` and `pose_path` so that it returns the TCP poses stored for a gripper in the gripper database.

```
{
   "name": "GetMultiSuctionGripperTCPs",
   "job_type": "CALL_GLOBAL_SERVICE",
   "node": "rc_gripper_db",
   "service": "get_grippers",
   "selected_return": "grippers[0].tcps",
   "pose_path": "pose_flange",
   "args": {
      "gripper_ids": [ "MultiSuctionGripper" ]
   },
   "additional_return": {
      "primary": [
         { "json_path": "id", "target_field": "data_4", "type": "map", "map": "tcp_names" },
         { "json_path": ".grippers[0].flange_radius", "target_field": "data_5", "type": "float" }
      ]
   },
   "maps": {
      "tcp_names": { "1": "TCP1", "2": "TCP2", "3": "TCP3" }
   }
}
```

- `selected_return` uses index addressing, `grippers[0].tcps`, to iterate over the TCPs of the first returned gripper as the primary objects.
- `pose_path` is set to `pose_flange` because a TCP has no plain `pose` field but separate `pose_flange` and `pose_parent` poses; `pose_flange` is the one returned.
- `additional_return.primary` mixes both path forms: the object-relative `id` (the current TCP's ID, mapped to an integer via `tcp_names`) and the root-relative `.grippers[0].flange_radius` (a property of the parent gripper, the same for every TCP).

At run time, for a TCP with `id = "TCP1"` on a gripper with `flange_radius = 0.04` m, `GET_NEXT_POSE` returns the TCP's `pose_flange`, `data_4 = 1` (`"TCP1"` via `tcp_names`) and `data_5 = 40000` (`0.04 x 1,000,000`).

Example 3: pipeline service job with dynamic arguments

This BoxPick job lets the robot choose the rectangle dimensions to detect at trigger time, so the same job can pick different box sizes without being reconfigured. The four dimensions of the `RECTANGLE` item model are filled from the request fields `data_1` through `data_4`.

```
{
   "name": "BoxPickDynamicRectangle",
   "job_type": "CALL_PIPELINE_SERVICE",
   "pipeline": "0",
   "node": "rc_boxpick",
   "service": "compute_grasps",
   "selected_return": "grasps",
   "args": {
      "pose_frame": "external",
      "item_models": [
         {
            "type": "RECTANGLE",
            "rectangle": {
               "min_dimensions": { "x": 0.05, "y": 0.05 },
               "max_dimensions": { "x": 0.10, "y": 0.10 }
            }
         }
      ],
      "suction_surface_length": 0.05,
      "suction_surface_width": 0.05
   },
   "dynamic_args": [
      { "source_field": "data_1", "json_path": "item_models[0].rectangle.min_dimensions.x", "type
↪": "float" },
      { "source_field": "data_2", "json_path": "item_models[0].rectangle.min_dimensions.y", "type
↪": "float" },
      { "source_field": "data_3", "json_path": "item_models[0].rectangle.max_dimensions.x", "type
↪": "float" },
      { "source_field": "data_4", "json_path": "item_models[0].rectangle.max_dimensions.y", "type
↪": "float" }
   ],
   "additional_return": {
      "primary": [
         { "json_path": "quality", "target_field": "data_4", "type": "float" }
      ]
   }
}
```

- `dynamic_args` uses the same `json_path` grammar as `additional_return`, here indexing into `item_models[0]` to reach the rectangle dimensions inside `args`.
- The dimensions given in `args` are placeholders: the values injected from the request overwrite them when the job is triggered.
- The request fields `data_1` through `data_4` (filled by `dynamic_args`) and the response fields `data_4` through `data_10` (filled by `additional_return`) belong to separate messages, so reusing `data_4` for both the sent minimum-x dimension and the returned grasp `quality` is not a conflict.

At run time, if the robot triggers the job with `data_1 = 50000`, `data_2 = 50000`, `data_3 = 100000` and `data_4 = 100000`, the GRI divides each value by `1,000,000` and calls `rc_boxpick/compute_grasps` with `min_dimensions (0.05, 0.05)` m and `max_dimensions (0.10, 0.10)` m. Each subsequent `GET_NEXT_POSE` then returns a grasp pose with `data_4` set to that grasp's `quality x 1,000,000`.

6.4.2.5 Execution modes

The Generic Robot Interface supports two execution modes to optimize the robot's cycle time:

- Synchronous Execution: The robot triggers a job and waits for the first result to arrive. This mode should be chosen when results are required immediately.
- Asynchronous Execution: The robot starts a job and can continue with other operations while the job is running in the background. The job status can be queried and results can be retrieved when ready. This mode maximizes efficiency during long detection times.

6.4.3 Hand-Eye Calibration

A hand-eye calibration configuration can be defined for each camera pipeline to allow for programmatic hand-eye calibration using the GRI. Each hand-eye calibration configuration consists of the following information:

- `grid_height`: height of the calibration grid in meters
- `grid_width`: width of the calibration grid in meters
- `robot_mounted`: boolean that determines whether the camera is mounted on the robot
- `tcp_offset`: 0 for 6DOF robots. For 4DOF robots: 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.
- `tcp_rotation_axis`: -1 for 6DOF robots. For 4DOF robots: determines 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).

More detailed information about these settings and the hand-eye calibration in general is given in *Hand-eye calibration*.

The hand-eye calibration configurations can be set via the Web GUI or via the REST-API (see *Job and HEC_config API*).

6.4.4 GRI binary protocol specification

This specification defines the exact on-wire format for client-server messages. A message consists of a fixed 8-byte header and a body whose layout depends on the protocol version. Currently, there is only protocol version 1.

Note: All multi-byte integers are **little-endian**. Types are `uint8` (8-bit unsigned), `int16` (16-bit signed), `int32` (32-bit signed).

6.4.4.1 Message header (8 bytes)

Table 6.2: Message header definition

Field	Type	Size	Description
magic_number	uint32	4	ASCII tag *"GRI0"*, bytes *47 52 49 00* (little-endian)
protocol_version	uint8	1	Protocol version: currently *1*
message_length	uint8	1	Total message size (bytes), incl. header + body
pose_format	uint8	1	Pose data format (see *Pose formats*)
action	uint8	1	Command/action (see *Actions*)

6.4.4.2 Pose formats

The GRI always uses **millimeters** for representing a position. The following tables show different rotation formats that can be chosen to match to the rotation representation of the used robot. The formats are grouped by non-Euler rotation formats, Tait-Bryan-Euler rotation formats (all three axes are used) and proper Euler rotation formats (first and last rotation axis are the same).

Table 6.3: Non-Euler rotation formats

Name	Value	rot_1	rot_2	rot_3	rot_4	Units	Robot Example
QUAT_WXYZ	1	w	x	y	z	–	ABB
QUAT_XYZW	2	x	y	z	w	–	Fruitcore HORST
AXIS_ANGLE_RAD	3	rx	ry	rz	–	rad	Universal Robots

In the following notation primes indicate successive rotations in the intrinsic frame (e.g., Y' = rotation about Y-axis after first rotation). `_B` and `_F` determine the order in which the rotation components are given. `F` stands for *forward*, meaning that the rotation components are given in the same order as the rotation is applied, and `B` stands for *backward*, meaning the rotation components are given in reverse order. `_RAD` and `_DEG` determine whether the rotation components are given in radians or degrees, respectively, if applicable. So the format `EULER_ZYX_B_DEG` means that the intrinsic rotation order is z-y'-x" (first rotate around the z axis, then rotate around the new y axis, then rotate around the new x axis), the order in which the rotation components are given is backward (so the first rotation element is the angle around the x axis), and the angels are given in degrees.

Table 6.4: Tait-Bryan-Euler rotation formats. Primes indicate successive rotations in the intrinsic frame (e.g., Y' = rotation about Y-axis after first rotation). **_F (Forward)**: *[1st, 2nd, 3rd]* | **_B (Backward)**: *[3rd, 2nd, 1st]*, **_DEG: degrees** | **_RAD: radian**.

Name	Value	rot_1	rot_2	rot_3	rot_4	Units	Robot Example
EULER_XYZ_F_DEG	4	X	Y'	Z"	–	deg	
EULER_XYZ_F_RAD	5	X	Y'	Z"	–	rad	
EULER_XYZ_B_DEG	6	Z"	Y'	X	–	deg	
EULER_XYZ_B_RAD	7	Z"	Y'	X	–	rad	
EULER_XZY_F_DEG	8	X	Z'	Y"	–	deg	
EULER_XZY_F_RAD	9	X	Z'	Y"	–	rad	
EULER_XZY_B_DEG	10	Y"	Z'	X	–	deg	
EULER_XZY_B_RAD	11	Y"	Z'	X	–	rad	
EULER_YXZ_F_DEG	12	Y	X'	Z"	–	deg	
EULER_YXZ_F_RAD	13	Y	X'	Z"	–	rad	
EULER_YXZ_B_DEG	14	Z"	X'	Y	–	deg	
EULER_YXZ_B_RAD	15	Z"	X'	Y	–	rad	
EULER_YZX_F_DEG	16	Y	Z'	X"	–	deg	
EULER_YZX_F_RAD	17	Y	Z'	X"	–	rad	
EULER_YZX_B_DEG	18	X"	Z'	Y	–	deg	
EULER_YZX_B_RAD	19	X"	Z'	Y	–	rad	
EULER_ZXY_F_DEG	20	Z	X'	Y"	–	deg	
EULER_ZXY_F_RAD	21	Z	X'	Y"	–	rad	
EULER_ZXY_B_DEG	22	Y"	X'	Z	–	deg	
EULER_ZXY_B_RAD	23	Y"	X'	Z	–	rad	
EULER_ZYX_F_DEG	24	Z	Y'	X"	–	deg	KUKA
EULER_ZYX_F_RAD	25	Z	Y'	X"	–	rad	
EULER_ZYX_B_DEG	26	X"	Y'	Z	–	deg	FANUC, Mitsubishi, Yaskawa
EULER_ZYX_B_RAD	27	X"	Y'	Z	–	rad	

Table 6.5: Euler rotation formats. Primes indicate successive rotations in the intrinsic frame (e.g., Y' = rotation about Y-axis after first rotation). **_F (Forward)**: *[1st, 2nd, 3rd]* | **_B (Backward)**: *[3rd, 2nd, 1st]*, **_DEG: degrees** | **_RAD: radian**.

Name	Value	rot_1	rot_2	rot_3	rot_4	Units	Robot Example
EULER_XYX_F_DEG	28	X	Y'	X"	–	deg	
EULER_XYX_F_RAD	29	X	Y'	X"	–	rad	
EULER_XYX_B_DEG	30	X"	Y'	X	–	deg	
EULER_XYX_B_RAD	31	X"	Y'	X	–	rad	
EULER_XZX_F_DEG	32	X	Z'	X"	–	deg	
EULER_XZX_F_RAD	33	X	Z'	X"	–	rad	
EULER_XZX_B_DEG	34	X"	Z'	X	–	deg	
EULER_XZX_B_RAD	35	X"	Z'	X	–	rad	
EULER_YXY_F_DEG	36	Y	X'	Y"	–	deg	
EULER_YXY_F_RAD	37	Y	X'	Y"	–	rad	
EULER_YXY_B_DEG	38	Y"	X'	Y	–	deg	
EULER_YXY_B_RAD	39	Y"	X'	Y	–	rad	
EULER_YZY_F_DEG	40	Y	Z'	Y"	–	deg	
EULER_YZY_F_RAD	41	Y	Z'	Y"	–	rad	
EULER_YZY_B_DEG	42	Y"	Z'	Y	–	deg	
EULER_YZY_B_RAD	43	Y"	Z'	Y	–	rad	
EULER_ZXZ_F_DEG	44	Z	X'	Z"	–	deg	
EULER_ZXZ_F_RAD	45	Z	X'	Z"	–	rad	
EULER_ZXZ_B_DEG	46	Z"	X'	Z	–	deg	
EULER_ZXZ_B_RAD	47	Z"	X'	Z	–	rad	
EULER_ZYZ_F_DEG	48	Z	Y'	Z"	–	deg	Kawasaki
EULER_ZYZ_F_RAD	49	Z	Y'	Z"	–	rad	
EULER_ZYZ_B_DEG	50	Z"	Y'	Z	–	deg	
EULER_ZYZ_B_RAD	51	Z"	Y'	Z	–	rad	

All pose components (position **and** rotation) are `int32` scaled by **1,000,000**.

- Float to Int: `int = round(float * 1000000)`
- Int to Float: `float = int / 1000000.0`
- Positions in **millimeters** before scaling
- Angles in **degrees/radians** (per format) before scaling
- Quaternions unitless, same scaling
- `rot_4` unused for Euler/axis-angle (set to 0)
- Value range: `int32` limits each scaled component to ±2147.483647, i.e. positions to ±2147.48 mm per axis. Values outside this range cannot be transmitted; the GRI then sends no response.

6.4.4.3 Actions

The following actions can be sent.

Table 6.6: GRI actions

Name	Value	Description
STATUS	1	Get system readiness; maps readiness to `data_2` (1 or 0)
TRIGGER_JOB_SYNC	2	Execute job synchronously
TRIGGER_JOB_ASYNC	3	Start job asynchronously
GET_JOB_STATUS	4	Query async job status (see *Job status*)
GET_NEXT_POSE	5	Retrieve next available result
GET_RELATED_POSE	6	Retrieve next related pose
HEC_INIT	7	Initialize hand-eye calibration
HEC_SET_POSE	8	Provide/store calibration pose
HEC_CALIBRATE	9	Run calibration and save results

STATUS (1)

Returns system readiness information fetched from the *rc_visard* in `data_2` (1 if ready, 0 if not).

TRIGGER_JOB_SYNC (2)

Runs the job and returns the **first** result immediately; additional results are stored for later retrieval. On success with results, `error_code` will be zero and the pose will be filled. If no results are returned, `error_code` will be `NO_POSES_FOUND` (positive warning). It also reports:

- `data_1` = node's `return_code` value
- `data_2` = number of remaining *primary objects* (see *Primary and related objects*)
- `data_3` = number of remaining *related objects* (see *Primary and related objects*)

If the job does not finish within the processing timeout, or the sensor does not answer within the API timeout, `error_code` is `PROCESSING_TIMEOUT` and the job is marked `FAILED`.

TRIGGER_JOB_ASYNC (3)

Starts the job and returns immediately. The job's status can be polled with `GET_JOB_STATUS` (4) (see *Job status*) and the results can be fetched with `GET_NEXT_POSE` (5), as soon as the job is `DONE`.

GET_JOB_STATUS (4)

Returns the job status. It reports:

- `data_1` = node's `return_code` value
- `data_2` = job status (see table *Job status values*)

Error details flow through `error_code`. A job that timed out reports `FAILED` with `error_code` `PROCESSING_TIMEOUT`; for `TRIGGER_JOB_ASYNC` this is the case when the sensor does not answer within the API timeout.

GET_NEXT_POSE (5)

Returns the next result of the *primary object*. It also reports:

- `data_1` = node's `return_code` value
- `data_2` = number of remaining *primary objects* (see *Primary and related objects*)
- `data_3` = number of remaining *related objects* (see *Primary and related objects*)

When no more primary objects are available, it returns `NO_POSES_FOUND`. The job stays `DONE`, so the related objects of the last primary object can still be retrieved with `GET_RELATED_POSE`. If the job is still running, `GET_NEXT_POSE` returns `JOB_STILL_RUNNING`; if it failed or has not been triggered yet, it returns `NO_POSES_FOUND`.

GET_RELATED_POSE (6)

Returns the next pose of the *related object* corresponding to the current *primary object*. It also reports:

- `data_1` = node's `return_code` value

- `data_2` = number of remaining *primary objects* (see *Primary and related objects*)
- `data_3` = number of remaining *related objects* (see *Primary and related objects*)

If no related poses were found, it returns `NO_RELATED_POSES`.

HEC_INIT (7)

This action initializes the hand-eye calibration. It clears any existing calibration data, applies the pipeline's configuration parameters and prepares the system for recording new poses. `data_1` specifies the target pipeline.

HEC_SET_POSE (8)

This action is to be used eight times to record distinct robot poses with visible calibration pattern. The field `data_2` is used to specify the image storage slot (0-7). A previous pose in a slot will be overwritten if a slot is reused. Each pose must provide a different view of the calibration pattern, as described in *Hand-eye calibration*. `data_1` specifies the target pipeline.

HEC_CALIBRATE (9)

This action processes all recorded poses and calculates the transformation between camera and robot. It automatically saves successful calibration results. `data_1` specifies the target pipeline.

6.4.4.4 Job status

The following job status values can be returned.

Table 6.7: Job status values

Name	Value
INACTIVE	1
RUNNING	2
DONE	3
FAILED	4

6.4.4.5 Body definitions

There are different body definitions depending on whether it is a request that is sent or a response that is received. The request body consists of 54 bytes in total and its definition is given in table *Request body definition*.

Table 6.8: Request body definition

Field	Type	Size	Description
header	struct	8	Message header (see *Message header (8 bytes)*)
job_id	uint16	2	Unique job ID from job configuration
pos_x	int32	4	Position X (scaled by 10^6)
pos_y	int32	4	Position Y (scaled by 10^6)
pos_z	int32	4	Position Z (scaled by 10^6)
rot_1	int32	4	Rotation component 1 (scaled by 10^6)
rot_2	int32	4	Rotation component 2 (scaled by 10^6)
rot_3	int32	4	Rotation component 3 (scaled by 10^6)
rot_4	int32	4	Rotation component 4 (scaled by 10^6)
data_1	int32	4	Additional parameter 1
data_2	int32	4	Additional parameter 2
data_3	int32	4	Additional parameter 3
data_4	int32	4	Additional parameter 4

The job ID is the unique identifier from the job configuration. The usage of the fields `data_1...data_4` depends on the action and job. They are set to *0* if unused. For `TRIGGER_JOB_SYNC/TRIGGER_JOB_ASYNC`,

these fields can be filled with robot-provided values via the job's `dynamic_args` configuration, see *Returning poses and extra data*.

The response body consists of 80 bytes in total. Its definition is given in table *Response body definition*.

Table 6.9: Response body definition

Field	Type	Size	Description
header	struct	8	Protocol header
job_id	uint16	2	Processed job number
error_code	int16	2	GRI result status (severity by sign)
pos_x	int32	4	Position X (scaled by 10^6)
pos_y	int32	4	Position Y (scaled by 10^6)
pos_z	int32	4	Position Z (scaled by 10^6)
rot_1	int32	4	Rotation component 1 (scaled by 10^6)
rot_2	int32	4	Rotation component 2 (scaled by 10^6)
rot_3	int32	4	Rotation component 3 (scaled by 10^6)
rot_4	int32	4	Rotation component 4 (scaled by 10^6)
data_1	int32	4	Node's return code (0 if none)
data_2	int32	4	Additional result 2
data_3	int32	4	Additional result 3
data_4	int32	4	Additional result 4
data_5	int32	4	Additional result 5
data_6	int32	4	Additional result 6
data_7	int32	4	Additional result 7
data_8	int32	4	Additional result 8
data_9	int32	4	Additional result 9
data_10	int32	4	Additional result 10

Note: For `rc_measure`, mean_z is mapped to pos_x/pos_y/pos_z.

The fields `data_4` through `data_10` are filled according to the job's `additional_return` configuration, see *Returning poses and extra data*.

6.4.4.6 Error codes and semantics

The `error_code` is `int16` and encodes errors/warnings by sign:

- Negative < *0* = **error** (failure)
- Zero = *0* = **success**
- Positive > *0* = **warning** (success with caveat)

The tables below give the different error codes and are split by sign and sorted.

Success

Name	Value	Description
NO_ERROR	0	Operation successful

Negative error codes

Name	Value	Description
UNKNOWN_ERROR	-1	GRI internal, unspecified
INTERNAL_ERROR	-2	GRI internal system error
API_NOT_REACHABLE	-3	Cannot reach API
API_RESPONSE_ERROR	-4	API call failed or returned a negative code
PIPELINE_NOT_AVAILABLE	-5	Reserved, currently not used
INVALID_REQUEST_ERROR	-6	Malformed request
INVALID_REQUEST_LENGTH	-7	Wrong message length
INVALID_ACTION	-8	Unsupported action
PROCESSING_TIMEOUT	-9	Operation timed out
UNKNOWN_PROTOCOL_VERSION	-10	Protocol version not supported
WRONG_PROTOCOL_FOR_JOB	-11	Job configured for unsupported protocol version
JOB_DOES_NOT_EXIST	-12	Invalid job ID
MISCONFIGURED_JOB	-13	Invalid job configuration
HEC_CONFIG_ERROR	-14	Invalid configuration parameters
HEC_INIT_ERROR	-15	Calibration init failed
HEC_SET_POSE_ERROR	-16	Failed to record pose in specified slot
HEC_CALIBRATE_ERROR	-17	Unable to compute calibration from recorded poses
HEC_INSUFFICIENT_DETECTION	-18	Calibration grid not visible or not detected
TYPE_CONVERSION_ERROR	-19	Value not convertible to the configured type
UNMAPPED_VALUE_ERROR	-20	Value has no entry in the referenced maps table

Positive codes

Name	Value	Description
NO_POSES_FOUND	1	No results available
NO_RELATED_POSES	2	No related data found
NO_RETURN_SPECIFIED	3	Job configured with no return type
JOB_STILL_RUNNING	4	Async job not complete
MISSING_ADDITIONAL_DATA	5	additional_return path not found

Node return code semantics

The modules/nodes may return a `return_code`. This node return code is placed in the response `data_1` field (defaults to 0 if no code). The GRI's primary status is in `error_code` (sign-based semantics).

`API_RESPONSE_ERROR` is reported when the node returns a negative `return_code`, when the REST-API answers with an HTTP error or without a `response`, and when a parameter setting job could not set all parameters. If the REST-API error carries a `return_code` (e.g. for a module or pipeline that is not ready), that value is reported in `data_1`, except for parameter setting jobs.

Malformed messages

The GRI answers a malformed message with an error and, if the byte stream can no longer be trusted, closes the connection afterwards. The robot then has to reconnect.

Message	`error_code`	Connection
wrong `magic_number`	`INVALID_REQUEST_ERROR`	closed
`message_length` does not match the protocol version	`INVALID_REQUEST_LENGTH`	closed
header received, but the body is incomplete or does not arrive in time	`INVALID_REQUEST_ERROR`	closed
header incomplete (fewer than 8 bytes arrive in time)	no response	closed
unsupported `protocol_version`, with a `message_length` that matches the bytes sent	`UNKNOWN_PROTOCOL_VERSION`	stays open
complete message with invalid content (e.g. `HEC_SET_POSE` without a pose)	`INVALID_REQUEST_ERROR`	stays open

6.4.5 Integration with a robot

The Generic Robot Interface offers communication on port 7100.

Different robot platforms can be supported by implementing a TCP socket client following the GRI binary protocol (see *GRI binary protocol specification*). This requires a robot controller with TCP/IP support and the ability to pack robot poses into binary messages and to parse binary messages into robot poses.

The implementation steps are as follows:

1. Create TCP socket connection
2. Compose request message:
   - Set message header (8 bytes)
   - Set job ID (2 bytes)
   - Pack position (12 bytes, 3x int32)
   - Pack rotation (16 bytes, 4x int32)
   - Pack additional data (16 bytes, 4x int32)
3. Send request (54 bytes total)
4. Receive response (80 bytes total)
5. Parse response:
   - Message header (8 bytes)
   - Job ID (2 bytes)
   - Error code (2 bytes)
   - Position (12 bytes, 3x int32)
   - Rotation (16 bytes, 4x int32)
   - Additional data (40 bytes, 10x int32)

6.4.5.1 Byte interpretation in socket communication

Some robot scripting languages interpret individual socket bytes as signed values in the range `[-128, 127]` instead of unsigned `[0, 255]`. If this is the case, each byte has to be converted to unsigned **before** reconstructing `int32` values:

```
# Convert signed byte to unsigned
if byte_value < 0:
    byte_value = byte_value + 256
```

After conversion, reconstruct the `int32` using little-endian byte order, then apply signed interpretation to the most significant byte (MSB) to determine if the overall `int32` value is negative.

Note: All pose components use scaling as described in *Pose formats*.

6.4.6 Job and HEC_config API

The job definitions and the definitions of HEC_configs for hand-eye calibration can be set, retrieved and deleted via the following REST-API endpoints.

`GET /generic_robot_interface/hec_configs`
Get defined hand-eye calibration configurations

Template request

```
GET /api/v2/generic_robot_interface/hec_configs HTTP/1.1
```

Sample response

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

{
  "0": {
    "grid_height": 0.18,
    "grid_width": 0.26,
    "robot_mounted": true,
    "tcp_offset": 0,
    "tcp_rotation_axis": -1
  }
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

GET /generic_robot_interface/hec_configs/{pipeline}
Get hand-eye calibration configuration for the selected pipeline

Template request

```
GET /api/v2/generic_robot_interface/hec_configs/<pipeline> HTTP/1.1
```

Sample response

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

{
  "grid_height": 0.18,
  "grid_width": 0.26,
  "robot_mounted": true,
  "tcp_offset": 0,
  "tcp_rotation_axis": -1
}
```

Parameters

- **pipeline** (*string*) – pipeline of the hand-eye calibration configuration *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

PUT /generic_robot_interface/hec_configs/{pipeline}
Sets a hand-eye calibration configuration for the selected pipeline.

Template request

```
PUT /api/v2/generic_robot_interface/hec_configs/<pipeline> HTTP/1.1
Accept: application/json application/ubjson

{}
```

Sample response

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

{
  "return_code": {
    "message": "HEC configuration saved successfully",
    "value": 0
  }
}
```

Parameters

- **pipeline** (*string*) – pipeline of the hand-eye calibration configuration *(required)*

Request JSON Object

- **hand-eye calibration configuration** (*object*) – example args *(required)*

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

DELETE /generic_robot_interface/hec_configs/{pipeline}

Remove a hand-eye calibration configuration.

Template request

```
DELETE /api/v2/generic_robot_interface/hec_configs/<pipeline> HTTP/1.1
Accept: application/json application/ubjson
```

Parameters

- **pipeline** (*string*) – pipeline of the hand-eye calibration configuration *(required)*

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation
- 403 Forbidden – forbidden, e.g. because there is no valid license for this module.
- 404 Not Found – hec config for the given pipeline not found

GET /generic_robot_interface/jobs
Get defined jobs

Template request

```
GET /api/v2/generic_robot_interface/jobs HTTP/1.1
```

Sample response

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

{
  "0": {
    "args": {
      "pose_frame": "external",
      "tags": []
    },
    "job_type": "CALL_PIPELINE_SERVICE",
    "name": "detect_qr_code",
    "node": "rc_qr_code_detect",
    "pipeline": "0",
    "selected_return": "tags",
    "service": "detect"
  },
  "1": {
    "job_type": "SET_PARAMETERS_SERVICE",
    "name": "set_depth_full_quality",
    "node": "rc_stereomatching",
    "parameters": {
      "double_shot": true,
      "quality": "Full"
    },
    "pipeline": "0"
  }
}
```

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

GET /generic_robot_interface/jobs/{job_id}
Get selected job definition

Template request

```
GET /api/v2/generic_robot_interface/jobs/<job_id> HTTP/1.1
```

Sample response

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

{
  "args": {
    "pose_frame": "camera",
    "tags": []
  },
  "job_type": "CALL_PIPELINE_SERVICE",
```

(continues on next page)

(continued from previous page)

```
  "name": "detect_qr_code",
  "node": "rc_qr_code_detect",
  "pipeline": "0",
  "selected_return": "tags",
  "service": "detect"
}
```

Parameters

- **job_id** (*string*) – ID of the job *(required)*

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

PUT /generic_robot_interface/jobs/{job_id}

Sets a job definition for the selected job ID. The required keys depend on the chosen job_type.

Template request

```
PUT /api/v2/generic_robot_interface/jobs/<job_id> HTTP/1.1
Accept: application/json application/ubjson

{}
```

Sample response

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

{
  "job_id": "1",
  "return_code": {
    "message": "Job configuration updated successfully",
    "value": 0
  }
}
```

Parameters

- **job_id** (*string*) – ID of the job *(required)*

Request JSON Object

- **job definition** (*object*) – example args *(required)*

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation

DELETE /generic_robot_interface/jobs/{job_id}

Remove a job definition.

Template request

```
DELETE /api/v2/generic_robot_interface/jobs/<job_id> HTTP/1.1
Accept: application/json application/ubjson
```

Parameters

- **job_id** (*string*) – ID of the job *(required)*

Request Headers

- Accept – application/json application/ubjson

Response Headers

- Content-Type – application/json application/ubjson

Status Codes

- 200 OK – successful operation
- 403 Forbidden – forbidden, e.g. because there is no valid license for this module.
- 404 Not Found – job with given id not found

6.5 OPC UA interface

The *rc_visard* also offers an optional OPC UA interface running on TCP port 4840. The OPC UA server can be activated via a separate *license* (Section 7.4).

The OPC UA server provides access to parameters and services of all available software modules analogous to the REST-API. To browse the OPC UA Address Space use e.g. the freely available UAExpert GUI client.

The OPC UA server uses the `DataTypeDefinition` attribute (available in OPC UA version 1.04) for custom datatypes and also uses methods and variable length arrays. Please check if your OPC UA client supports this.

6.6 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 Basler's EKIBridge *license* (Section 7.4) to be purchased.

Note: The KUKA.EthernetKRL add-on software package version 2.2 up to version 5.x must be activated on the robot controller to use this component.

The EKI Bridge can be used to programmatically to

- 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.

Note: A known limitation of the EKI Bridge is that strings representing valid numbers will be converted to int/float. Hence user-defined names (like ROI IDs, etc.) should always contain at least one letter so they can be used in service call arguments.

6.6.1 Ethernet connection configuration

The EKI Bridge listens on port 7000 for EKI XML messages and transparently bridges the *rc_visard*'s *REST-API v2* (Section 6.2). 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 modules* (Section 5).

The Ethernet connection to the *rc_visard* on the robot controller is configured using 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`.

The IP of the *rc_visard* in the network needs to be filled into the XML file.

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.

6.6.2 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"/>
```

(continues on next page)

(continued from previous page)

```
    <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.

6.6.2.1 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 6.10: Return codes of the EKI Bridge component

Code	Description
0	Success
-1	Parsing error in the conversion from XML to JSON
-2	Internal error
-5	Connection error from the REST-API
-9	Missing or invalid license for EKI Bridge component

Note: The EKI Bridge can also return return code values specific to individual nodes. They are documented in the respective *software module* (Section 5).

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.

6.6.3 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 modules* (Section 5). 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",
```

(continues on next page)

(continued from previous page)

```
      "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"
    }
  }
}
```

is converted to the XML fragment

```
<item uuid="..." confidence="...">
  <rectangle x="..." y="...">
  </rectangle>
</item>
```

6.6.3.1 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_load_carrier_db's get_load_carriers service (see *Load-CarrierDB*, Section 5.6.1) is:

```
<SEND>
  <XML>
    <ELEMENT Tag="req/node/rc_load_carrier_db" 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_load_carrier_db'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_load_carrier_db-get_load_carriers", "req/args/load_carrier_ids/
↪le", "load_carrier1")
RET = EKI_Send("rc_load_carrier_db-get_load_carriers", "req/args/load_carrier_ids/le")
```

The second telegram includes all tags and triggers the request to the rc_load_carrier_db node:

```
<req>
  <node>
    <rc_load_carrier_db></rc_load_carrier_db>
  </node>
  <service>
    <get_load_carriers></get_load_carriers>
  </service>
  <args>
```

(continues on next page)

(continued from previous page)

```
    <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_load_carrier_db-get_load_carriers", "req/args/load_carrier_ids/
↪le", "load_carrier2")
RET = EKI_Send("rc_load_carrier_db-get_load_carriers", "req")
```

6.6.3.2 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 5.4.3) is:

```
<RECEIVE>
  <XML>
    <ELEMENT Tag="res/timestamp/@sec" Type="INT"/>
    <ELEMENT Tag="res/timestamp/@nsec" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/tags/le/pose_frame" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/timestamp/@sec" Type="INT"/>
    <ELEMENT Tag="res/tags/le/timestamp/@nsec" Type="INT"/>
    <ELEMENT Tag="res/tags/le/pose/@X" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@Y" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@Z" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@A" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@B" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/pose/@C" Type="REAL"/>
    <ELEMENT Tag="res/tags/le/instance_id" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/id" Type="STRING"/>
    <ELEMENT Tag="res/tags/le/size" Type="REAL"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

For arrays, the response includes multiple instances of the same XML element. Each element is written into a separate buffer within EKI and can be read from the buffer with KRL instructions. The number of instances can be requested with `EKI_CheckBuffer` and each instance can then be read by calling `EKI_Get<Type>`.

As an example, the tag poses received after a call to the `rc_april_tag_detect`'s `detect` service can be read in KRL using the following code:

```
DECL EKI_STATUS RET
DECL INT i
DECL INT num_instances
DECL FRAME poses[32]

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.

6.6.3.3 Memory limits and array trimming

When returning large arrays, KUKA EKI enforces two main memory limits:

- `BUFFERING` limit (queue depth): The maximum number of elements that are stored in a single array bucket. A single response often contains multiple independent lists (e.g., `res/grasps/le/...` and `res/matches/le/...`), which are stored in entirely separate buckets inside the robot controller. No single bucket may exceed the configured `BUFFERING` limit, otherwise the connection drops.
- `BUFFSIZE` limit (message size): The maximum total byte size of a single XML telegram on the network. The absolute KUKA maximum is 65534 bytes (~64 kB).

Independent lists within the same response (e.g. `grasps` and `matches`) are conceptually different and naturally have different lengths. Furthermore, if a payload exceeds the 64kB `BUFFSIZE` limit, the EKI Bridge will automatically delete elements from the end of the response to fit it on the network. This means the bridge might truncate items from one list while leaving another untouched.

Note: All child elements of a list element (after le/*) belong to the same array of a REST-API response and are guaranteed to have the same lengths even after pruning.

Always use `EKI_CheckBuffer` on the specific array you are about to read to get its true length. Do not assume its length matches that of a different array.

6.6.4 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"/>
```

(continues on next page)

(continued from previous page)

```
    <ELEMENT Tag="res/parameters/<parameter_x>/@min" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_x>/@max" Type="INT"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@value" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@default" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@min" Type="REAL"/>
    <ELEMENT Tag="res/parameters/<parameter_y>/@max" Type="REAL"/>
    <ELEMENT Tag="res/return_code/@value" Type="INT"/>
    <ELEMENT Tag="res/return_code/@message" Type="STRING"/>
    <ELEMENT Tag="res" Set_Flag="998"/>
  </XML>
</RECEIVE>
```

The request is interpreted as a *get* request if all parameter's `value` attributes are empty. If any `value` attribute is non-empty, it is interpreted as *set* request of the non-empty parameters.

As an example, the current value of all parameters of `rc_stereomatching` can be queried using 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>
```

6.6.5 Migration to firmware version 22.01

From firmware version 22.01 on the EKI Bridge reflects *rc_visard*'s *REST-API v2* (Section 6.2).

This requires the following changes:

- Configuring load carriers, grippers and regions of interest is now only accessible in the global database modules:
  - Use the `rc_load_carrier_db` XML files for getting, setting and deleting of load carriers.
  - Use the `rc_gripper_db` XML files for getting, setting and deleting of grippers.
  - Use the `rc_roi_db` XML files for getting, setting and deleting of regions of interest.
- Load carrier detection and filling level detection is now only accessible via the `rc_load_carrier` node.
  - Use the `rc_load_carrier` XML files for `detect_load_carriers` and `detect_filling_level` services.

6.6.6 Troubleshooting

SmartPad error message: Limit of element memory reached

This error occurs when the number of received elements in any array exceeds the configured `BUFFERING` limit in the EKI XML configuration file. The connection drops before the KRL program can read the data.

- Increase the `BUFFERING` limit in the EKI XML configuration file on the robot controller to match the maximum number of expected elements.
- Decrease the *Maximum Matches* (or similar) parameter in the *rc_visard* detection module to ensure it returns fewer items.

The total memory available for storing EKI values on the robot is limited by default to 1MB across all connections. If you increase the limits too much or have too many parallel arrays, the robot may fail to initialize the connection. Please refer to the official KUKA documentation (KST_EthernetKRL) to increase this system memory pool.

SmartPad error message: Reading from empty buffer

This error occurs when the KRL program attempts to read an array element that was never received.

- Payload trimming: The EKI Bridge might have trimmed elements from the end of the response to fit within the 64kB `BUFFSIZE` limit, causing independent arrays to have different lengths (see *Memory limits and array trimming*). Check the EKI Bridge logs for warnings like "Trimmed XML tree" or "Pruned response items".
- Mismatched loop counts: The KRL program loops further than the actual received elements. Ensure you are using `EKI_CheckBuffer` on the specific array you are reading.

- KRL array size: Ensure that the arrays defined in your KRL `.dat` and `.src` files are large enough to hold all incoming data.

6.7 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.

6.7.1 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.

6.7.2 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 *Camera module* (Section 5.1) and the *Stereo matching module* (Section 5.2) module.

6.7.3 Important standard GenICam features

6.7.3.1 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 6.7.6).

`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.

`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 6.7.6).

`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.

6.7.3.2 Category: AcquisitionControl

`AcquisitionFrameRate`

- type: Float, ranges from 1 Hz to 25 Hz
- default: 25 Hz
- description: Frame rate of the camera

`ExposureAuto`

- type: Enumeration, one of `Continuous`, `Out1High`, `AdaptiveOut1`, `HDR` or `Off`
- default: `Continuous`
- description: Combines `exp_control` and `exp_auto_mode`. `Off` maps to *Manual* exposure control. `Continuous`, `Out1High` or `AdaptiveOut1` enable *Auto* exposure control with the respective auto exposure mode where `Continuous` maps to the *Normal* `exp_auto_mode`. `HDR` enables high-dynamic-range exposure control.

`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.

6.7.3.3 Category: AnalogControl

`GainSelector` (read-only)

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

`Gain`

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

`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.

`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.

6.7.3.4 Category: DigitalIOControl

`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 6.11: Meaning of bits of `LineStatusAll` field.

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

`LineSource`

- type: Enumeration, one of `ExposureActive`, `ExposureAlternateActive`, `Low` or `High`
- default: `Low`
- description: Mode for output line selected by `LineSelector` as described in the IO-Control module (*out1_mode and out2_mode*, Section 5.5.4.1). See also parameter `AcquisitionAlternateFilter` for filtering images in `ExposureAlternateActive` mode.

6.7.3.5 Category: TransportLayerControl / PtpControl

`PtpEnable`

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

6.7.3.6 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 implicitly selected by `DepthQuality`.

`Scan3dBaseline` (read-only)

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

`Scan3dPrinciplePointU` (read-only)

- type: Float
- description: Horizontal location of the principal 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 implicitly selected by `DepthQuality`.

`Scan3dPrinciplePointV` (read-only)

- type: Float
- description: Vertical location of the principal 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 implicitly 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.

6.7.3.7 Category: ChunkDataControl

`ChunkModeActive`

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

6.7.4 Custom GenICam features of the *rc_visard*

6.7.4.1 Category: DeviceControl

`RcSystemReady` (read-only)

- type: Boolean
- description: Returns whether the device's boot process has completed and all modules are running.

`RcParamLockDisable`

- type: Boolean
- default: False
- description: If set to true, the camera and depth image parameters are not locked when a GigE Vision client is connected to the device. Please note that depending on the connected GigE Vision client, parameter changes by other applications (e.g. the Web GUI) might not be noticed by the GigE Vision client, which could lead to unwanted results.

6.7.4.2 Category: AcquisitionControl

`AcquisitionAlternateFilter`

- 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 5.5.4.1).

`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.

`ExposureRegionOffsetX`

- type: Integer in the range of 0 to the maximum image width
- default: 0
- description: Horizontal offset of exposure region in pixel.

`ExposureRegionOffsetY`

- type: Integer in the range of 0 to the maximum image height
- default: 0
- description: Vertical offset of exposure region in pixel.

`ExposureRegionWidth`

- type: Integer in the range of 0 to the maximum image width
- default: 0
- description: Width of exposure region in pixel.

`ExposureRegionHeight`

- type: Integer in the range of 0 to the maximum image height
- default: 0
- description: Height of exposure region in pixel.

`RcExposureAutoAverageMax`

- type: Float in the range of 0 to 1
- default: 0.75

- description: Maximum brightness for the auto exposure function 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 as value between 0 (dark) and 1 (bright).

6.7.4.3 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.

6.7.4.4 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. 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 5.2.1.2).

`DepthDoubleShot`

- type: Boolean
- default: `False`
- description: `True` for improving the stereo matching result of a scene recorded with a projector by filling holes with depth information computed from images without projector pattern. (*Double-Shot*, Section 5.2.1.2).

`DepthStaticScene`

- type: Boolean
- default: `False`
- description: `True` for averaging 8 consecutive camera images for improving the stereo matching result. (*Static*, Section 5.2.1.2).

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

- type: Boolean
- default: `False`
- description: `True` for advanced smoothing of disparity values. (*Smoothing*, Section 5.2.1.2).

`DepthFill`

- type: Integer, ranges from 0 pixel to 4 pixels
- default: 3 pixels
- description: Value in pixels for *Fill-In* (Section 5.2.1.2).

`DepthSeg`

- type: Integer, ranges from 0 pixel to 4000 pixels
- default: 200 pixels
- description: Value in pixels for *Segmentation* (Section 5.2.1.2).

`DepthMinConf`

- type: Float, ranges from 0.0 to 1.0
- default: 0.0
- description: Value for *Minimum Confidence* filtering (Section 5.2.1.2).

`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 5.2.1.2).

`DepthMaxDepth`

- type: Float, ranges from 0.1m to 100.0 m
- default: 100.0 m
- description: Value in meters for *Maximum Distance* filtering (Section 5.2.1.2).

`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 5.2.1.2).

6.7.5 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 6.7.7).
- `ChunkPartIndex` provides the index of the image part in this MultiPart block for the selected component (`ChunkComponentSelector`).
- `ChunkRcOut1Reduction` gives a ratio of how much the brightness of the images with GPIO Out1 LOW is lower than the brightness of the images with GPIO Out1 HIGH. For example, a value of 0.2 means that the images with GPIO Out1 LOW have 20% less brightness than the images with GPIO Out1 HIGH. This value is only available if `exp_auto_mode` of the stereo camera is set to `AdaptiveOut1` or `Out1High`.

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

6.7.6 Provided image streams

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

Component name	PixelFormat	Description
`Intensity`	`Mono8` (monochrome cameras) `YCbCr411_8` (color cameras)	Left rectified camera image
`IntensityCombined`	`Mono8` (monochrome cameras) `YCbCr411_8` (color cameras)	Left rectified camera image stacked on right rectified camera image
`Disparity`	`Coord3D_C16`	Disparity image in desired resolution, i.e., `DepthQuality` of `Full`, `High`, `Medium` or `Low`
`Confidence`	`Confidence8`	Confidence image
`Error`	`Error8` (custom: 0x81080001)	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.

6.7.7 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 principal 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*.

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

$$d_{ik} = d16_{ik} \cdot \mathrm{Scan3dCoordinateScale}$$

The 3D reconstruction in meters can be written with the GenICam parameters as:

$$\begin{aligned} P_x &= (i + 0.5 - \mathrm{Scan3dPrincipalPointU}) \frac{\mathrm{Scan3dBaseline}}{d_{ik}}, \\ P_y &= (k + 0.5 - \mathrm{Scan3dPrincipalPointV}) \frac{\mathrm{Scan3dBaseline}}{d_{ik}}, \\ P_z &= \mathrm{Scan3dFocalLength} \frac{\mathrm{Scan3dBaseline}}{d_{ik}}. \end{aligned}$$

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*, the depth error z_{eps} in meters can be computed with GenICam parameters as

$$\begin{aligned} d_{ik} &= d16_{ik} \cdot \mathrm{Scan3dCoordinateScale}, \\ z_{eps} &= \frac{e_{ik} \cdot \mathrm{Scan3dCoordinateScale} \cdot \mathrm{Scan3dFocalLength} \cdot \mathrm{Scan3dBaseline}}{(d_{ik})^2}. \end{aligned}$$

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 module* (Section 5.2).

6.8 gRPC image stream interface

The gRPC image streaming interface can be used as an alternative to the *GigE Vision / GenICam interface* (Section 6.7) for getting camera images and synchronized sets of images (e.g. left camera image and corresponding disparity image).

gRPC is a remote procedure call system that also supports streaming. It uses Protocol Buffers (see https://developers.google.com/protocol-buffers/) as interface description language and data serialization. For a gRPC introduction and more details please see the official website (https://grpc.io/).

The advantages of the gRPC interface in comparison to GigE Vision are:

- It is simpler to use in own programs than GigE Vision.
- There is gRPC support for a lot of programming languages (see https://grpc.io/).
- The communication is based on TCP instead of UDP and therefore it also works over less stable networks, e.g. WLAN.

The disadvantages of the gRPC interface in comparison to GigE Vision are:

- It does not support changing parameters, but the *REST-API interface* (Section 6.2) can be used for changing parameters.
- It is not a standard vision interface like GigE Vision.

The *rc_visard* provides synchronized image sets via gRPC server side streams on port 50051.

The communication is started by sending an `ImageSetRequest` message to the server. The message contains the information about requested images, i.e. left, right, disparity, confidence and disparity_error images can be enabled separately.

After getting the request, the server starts continuously sending `ImageSet` messages that contain all requested images with all parameters necessary for interpreting the images. The images that are contained in an `ImageSet` message are synchronized, i.e. they are all captured at the same time. The only exception to this rule is if the *out1_mode* (Section 5.5.4.1) is set to `AlternateExposureActive`. In this case, the camera and disparity images are taken 40 ms apart, so that the GPIO Out1 is LOW when the left and right images are taken, and HIGH for the disparity, confidence and error images. This mode is useful when a random dot projector is used, because the projector would be off for capturing the left and right image, and on for the disparity image, which results in undisturbed camera images and a much denser and more accurate disparity image.

Streaming of images is done until the client closes the connection.

An `ImageEventsRequest` message can be sent to start streaming `ImageEvents`. This message contains the `depth_acquisition_done` Event, which signals that the image acquisition for depth computation has finished. It also contains the `imageset_timestamp` of the corresponding `ImageSet`. This event cen be used to optimize cycle time in a robotic application, because it signals when it is safe to move the camera or the scene after triggering a detection.

6.8.1 gRPC service definition

```
syntax = "proto3";

message Time
{
  int32 sec = 1; ///< Seconds
  int32 nsec = 2; ///< Nanoseconds
}

message Gpios
{
  uint32 inputs  = 1; ///< bitmask of available inputs
  uint32 outputs = 2; ///< bitmask of available outputs
  uint32 values  = 3; ///< bitmask of GPIO values
}

message Image
{
  Time timestamp              = 1; ///< Acquisition timestamp of the image
  uint32 height               = 2; ///< image height (number of rows)
  uint32 width                = 3; ///< image width (number of columns)
  float focal_length          = 4; ///< focal length in pixels
```

(continues on next page)

(continued from previous page)

```
  float principal_point_u     = 5; ///< horizontal position of the principal point
  float principal_point_v     = 6; ///< vertical position of the principal point
  string encoding             = 7; ///< Encoding of pixels ["mono8", "mono16", "rgb8"]
  bool is_bigendian           = 8; ///< is data bigendian, (in our case false)
  uint32 step                 = 9; ///< full row length in bytes
  bytes data                  = 10; ///< actual matrix data, size is (step * height)
  Gpios gpios                 = 11; ///< GPIOs as of acquisition timestamp
  float exposure_time         = 12; ///< exposure time in seconds
  float gain                  = 13; ///< gain factor in decibel
  float noise                 = 14; ///< noise
  float out1_reduction        = 16; ///< Fraction of reduction (0.0 - 1.0) of exposure time for␣
↪images with GPIO Out1=Low in exp_auto_mode=AdaptiveOut1
  float brightness            = 17; ///< Current brightness of the image as value between 0 and 1
  float gamma                 = 18; ///< Current gamma value of the image, where 1.0 is a linear␣
↪mapping. Lower gamma values let dark image parts appear brighter.
  bool auto_exposure_adapting = 19; ///< True if auto exposure is currently still adapting
}

message DisparityImage
{
  Time timestamp           = 1; ///< Acquisition timestamp of the image
  float scale              = 2; ///< scale factor
  float offset             = 3; ///< offset in pixels (in our case 0)
  float invalid_data_value = 4; ///< value used to mark pixels as invalid (in our case 0)
  float baseline           = 5; ///< baseline in meters
  float delta_d            = 6; ///< Smallest allowed disparity increment. The smallest␣
↪achievable depth range resolution is delta_Z = (Z^2/image.focal_length*baseline)*delta_d.
  Image image              = 7; ///< disparity image
}

message Mesh
{
  Time timestamp           = 1; ///< Acquisition timestamp of disparity image from which the mesh␣
↪is computed
  string format            = 2; ///< currently only "ply" is supported
  bytes data               = 3; ///< actual mesh data
}

message ImageSet
{
  Time timestamp             = 1;
  Image left                 = 2;
  Image right                = 3;
  DisparityImage disparity   = 4;
  Image disparity_error      = 5;
  Image confidence           = 6;
  Mesh mesh                  = 7;
}

message MeshOptions
{
  uint32 max_points            = 1; ///< limit maximum number of points, zero means default (up␣
↪to 3.1MP), minimum is 1000
  enum BinningMethod {
    AVERAGE = 0;                   ///< average over all points in bin
    MIN_DEPTH = 1;                 ///< use point with minimum depth (i.e. closest to camera) in␣
↪bin
  }
  BinningMethod binning_method = 2; ///< method used for binning if limited by max_points
  bool watertight              = 3; ///< connect all edges and fill all holes, e.g. for collision␣
↪checking
```

(continues on next page)

(continued from previous page)

```
  bool textured                = 4; ///< add texture information to mesh
}

message ImageSetRequest
{
  bool left_enabled            = 1;
  bool right_enabled           = 2;
  bool disparity_enabled       = 3;
  bool disparity_error_enabled = 4;
  bool confidence_enabled      = 5;
  bool mesh_enabled            = 6;
  MeshOptions mesh_options     = 7;
  bool color                   = 8; ///< send left/right image as color (rgb8) images
}

service ImageInterface
{
  // A server-to-client streaming RPC.
  rpc StreamImageSets(ImageSetRequest) returns (stream ImageSet) {}
}

message Event
{
  Time timestamp = 1; ///< timestamp of the event
  string message = 2; ///< optional message of the event
}

message ImageEvents
{
  Time imageset_timestamp      = 1; ///< timestamp of the ImageSet that the event belongs to
  Event depth_acquisition_done = 2; ///< Depth image acquisition is done (e.g. stereo images␣
↪captured)
}

message ImageEventsRequest
{
  bool depth_acquisition_done_enabled = 1; ///< send event when depth acquisition is done
}

service ImageEventsInterface
{
  // A server-to-client streaming RPC.
  rpc StreamImageEvents(ImageEventsRequest) returns (stream ImageEvents) {}
}
```

6.8.1.1 Image stream conversions

The disparity image contains 16 bit unsigned integer values. These values must be multiplied by the `scale` value given in the `DisparityImage` message to get the disparity values d in pixels. To compute the 3D object coordinates from the disparity values, the baseline and the focal length as well as the principal point are required. These parameters are transmitted as `baseline` = t in the `DisparityImage` message, and `focal_length` = f, `principal_point_u` = c_x and `principal_point_v` = c_y in the `ImageData` message. The focal length and principal point depend on the resolution of the camera image and need to be scaled to the resolution of the disparity image. 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*.

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

$$d_{ik} = d16_{ik} \cdot \text{scale}$$

The 3D reconstruction in meters can be written as:

$$P_x = (i + 0.5 - c_x)\,\frac{t}{d_{ik}},$$
$$P_y = (k + 0.5 - c_y)\,\frac{t}{d_{ik}},$$
$$P_z = f\,\frac{t}{d_{ik}}.$$

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 `DisparityImage` message to get the disparity-error values d_{eps} in pixels. According to the description in *Confidence and error images*, the depth error z_{eps} in meters can be computed as

$$d_{ik} = d16_{ik} \cdot \text{scale},$$
$$z_{eps} = \frac{e_{ik} \cdot \text{scale} \cdot f \cdot t}{(d_{ik})^2}.$$

For more information about disparity, error, and confidence images, please refer to *Stereo matching module* (Section 5.2).

6.9 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 at the last saved time (saved on reboot and every 15 minutes) 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 6.2) and seen on the *Web GUI*'s (Section 6.1) *System* page.

Note: Depending on the reachability of NTP servers or PTP masters it might take up to several minutes until the time is synchronized.

6.9.1 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 4.4.2) (which is the default setting), it will also request NTP servers from the DHCP server and try to use those.

Additionally, the user can specify up to three NTP servers manually using the REST-API's **/system/ntp** endpoint. A more convenient way is setting the NTP servers on the *Web GUI*'s (Section 6.1) *System Time* page.

6.9.2 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 6.7) 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.

6.9.3 Setting time manually

The *rc_visard* allows to set the current date and time manually using the REST-API's **/system/time** endpoint, if no time synchronization is active (see *System and logs*, Section 6.2.3.3). A more convenient way is setting the system time on the *Web GUI*'s (Section 6.1) *System Time* page.

7 Maintenance

Warning: The customer does not need to open the *rc_visard*'s housing to perform maintenance. Unauthorized opening will void the warranty.

7.1 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.

7.2 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 5.5.3).

7.3 Creating and restoring backups of settings

The *rc_visard* offers the possibility to download the current settings as backup or for transferring them to a different *rc_visard* or *rc_cube*.

The current settings of the *rc_visard* can be downloaded on the *Web GUI*'s (Section 6.1) *System* page in the *rc_visard Settings* section. They can also be downloaded via the *rc_visard*'s *REST-API interface* (Section 6.2) using the `GET /system/backup` request.

For downloading a backup, the user can choose which settings to include:

- `nodes`: the settings of all modules (parameters, preferred orientations and sorting strategies)
- `load_carriers`: the configured load carriers
- `regions_of_interest`: the configured 2D and 3D regions of interest
- `grippers`: the configured grippers (without the CAD elements)

The returned backup should be stored as a .json file.

The templates of the SilhouetteMatch module are not included in the backup but can be downloaded manually using the REST-API or the Web GUI (see *Template API*, Section 5.4.7.14).

A backup can be restored to the *rc_visard* on the *Web GUI*'s (Section 6.1) *System* page in the *rc_visard Settings* section by uploading the backup .json file. In the *Web GUI* the settings included in the backup are shown and can be chosen for restore. The corresponding *REST-API interface* (Section 6.2) call is `POST /system/backup`.

Warning: When restoring load carriers, all existing load carriers on the *rc_visard* will get lost and will be replaced by the content of the backup. The same applies to restoring grippers and regions of interest.

When restoring a backup, only the settings which are applicable to the *rc_visard* are restored. Parameters for modules that do not exist on the device or do not have a valid license will be skipped. If a backup can only be restored partially, the user will be notified by warnings.

7.4 Updating the software license

Licenses that are purchased from Basler for enabling additional features can be installed via the *Web GUI*'s (Section 6.1) *System* → *Firmware & License* page. The *rc_visard* has to be rebooted to apply the licenses.

7.5 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 6.1) *System* → *Logs* page. If contacting the support (*Contact*, Section 10), 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 *System* → *Logs* page.

Aside from the Web GUI, the logs are also accessible via the *rc_visard*'s *REST-API interface* (Section 6.2) using the `GET /logs` and `GET /logs/{log}` requests.

7.6 Updating the firmware

Information about the current firmware image version can be found on the *Web GUI*'s (Section 6.1) *System* → *Firmware & License* page. It can also be accessed via the *rc_visard*'s *REST-API interface* (Section 6.2) using the `GET /system` request. Users can use either the Web GUI or the REST-API to update the firmware.

Warning: When upgrading from a version prior to 21.07, all of the software modules' configured parameters will be reset to their defaults after a firmware update. Only when upgrading from version 21.07 or higher, the last saved parameters will be preserved. Please make sure these settings are persisted on the application-side or client PC (e.g., using the *REST-API interface*, Section 6.2) 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-specified device name,
- the latest result of the *Hand-eye calibration* (Section 5.5.1), 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 5.5.3), 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.

> **Warning:** Make sure the firmware version to upload is still within the software maintenance period of your *rc_visard*. You can see the firmware version constraints on the *rc_visard*'s Web GUI on the *System* → *Firmware & License* page. If the latest firmware version exceeds the software maintenance period, a new license must be purchased to use a newer firmware.

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 *System* → *Firmware & License* page and press the "Upload *rc_visard* Update" button. Select the desired update image file (file extension `.mender`) from the local file system and open it to start the update.

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.

> **Note:** Depending on the web browser, the update progress status shown in the progress bar may indicate the completion of the update too early. Please wait until a notification window opens, which indicates the end of the update process. Expect an overall update time of at least five minutes.

> **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 notification window is opened, which offers to reboot the device immediately or to postpone the reboot. To reboot the *rc_visard* at a later time, use the *Reboot* button on the Web GUI's *System* page.

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* → *Firmware & License* page or via the REST-API's *GET /system/update* request.

Please contact Basler in case the firmware update could not be applied successfully.

7.7 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*.

Warning: System rollbacks are in general not supported by the UserSpace. Before performing a system rollback, all UserSpace apps should be backed up to avoid data loss.

Note: Using the latest firmware as provided by Basler 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 6.2) 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 6.2.5). Like the update process, the rollback requires a subsequent device reboot to activate the restored firmware version.

7.8 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 6.2) using the `PUT /system/reboot` request, or manually on the *Web GUI*'s (Section 6.1) *System* page.

The reboot is finished when the LED turns green again.

8 Accessories

8.1 Connectivity

To set up and operate an *rc_visard* camera, certain cables and a power supply are necessary. Basler offers these accessories that have been specifically tested for use with Basler cameras to ensure optimum performance. Other makes are available (see *Wiring*).

- Cable GigE, M12 A 8P Plug, RJ45, 10 m: https://www.baslerweb.com/en/shop/cable-gige-m12-a-8p-plug-rj45-10-m/
- Power Adapter Cable, M12 A 8P Rec., DC 5.5/2.1mm (30 cm): https://www.baslerweb.com/en/shop/power-cable-m12-a-8p-rec-dc-5-5-2-1mm/
- Power Supply 24V/60W, DC Jack 5.5/2.1mm: https://www.baslerweb.com/en/shop/power-supply-24v-60w-dc-jack-5-5-2-1mm/

If the camera is to be installed in industrial environments where exposure to strong ESD sources may occur (e.g., from frequency converters, contactors, large electric motors, or arc discharges) or where EN 61000-6-2 is required, the customer must use a DIN rail power supply that satisfies EN 61000-6-2 in terms of immunity, increased temperature range, and secure connectors.

8.2 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.

8.2.1 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

8.2.2 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)

8.2.3 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.

8.3 Spare parts

No user-serviceable spare parts are currently available for *rc_visard* devices.

9 Troubleshooting

9.1 LED colors

During the boot process, the LED will change color several times to indicate stages in the boot process:

Table 9.1: 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 9.2: LED color trouble codes

LED color	Warning or error state
off	no power to the sensor
flashing every second	no network connectivity
blue flashing	system messages available, check the Web GUI or /system/messages
fast flashing red	high-temperature warning (case has exceeded 60 °C)
red	Some process has terminated and failed to restart.

9.2 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 starts flashing 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.

9.3 Connectivity issues

LED flashes every second

If the LED flashes every 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 every second (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 every second (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 4.4) 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*.

9.4 Camera-image issues

The camera image is too bright

- If the camera is in manual exposure mode, decrease the exposure time, or

- switch to auto-exposure mode.

The camera image is too dark

- If the camera is in manual exposure mode, increase the exposure time, or
- switch to auto-exposure mode.

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.

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 10).

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.

The camera image is fuzzy

- Check whether the lenses are dirty and clean them if so (see *Lens cleaning*, Section 7.1).
- If none of the above applies, a severe hardware problem might exist. Please *contact support* (Section 10).

The camera image frame rate is too low

- Increase the image frame rate.
- The maximal frame rate of the cameras is 25 Hz.

9.5 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 9.4) if applicable.
- Check whether the scene has enough texture and install an external pattern projector if required.
- Decrease the *Minimum Distance* (Section 5.2.1).
- Increase the *Maximum Distance* (Section 5.2.1).
- Check whether the object is too close to the cameras. Consider the different depth ranges of the camera variants.
- Decrease the *Minimum Confidence* (Section 5.2.1).
- Increase the *Maximum Depth Error* (Section 5.2.1).

- Choose a lesser *Disparity Image Quality* (Section 5.2.1). Lower resolution disparity images are generally less sparse.
- Check the cameras' calibration and recalibrate if required (see *Camera calibration*, Section 5.5.3).

The disparity images' frame rate is too low

- Check and increase the frame rate of the camera images. 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 5.2.1).
- Increase the *Minimum Distance* (Section 5.2.1) 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 camera variants.
- Decrease the *Minimum Distance* (Section 5.2.1).

The disparity image does not show distant objects

- Increase the *Maximum Distance* (Section 5.2.1).
- Increase the *Maximum Depth Error* (Section 5.2.1).
- Decrease the *Minimum Confidence* (Section 5.2.1).

The disparity image is too noisy

- Increase the *Segmentation value* (Section 5.2.1).
- Increase the *Fill-In value* (Section 5.2.1).

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 5.2.1).

The disparity image does not show small structures

- Decrease the *Segmentation value* (Section 5.2.1).
- Decrease the *Fill-In value* (Section 5.2.1).

9.6 Dynamics issues

State estimates are unavailable

- Check in the Web GUI that pose estimation has been switched on (see *Parameters*, Section 5.3.2.1).
- 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 5.3.2.1).
- Check whether the *camera pose stream* has enough accuracy.

Pose estimation has jumps

- Has the SLAM module 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 5.3.2.1).

Pose frequency is too low

- Use the real-time pose stream with a 200 Hz update rate. See *Stereo INS* (Section 5.3.3).

Delay/Latency of pose is too great

- Use the real-time pose stream. See *Stereo INS* (Section 5.3.3).

9.7 GigE Vision/GenICam issues

No images

- Check that the modules are enabled. See `ComponentSelector` and `ComponentEnable` in *Important GenICam parameters* (Section 6.7.2).

10 Contact

10.1 Support

For support issues, please see https://www.baslerweb.com/en/sales-support/support-contact/ or contact

Technical Support Team

Europe, Middle East, Africa
+49 4102 463 515
support.europe@baslerweb.com

The Americas
+1 610 280 0171
support.usa@baslerweb.com

Asia-Pacific
+65 6367 1355
support.asia@baslerweb.com

For specific issues, please download the log files (see *Downloading log files*, Section 7.5) and, if applicable, download a snapshot of the last detection: Every software module allows to download the last detection on its *Web GUI* (Section 6.1) page by clicking the blue camera button below the image streams.

Downloading the last detection can also be automated by using the `trigger_dump` service of the respective software module, e.g. see *Services* (Section 5.4.2.7) for the LoadCarrier module.

10.2 Downloads

Software SDKs, etc. can be downloaded from https://www.baslerweb.com/3d-software.

10.3 Address

Basler AG
An der Strusbek 60-62
22926 Ahrensburg
Germany

Web: http://www.baslerweb.com
Email: info@baslerweb.com
Phone: +49 4102 463 0

11 Appendix

11.1 Pose formats

A pose consists of a translation and rotation. The translation defines the shift along the x, y and z axes. The rotation can be defined in many different ways. The *rc_visard* uses quaternions to define rotations and translations are given in meters. This is called the XYZ+quaternion format. This chapter explains the conversion between different common conventions and the XYZ+quaternion format.

It is quite common to define rotations in 3D by three angles that define rotations around the three coordinate axes. Unfortunately, there are many different ways to do that. The most common conventions are Euler and Cardan angles (also called Tait-Bryan angles). In both conventions, the rotations can be applied to the previously rotated axis (intrinsic rotation) or to the axis of a fixed coordinate system (extrinsic rotation).

We use x, y and z to denote the three coordinate axes. x', y' and z' refer to the axes that have been rotated one time. Similarly, x'', y'' and z'' are the axes after two rotations.

In the (original) Euler angle convention, the first and the third axis are always the same. The rotation order z-x'-z'' means rotating around the z-axis, then around the already rotated x-axis and finally around the (two times) rotated z-axis. In the Cardan angle convention, three different rotation axes are used, e.g. z-y'-x''. Cardan angles are often also just called Euler angles.

For each intrinsic rotation order, there is an equivalent extrinsic rotation order, which is inverted, e.g. the intrinsic rotation order z-y'-x'' is equivalent to the extrinsic rotation order x-y-z.

Rotations around the x, y and z axes can be defined by quaternions as

$$r_x(\alpha) = \begin{pmatrix} \sin\frac{\alpha}{2} \\ 0 \\ 0 \\ \cos\frac{\alpha}{2} \end{pmatrix}, \qquad r_y(\beta) = \begin{pmatrix} 0 \\ \sin\frac{\beta}{2} \\ 0 \\ \cos\frac{\beta}{2} \end{pmatrix}, \qquad r_z(\gamma) = \begin{pmatrix} 0 \\ 0 \\ \sin\frac{\gamma}{2} \\ \cos\frac{\gamma}{2} \end{pmatrix},$$

or by rotation matrices as

$$r_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix},$$
$$r_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix},$$
$$r_z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The extrinsic rotation order x-y-z can be computed by multiplying the individual rotations in inverse order, i.e. $r_z(\gamma)r_y(\beta)r_x(\alpha)$.

Based on these definitions, the following sections explain the conversion between common conventions and the XYZ+quaternion format.

Note: Please be aware of units for positions and orientations. *rc_visard* devices always specify positions in meters, while most robot manufacturers use millimeters or inches. Angles are typically specified in degrees, but may sometimes also be given in radians.

11.1.1 Rotation matrix and translation vector

A pose can also be defined by a rotation matrix R and a translation vector T.

$$R = \begin{pmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{pmatrix}, \qquad T = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}.$$

The pose transformation can be applied to a point P by

$$P' = RP + T.$$

11.1.1.1 Conversion from rotation matrix to quaternion

The conversion from a rotation matrix (with $det(R) = 1$) to a quaternion $q = \begin{pmatrix} x & y & z & w \end{pmatrix}$ can be done as follows.

$$\begin{aligned} x &= \text{sign}(r_{21} - r_{12}) \frac{1}{2} \sqrt{\text{max}(0, 1 + r_{00} - r_{11} - r_{22})} \\ y &= \text{sign}(r_{02} - r_{20}) \frac{1}{2} \sqrt{\text{max}(0, 1 - r_{00} + r_{11} - r_{22})} \\ z &= \text{sign}(r_{10} - r_{01}) \frac{1}{2} \sqrt{\text{max}(0, 1 - r_{00} - r_{11} + r_{22})} \\ w &= \frac{1}{2} \sqrt{\text{max}(0, 1 + r_{00} + r_{11} + r_{22})} \end{aligned}$$

The sign operator returns -1 if the argument is negative. Otherwise, 1 is returned. It is used to recover the sign for the square root. The max function ensures that the argument of the square root function is not negative, which can happen in practice due to round-off errors.

11.1.1.2 Conversion from quaternion to rotation matrix

The conversion from a quaternion $q = \begin{pmatrix} x & y & z & w \end{pmatrix}$ with $||q|| = 1$ to a rotation matrix can be done as follows.

$$R = 2 \begin{pmatrix} \frac{1}{2} - y^2 - z^2 & xy - zw & xz + yw \\ xy + zw & \frac{1}{2} - x^2 - z^2 & yz - xw \\ xz - yw & yz + xw & \frac{1}{2} - x^2 - y^2 \end{pmatrix}$$

11.1.2 ABB pose format

ABB robots use a position X, Y, Z and a quaternion $Q1$, $Q2$, $Q3$, $Q4$ for describing a pose, similar to *rc_visard* devices. However, the position must be given in millimeters and the quaternion order is as follows:

$q = \begin{pmatrix} x & y & z & w \end{pmatrix} = \begin{pmatrix} Q2 & Q3 & Q4 & Q1 \end{pmatrix}$.

11.1.3 FANUC XYZ-WPR format

The pose format that is used by FANUC robots consists of a position XYZ in millimeters and an orientation WPR that is given by three angles in degrees, with W rotating around x-axis, P rotating around

y-axis and R rotating around z-axis. The rotation order is x-y-z and computed by $r_z(R)r_y(P)r_x(W)$.

11.1.3.1 Conversion from FANUC-WPR to quaternion

The conversion from the WPR angles in degrees to a quaternion $q = (\ x \quad y \quad z \quad w\ )$ can be done by first converting all angles to radians

$$W_r = W\frac{\pi}{180},$$
$$P_r = P\frac{\pi}{180},$$
$$R_r = R\frac{\pi}{180},$$

and then calculating the quaternion with

$$x = \cos(R_r/2)\cos(P_r/2)\sin(W_r/2) - \sin(R_r/2)\sin(P_r/2)\cos(W_r/2),$$
$$y = \cos(R_r/2)\sin(P_r/2)\cos(W_r/2) + \sin(R_r/2)\cos(P_r/2)\sin(W_r/2),$$
$$z = \sin(R_r/2)\cos(P_r/2)\cos(W_r/2) - \cos(R_r/2)\sin(P_r/2)\sin(W_r/2),$$
$$w = \cos(R_r/2)\cos(P_r/2)\cos(W_r/2) + \sin(R_r/2)\sin(P_r/2)\sin(W_r/2).$$

11.1.3.2 Conversion from quaternion to FANUC-WPR

The conversion from a quaternion $q = (\ x \quad y \quad z \quad w\ )$ with $||q|| = 1$ to the WPR angles in degrees can be done as follows.

$$R = \text{atan}_2(2(wz + xy), 1 - 2(y^2 + z^2))\frac{180}{\pi}$$
$$P = \text{asin}(2(wy - zx))\frac{180}{\pi}$$
$$W = \text{atan}_2(2(wx + yz), 1 - 2(x^2 + y^2))\frac{180}{\pi}$$

11.1.4 Franka Pose Format

Franka robots use a transformation matrix T to define a pose. A transformation matrix combines a rotation matrix R and a translation vector $t = (\ x \quad y \quad z\ )^T$.

$$T = \begin{pmatrix} r_{00} & r_{01} & r_{02} & x \\ r_{10} & r_{11} & r_{12} & y \\ r_{20} & r_{21} & r_{22} & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The pose given by Franka's "Measure Pose" App consists of a translation x, y, z in millimeters and a rotation x, y, z in degrees. The rotation convention is z-y'-x'' (i.e. x-y-z) and is computed by $r_z(z)r_y(y)r_x(x)$.

11.1.4.1 Conversion from transformation matrix to quaternion

The conversion from a rotation matrix (with $det(R) = 1$) to a quaternion $q = (\ q_x \quad q_y \quad q_z \quad q_w\ )$ can be done as follows:

$$
\begin{aligned}
q_x &= \text{sign}(r_{21} - r_{12})\frac{1}{2}\sqrt{\text{max}(0, 1 + r_{00} - r_{11} - r_{22})} \\
q_y &= \text{sign}(r_{02} - r_{20})\frac{1}{2}\sqrt{\text{max}(0, 1 - r_{00} + r_{11} - r_{22})} \\
q_z &= \text{sign}(r_{10} - r_{01})\frac{1}{2}\sqrt{\text{max}(0, 1 - r_{00} - r_{11} + r_{22})} \\
q_w &= \frac{1}{2}\sqrt{\text{max}(0, 1 + r_{00} + r_{11} + r_{22})}
\end{aligned}
$$

The sign operator returns -1 if the argument is negative. Otherwise, 1 is returned. It is used to recover the sign for the square root. The max function ensures that the argument of the square root function is not negative, which can happen in practice due to round-off errors.

11.1.4.2 Conversion from Rotation-XYZ to quaternion

The conversion from the x, y, z angles in degrees to a quaternion $q = (\ q_x \quad q_y \quad q_z \quad q_w\ )$ can be done by first converting all angles to radians

$$
\begin{aligned}
X_r &= x\frac{\pi}{180}, \\
Y_r &= y\frac{\pi}{180}, \\
Z_r &= z\frac{\pi}{180},
\end{aligned}
$$

and then calculating the quaternion with

$$
\begin{aligned}
q_x &= \cos(Z_r/2)\cos(Y_r/2)\sin(X_r/2) - \sin(Z_r/2)\sin(Y_r/2)\cos(X_r/2), \\
q_y &= \cos(Z_r/2)\sin(Y_r/2)\cos(X_r/2) + \sin(Z_r/2)\cos(Y_r/2)\sin(X_r/2), \\
q_z &= \sin(Z_r/2)\cos(Y_r/2)\cos(X_r/2) - \cos(Z_r/2)\sin(Y_r/2)\sin(X_r/2), \\
q_w &= \cos(Z_r/2)\cos(Y_r/2)\cos(X_r/2) + \sin(Z_r/2)\sin(Y_r/2)\sin(X_r/2).
\end{aligned}
$$

11.1.4.3 Conversion from quaternion and translation to transformation

The conversion from a quaternion $q = (\ q_x \quad q_y \quad q_z \quad q_w\ )$ and a translation vector $t = (\ x \quad y \quad z\ )^T$ to a transformation matrix T can be done as follows:

$$
T = \begin{pmatrix}
1 - 2s(q_y^2 + q_z^2) & 2s(q_xq_y - q_zq_w) & 2s(q_xq_z + q_yq_w) & x \\
2s(q_xq_y + q_zq_w) & 1 - 2s(q_x^2 + q_z^2) & 2s(q_yq_z - q_xq_w) & y \\
2s(q_xq_z - q_yq_w) & 2s(q_yq_z + q_xq_w) & 1 - 2s(q_x^2 + q_y^2) & z \\
0 & 0 & 0 & 1
\end{pmatrix}
$$

where $s = ||q||^{-2} = \frac{1}{q_x^2 + q_y^2 + q_z^2 + q_w^2}$ and $s = 1$ if q is a unit quaternion.

11.1.4.4 Conversion from quaternion to Rotation-XYZ

The conversion from a quaternion $q = (\ q_x \quad q_y \quad q_z \quad q_w\ )$ with $||q|| = 1$ to the x, y, z angles in degrees can be done as follows.

$$x = \text{atan}_2(2(q_w q_z + q_x q_y), 1 - 2(q_y^2 + q_z^2))\frac{180}{\pi}$$
$$y = \text{asin}(2(q_w q_y - q_z q_x))\frac{180}{\pi}$$
$$z = \text{atan}_2(2(q_w q_x + q_y q_z), 1 - 2(q_x^2 + q_y^2))\frac{180}{\pi}$$

11.1.4.5 Pose representation in RaceCom messages and state machines

In RaceCom messages and in state machines a pose is usually defined as one-dimensional array of 16 float values, representing the transformation matrix in column-major order. The indices of the matrix entries below correspond to the array indices

$$T = \begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

11.1.5 Fruitcore HORST pose format

Fruitcore HORST robots use a position in meters and a quaternion with $q_0 = w$, $q_1 = x$, $q_2 = y$ and $q_3 = z$ for describing a pose, like *rc_visard* devices. There is no conversion needed.

11.1.6 Kawasaki XYZ-OAT format

The pose format that is used by Kawasaki robots consists of a position XYZ in millimeters and an orientation OAT that is given by three angles in degrees, with O rotating around z axis, A rotating around the rotated y axis and T rotating around the rotated z axis. The rotation convention is z-y'-z'' (i.e. z-y-z) and computed by $r_z(O)r_y(A)r_z(T)$.

11.1.6.1 Conversion from Kawasaki-OAT to quaternion

The conversion from the OAT angles in degrees to a quaternion $q = (\ x \quad y \quad z \quad w\ )$ can be done by first converting all angles to radians

$$O_r = O\frac{\pi}{180},$$
$$A_r = A\frac{\pi}{180},$$
$$T_r = T\frac{\pi}{180},$$

and then calculating the quaternion with

$$x = \cos(O_r/2)\sin(A_r/2)\sin(T_r/2) - \sin(O_r/2)\sin(A_r/2)\cos(T_r/2),$$
$$y = \cos(O_r/2)\sin(A_r/2)\cos(T_r/2) + \sin(O_r/2)\sin(A_r/2)\sin(T_r/2),$$
$$z = \sin(O_r/2)\cos(A_r/2)\cos(T_r/2) + \cos(O_r/2)\cos(A_r/2)\sin(T_r/2),$$
$$w = \cos(O_r/2)\cos(A_r/2)\cos(T_r/2) - \sin(O_r/2)\cos(A_r/2)\sin(T_r/2).$$

11.1.6.2 Conversion from quaternion to Kawasaki-OAT

The conversion from a quaternion $q = (\ x \quad y \quad z \quad w\ )$ with $||q|| = 1$ to the OAT angles in degrees can be done as follows.

If $x = 0$ **and** $y = 0$ the conversion is

$$O = \text{atan}_2(2(z - w), 2(z + w))\frac{180}{\pi}$$
$$A = \text{acos}(w^2 + z^2)\frac{180}{\pi}$$
$$T = \text{atan}_2(2(z + w), 2(w - z))\frac{180}{\pi}$$

If $z = 0$ **and** $w = 0$ the conversion is

$$O = \text{atan}_2(2(y - x), 2(x + y))\frac{180}{\pi}$$
$$A = \text{acos}(-1.0)\frac{180}{\pi}$$
$$T = \text{atan}_2(2(y + x), 2(y - x))\frac{180}{\pi}$$

In all other cases the conversion is

$$O = \text{atan}_2(2(yz - wx), 2(xz + wy))\frac{180}{\pi}$$
$$A = \text{acos}(w^2 - x^2 - y^2 + z^2)\frac{180}{\pi}$$
$$T = \text{atan}_2(2(yz + wx), 2(wy - xz))\frac{180}{\pi}$$

11.1.7 KUKA XYZ-ABC format

KUKA robots use the so called XYZ-ABC format. XYZ is the position in millimeters. ABC are angles in degrees, with A rotating around z axis, B rotating around y axis and C rotating around x axis. The rotation convention is z-y'-x'' (i.e. x-y-z) and computed by $r_z(A)r_y(B)r_x(C)$.

11.1.7.1 Conversion from KUKA-ABC to quaternion

The conversion from the ABC angles in degrees to a quaternion $q = (\ x \quad y \quad z \quad w\ )$ can be done by first converting all angles to radians

$$A_r = A\frac{\pi}{180},$$
$$B_r = B\frac{\pi}{180},$$
$$C_r = C\frac{\pi}{180},$$

and then calculating the quaternion with

$$x = \cos(A_r/2)\cos(B_r/2)\sin(C_r/2) - \sin(A_r/2)\sin(B_r/2)\cos(C_r/2),$$
$$y = \cos(A_r/2)\sin(B_r/2)\cos(C_r/2) + \sin(A_r/2)\cos(B_r/2)\sin(C_r/2),$$
$$z = \sin(A_r/2)\cos(B_r/2)\cos(C_r/2) - \cos(A_r/2)\sin(B_r/2)\sin(C_r/2),$$
$$w = \cos(A_r/2)\cos(B_r/2)\cos(C_r/2) + \sin(A_r/2)\sin(B_r/2)\sin(C_r/2).$$

11.1.7.2 Conversion from quaternion to KUKA-ABC

The conversion from a quaternion $q = (\ x \quad y \quad z \quad w\ )$ with $||q|| = 1$ to the ABC angles in degrees can be done as follows.

$$A = \text{atan}_2(2(wz + xy), 1 - 2(y^2 + z^2))\frac{180}{\pi}$$
$$B = \text{asin}(2(wy - zx))\frac{180}{\pi}$$
$$C = \text{atan}_2(2(wx + yz), 1 - 2(x^2 + y^2))\frac{180}{\pi}$$

11.1.8 Mitsubishi XYZ-ABC format

The pose format that is used by Mitsubishi robots is the same as that for KUKA robots (see *KUKA XYZ-ABC format*, Section 11.1.7), except that A is a rotation around x axis and C is a rotation around z axis. Thus, the rotation is computed by $r_z(C)r_y(B)r_x(A)$.

11.1.8.1 Conversion from Mitsubishi-ABC to quaternion

The conversion from the ABC angles in degrees to a quaternion $q = (\ x \quad y \quad z \quad w\ )$ can be done by first converting all angles to radians

$$A_r = A\frac{\pi}{180},$$
$$B_r = B\frac{\pi}{180},$$
$$C_r = C\frac{\pi}{180},$$

and then calculating the quaternion with

$$x = \cos(C_r/2)\cos(B_r/2)\sin(A_r/2) - \sin(C_r/2)\sin(B_r/2)\cos(A_r/2),$$
$$y = \cos(C_r/2)\sin(B_r/2)\cos(A_r/2) + \sin(C_r/2)\cos(B_r/2)\sin(A_r/2),$$
$$z = \sin(C_r/2)\cos(B_r/2)\cos(A_r/2) - \cos(C_r/2)\sin(B_r/2)\sin(A_r/2),$$
$$w = \cos(C_r/2)\cos(B_r/2)\cos(A_r/2) + \sin(C_r/2)\sin(B_r/2)\sin(A_r/2).$$

11.1.8.2 Conversion from quaternion to Mitsubishi-ABC

The conversion from a quaternion $q = (\ x \quad y \quad z \quad w\ )$ with $||q|| = 1$ to the ABC angles in degrees can be done as follows.

$$A = \text{atan}_2(2(wx + yz), 1 - 2(x^2 + y^2))\frac{180}{\pi}$$
$$B = \text{asin}(2(wy - zx))\frac{180}{\pi}$$
$$C = \text{atan}_2(2(wz + xy), 1 - 2(y^2 + z^2))\frac{180}{\pi}$$

11.1.9 Universal Robots pose format

The pose format that is used by Universal Robots consists of a position XYZ in millimeters and an orientation in angle-axis format $V = (\ RX \quad RY \quad RZ\ )^T$. The rotation angle θ in radians is the length

of the rotation axis U.

$$V = \begin{pmatrix} RX \\ RY \\ RZ \end{pmatrix} = \begin{pmatrix} \theta u_x \\ \theta u_y \\ \theta u_z \end{pmatrix}$$

V is called a rotation vector.

11.1.9.1 Conversion from angle-axis format to quaternion

The conversion from a rotation vector V to a quaternion $q = \begin{pmatrix} x & y & z & w \end{pmatrix}$ can be done as follows.

We first recover the angle θ in radians from the rotation vector V by

$$\theta = \sqrt{RX^2 + RY^2 + RZ^2}.$$

If $\theta = 0$, then the quaternion is $q = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$, otherwise it is

$$\begin{aligned} x &= RX \frac{\sin(\theta/2)}{\theta}, \\ y &= RY \frac{\sin(\theta/2)}{\theta}, \\ z &= RZ \frac{\sin(\theta/2)}{\theta}, \\ w &= \cos(\theta/2). \end{aligned}$$

11.1.9.2 Conversion from quaternion to angle-axis format

The conversion from a quaternion $q = \begin{pmatrix} x & y & z & w \end{pmatrix}$ with $||q|| = 1$ to a rotation vector in angle-axis form can be done as follows.

We first recover the angle θ in radians from the quaternion by

$$\theta = 2 \cdot \text{acos}(w).$$

If $\theta = 0$, then the rotation vector is $V = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$, otherwise it is

$$\begin{aligned} RX &= \theta \frac{x}{\sqrt{1 - w^2}}, \\ RY &= \theta \frac{y}{\sqrt{1 - w^2}}, \\ RZ &= \theta \frac{z}{\sqrt{1 - w^2}}. \end{aligned}$$

11.1.10 Yaskawa Pose Format

The pose format that is used by Yaskawa robots consists of a position XYZ in millimeters and an orientation that is given by three angles in degrees, with Rx rotating around x-axis, Ry rotating around y-axis and Rz rotating around z-axis. The rotation order is x-y-z and computed by $r_z(Rz)r_y(Ry)r_x(Rx)$.

11.1.10.1 Conversion from Yaskawa Rx, Ry, Rz to quaternion

The conversion from the Rx, Ry, Rz angles in degrees to a quaternion $q = (\ x \quad y \quad z \quad w\ )$ can be done by first converting all angles to radians

$$
\begin{aligned}
X_r &= Rx\frac{\pi}{180}, \\
Y_r &= Ry\frac{\pi}{180}, \\
Z_r &= Rz\frac{\pi}{180},
\end{aligned}
$$

and then calculating the quaternion with

$$
\begin{aligned}
x &= \cos(Z_r/2)\cos(Y_r/2)\sin(X_r/2) - \sin(Z_r/2)\sin(Y_r/2)\cos(X_r/2), \\
y &= \cos(Z_r/2)\sin(Y_r/2)\cos(X_r/2) + \sin(Z_r/2)\cos(Y_r/2)\sin(X_r/2), \\
z &= \sin(Z_r/2)\cos(Y_r/2)\cos(X_r/2) - \cos(Z_r/2)\sin(Y_r/2)\sin(X_r/2), \\
w &= \cos(Z_r/2)\cos(Y_r/2)\cos(X_r/2) + \sin(Z_r/2)\sin(Y_r/2)\sin(X_r/2).
\end{aligned}
$$

11.1.10.2 Conversion from quaternion to Yaskawa Rx, Ry, Rz

The conversion from a quaternion $q = (\ x \quad y \quad z \quad w\ )$ with $||q|| = 1$ to the Rx, Ry, Rz angles in degrees can be done as follows.

$$
\begin{aligned}
Rx &= \text{atan}_2(2(wx + yz), 1 - 2(x^2 + y^2))\frac{180}{\pi} \\
Ry &= \text{asin}(2(wy - zx))\frac{180}{\pi} \\
Rz &= \text{atan}_2(2(wz + xy), 1 - 2(y^2 + z^2))\frac{180}{\pi}
\end{aligned}
$$

HTTP Routing Table

/cad

GET /cad/gripper_elements, 284
GET /cad/gripper_elements/{id}, 285
PUT /cad/gripper_elements/{id}, 285
DELETE /cad/gripper_elements/{id}, 286

/datastreams

GET /datastreams, 308
GET /datastreams/{stream}, 309
PUT /datastreams/{stream}, 310
DELETE /datastreams/{stream}, 310

/generic_robot_interface

GET /generic_robot_interface/hec_configs, 362
GET /generic_robot_interface/hec_configs/{pipeline}, 363
GET /generic_robot_interface/jobs, 364
GET /generic_robot_interface/jobs/{job_id}, 365
PUT /generic_robot_interface/hec_configs/{pipeline}, 363
PUT /generic_robot_interface/jobs/{job_id}, 366
DELETE /generic_robot_interface/hec_configs/{pipeline}, 364
DELETE /generic_robot_interface/jobs/{job_id}, 366

/logs

GET /logs, 311
GET /logs/{log}, 312

/nodes

GET /nodes, 294
GET /nodes/{node}, 295
GET /nodes/{node}/services, 296
GET /nodes/{node}/services/{service}, 296
GET /nodes/{node}/status, 297
PUT /nodes/{node}/services/{service}, 297

/pipelines

GET /pipelines/{pipeline}/nodes, 298
GET /pipelines/{pipeline}/nodes/{node}, 300
GET /pipelines/{pipeline}/nodes/{node}/parameters, 300
GET /pipelines/{pipeline}/nodes/{node}/parameters/{param}, 303
GET /pipelines/{pipeline}/nodes/{node}/services, 304
GET /pipelines/{pipeline}/nodes/{node}/services/{service}, 305
GET /pipelines/{pipeline}/nodes/{node}/status, 307
PUT /pipelines/{pipeline}/nodes/{node}/parameters, 301
PUT /pipelines/{pipeline}/nodes/{node}/parameters/{param}, 303
PUT /pipelines/{pipeline}/nodes/{node}/services/{service}, 306

/system

GET /system, 314
GET /system/backup, 315
GET /system/dns, 316
GET /system/license, 317
GET /system/max_power_test, 318
GET /system/network, 318
GET /system/network/settings, 319
GET /system/ntp, 320
GET /system/rollback, 322
GET /system/time, 323
GET /system/ui_lock, 323
GET /system/update, 325
POST /system/backup, 315
POST /system/license, 318
POST /system/max_power_test, 318
POST /system/ui_lock, 324
POST /system/update, 325
PUT /system/dns, 316
PUT /system/network/settings, 319
PUT /system/ntp, 321
PUT /system/reboot, 322
PUT /system/rollback, 322
PUT /system/time, 323
DELETE /system/ui_lock, 324

/templates

GET /templates/rc_boxpick, 159
GET /templates/rc_boxpick/{id}, 160
GET /templates/rc_silhouettematch, 210
GET /templates/rc_silhouettematch/{id}, 211
PUT /templates/rc_boxpick/{id}, 160
PUT /templates/rc_silhouettematch/{id}, 211
DELETE /templates/rc_boxpick/{id}, 161

DELETE /templates/rc_silhouettematch/{id},
212

Index

Symbols

3D coordinates, 49
    disparity image, 49
3D modeling, 49, 53

A

acceleration, 53, 54
acquisition mode
    disparity image, 41
AcquisitionAlternateFilter
    GenICam, 381
AcquisitionFrameRate
    GenICam, 377
AcquisitionMultiPartMode
    GenICam, 381
active partition, 393
AdaptiveOut1
    auto exposure mode, 35
angular
    velocity, 53, 54
AprilTag, **95**
    pose estimation, 98
    re-identification, 99
    return codes, 106
    services, 101
auto
    exposure, 35
auto exposure, 34, 35
auto exposure mode, 35
    AdaptiveOut1, 35
    Normal, 35
    Out1High, 35

B

backup
    settings, 391
BalanceRatio
    GenICam, 378
BalanceRatioSelector
    GenICam, 378
BalanceWhiteAuto
    GenICam, 378
base-plane
    SilhouetteMatch, 175
base-plane calibration
    SilhouetteMatch, 175
Baseline
    GenICam, 382
Baumer
    IpConfigTool, 30
bin packing, 161
bin picking, 107, 128
BoxPick, **128**
    dimensioning, 131
    filling level, 79
    grasp, 131
    item models, 129
    load carrier, 78, 255
    parameters, 134
    plane, 269
    preferred orientation, 133
    RECTANGLE, 130
    region of interest, 263
    return codes, 158
    services, 141
    sorting, 132
    status, 141
    template api, 159
    template deletion, 159
    template download, 159
    template upload, 159
    texture, 130
    TEXTURED_BOX, 130
    views, 130

C

cables, 21, 395
CAD model, 19
calibration
    camera, 244
    camera to IMU, 53
    hand-eye calibration, 217
    rectification, 39
calibration grid, 245
camera
    calibration, 244
    frame rate, 33
    gamma, 34
    parameters, 31
    pose stream, 53
    Web GUI, 31
camera calibration
    monocalibration, 249
    parameters, 250
    services, 250
    stereo calibration, 247

camera to IMU
    calibration, 53
    transformation, 53
Chunk data
    GenICam, 380
collision check, 235, 273
CollisionCheck, **235**
    return codes, 243
compartment
    load carrier, 258
ComponentEnable
    GenICam, 377
ComponentIDValue
    GenICam, 376
components
    rc_visard, 16
ComponentSelector
    GenICam, 376
Confidence
    GenICam image stream, 384
confidence, **49**
    minimum, 46
connectivity, 395
conversions
    GenICam image stream, 385
    gRPC image stream, 388
cooling, 20
coordinate frames
    dynamics, 53
    mounting, 24
    state estimation, 51
corners
    visual odometry, 59, 61
correspondences
    visual odometry, 59

D

data
    IMU, 54
    inertial measurement unit, 54
data model
    REST-API, 326
data stream
    dynamics, 52, 53
    imu, 54
    pose, 53
    pose_rt, 53
    REST-API, 308
data-type
    REST-API, 326
definition
    load carrier, 255
depth error
    maximum, 46
depth image, 39, 48, **49**, 49
    Web GUI, 39
depth measurement, 71
DepthAcquisitionMode
    GenICam, 382
DepthAcquisitionTrigger
    GenICam, 382
DepthDoubleShot
    GenICam, 382
DepthFill
    GenICam, 383
DepthMaxDepth
    GenICam, 383
DepthMaxDepthErr
    GenICam, 383
DepthMinConf
    GenICam, 383
DepthMinDepth
    GenICam, 383
DepthQuality
    GenICam, 382
DepthSeg
    GenICam, 383
DepthSmooth
    GenICam, 383
DepthStaticScene
    GenICam, 382
detection
    load carrier, 78
    tag, 94
DHCP, **12**
DHCP, 29
dimensioning
    plane, 131, 269
dimensions
    load carrier, 255
    rc_visard, 18
disable parameter lock
    GenICam, 380
discovery GUI, 27
Disparity
    GenICam image stream, 384
disparity, 39, 48
disparity error, 49
disparity image, 39, 48
    3D coordinates, 49
    acquisition mode, 41
    double_shot, 44
    exposure adaptation timeout, 42
    parameters, 39
    quality, 42
    smooth, 45
    static_scene, 44
    Web GUI, 39
disparity range
    visual odometry, 61
DNS, **12**
DOF, **12**
double_shot
    disparity image, 44
    GenICam, 382
download

images, 39
log files, 392
point cloud, 50
settings, 391
dynamics
coordinate frames, 53
data stream, 52, 53
jump flag, 53
REST-API, 308
services, 54
Web GUI, 59
dynamics stream, 52, 53

E

egomotion, 59
eki, 367
Error
GenICam image stream, 384
error, **49**
hand-eye calibration, 223
pose, 64
Ethernet
pin assignments, 21
exposure
auto, 34, 35
HDR, 34
manual, 34
exposure adaptation timeout
disparity image, 42
exposure region, 36
exposure time, 36
maximum, 35
ExposureAuto
GenICam, 377
ExposureRegionHeight
GenICam, 381
ExposureRegionOffsetX
GenICam, 381
ExposureRegionOffsetY
GenICam, 381
ExposureRegionWidth
GenICam, 381
ExposureTime
GenICam, 377
ExposureTimeAutoMax
GenICam, 381
external reference frame
hand-eye calibration, 213

F

features
visual odometry, 62
fill-in, 45
GenICam, 383
filling level
BoxPick, 79
ItemPick, 79
LoadCarrier, 79
SilhouetteMatch, 79
firmware
mender, 392
rollback, 393
update, 392
version, 392
focal length, 39
focal length factor
GenICam, 382
FocalLengthFactor
GenICam, 382
fps, *see* frame rate
frame rate, 17
camera, 33
GenICam, 377
pose, 52, 53
visual odometry, 59

G

Gain
GenICam, 378
gain factor, 35, 37
gamma
camera, 34
Generic Robot Interface, 345
GenICam, **12**
GenICam
AcquisitionAlternateFilter, 381
AcquisitionFrameRate, 377
AcquisitionMultiPartMode, 381
BalanceRatio, 378
BalanceRatioSelector, 378
BalanceWhiteAuto, 378
Baseline, 382
Chunk data, 380
ComponentEnable, 377
ComponentIDValue, 376
ComponentSelector, 376
DepthAcquisitionMode, 382
DepthAcquisitionTrigger, 382
DepthDoubleShot, 382
DepthFill, 383
DepthMaxDepth, 383
DepthMaxDepthErr, 383
DepthMinConf, 383
DepthMinDepth, 383
DepthQuality, 382
DepthSeg, 383
DepthSmooth, 383
DepthStaticScene, 382
disable parameter lock, 380
double_shot, 382
ExposureAuto, 377
ExposureRegionHeight, 381
ExposureRegionOffsetX, 381
ExposureRegionOffsetY, 381
ExposureRegionWidth, 381
ExposureTime, 377

ExposureTimeAutoMax, 381
fill-in, 383
focal length factor, 382
FocalLengthFactor, 382
frame rate, 377
Gain, 378
Height, 377
HeightMax, 377
LineSelector, 378
LineSource, 379
LineStatus, 378
LineStatusAll, 378
maximum depth error, 383
maximum distance, 383
minimum confidence, 383
minimum distance, 383
PixelFormat, 377, 384
PtpEnable, 379
quality, 382
RcExposureAutoAverageMax, 381
RcExposureAutoAverageMin, 382
Scan3dBaseline, 379
Scan3dCoordinateOffset, 380
Scan3dCoordinateScale, 379
Scan3dDistanceUnit, 379
Scan3dFocalLength, 379
Scan3dInvalidDataFlag, 380
Scan3dInvalidDataValue, 380
Scan3dOutputMode, 379
Scan3dPrincipalPointU, 379
Scan3dPrincipalPointV, 379
segmentation, 383
smooth, 383
static_scene, 382
system ready, 380
timestamp, 384
Width, 377
WidthMax, 377
GenICam image stream
Confidence, 384
conversions, 385
Disparity, 384
Error, 384
Intensity, 384
IntensityCombined, 384
GigE, **12**
GigE Vision, **12**
GigE Vision, *see* GenICam
IP address, 30
GPIO
pin assignments, 22
grasp computation, 107, 128
GRI, 345
gripper CAD element api, 284
gripper CAD element deletion, 284
gripper CAD element download, 284
gripper CAD element upload, 284
GripperDB, **273**
return codes, 284
gRPC, 385
gRPC image stream
conversions, 388

H

hand-eye calibration
calibration, 217
error, 223
external reference frame, 213
mounting, 213
parameters, 223
robot frame, 213
slot, 220
Height
GenICam, 377
HeightMax
GenICam, 377
host name, 29
housing temperature
LED, 20
humidity, 20

I

image
timestamp, 50, 384
image features
visual odometry, 59
image noise, 35
images
download, 39
IMU, **12**
IMU
data, 54
inertial measurement unit, 59
imu
data stream, 54
inactive partition, 393
inertial measurement unit
data, 54
IMU, 59
inner volume
load carrier, 255
INS, **12**
installation, 26
Intensity
GenICam image stream, 384
IntensityCombined
GenICam image stream, 384
IP, **12**
IP address, **12**
IP address, 28
GigE Vision, 30
IP54, 20
IpConfigTool
Baumer, 30
ItemPick, **107**
filling level, 79

grasp, 107
load carrier, 78, 255
parameters, 110
preferred orientation, 109
region of interest, 263
return codes, 128
services, 115
sorting, 108
status, 114

J

jump flag
dynamics, 53
SLAM, 53

K

keyframes, 59
visual odometry, 59, 61

L

LED, 26
colors, 397
housing temperature, 20
linear
velocity, 53
LineSelector
GenICam, 378
LineSource
GenICam, 379
LineStatus
GenICam, 378
LineStatusAll
GenICam, 378
Link-Local, **12**
Link-Local, 29
load carrier
BoxPick, 78, 255
compartment, 258
definition, 255
detection, 78
dimensions, 255
inner volume, 255
ItemPick, 78, 255
orientation prior, 255
pose, 255
rim, 255
SilhouetteMatch, 78, 255
load carrier detection, 78
load carrier model, 255
LoadCarrier, **78**
filling level, 79
parameters, 81
return codes, 93
services, 83
LoadCarrierDB, **255**
return codes, 262
services, 259
log files
download, 392
logs
REST-API, 311
loop closure, 64

M

MAC address, **12**
MAC address, 29
manual exposure, 34, 36
maximum
depth error, 46
exposure time, 35
maximum depth error, 46
GenICam, 383
maximum distance, 44
GenICam, 383
mDNS, **12**
Measure, **71**
parameters, 73
plane estimation, 72
return codes, 78
services, 73
mender
firmware, 392
minimum
confidence, 46
minimum confidence, 46
GenICam, 383
minimum distance, 43
GenICam, 383
monocalibration
camera calibration, 249
motion blur, 35
mounting, 23
hand-eye calibration, 213

N

network cable, 395
network configuration, 28
node
REST-API, 293
Normal
auto exposure mode, 35
NTP, **12**
NTP
synchronization, 389

O

object detection, 173
OPC UA, 367
operating conditions, 20
orientation prior
load carrier, 255
Out1High
auto exposure mode, 35

P

Pack, **161**

packing volume, 162
place pose, 162
preferred orientation, 162
return codes, 173
services, 166
starting point, 162
status, 166
parameter
REST-API, 293
parameters
camera, 31
camera calibration, 250
disparity image, 39
hand-eye calibration, 223
services, 38
visual odometry, 59
pin assignments
Ethernet, 21
GPIO, 22
power, 22
PixelFormat
GenICam, 377, 384
place pose computation, 161
plane, 269
dimensioning, 131
plane estimation, 71
Measure, 72
point cloud, 49
download, 50
pose
data stream, 53
error, 64
frame rate, 52, 53
load carrier, 255
timestamp, 52
pose estimation, *see* state estimation
AprilTag, 98
QR code, 98
pose stream, 53
camera, 53
pose_rt
data stream, 53
power
pin assignments, 22
power cable, 395
power supply, **20**, 396
PrimitiveDB, **269**
return codes, 273
services, 271
protection class, 20
PTP, **12**
PTP
synchronization, 379, 390
PtpEnable
GenICam, 379

Q

QR Code
return codes, 106
QR code, **95**
pose estimation, 98
re-identification, 99
services, 101
quality
disparity image, 42
GenICam, 382
quaternion
rotation, 53

R

rc_dynamics, 342
rc_visard
components, 16
RcExposureAutoAverageMax
GenICam, 381
RcExposureAutoAverageMin
GenICam, 382
re-identification
AprilTag, 99
QR code, 99
real-time pose, 52, 53
reboot, 394
rectification, 39
reset, 27
resolution, 17
REST-API, 290
data model, 326
data stream, 308
data-type, 326
dynamics, 308
entry point, 290
logs, 311
node, 293
parameter, 293
services, 294
status value, 293
system, 311
version, 290
restore
settings, 391
return codes
AprilTag, 106
BoxPick, 158
CollisionCheck, 243
GripperDB, 284
ItemPick, 128
LoadCarrier, 93
LoadCarrierDB, 262
Measure, 78
Pack, 173
PrimitiveDB, 273
QR Code, 106
RoiDB, 269
SilhouetteMatch, 209
rim
load carrier, 255

robot frame
    hand-eye calibration, 213
ROI, 263
RoiDB, **263**
    return codes, 269
    services, 264
rollback
    firmware, 393
rotation
    quaternion, 53

S

Scan3dBaseline
    GenICam, 379
Scan3dCoordinateOffset
    GenICam, 380
Scan3dCoordinateScale
    GenICam, 379
Scan3dDistanceUnit
    GenICam, 379
Scan3dFocalLength
    GenICam, 379
Scan3dInvalidDataFlag
    GenICam, 380
Scan3dInvalidDataValue
    GenICam, 380
Scan3dOutputMode
    GenICam, 379
Scan3dPrincipalPointU
    GenICam, 379
Scan3dPrincipalPointV
    GenICam, 379
SDK, **12**
segmentation, 45
    GenICam, 383
self-calibration, 244
sensor fusion, 59
serial number, 27, 29
services
    AprilTag, 101
    camera calibration, 250
    dynamics, 54
    parameters, 38
    QR code, 101
    REST-API, 294
    tag detection, 101
    visual odometry, 62
set
    time, 390
settings
    backup, 391
    download, 391
    restore, 391
    upload, 391
SGM, **12**
silhouette, 173
SilhouetteMatch, **173**
    base-plane, 175
    base-plane calibration, 175
    collision check, 182
    detection of objects, 179
    filling level, 79
    grasp points, 176
    load carrier, 78, 255
    object template, 176
    parameters, 182
    preferred orientation, 178
    region of interest, 176, 263
    return codes, 209
    services, 188
    sorting, 179
    status, 187
    template api, 210
    template deletion, 210
    template download, 210
    template upload, 210
Simultaneous Localization and Mapping, *see* SLAM
SLAM, **13**
SLAM, **64**
    jump flag, 53
    Web GUI, 64
slot
    hand-eye calibration, 220
smooth
    disparity image, 45
    GenICam, 383
spare parts, 396
specifications
    rc_visard, 17
state estimate, 52
state estimation
    coordinate frames, 51
static_scene
    disparity image, 44
    GenICam, 382
status value
    REST-API, 293
stereo calibration
    camera calibration, 247
stereo region of interest, 43
Swagger UI, 338
synchronization
    NTP, 389
    PTP, 379, 390
    time, 379, 389
system
    REST-API, 311
system ready
    GenICam, 380

T

tag detection, **94**
    families, 95
    pose estimation, 98
    re-identification, 99

services, 101
TCP, **13**
temperature range, 20
time
set, 390
synchronization, 379, 389
timestamp
GenICam, 384
image, 50, 384
pose, 52
transformation
camera to IMU, 53
translation, 53
tripod, 23

U

UDP, **13**
update
firmware, 392
upload
settings, 391
URI, **13**
URL, **13**

V

velocity
angular, 53, 54
linear, 53
version
firmware, 392
REST-API, 290
visual odometry, **59**
corners, 59, 61
correspondences, 59
disparity range, 61
features, 62
frame rate, 59
image features, 59
keyframes, 59, 61
parameters, 59
services, 62
Web GUI, 59
VO, *see* visual odometry

W

Web GUI, **287**
backup, 391
camera, 31
depth image, 39
disparity image, 39
dynamics, 59
logs, 392
SLAM, 64
update, 392
visual odometry, 59
white balance, 37
Width
GenICam, 377
WidthMax
GenICam, 377

X

XYZ+quaternion, **13**
XYZABC, **13**

rc_visard 3D Stereo Sensor

ASSEMBLY AND OPERATING MANUAL

Basler AG

An der Strusbek 60-62
22926 Ahrensburg
Germany

info@baslerweb.com
https://www.baslerweb.com

Software: https://www.baslerweb.com/3d-software
Hardware: https://www.baslerweb.com/stereo

For customer support, contact

+49 4102 463 515 support.europe@baslerweb.com
+1 610 280 0171 support.usa@baslerweb.com
+65 6367 1355 support.asia@baslerweb.com