Software manual
2D Grasping-Kit
Native Protocol V3

Original software manual

Hand in hand for tomorrow

Native Protocol

Low-level pipeline interface for robot controllers.

The *Native Protocol* is a communication protocol sitting on top of the TCP/IP stack.
The protocol design is simple and straightforward, and thus feasible to implement even on the most archaic robots and industrial controllers.

Conventions

- **Port:** The default port is 42001
- **Endianness:** The order of multi-byte fields is big-endian
- **Floating-point numbers:** Single IEEE-Standard (IEC-60559)
- **Signed integers:** Signed integers are represented in two's complement notation.
- **Units of measure:** Unless otherwise specified, lengths are measured in meters [m], and angles are measured in radians [rad].

Data Types

We define the following data types:

- **signed integers:** int8, int16, int32
- **unsigned integers:** uint8, uint16,
- uint32 **floating point numbers:** float32

The number suffix of a data type denotes its size in bits (e. g. int16 is two bytes long).

Versioning

The server is backwards compatible.
Clients using earlier protocol versions will be served as expected and won't even notice the version mismatch. The server always replies with the same version number in the prefix as the one used in the requests.

However, if a server is outdated such that its protocol version is lower than the client's protocol version, the server will always reply with an empty message with the *version*-field set to the server's highest supported protocol version. In this case, clients are advised to alert the user that their server is outdated and incompatible with their current client version.

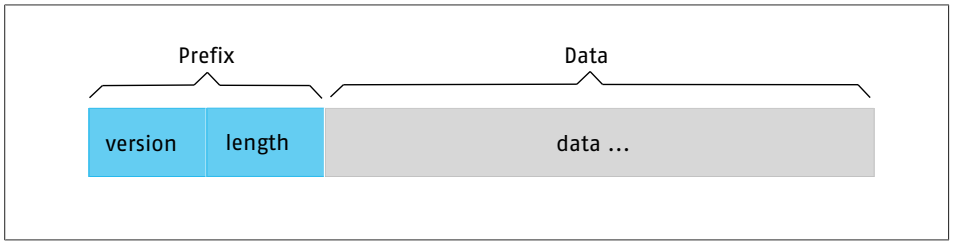
Layout

Each message frame comprises two parts: an invariant prefix that remains consistent across different protocol versions and a version-specific section. The details of these parts are elaborated in their respective chapters.

Native Protocol Prefix

Invariant part of the low-level pipeline interface for robot controllers.

Layout



Prefix

All messages start with a *prefix* defining the protocol version and the size of the remaining message. *The prefix is invariant and does not change between different protocol versions.*

Byte offset	0	2	6
Name	version	length	...
Type	uint16	uint32	...
	Prefix		**Data**

▼ Details

version

The protocol version.

length

The size of the **remaining** message in bytes (max 4GB), excluding the *prefix* itself (6B).

Data

The actual message content. The structure is version-specific and is described in the version chapters.

Native Protocol V3

Version 3 of the low-level pipeline interface for robot controllers.

General

- **Message size:** Every message is 80 bytes long. Always. **Data**
- **types:** Only integers (signed and unsigned). No floats.

Layout

Every message consists of an invariant prefix, a header and a body.

Byte offset	0	6	10 – 79
Data	Prefix	Header	Body

Requests and responses adhere to a consistent layout, which allows a static mapping between fields and their byte indices in a frame. However, depending on the message context, only relevant sections of the body carry meaningful values, while other parts are to be disregarded. The parts to be disregarded are denoted as *offset* or *padding*.

Request

Byte offset	**Data**	**Type**	
0	version	uint16	**Prefix**
2	length	uint32	
6	comm type	uint8	**Header**
7	reply code	uint8	
8	reply counter	uint8	
9	msg type	uint8	
10	client	uint8	**Body**
11	grasp mode	uint8	
12	object class	uint16	
14	tool	uint8	
15	pose format	uint8	
16	grasp feedback	uint8	
17	*offset*	–	
18	robot pose	int32[7]	
46	project index	uint32	
50 – 79	*padding*	–	

Response

Byte offset	**Data**	**Type**	
0	version	uint16	**Prefix**
2	length	uint32	
6	comm type	uint8	**Header**
7	reply code	uint8	
8	reply counter	uint8	
9	msg type	uint8	
10	version	uint16	**Body**
12	state	uint8	
13	grasp mode	uint8	
14	object class	uint16	
16	object instance	uint16	
18	stroke	int32	
22	angle offset	int32	
26	center offset	int32[3]	
38	tool	uint8	
39	pose format	uint8	
40	grasp pose	int32[7]	
68	object count	uint16	
70	candidate count	uint16	
72 – 79	*padding*	–	

Prefix

	version	length
value	3	74

The layout is described in the prefix chapter.

*Note: the length field denotes the **remaining** message size, which is 74 bytes.*

Header

The *header* is the part that encodes the message type and its intent.

Byte offset	0	6	7	8	9	10 – 79
Data	…	comm type	reply code	reply counter	msg type	…
Type	…	uint8	uint8	uint8	uint8	…
	Prefix	**Header**				**Body**

▼ Details

comm type

value	description
01	request
02	response

reply code

The reply code indicates the result status of the processed request.
Clients should always check the reply code in every single response message and only progress if the reply code is set to SUCCESS. If the reply code is not set to SUCCESS, then the rest of the message shall be ignored. However, the reply counter will still be incremented.

value	name	description
General		
1	SUCCESS	The server has handled the request succesfully
2	ERROR	The server has encountered an error while processing the request
Message-specific		
3	NO_OBJECT	No object found
4	NO_GRASP	No grasp found
5	INVALID_OBJECT_CLASS	Object class is either invalid or does not exist

Note: The reply code is ignored in requests.

reply counter

A counter that increments with each reply, resetting to zero when it reaches

256. *Note: The reply counter is ignored in requests.*

msg type

The message types are explained in detail in the next paragraph.

Message Types

The following table summarizes all availabe message types.

value	msg type	description
General		
0	reserved/unused	
1	GET_PROTOCOL_VERSION	Get the server's protocol version
2	GET_STATE	Get the server's current state
3	REGISTER_CLIENT	Register the client's robot system to the server
4	SET_PROJECT	Set the active project on the server
Grasping		
16	GET_GRASP	Get a grasp for the current scene
17	GRASP_FEEDBACK	Provide feedback to the server about the last grasp result
Detection		
32	GET_OBJECT_COUNT	Get the number of objects in the current scene
Robot State		
48	ROBOT_POSE	notify the server about the current robot pose

Messages

General

GET_PROTOCOL_VERSION

Returns the server's highest supported protocol version.
This is useful to see if the client's version is outdated or not. If this is the case, the client is advised to inform the user accordingly.

Request

Byte offset	0	2	6	7	8	9	10 – 79
Data	3	74	1	0	0	1	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–
	Prefix		**Header**				**Body**

Response

Byte offset	0	2	6	7	8	9	10	12 – 79
Data	3	74	2	reply code	counter	1	version	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	uint16	–
	Prefix		**Header**				**Body**	

▼ Details

> **version**
>
> The server's highest supported protocol version.

GET_STATE

Retrieves the server's state.

Request

Byte offset	0	2	6	7	8	9	10 – 79
Data	3	74	1	0	0	2	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–
	Prefix		**Header**				**Body**

Response

Byte offset	0	2	6	7	8	9	10	12	13 – 79
Data	3	74	2	reply code	counter	2	*offset*	state	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–	uint8	–
	Prefix		**Header**				**Body**		

▼ Details

state

value	name	description
1	INIT	The server is initializing
2	OPERATIONAL	The server is operational and ready to serve requests
3	STOPPED	The server has stopped operating
4	ERROR	The server has encountered a critical error

REGISTER_CLIENT

Registers the client's robot system (for informational purposes).
This message should be sent right after the connection has been established. However, there are no restrictions on when to send and how often. If the client's robot system is not officially supported (see client table in the request details), then simply ignore this message.

Request

Contains information about the client's robot system.

Byte offset	0	2	6	7	8	9	10	11 – 79
Data	3	74	1	0	0	3	client	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	uint8	–
	Prefix		**Header**				**Body**	

▼ Details

client

value	name	vendor
1	UR	Universal Robots
2	Kuka	Kuka
3	Yaskawa	Yaskawa
4	Fanuc	Fanuc
5	ABB	ABB
6	HORST	fruitcore robotics
128	Siemens PLC	Siemens

Response

The response body is empty.

Byte offset	0	2	6	7	8	9	10 – 79
Data	3	74	2	reply code	counter	3	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–
	Prefix		**Header**				**Body**

SET_PROJECT

Sets the active project on the server.

Request

Byte offset	0	2	6	7	8	9	10	46	50 – 79
Data	3	74	1	0	0	4	*offset*	project index	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–	uint32	–
	Prefix		**Header**				**Body**		

▼ Details

project index

The index of the project to activate.

Response

The response body is empty. The reply code indicates whether the operation was successful or not.

Byte offset	0	2	6	7	8	9	10 – 79
Data	3	74	2	reply code	counter	4	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–
	Prefix		**Header**				**Body**

Grasping

GET_GRASP

Requests a grasp for the current scene.

Request

Byte offset	0	2	6	7	8	9	10	11	12	14	15	16 – 79
Data	3	74	1	0	0	16	*offset*	grasp mode	object class	tool	pose format	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–	uint8	uint16	uint8	uint8	–
	Prefix		**Header**				**Body**					

▼ Details

grasp mode

value	name	description
1	ACTIVE_GRASP	The returned grasp must match the target object's active user-defined grasp definition
2	ANY_GRASP	The returned grasp must match any of the target object's user-defined grasp definitions
3	AUTO_GRASP	ANY_GRASP mode with model-free grasp-planning as fallback

object class

The class id of the target object. If the value is zero, then the class id is ignored and the target object is selected randomly.

tool

The tool to be used for the grasp.

value	name	description
1	EXTERIOR	Exterior two-finger parallel gripper
2	INTERIOR	Interior two-finger parallel gripper
3	CONTACT	Suction, magnet or adhesion with a contact area

pose format

Specifies the pose format to use in the response message. Refer to the *Pose Formats* section to see all supported formats.

Response

Byte offset	Data	Type	
0	3	uint16	**Prefix**
2	74	uint32	
6	2	uint8	**Header**
7	reply code	uint8	
8	counter	uint8	
9	16	uint8	
10	*offset*	–	**Body**
13	grasp mode	uint8	
14	object class	uint16	
16	object instance	uint16	
18	stroke	int32	

22	angle offset	int32
26	center offset	int32[3]
38	tool	uint8
39	pose format	uint8
40	grasp pose	int32[7]
68	object count	uint16
70	candidate count	uint16
72 – 79	*padding*	–

▼ Details

grasp mode

The used grasp mode, see request details for the enumeration values.

Note: This grasp mode is not necessarily the same as the requested grasp mode. For example, if an auto grasp is requested, the system will still prefer user-defined grasps and only resort to auto grasps if former either does not exist or is not applicable.

object class

The class id of the target object.

object instance

The instance id of the target object.

stroke

The distance in micrometers between both fingers to set before approaching the object. This field is only relevant for grippers and can be ignored for other types of tools.

Note: The gripper is at position zero when both fingers touch each other.

angle offset

Angular offset in microdegrees between the object model and the robot flange at position zero.

If an object has to be placed in a pre-defined orientation, align the robot flange by adjusting its position to match the angular offset (after grasping the object). Additionally, apply any custom use-case-specific offset as needed.

Note: The angular offset is not applicable when dealing with deformable objects or during auto grasps.

center offset

The translational [x,y,z]-offset in micrometers from the object center to the grasp point, expressed in the robot's base coordinate frame.

tool

The type of tool used in the request (same enumeration value).

pose format

The pose format used for the *pose*-field. Refer to the *Pose Formats* section to see all supported formats.

grasp pose

The grasp pose in the robot's base coordinate frame. The type of this field depends on the value in the *pose format*-field.

object count

The number of all detected objects (including the target object itself).

candidate count

The number of detected objects of the requested class id (including the target object itself).

Note: If the requested class id was 0 (= random), then all objects are counted.

GRASP_FEEDBACK

Provides *optional* feedback to the server about the last grasp result.

Request

Byte offset	0	2	6	7	8	9	10	16	17 – 79
Data	3	74	1	0	0	17	*offset*	grasp feedback	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–	uint8	–
	Prefix		**Header**				**Body**		

▼ Details

> **grasp feedback**
>
> Feedback about the last grasp result.
>
> | value | name | description |
> |---|---|---|
> | 1 | OK | The grasp was OK |
> | 2 | BAD | The grasp was not OK |

Response

The response body is empty.

Byte offset	0	2	6	7	8	9	10 – 79
Data	3	74	2	reply code	counter	17	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–
	Prefix		**Header**				**Body**

Detection

GET_OBJECT_COUNT

Requests the number of objects in the current scene.

Request

Byte offset	0	2	6	7	8	9	10	12	14 – 79
Data	3	74	1	0	0	32	*offset*	object class	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–	uint16	–
	Prefix		**Header**				**Body**		

▼ Details

> **object class**
>
> The class id of the objects to count. If the value is zero, then all objects of all classes are counted.

Response

Byte offset	0	2	6	7	8	9	10	14	16	68	70	72 – 79
Data	3	74	2	reply code	counter	32	*offset*	object class	*offset*	object count	candidate count	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	–	uint16	–	uint16	uint16	–
	Prefix		**Header**				**Body**					

▼ Details

object class

The requested class id of the objects to count. If the value is zero, then all objects of all classes are counted. **object count**

The number of detected objects of all classes.

candidate count

The number of detected objects of the requested class.

Note: If the requested class id was zero, then all objects are counted.

Robot State

ROBOT_POSE

Notifies the server about the current pose of the end-effector in the robot's base coordinate frame.

Request

Byte offset	0	2	6	7	8	9	10 - 14	15	16	18	46 - 79
Data	3	74	1	0	0	48	*offset*	pose format	*offset*	robot pose	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	-	uint8	-	int32[7]	-
	Prefix		**Header**				**Body**				

▼ Details

pose format

Specifies the format of the robot pose. Refer to the Pose Formats section to see all supported formats.

robot pose

The pose to be transmitted to the server.
Child frame: end-effector (TCP), parent frame: robot base.

Response

The response body is empty.

Byte offset	0	2	6	7	8	9	10 - 79
Data	3	74	2	reply code	counter	48	*padding*
Type	uint16	uint32	uint8	uint8	uint8	uint8	-
	Prefix		**Header**				**Body**

Visualization

A visualization of the latest detection result is provided as an image resource that can be accessed via HTTP at:

`http://<Server-IP>/monitor/latest_result`

Pose Formats

Following pose formats are supported:

value	name	type	layout	description
General formats				
1	Quaternion	int32[7]	position: x, y, z; orientation: qx, qy, qz, qw	Position and *unnormalized* quaternion. *Note: To normalize the quaternion, divide it by 10^6.* **Units:** Micrometer and radian.
2	Axis-Angle	int32[7]	position: x, y, z; orientation: x, y, z; –	Position and unnormalized rotation vector. **Units:** Micrometer and microradian. **Vendor:** Universal Robots
Vendor-specific formats				
16	WPR	int32[7]	position: x, y, z; orientation: W, P, R; –	Position and orientation **WPR** with **W** rotating around the *fixed* x-axis, then **P** rotating around the *fixed* y-axis, then **R** rotating around the *fixed* z-axis. **Rotation convention:** x-y-z (extrinsic). **Units:** Micrometer and microdegree. **Vendors:** Fanuc, Yaskawa
17	ABC	int32[7]	position: x, y, z; orientation: A, B, C; –	Position and orientation **ABC** with **A** rotating around the z-axis, then **B** rotating around the y'-axis, then **C** rotating around the x''-axis. **Rotation convention:** z-y'-x'' (intrinsic). **Units:** Micrometer and microdegree. **Vendor:** Kuka

Examples

Lets assume we've set up a TCP connection to the server successfully.
For simple grasping tasks, the most basic sequence of messages is:

▾ 1. GET_PROTOCOL_VERSION

The first thing we want to do after setting up a connection is to check the server's highest supported protocol version.

Request

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	1	uint8	**Header**
7	reply code	0	uint8	
8	reply counter	0	uint8	
9	msg type	1	uint8	
10 - 79	*padding*	0	-	**Body**

Data

```
000 003 000 000 000 074 001 000
000 001 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

Response

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	2	uint8	**Header**
7	reply code	1	uint8	
8	reply counter	0	uint8	
9	msg type	1	uint8	
10	version	3	uint16	**Body**
12 - 79	*padding*	0	-	

Data

```
000 003 000 000 000 074 002 001
000 001 000 003 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

In this example, the server answers with *version* = 3. This means that our client is up-to-date.

▼ . 2. GET_STATE

Before sending further requests to the server, we want to check the server's state.

Request

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	1	uint8	**Header**
7	reply code	0	uint8	
8	reply counter	0	uint8	
9	msg type	2	uint8	
10 – 79	*padding*	0	–	**Body**

Data

```
000 003 000 000 000 074 001 000
000 002 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

Response

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	2	uint8	**Header**
7	reply code	1	uint8	
8	reply counter	1	uint8	
9	msg type	1	uint8	
10	*offset*	0	–	**Body**
12	state	2	–	
13 – 79	*padding*	0	–	

Data

```
000 003 000 000 000 074 002 001
001 002 000 002 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

In this example, the server answers with *state* = 2. This means that the server is operational and we can proceed.

▼ . 3. REGISTER_CLIENT

We register our client system, which is a Siemens PLC in this example.

Request

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	1	uint8	**Header**
7	reply code	0	uint8	
8	reply counter	0	uint8	
9	msg type	3	uint8	
10	client	128	uint8	**Body**
11 – 79	*padding*	0	–	

Data

```
000 003 000 000 000 074 001 000
000 003 128 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

Response

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	2	uint8	**Header**
7	reply code	1	uint8	
8	reply counter	2	uint8	
9	msg type	3	uint8	
10 – 79	*padding*	0	–	**Body**

Data

```
000 003 000 000 000 074 002 001
002 003 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

The response body is empty.

▼ 4. SET_PROJECT

We activate the project with index 5, which contains the objects we want to detect and grasp.

Request

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	1	uint8	**Header**
7	reply code	0	uint8	
8	reply counter	0	uint8	
9	msg type	4	uint8	
10	*offset*	0	-	**Body**
46	project index	5	uint32	
50 - 79	*padding*	0	-	

Data

```
000 003 000 000 000 074 001 000
000 004 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 005 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

Response

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	2	uint8	**Header**
7	reply code	1	uint8	
8	reply counter	3	uint8	
9	msg type	4	uint8	
10 - 79	*padding*	0	-	**Body**

Data

```
000 003 000 000 000 074 002 001
003 004 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

In this example activating the project with index 5 was successful.
The response body is empty.

Note: Check the *reply code* in the response header and proceed only if the *reply code* is 1 (indicating success).

▼ 5. GET_GRASP

We request an auto grasp, where the target object class is random, the tool is an exterior gripper and the grasp pose shall be returned in quaternion representation.

Request

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	1	uint8	**Header**
7	reply code	0	uint8	
8	reply counter	0	uint8	
9	msg type	16	uint8	
10	*offset*	0	–	**Body**
11	grasp mode	3	uint8	
12	object class	0	uint16	
14	tool	1	uint8	
15	pose format	1	uint8	
16 – 79	*padding*	0	–	

Data

```
000 003 000 000 000 074 001 000
000 016 000 003 000 000 001 001
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
000 000 000 000 000 000 000 000
```

Response

Index	Name	Value	Type	
0	version	3	uint16	**Prefix**
2	length	74	uint32	
6	comm type	2	uint8	**Header**
7	reply code	1	uint8	
8	reply counter	4	uint8	
9	msg type	16	uint8	
10	*offset*	0	–	**Body**
13	grasp mode	2	uint8	
14	object class	3	uint16	
16	object instance	0	uint8	
18	stroke	86000	int32	
22	angle offset	-25210142	int32	
26	center offset	[-911, -310, 68]	int32[3]	
38	tool	1	uint8	
39	pose format	1	uint8	
40	grasp pose	[853798, 1111998, 502790, 775990, 630744, 0, 0]	int32[7]	
68	object count	2	uint16	
70	candidate count	2	uint16	
72 – 79	*padding*	0	–	

Data

```
000 003 000 000 000 074 002 001
004 016 000 000 000 002 000 002
000 000 000 001 079 240 254 127
082 226 255 255 252 113 255 255
254 202 000 000 000 068 001 001
000 013 007 038 000 016 247 190
000 007 172 006 000 011 215 054
000 009 159 216 000 000 000 000
000 000 000 000 000 002 000 000
000 000 000 000 000 000 000 000
```

In this example, the returned grasp is a user-defined grasp (grasp mode = 2). Even though we've requested an auto grasp, the system always prefers user-defined grasps and only resorts to auto grasps if former either does not exist or is not applicable.
In the response, we can also see that the *candidate count* is 2, which means that after applying the grasp there will be one candidate object left in the scene.

Note: Requesting a grasp may fail for various reasons, such as an uncalibrated camera, inability to locate the object, or an impractical grasp. It is crucial to verify the *reply code* in the response header and proceed only if the *reply code* is 1 (indicating success).

Steps 1 to 4 are performed only once. Step 5 typically iterates in a loop.

SCHUNK SE & Co. KG
Spanntechnik
Greiftechnik
Automatisierungstechnik

Bahnhofstr. 106 – 134
D-74348 Lauffen/Neckar
Tel. +49-7133-103-0
schunk.com
info@de.schunk.com

Follow us