Seervision Robotics API Specification

Version 1.0.0 DRAFT

1 About this document

1.1 Intended purpose

The purpose of the present document is to clearly and unambiguously document the Seervision Robotics (**sv::robo**) API and provide a basis upon which it can be implemented.

1.2 Versioning

This document is versioned via a **major**.**minor**.**patch** scheme following the Semantic Versioning Specification 2.0.0.

The most important implications if this are:

- Changes in patch version are “bug fixes” that do not break compatibility and neither change nor add functionality, e.g. version 0.3.1 only added some clarifications to this document compared to version 0.3.0.
- Changes in minor version do not break compatibility to previous versions but may add new functionality, e.g. version 0.4.0 will be compatible with 0.3.1 but may introduce new functionality. Changes in minor version can also be used to mark functionality as *deprecated*.
- Changes in major version may break compatibility to previous versions, e.g. version 0.3.1 may not be compatible with version 1.0.0.

All changes to the API are recorded in appendix under 7 Changelog at the end of this document.

Table of contents

2 General

The communication between the Seervision system, its *"director of photography"* (DOP), and any head supporting the *sv::robo API* consists of messages of different types and has a request/response structure, i.e., the *server* (remote head) is responding to requests by the *client* (Seervison DOP). As such **all requests can be sent asynchronously** and responses are expected to arrive within good time.

In the following, the term *axis* refers to an actuated axis, e.g. pan, tilt, zoom, focus, iris or other possible axes.

2.1 API incarnations

This API comes in multiple flavors (we call them *API incarnations*) to address slightly different capabilities on the robotics side. All incarnations share the communication structure and message format described in 3 Communication channel and 4 Message format. They only differ in the possible states of each axis and which transitions between axis states are possible, see 5 Axis state.

2.2 Open issues

- Specify directions of rotation for pan/tilt
- Add timestamp to measurements

3 Communication channel

The communication occurs over UDP, port **59629**.

4 Message format

4.1 Notation

We denote hexadecimal numbers via `0x` followed by numbers and letters consisting of `0123456789abcdef`, e.g. `0x32` is the number `50` in hex.

To describe the API, we use a JSON-like notation that is augmented to include types:

- `[ ... ]` is an array
  - Variable size arrays with a fixed (homogeneous) type are described as `[ type ... ]`, e.g. `[ float32 ... ]` for an array of floats.

    - Fixed size arrays with variable (heterogeneous) types are described as `[ type1 (name1), type2 (name2), ... ]`, where the names are not part of the protocol, they just help to understand the structure and give a recommendation for the naming in the implementation. E.g. `[ float32 (value), uint32 (number) ]`
- `{ ... }` is a key-value map
    - Maps have the following structure `{ key_type (key1): value_type (value1), (key2): value_type (value2), ... }`, e.g. `{ uint32 (key1): float32 (value1), (key2): float32 (value1) }`
- To simplify the notation, we often define our own objects (types). This happens e.g. as follows:

```
Header: [ uint32 (number), uint32 (type)]
```

    These types may appear in later message definitions. E.g. as

```
Message: [ Header (header), Payload (payload) ]
```

4.2 General message structure

All messages in this protocol follow the same structure consisting of a header with a fixed structure and a payload with a structure depending on the message type. Messages are serialized and deserialized using MessagePack, an efficient binary serialization format with implementations available for many languages including C/C++. We recommend msgpack-c for C/C++ users.

Every message consists of a header and a payload stored in an array:

```
Message: [ Header (header), Payload (payload) ]
```

where `Header` is the header structure, which is the same for all messages, and `Payload` is the payload structure which is different (a different structure/type) for each message type. The naming convention for the `Payload` in this document is:

- For the request message from client to server: `MessageTypeRequest`
- For the response message from server to client: `MessageTypeResponse`

Where `MessageType` is replaced with the name/type of the message.

Example

With *msgpack-c* the message can be implemented as the following struct

```
struct Message {
  Header header;
  Payload payload;
  MSG_PACK_DEFINE_ARRAY(header, payload);
};
```

4.3 Header structure

Each header contains a 4-byte *message number* to validate the integrity of the message and to correctly synchronize request and response; and the type of the message encoded as a 4-byte number.

```
Header: [ uint32 (session number), uint32 (number), uint32 (type)]
```

The field `session` number serves as a placeholder for future features where the server can be controlled through different sessions by multiple clients.

The `number` needs to be set for each request message and the corresponding response message needs to contain the same number. This allows the client to synchronize requests and responses. It is mandatory for the server to set the number of each response message to the message number of the corresponding request, however it is optional for the client to use this number in any way. The server cannot rely on there being any relationship between *message numbers* of consecutive messages, i.e., message numbers can be assigned perfectly arbitrarily by the client.

The `type` encodes the different message types as follows

Message type `Identifier (uint32)`	**Name**	**Description**
`0 (0x00)`	**Update reference / Get measurement**	*Request*: Update reference values for selected axes *Response*: Get measurements for all supported axes
`1 (0x01)`	**Set parameters**	*Request*: Set selected internal parameters of the head as a list of name-value pairs *Response*: Status of each parameter indicating whether it was set and why if not
`2 (0x02)`	**Get parameters**	*Request*: Request selected internal parameters of the head as a list of names *Response*: Receive list of values of the requested parameters
`3 (0x03)`	**Request state action**	*Request*: For each axis request an action on the state machine, e.g. transitioning to a specific state *Response*: Status of each axis indicating the current state and whether the state action was executed successfully and why if not
`4 (0x04)`	**Discover**	*Request*: Nil *Response*: Version information and network information (IP, mask, MAC)

`5 (0x05)`	**Set Network Data**	*Request:* Request new IP and network mask by device MAC *Response:* Status of the request

4.4 Payload structure with axes

Most data sent via the described protocol can be assigned to an axis (e.g. a reference or a measurement). We characterize axes via a unique `Identifier` (see below) of type `uint32`. Therefore message payloads are usually structured as a map from axis identifiers to the axis-specific payload or data, i.e.

```
{ uint32 (axis): Data (axis data), ... }
```

The following table describes named axis identifiers. Note that which axes are supported/implemented is decided by the server-side implementation. Additional axes can be added to the protocol upon request

Axis `Identifier` (uint32)	**Axis name**	**Description**
`0 (0x00)`	global	Used for all non-axis-specific values
`1 (0x01)`	pan	
`2 (0x02)`	tilt	
`3 (0x03)`	roll	
`4 (0x04)`	zoom	
`5 (0x05)`	focus	
`6 (0x06)`	iris	
`7 (0x07)`	x	
`8 (0x08)`	y	
`9 (0x09)`	z	
`10 (0x0a)`	range	For object distance (ranging) measurements

4.5 Discover message

The echo message can be used to ping the robot or discover its network settings. It can also be used as a default message type.

The request is an empty message (Header only).

```
DiscoverRequest: nil
```

The response carries the following information about the API version and information about network settings:

```
DiscoverResponse: {
      uint32 (version): [uint32 (major api version), uint32 (minor api version),
uint32 (api type)],
      uint32 (network info): [ NetworkInfo, ...]
}
```

Note that Payload in this case is a map with two keys. The key `network info` points to an array of data. Each element is a `NetworkInfo` type which in itself is an array of data that identifies the essential network settings:

```
NetworkInfo: [string(IP), string(network mask), const string(MAC)]
```

The `version` and `network info` keys in the above `DiscoverResponse` are set as defined below:

Discover info Identifier (uint32)	**Name**	**Description**
`0 (0x00)`	**version**	API version and type implemented by the server
`1 (0x01)`	network info	Information about the networking parameters

The `major api version` and `minor api version` indicate the API version as outlined in 1.2 Versioning. The patch version is omitted because it only impacts the API documentation, not its implementation. The `api type` defines which set of functionalities is implemented on the server-side, see 2.1 API incarnations and 6.1 Supported API incarnations.

`NetworkInfo` contains the `IP`, network mask and MAC address under which the robot is reachable.

Example

We have a server that implements the API version 0.4.3 and the *light* API incarnation, meaning `api type` is equal to `1 (0x01)`, see 6.1 Supported API incarnations.. Moreover, it is reachable under the IP `192.168.0.12` with the network mask `255.255.255.0` and the MAC address `06:55:d7:e4:5c:fd`. This server would respond to a Discover request message with a Discover response with the following payload.

```
DiscoverResponse: {
  version: [0, 4, 1],
  network info: ["192.168.0.12", "255.255.255.0", "06:55:d7:e4:5c:fd"]
```

```
}
```

`Where version` and `network info` need to be replaced with their corresponding identifiers, i.e., `0` and `1`, respectively.

4.6 Update reference/get measurement message

The payload of the reference messages are maps from axis to data. The axis data is itself a map from an `Identifier (uint32)` to a `value (float32)` with one exception. In case of the `Identifier` being `timestamp` (which is only sent by the server and not the client), the value it points to is a `uint64`. For request messages these values are reference and for response messages these values are measurements. The identifier encodes what the value is including its unit, e.g. an angular position in [deg] or an angular velocity in [deg/s] used for pan/tilt axes, or a unit position value in [0,1] used for zoom/focus axes. The available/supported identifiers/types can be found in 6.2 Supported payload types for reference/measure messages.
Important: All references and measurements are `float32` and no other types are supported for these values.

4.6.1 Request (update references)

```
ReferenceRequest: {
  uint32 (axis): { uint32 (valueIdentifier): float32 (reference value), ... },
  ...
}
```

For every axis, a valid combination of keys for the `ReferenceRequest` exists (e.g. a pan axis might accept a single angular velocity value). When a request contains an axis with value nil, e.g. `ReferenceRequest: { pan: nil }` then the server must keep applying the previous reference(s) of that axis. Moreover, it must return the measurements for that axis in its response, see below. If an axis is present in the request and not nil, then it must contain **all** the required references and should be rejected by returning the **Invalid** status (see the response below) if it does not.

The server is expected to implement reasonable and safe defaults for references that have not yet been set via a reference request message.

Important: Reference requests only update the reference when the axis is in *running* state (see 5.1 States). Therefore, they also can only lead to motion when the axis is in *running* state. If the server receives a reference request while it is not in *running* state it will return a `WrongState` error.

Example

We consider a robot supporting three axes:

- `pan` with a single reference of type `angularVelocity`

- `x` with a single reference of type `position`, and
- `zoom` with a single reference of type `unitPosition`.

Note that in an actual implementation of the following messages `pan, x, zoom` and `angularVelocity`, `position`, `unitPosition` would need to be replaced by their corresponding identifiers.

The following complete reference request message was successfully sent to the server

```
ReferenceRequest: {
  pan: { angularVelocity: 0.1 },
  x: { position: 1.2 },
  zoom: { unitPosition: 0.7 }
}
```

Afterwards, another, now incomplete, reference request message is sent

```
ReferenceRequest: {
  pan: nil,
  x: { position: 1.0 }
}
```

which yields a message with the following payload:

```
ReferenceRequest: {
  1: nil,
  7: { 1: 1.0 }
}
```

Based on this and the previous reference request message, the following references are applied:

```
{ pan: { angularVelocity: 0.1 },
  x: { position: 1.0 },
  zoom: { unitPosition: 0.7 } }
```

4.6.2 Response (get measurements)

```
ReferenceResponse: {
  uint32 (axis): [ uint32 (status),
                   { uint32 (valueIdentifier): float32 (measurement value), ...,
                     uint32 (optional timestamp): uint64 (time) }
                 ],
  ...
}
```

For every axis in the request, a response is returned. If the status of the response is **Success**, **Unchanged** or **Invalid**, all measurements for that axis must be returned. If the status is **Error**, a measurement should be returned, if possible.

The reference status contained in the payload of the response message indicates whether the axis reference was updated successfully.

Reference status `Identifier (uint32)`	Name	Description
`0 (0x00)`	**Success**	Acknowledges that the requested axis reference was updated successfully
`1 (0x01)`	**Unchanged**	The axis reference was not updated because the axis or reference value were not supplied in the corresponding request
`2 (0x02)`	**Invalid**	The axis reference was not set because it was out-of-range, i.e., either too small or too large, or it was not finite, i.e. it was NaN or Inf
`3 (0x03)`	**Error**	The axis reference could not be set because faults were present in the axis, see 5.3 Faults
`4 (0x04)`	**NonExistent**	The axis reference does not exist on this robot
`5 (0x05)`	**WrongState**	If the axis is not in running state.

Example

As in the previous example, we consider a robot supporting the axes `pan`, `x` and `zoom`. The axes provide the following measurements:

- `pan` supports a single measurement of type `angularPosition`
- `x` supports two measurements of types `position` and `velocity`, and
- `zoom` supports a single measurement of type `unitPosition`.

After receiving the above (second) reference request message (restated here for convenience)

```
ReferenceRequest: {
  pan: nil,
  x: { position: 1.0 }
}
```

the server responds with a response containing all available measurements of the requested axes

```
ReferenceResponse: {
  pan: [ Unchanged, { angularPosition: 21.0 }],
```

```
  x: [ Success, { position: 1.04, velocity: -0.93 } ]
}
```

Notice that no measurement for the zoom axis is returned since it was not requested.

4.6.3 Timestamp

If the response contains a timestamp, it is represented as `uint64 (microseconds)`, where time is measured since Epoch according to POSIX specifications available here.

4.7 Set parameters message

Parameters are values that parametrize the server-side implementation. They can either be *mutable* (they can be changed via a set parameter message) or *immutable* (they cannot be changed - only inspected). Moreover, parameters can either apply to axes or apply independently of an axis, i.e., the `global` axis.

Parameters can have different types but must have one of the following types

```
bool, int32, int64, uint32, uint64, float32, float64
```

Each parameter is identified via a unique `Identifier (uint32)`, the supported parameters are given in 6.3 Supported parameters.

4.7.1 Request (set parameter)

Parameter is a map from parameter identifiers to values

```
Parameters: {
  uint32 (id): any_of<bool, int32, int64, uint32, uint64,
                                  float32, float64> (value)
  ...
}
```

The request message consists of a map from axes to parameters, where it is optional to include axes without any parameters to be set

```
SetParameterRequest: { uint32 (axis): Parameters parameters, ... }
```

Note

We recommend limiting set parameter messages such that the payload does not exceed 500 bytes. A good rule of thumb is fewer than 50 parameters - if not float64 parameters are used. This is because the maximum (definitely safe for all cases) UDP packet payload is 508 bytes [1], this leads to our maximum (definitely safe for all cases) payload of 500 bytes.

4.7.2 Response (set parameter)

For each parameter that has been requested to be set, a status is returned

```
ParameterStatuses: { uint32 (id): uint32 (status), ... }
```

As in the request, axes map to parameter statuses

```
SetParameterResponse: { uint32 (axis): ParameterStatuses statuses, ... }
```

The parameter status indicates whether a parameter was set successfully and why if not.

Parameter status `Identifier` `(uint32)`	**Name**	**Description**
`0 (0x00)`	**Success**	Acknowledges that the parameter was set successfully
`1 (0x01)`	**NonExistent**	The parameter could not be set because it does not exist on the server
`2 (0x02)`	**Invalid**	The axis parameter was not set because it was out-of-range, i.e., either too small or too large, or it was not finite, i.e. it was NaN or Inf
`3 (0x03)`	**Denied**	The parameter could not be set because it is cannot be written (the parameter is immutable)

4.8 Get parameters message

See 4.7 Set parameters message for more details on parameters.

4.8.1 Request

```
GetParametersRequest: { uint32 (axis): [ uint32 (id) ... ] }
```

4.8.2 Response

```
GetParametersResponse: {
   uint32 (axis): Parameters parameters, ...
}
```

Only parameters that exist on the server and axis are returned.

Note

We recommend limiting get parameter messages such that the payload does not exceed 500 bytes. A good rule of thumb is fewer than 40 parameters - if no float64 parameters are used. This is because the maximum (definitely safe for all cases) UDP packet payload is 508 bytes [1], this leads to our maximum (definitely safe for all cases) payload of 500 bytes.

4.9 Request state action

The request state action message can be used to poll the current state of all available axes, request transitions axis state transitions and to reset faults.

4.9.1 Request

The request is simply a desired state for each axis that a state change should be performed. Additionally, one can either poll for the state response (Poll State) or ask for fault reset (Reset Fault) by sending the respective request.

```
StateActionRequest: { uint32 (axis): uint32 (state action) }
```

State action `Identifier (uint32)`	Name	Description
`0 (0x00)`	**Poll state**	No action is requested. Can be used to poll for the current state.
`1 (0x01)`	**Request disconnected state**	
`2 (0x02)`	**Request disabled state**	
`3 (0x03)`	**Request ready state**	
`4 (0x04)`	**Request running state**	
`5 (0x05)`	**Request stopping state**	
`6 (0x06)`	**Request auto calibration state**	
`7 (0x07)`	**Request manual**	

	calibration state	
`8 (0x08)`	**Reserved**	
`9 (0x09)`	**Reset faults**	Reset all faults. This does not cause a state transition.

4.9.2 Response

The response carries the current state and an array of active faults for all axes present in the request. Faults are described in 5.3 Faults.

```
StateActionResponse: { uint32 (axis): [ uint32 (state),
                                        [ uint16 ... ] (faults) ] ...
                     }
```

Important: There is no guarantee that the requested state is applied immediately and that the response already contains the requested state as the current state.

State `Identifier (uint32)`	Name	Description
`0 (0x00)`	**Reserved**	Not a valid state. Should never be returned
`1 (0x01)`	**Disconnected**	No communication channel has been established yet with the axis
`2 (0x02)`	**Disabled**	The axis is connected, but not powered/no torque
`3 (0x03)`	**Ready**	The axis is possibly powered, but does not move
`4 (0x04)`	**Running**	The axis is powered and reference requests lead to motion
`5 (0x05)`	**Stopping**	The axis is aborting any movement. As soon as it stopped, a state transition is initiated
`6 (0x06)`	**Auto calibration**	The axis calibrates itself. It returns to *ready* as soon as it finished
`7 (0x07)`	**Manual calibration**	The axis allows external calibration to happen
`8 (0x08)`	**Disarmed**	The robot is disarmed, i.e. there is no torque

4.10 Change Network message

Change network message can be used to change the current IP address or network mask of the robot. The message can either be sent directly to the robot or broadcast on LAN.

4.10.1 Request

The request is the desired IP address and network mask for the given MAC address. For the sake of consistency, this payload is also a map.

```
NetworkChangeRequest: { uint32 (network info): NetworkInfo }
```

4.10.2 Response

The response contains status which indicates whether a network was changed successfully and current values of network data.

```
NetworkChangeResponse: { uint32 (status): bool (status), uint32 (network info):
NetworkInfo }
```

Change Network `Identifier (uint32)`	Name	Description
`0 (0x00)`	`status`	Response status
`0 (0x01)`	`network info`	Network change request and response key value

5 Axis state

Each robot implementing the sv::robo API as a server, based on its functionality, supports a number of axes such as e.g. pan, tilt, zoom and/or focus. As such, each axis needs to keep track of the following:

- Its internal state encoded by a `uint32 (axis state)`
- An array of present (not yet reset) faults `[ uint16 ... ] (faults)`
- Whether the axis is calibrated or not `bool (calibrated)`

Notes

- We are only using the JSON-like notation of 4.1 Notation for convenience, the above quantities are not message payloads that are exchanged.
- We recommend implementing `bool (calibrated)` as an immutable parameter available for all axes whose default indicates whether an axis must be calibrated or not.

5.1 States

Each axis always needs to be in one of the following states. Note that depending on the API incarnation only some essential states need to be implemented. The other states are optional. Valid state transitions and how they can occur are described in 5.2 State transitions.

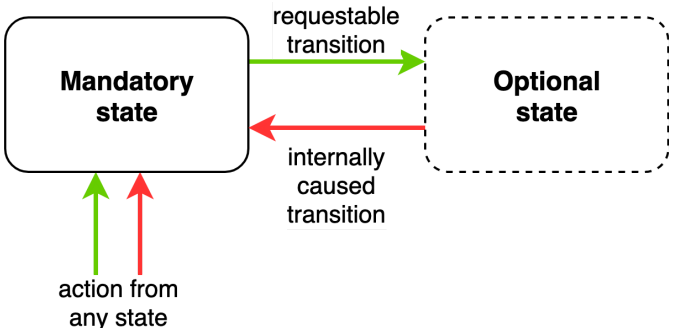
Name	Description
Disconnected	Communication with the axis is not established. This is the default state of each axis when the robot is started.
Disabled	The axis is inert, specifically motors are not powered or do not produce any torque.
Ready	The axis is powered and makes sure there is no motion, e.g. the pan or tilt position is kept. This means the motors can produce torque but should not move the axis.
Running	The axis is operational and can move. This is the only state where axis references can actually lead to motion in the axis.
Stopping	The axis is in the process of performing a controlled stop. This stop can be requested via the sv::robo API or in cases of faults that require the axis to stop it can also be initiated by the server-side firmware running on the robot. When the axis has come to a stop the server-side firmware executes a transition to either *disconnected*, *disabled* or *ready* depending on whether there are faults present in the axis and if there are on their severity. See 5.3 Faults for more details
Auto calibration	This initiates an auto calibration procedure to calibrate the axis. Once the action terminates (successfully or unsuccessfully) the server-side firmware automatically transitions to the *ready* state. If the auto calibration procedure was successful the is set to `calibrated = true`.
Manual calibration	This puts the axis in a state where it can be calibrated manually with the user's help. When the calibration is completed the client-side can send a request ready state message at which point `calibrated = true` is set.
Disarmed	The axis was disarmed via some external action, e.g. a disarm button. In this state the axis is inert, specifically motors are not powered or do not produce any torque (same as disabled). This state can only be left via some other external action to again arm the axis. In this case the next state is *disabled*. The sv::robo API intentionally does not provide a mechanism for leaving this state via a request state action message.

5.2 State transitions

Because each axis has its own state, a state machine needs to be implemented for each axis with transitions as illustrated below.

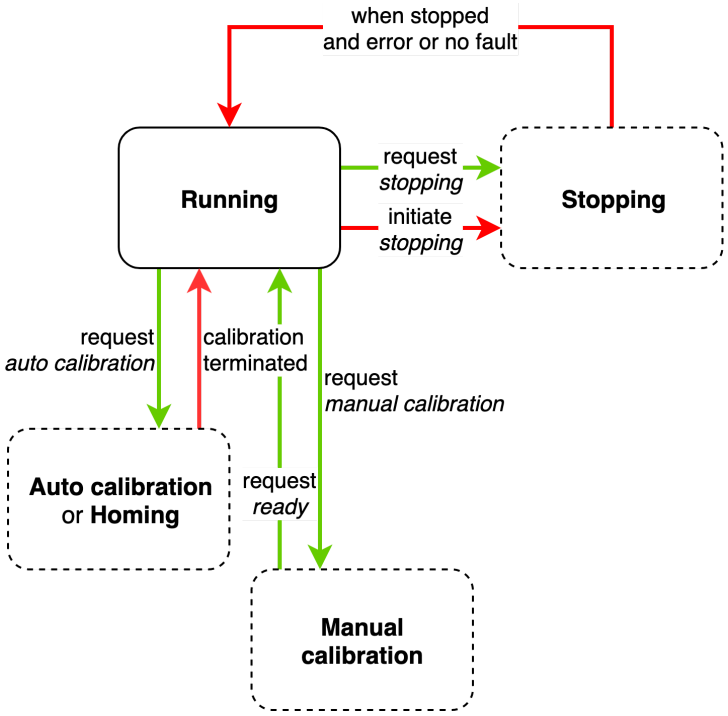
5.2.1 Legend

Only mandatory states need to be implemented. The presence of faults causes any request for a state transition to be denied until all faults are reset via the request reset faults.



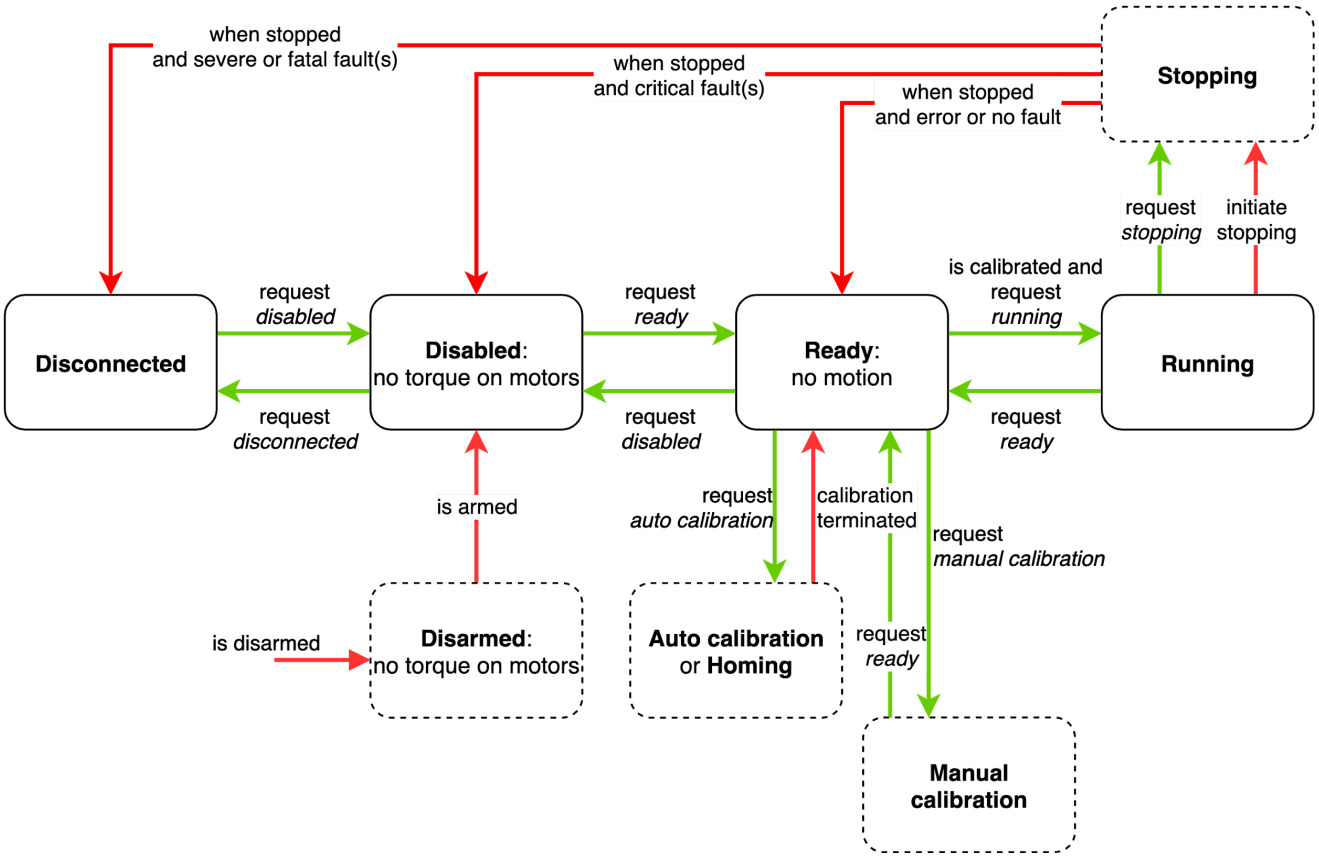
5.2.2 Light incarnation

For the light incarnation only the state *running* is essential and needs to be implemented. *Manual* and *auto calibration*, as well as the *stopping* state are optional.



5.2.3 Nominal incarnation

For the nominal version the states *disconnected*, *disabled*, *ready* and *running* are essential and need to be implemented. *Disarmed*, *manual* and *auto calibration*, as well as the *stopping* state are optional.



5.3 Faults

There are four levels of faults. A fault is encoded in a `uint16` , where the two most significant bits encode the level and the remaining 14bits encode the error code. This code might convey more detailed information about the fault and the system might take appropriate actions if it knows about the specific fault. See 6.4 Supported faults for a list of available faults.

Level	Range `(uint16)`	Description
Error	`0x0000` to `0x3fff`	When this fault occurs, the operation is stopped and the axis is brought to the *ready* state. A *reset fault* state action request can be used to reset this fault, after which the axis can be brought into *running* state again.
Critical	`0x4000` to `0x6fff`	When this fault occurs, the operation is stopped and the axis is disabled, i.e., it is brought into the *disabled* state. A *reset fault* state action request

		can be used to reset this fault, after which the axis can be brought into *ready* and then *running* state again.
Severe	`0x8000` to `0xafff`	When this fault occurs, the operation is stopped and the axis is disconnected, i.e., it is brought into the *disconnected* state. A *reset fault* state action request can be used to reset this fault, after which the axis can be brought into *disabled*, then *ready* and then *running* state again.
Fatal	`0xb000` to `0xffff`	When this fault occurs, the operation is stopped and the axis is disconnected, i.e., it is brought into the *disconnected* state. This fault indicates that something is majorly broken and no safe usage is possible anymore. The robot needs to be inspected by the manufacturer. This fault can therefore not be reset via the sv::robo API. It should not even be resettable by a reboot of the robot.

When multiple faults occur at the same time the response is determined by the most severe fault.

Important: Any of these faults can only be reset by the client via the sv::robo API (if at all), never by the server itself.

6 Appendix

6.1 Supported API incarnations

Value `Identifier (uint32)`	Name	Description
0 (0x00)	light	Nominal functionality with a 4-stage core state machine
1 (0x01)	nominal	Light functionality with 1-stage core state machine

6.2 Supported payload types for reference/measure messages

Value `Identifier (uint32)`	Name	Description
0 (0x00)	any	Not specified value
1 (0x01)	position	Position in [m]
2 (0x02)	velocity	Velocity in [m/s]
3 (0x03)	acceleration	Acceleration in [m/s^2]
4 (0x04)	unitPosition	Relative position in [0, 1]
5 (0x05)	unitVelocity	Velocity in [0, 1]/s
6 (0x06)	unitAcceleration	Acceleration in [0, 1]/s^2
7 (0x07)	angularPosition	Angular position in [deg]
8 (0x08)	angularVelocity	Angular velocity in [deg/s]
9 (0x09)	angularAcceleration	Angular acceleration in [deg/s^2]
10 (0x0a)	current	Current in [A]
11 (0x0b)	torque	Torque in [Nm]
12 (0x0c)	timestamp	time since epoch according to POSIX specifications available here. [usec]

6.3 Supported parameters

Parameters are always set for a specific axis (see 4.7 Set parameters message and 4.4 Payload structure with axes)

6.3.1 Parameter properties

- Parameters that must be supported by the server are marked as `mandatory`. Parameters that must be supported if a specific functionality is available are marked as conditional
- Parameters that are immutable (cannot be changed) are denoted by `const`. Parameters that are not marked as `const` may (or may not) be immutable depending on the server implementation. The response to a `setParameterRequest` for a that cannot be changed on the server, because it is immutable, must return a response with status **Denied**.
-

Examples

- The parameter `major api version` is `mandatory const`. That means it must be supported by any server implementation. Moreover, no server implementation should allow for this parameter to be changed via a `setParameterRequest` of this API.
- The parameter `watchdog enabled` is `conditional` that means it must be supported if a watchdog functionality is implemented (as indicated in its description). Moreover, since it is not marked as `const` it can be mutable or immutable. E.g., if a server implementation has a watchdog implemented (and enabled) but does not support disabling it (i.e., the parameter is immutable), then it can simply respond to a `setParameterRequest` with status **Denied**.

6.3.2 List of supported parameters

Below is a list of all currently supported parameters.

Parameter `Identifier (uint32)`	**type (name)**	**Description**
`0`	`mandatory const uint32 (major api version)`	The sv::robo::API major version. Every change in major version should be expected to break backwards compatibility
`1`	`mandatory const uint32 (minor api version)`	The sv::robo::API minor version. Changes in minor version do not break backwards compatibility
`2`	`mandatory const uint32 (api incarnation)`	The API incarnation, see 2.1 API incarnations
`3`	`const uint32`	The maximum number of parameters the server

	(max parameters response)	processes and responds to for set/get parameter requests. If the parameter is unset or zero then there is no limit
4	bool (watchdog enabled)	Whether the safety watchdog is enabled. If the parameter is unset, then there is no watchdog
5	float32 (watchdog timeout)	Safety watchdog timeout in seconds. If the parameter is unset, then there is no watchdog
6	float32 (minimal limit)	The minimal axis limit
7	float32 (maximal limit)	The maximal axis limit

6.4 Supported faults

Below is the list of currently supported faults.

Fault `Identifier (uint32)`	Name	Description
Errors		
`0x0000`	Generic error	
Critical faults		
`0x4000`	Generic critical fault	
Severe faults		
`0x8000`	Generic severe fault	
`0x8001`	Connection failure	Could not connect to axis
Fatal fault		
`0xb000`	Generic fatal fault	

7 Changelog

7.1 Version 0.4.0 -> 1.0.0

- All payloads have been changed to be maps.
- A new error `WrongState` is added in case an update reference is requested for an axis which is not yet in running state.
- A new field `session number` has been added to the header.
- Network discovery has been added.
- Change network message has changed.
- An optional `timestamp` field is added to the `get measurement` response of an axis.

7.2 Version 0.3.1 -> 0.4.0

- The response to get and set parameter requests now needs to give a response for each parameter (and axis) requested even if it does not exist. To allow the server to cope with requests of arbitrary size, it may drop all except N >= 1 parameters, where N can be inspected as parameter with `Identifier = 3`. This means only parameters for which a response is returned have been processed by the server and all other parameters can be requested again if needed.
- Added parameter properties `mandatory` and `conditional`. Clarified their meaning and usage in 6.3.1 Parameter properties
- The list of parameters in 6.3 Supported parameters has been extended with
    - The maximum number of parameters that the server responds to have been added as a `const` parameter with `Identifier = 3`
    - The watchdog enabled and watchdog timeout parameters
    - The min and max limit parameters
- The versioning section was updated to clarify that Semantic Versioning 2.0.0 will be followed in the versioning of this API

7.3 Version 0.3.0 -> 0.3.1

- Added Changelog
- Added section explaining how this document is versioned