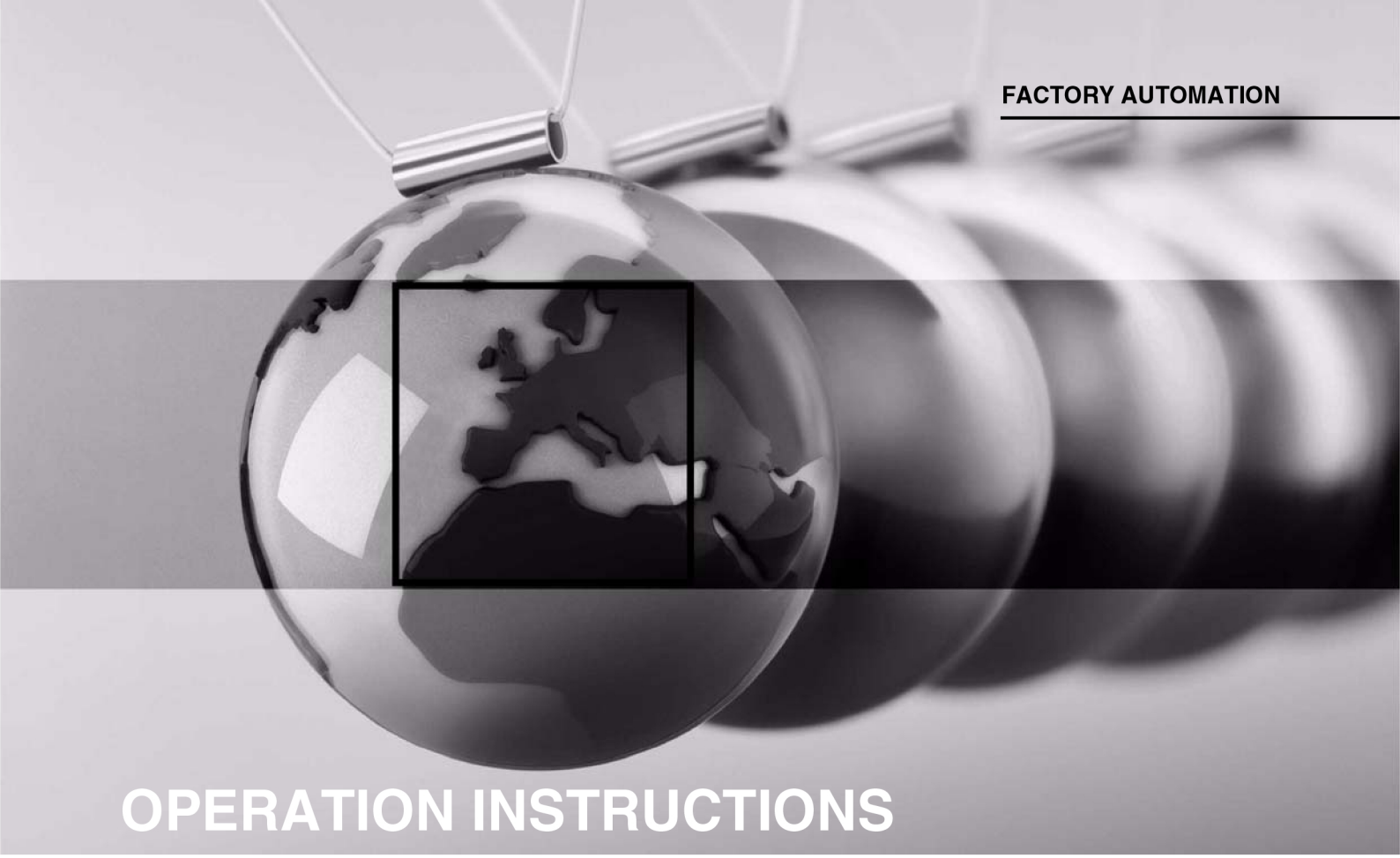


OPERATION INSTRUCTIONS

OMDxxx–R2000

Ethernet communication protocol

Protocol version 1.04

PEPPERL+FUCHS
SENSING YOUR NEEDS

Contents

1 Protocol basics

This chapter describes the basics of the Pepperl+Fuchs scan data protocol (PFSDP).

1.1 Basic design

The communication protocol specification is based on the following basic design decisions:

- A simple command protocol using HTTP requests (and responses) is provided in order to parametrize and control the sensor. The HTTP can be accessed using a standard web browser or by establishing temporary TCP/IP connections to the HTTP port.
- Sensor process data (scan data) is received from the sensor using a separate TCP/IP or UDP/IP channel. A TCP channel is recommended for every application that requires a reliable and error proof transmission of scan data. An UDP channel is recommended for applications in need of minimum latency transmission of scan data.

Output of scan data is always conform to the following conventions:

- Data output is performed as packets with a packet size adapted to the common Ethernet frame size (TCP as well as UDP).
- A single packet always contains data of a single continuous scan only. Scan data output always starts with a new packet for every (new) scan.
- For scan data output the user application can select from multiple data types with different levels of information detail. This way a client can decide to receive only the amount of data needed for its individual application – reducing traffic. Furthermore this provides an easy way to implement future extensions to the scan data output (e.g. adding additional information) as well.
- The byte order for all binary data is *Little Endian* (least significant byte first). The DSP of the sensor and PC CPUs both use Little Endian – thus no conversions need to take place.
- The sensor capabilities restrict the number of active (concurrent) client connections. This does not imply though that the device can handle multiple concurrent connections with the maximum amount of (scan) data. Basically it is the users responsibility to design his (client) system or application in a way that the sensor can handle the amount of requested data without getting overloaded.

1.2 HTTP command protocol

The HTTP command protocol provides a simple unified way to control sensor operation for application software. HTTP commands are used to configure sensor measurement as well as to read and change sensor parameters. Furthermore it is used to set up (parallel) TCP or UDP data channels providing sensor scan data.

This section describes the basic HTTP command protocol and various commands available to the user. Transmission of scan data using an additional TCP or UDP channel is explained in section 3.4.

Please note:
The R2000 provides full support for HTTP/1.1 – but does currently not support persistent connections (which is optional according to the HTTP/1.1 standard [5]). Each HTTP response includes the `"Connection: close"` header to inform the client that a subsequent HTTP request requires a separate TCP/IP connection to the sensor.

1.2.1 Sending commands

Sending commands to the sensor is done using the Hypertext Transfer Protocol (HTTP) as defined by RFC 2616 [5]. Each HTTP command is constructed as **Uniform Resource Identifier** (URI) according to RFC 3986 [7] with the following basic structure:

```
<scheme>:<authority>/<path>?<query>#<fragment>
```

A typical HTTP request to the sensor looks like:

```
http://<sensor IP address>/cmd/<cmd_name>?<argument1=value>&<argument2=value>
```

Thus, in terms of an URI a valid HTTP command is composed of the following parts:

- **scheme** is always 'http://'
- **authority** is represented by the IP address of the sensor (and a port number, if necessary)
- **path** consists of the prefix 'cmd/' and the name of the requested command ('<cmd_name>')
- **query** lists additional arguments for the specific command
- **fragment** is currently not used – anything following the hash mark will be ignored

Please note:
The order of the command arguments (within <query>) is interchangeable at will. Sole exception is the argument handle (see section 3.3), which has to be specified always first in order to identify the client scan data connection – provided that this is required for the requested command.

Please note:
The number of command arguments (within <query>) is limited to 100. Furthermore, the maximum length of a HTTP request URI is limited to 16 kB. Typical user application do not exceed these limitation, though.

1.2.2 Query argument encoding

The query part of the command URI (see section 1.2.1) is used to transport additional arguments to HTTP commands (compliant to RFC 3986 [7]). This section describes the composition of arguments as "key=value" pairs.

HTTP command arguments are composed using the following scheme ("key=value" pairs):

```
key=value[;value][&key=value]
```

The key denotes an argument that receives one or more values. Multiple values for a single argument are separated by a semicolon ';'. A single command takes multiple arguments separated by an ampersand '&'.

Please note:
Some characters are reserved within an URI and need to be percent encoded according to the rules of RFC 3986 [7]. Most notably, if parameter values contain URI delimiters like the question mark '?', equal sign '=' or the ampersand '&', these characters need to be escaped on the URI.

1.2.3 Replies to commands

After sending a command to the sensor the following replies can be received:

- **HTTP status code**
  A HTTP command will be answered with a standard HTTP status code first. This code indicates whether the command (i.e. URI) is known and has been received correctly. An error code is returned only if the URI is invalid or could not be processed. Please refer to section 1.2.5 for a detailed description of HTTP status codes used by the R2000.
- **Command error code**
  Normally the HTTP status code is read as 'OK'. In this case the result of the command processing can be evaluated using two return values: error_code and error_text. error_code contains a numeric result code for the command call, while error_text provides a textual error description. Both values are returned using JSON encoding [9]. Section 1.2.6 provides a detailed description of all R2000 command error codes.
- **Command reply data**
  The body of a command reply contains any requested payload data. This data is always transmitted using JSON encoding [9]. Large amounts of data might be output using base64 encoded JSON arrays.

Please note:
The character encoding used for all JSON encoded command replies of the R2000 is always UTF-8 (RFC 7159 [9]).

1.2.4 HTTP request and reply – low level example

This section shows an example, how a HTTP request is transmitted to the sensor without using a web-browser. Lets assume, that the following HTTP request shall be send:

```
http://<sensor IP address>/cmd/get_parameter?list=scan_frequency
```

This request is translated into a simple string (using HTTP/1.0 in this example):

```
GET /cmd/get_parameter?list=scan_frequency HTTP/1.0\r\n\r\n
```

This string is then send as payload data of a TCP/IP packet to the sensor. The sensor then sends back a TCP/IP packet with the HTTP reply as payload data. The HTTP reply can be parsed as simple text string with the following content:

```
HTTP/1.0 200 OK\r\n
Expires: -1\r\n
Pragma: no-cache\r\n
Content-Type: text/plain\r\n
Connection: close\r\n
\r\n
{\r\n
"scan_frequency":50,\r\n
"error_code":0,\r\n
"error_text":"success"\r\n
}\r\n
```

The most important parts of this HTTP reply are the first line containing the HTTP error code and the last few lines containing the requested information wrapped within a single JSON-encoded [9] object denoted by a pair of curly brackets.

Please note:
It is highly recommended to use a third party HTTP library instead of a new custom implementation. Standards-compliant HTTP client implementations are widely available for most operation systems and hardware platforms (e.g. `Libwww` [11] or `libcURL` [12]).

1.2.5 HTTP status codes

The following table lists all HTTP status codes used by the device:

status code	message	description
200	OK	request successfully received
400	Bad Request	wrong URI syntax or URI too long
403	Forbidden	permission denied for this URI
404	Not Found	unknown command code or unknown URI
405	Method not allowed	invalid method requested (currently only `GET` is allowed)

Examples for (invalid) queries causing a HTTP error

request	status code	error message
http://<ip>/cmd/nonsense	400	"unrecognized command"
http://<ip>/cmd/get_parameter&test	400	"unrecognized command"
http://<ip>/cmd/get_parameter?list	400	"parameter without value"
http://<ip>/test	404	"file not found"
http://<ip>/test/	404	"file not found"
http://<ip>/test/file	404	"file not found"

1.2.6 Sensor error codes

The following table lists some generic error codes (`error_code`) returned by the device:

error code	description (`error_text`)
0	"success"
100	"unknown argument '%s'"
110	"unknown parameter '%s'"
120	"invalid handle or no handle provided"
130	"required argument '%s' missing"
200	"invalid value '%s' for argument '%s'"
210	"value '%s' for parameter '%s' out of range"
220	"write-access to read-only parameter '%s'"
230	"insufficient memory"
240	"resource already/still in use"
333	"(internal) error while processing command '%s'"

Please note:
The error message in `error_text` might slightly vary depending on the firmware version and the specific error condition of the actual command. Do not expect to receive error messages exactly as listed above.

Examples for (invalid) commands provoking sensor error codes

command (query)	code	error message
/cmd/get_protocol_info?list=test	100	"Unknown argument 'list'"
/cmd/get_parameter?list=test	110	"Unknown parameter 'test'"
/cmd/start_scanoutput	120	"Invalid handle or no handle provided"
/cmd/start_scanoutput?handle=test	120	"Invalid handle or no handle provided"
/cmd/set_parameter?ip_address=777	200	"Invalid value '777' for argument 'ip_address'."
/cmd/set_parameter?scan_frequency=999	210	"Value '999' for parameter 'scan_frequency' is out of range."
/cmd/set_parameter?serial=123456	220	"Write-access to read-only parameter 'serial'"

1.2.7 Protocol information (`get_protocol_info`)

Ethernet protocol users should be aware that depending on the protocol version some commands might not be available or might show different behavior. For this reason user applications should always check the protocol version using the dedicated command `get_protocol_info` which returns basic version information on the communication protocol:

parameter name	type	description
protocol_name	string	Protocol name (currently always 'pfsdp')
version_major	uint	Protocol major version (e.g. 1 for 'v1.02', 3 for 'v3.10')
version_minor	uint	Protocol minor version (e.g. 2 for 'v1.02', 10 for 'v3.10')
commands	string	List of all available HTTP commands

This document refers to protocol version '1.04' which is implemented by R2000 firmware version '1.60' and newer.

Please note:
The command `get_protocol_info` will return the above information on every sensor – regardless of its firmware version. All other commands and their return values might be changed by updates to the communication protocol, though. Therefore it is strongly recommended to check the protocol version first.

Example

Query: `http://<sensor IP address>/cmd/get_protocol_info`

Reply:

```
{
    "protocol_name":"pfsdp",
    "version_major":1,
    "version_minor":4,
    "commands":[
    "get_protocol_info",
    "list_parameters",
    "get_parameter",
    "set_parameter",
    "reboot_device",
    "factory_reset",
    "reset_parameter",
    "request_handle_udp",
    "request_handle_tcp",
    "feed_watchdog",
    "set_scanoutput_config",
    "get_scanoutput_config",
    "start_scanoutput",
    "stop_scanoutput",
    "release_handle",
    "get_iq_parameter",
    "set_iq_parameter",
    "list_iq_parameters"
    ],
    "error_code":0,
    "error_text":"success"
    }
```

2 Sensor parametrization using HTTP

2.1 Parameter types

The sensor provides access to different types of parameters. The following table gives a quick overview of the relevant types, a more detailed description follows in separate sub-sections:

type	description
`enum`	enumeration type with a set of named values (strings)
`bool`	boolean values (`on` / `off`)
`bitfield`	a set of boolean flags
`int`	signed integer values
`uint`	unsigned integer values
`double`	floating point values with double precision
`string`	strings composed of UTF-8 characters
`ipv4`	Internet Protocol version 4 addresses or network masks
`ntp64`	NTP timestamp values
`binary`	binary data
`array`	collection of values of the same type

Independently of their type, each parameter belongs to one of the following access groups:

access	description
sRO	static Read-Only access (value never changes)
RO	Read-Only access (value might change during operation)
RW	Read-Write access (non-volatile storage)
vRW	volatile Read-Write access (lost on reset)

Most sensor parameters are stored in non-volatile memory. Thus their value also persists a power-cycle of the device.

Please note:
Non-volatile storage has a limited number of write cycles only (typically > 10.000 cycles). Therefore all non-volatile parameters should be written only if necessary.

2.1.1 Enumeration values (`enum`)

Notes on parameters using enumeration values (`enum`):

- An enumeration type parameter accepts a single value out of a list of predefined values.
- Each enumeration values is defined by a string ('named' value).
- Each enumeration value is typically (but not necessarily) unique to the specific parameter.
- Each `enum` parameter can hold only a single value at a time.
- URI: Named enumeration values use non-reserved ASCII characters only and need no percent encoding [7] when specified as argument to a command on the URI.

2.1.2 Boolean values (`bool`)

Notes on parameters using boolean values (`bool`):

- Boolean parameters are a special case of enumeration parameters.
- Only the named values `on` and `off` are accepted.
- Each `bool` parameter can hold only a single value at a time.

2.1.3 Bit fields (`bitfield`)

Notes on parameters using bit fields (`bitfield`):

- Bit fields combine multiple boolean flags into an unsigned integer value.
- Each flag occupies a single bit of the integer.
- Not every bit of the integer needs to be assigned to a flag.
- Bits might be marked as *reserved*. These should always be zero.
- Bit field parameters are read and written using the integer representation.

2.1.4 Integer values (`int`, `uint`)

Notes on parameters using signed integer values (`int`) and unsigned integer value (`uint`):

- Unless denoted differently, the value range of integer values is limited to 32 bit.
- Leading zeros are accepted when writing a value (they will be ignored).
- Neither a hexadecimal nor an octal representation of integer values is supported.

2.1.5 Double values (`double`)

Notes on parameters using double precision floating point values (`double`):

- A dot '.' is used as decimal mark (separating the decimal part from the fractional part of a `double` number).
- The floating point decimal format (xxx.yyy) should be used when accessing `double` parameters. The floating point exponential format (xxx.yyy Ezzz) is not supported.
- The number of significant digits of the fractional part of a `double` value might be limited for some parameters. Excess digits are rounded or discarded.

2.1.6 String values (`string`)

Notes on parameters using string values (`string`):

- Strings represent a set characters.
- All characters of the string need to be encoded in UTF-8 format [6].
- The maximum size of a string is usually limited. Please refer to the description of the specific parameter for its actual size limitation.
- URI: For write access to a `string` parameter, its new value is implicitly delimited by the surrounding '=' and '`&`' within the URI (see RFC 3986 [7]). Any additionally added delimiter (e.g. '"') will be interpreted as part of the string.
- URI: Some characters are reserved within an URI and need to be percent encoded [7] (see section 1.2.2 for details).

When parsing a `string`-typed parameter within an UTF-8 encoded command URI the sensor performs the following steps:

1. Dissect the URI into it individual parts
2. Resolve percent encoded characters
3. Check string for a valid UTF-8 encoding
4. Process the string (UTF-8 bytes), e.g. store it into non-volatile memory

When the sensor outputs a `string`-typed parameter in JSON format, it applies escaping for the following reserved UTF-8 characters (as required by RFC 7159 [9] section 2.5):

character	code	replacement
backspace	U+0008	'\b'
tabulator	U+0009	'\t'
new line	U+000A	'\n'
form feed	U+000C	'\f'
carriage return	U+000D	'\r'
double quote	U+0022	'\"'
solidus	U+002F	currently not replaced
backslash	U+005C	'\\'
other control characters	U+0000 ...U+001F	'\uXXXX'

2.1.7 IPv4 address and network mask values (`IPv4`)

Notes on parameters using IPv4 network addresses and subnet masks (`IPv4`):

- Addresses and network masks need to follow the rules of the Internet Protocol specification (RFC-791 [1])
- Addresses are denoted as `string` values in human-readable *dotted decimal* notation (i.e. `10.0.10.9`)
- Subnet masks are denoted as `string` values in human-readable *dotted decimal* notation (i.e. `255.255.0.0`)

2.1.8 NTP timestamp values (`ntp64`)

Notes on parameters using NTP timestamps (`ntp64`):

- NTP timestamps are part of the Network Time Protocol (NTP) as defined by RFC 1305 [2].
- NTP timestamps are represented as a 64 bit unsigned fixed-point integer number (`uint64`) in seconds in reference to a specific point in time. The most significant 32 bit represent the integer part (seconds), the lower 32 bit the fractional part.
- *Absolute* timestamps (synchronized time) refer to the time elapsed since 1 January 1900.
- *Relative* timestamps (raw system time) refer to the time elapsed since power-on of the sensor.

Please refer to section 3.1.5 for more details on timestamps.

2.1.9 Binary data (`binary`)

Notes on parameters using binary data (`binary`):

- Parameters of type `binary` store binary data without further data-specific knowledge. Parameter values are simply treated as a collection of bytes. The interpretation of binary data is specific to the individual parameter. See the description of the specific parameter for details.
- For `binary` parameters usually only size checking is performed. The maximum size of data depends on the specific parameter.
- Read access to a `binary` parameter returns its value as `base64` encoded string within the JSON reply (see section 1.2.3). The `base64` encoding transforms an 8 bit data stream to a string with a particular set of 64 ASCII characters (6 bit) that are printable and common to most character encodings (see RFC 4648 [8] for details). This encoding requires 33 % more storage space.
- Write access to a `binary` parameter requires the binary data to be encoded as `base64url` [8] string on the URI. The `base64url` encoding is very similar to the `base64` encoding but uses a slightly different character set, that avoids using reserved URI characters.

2.1.10 Collection of values (`array`)

Notes on parameters using a collection of values (`array`):

- Parameters of type `array` store multiple *elements* of the same type. The type of the elements is specific to the individual parameter. See the description of the specific parameter for details.
- For `array` parameters both the number of elements as well as the value of each element is checked. The correct number of elements and allowed element values depend on the specific parameter.
- Read access to a `array` parameter returns its value as a encoded JSON array with the value for each element.
- Write access to a `array` parameter requires the array elements to be provided as a comma-separated list of values.
- Individual access to a specific element of an `array` parameter is not supported.

2.2 Commands for sensor parametrization

This section describes all commands available for manipulation of global sensor parameters.

2.2.1 `list_parameters` – list parameters

The command `list_parameters` returns a list of all available global sensor parameters.

Example

Query: `http://<sensor IP address>/cmd/list_parameters`

Reply:

```
{
    "parameters":[
    "vendor",
    "product",
    "part",
    "serial",
    "revision_fw",
    "revision_hw",
    "max_connections",
    "feature_flags",
    "radial_range_min",
    "radial_range_max",
    "radial_resolution",
    "angular_fov",
    "angular_resolution",
    "ip_mode",
    "ip_address",
    "subnet_mask",
    "gateway",
    "scan_frequency",
    "scan_direction",
    "samples_per_scan",
    "scan_frequency_measured",
    "status_flags",
    "load_indication",
    "device_family",
    "mac_address",
    "hmi_display_mode",
    "hmi_language",
    "hmi_button_lock",
    "hmi_parameter_lock",
    "ip_mode_current",
    "ip_address_current",
    "subnet_mask_current",
    "gateway_current",
    "system_time_raw",
    "user_tag",
    "user_notes",
    "locator_indication",
    ],
    "error_code":0,
    "error_text":"success"
    }
```

2.2.2 `get_parameter` – read a parameter

The command `get_parameter` reads the current value of one or more global sensor parameters:

```
http://<sensor IP address>/cmd/get_parameter?list=<param1>;<param2>
```

Command arguments

- `list` – semicolon separated list of parameter names (optional)

If the argument `list` is not specified the command will return the current value of all available parameters.

Example

Query: `http://<sensor IP address>/cmd/get_parameter?list=scan_frequency;scan_frequency_measured`

Reply:
```
{
    "scan_frequency":50,
    "scan_frequency_measured":49.900000,
    "error_code":0,
    "error_text":"success"
    }
```

2.2.3 `set_parameter` – change a parameter

Using the command `set_parameter` the value of any write-accessible global sensor parameter can be changed:

```
http://<sensor IP address>/cmd/set_parameter?<param1>=<value>&<param2>=<value>
```

Command arguments

- `<param1> = <value>` – new <value> for parameter <param1>
- `<param2> = <value>` – new <value> for parameter <param2>
- `...`

Please note:
The command `set_parameter` returns an error message, if any parameter specified as command argument is unknown or a read-only parameter. The return values `error_code` and `error_text` have appropriate values in this case (see section 1.2.6).

Example

Query: `http://<sensor IP address>/cmd/set_parameter?scan_frequency=50`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

2.2.4 `reset_parameter` – reset a parameter to its default value

The command `reset_parameter` resets one or more global sensor parameters to their factory default values:

```
http://<sensor IP address>/cmd/reset_parameter?list=<param1>;<param2>
```

Command arguments

- `list` – semicolon separated list of parameter names (optional)

Please note:
If the argument `list` is not specified the command will load the factory default value for all parameters writeable with `set_parameter`!

Please note:
This command applies to global R/W parameters accessible via the command `set_parameter` only. If the argument `list` contains an unknown or a read only parameter, an error message will be returned.

Please note:
Resetting a parameter to its default value might require a device restart in order to take effect. For example, this applies to all Ethernet configuration parameters (see section 2.5).

Example

Query: `http://<sensor IP address>/cmd/reset_parameter?list=scan_frequency;scan_direction`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

2.2.5 `reboot_device` – restart the sensor firmware

The command `reboot_device` triggers a soft reboot of the sensor firmware:

```
http://<sensor IP address>/cmd/reboot_device
```

Command arguments

The command accepts no additional arguments. The reboot is performed shortly after the HTTP reply has been sent.

Please note:
A reboot terminates all running scan data output. All scan data handles are invalidated and have to be renewed from scratch after reboot (see section 3.4).

Please note:
A device reboot takes up to 60 s (depending on the sensor configuration). The reboot is completed as soon as the sensor answers to HTTP command requests again and the system status flag *Initialization* (see section 2.8.2) is cleared.

Example

Query: `http://<sensor IP address>/cmd/reboot_device`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

2.2.6 `factory_reset` – reset the sensor to factory settings

The command `factory_reset` performs a complete reset of all sensor settings to factory defaults and reboots the device. Its result is similar to a call of `reset_parameter` without any arguments followed by a call to `reboot_device`.

Command arguments

The command accepts no additional arguments. The factory reset and device reboot is performed shortly after the HTTP reply has been sent.

Please note:
The factory reset performs a device reboot, because some changes take effect at sensor boot time only (e.g. all changes to Ethernet configuration parameters – see section 2.5).

PFSDP compatibility note:
The command `factory_reset` is available on devices with PFSDP version 1.01 or newer.

Example

Query: `http://<sensor IP address>/cmd/factory_reset`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

2.3 Basic sensor information

This section describes all sensor parameters which are available to the user.

2.3.1 Parameter overview

The following table lists numerous parameters (mostly read-only) which provide basic sensor information:

parameter name	type	description	access
`device_family`	`uint`	Numeric unique identifier (see below)	sRO
`vendor`	`string`	Vendor name (max. 100 chars)	sRO
`product`	`string`	Product name (max. 100 chars)	sRO
`part`	`string`	Part number (max. 32 chars)	sRO
`serial`	`string`	Serial number (max. 32 chars)	sRO
`revision_fw`	`string`	Firmware revision (max. 32 chars)	sRO
`revision_hw`	`string`	Hardware revision (max. 32 chars)	sRO
`user_tag`	`string`	User defined name (max. 32 chars)	RW
`user_notes`	`string`	User notes (max. 1000 bytes)	RW

These entries are comparable to generic information available on IO-Link devices. In contrast to IO-Link most strings have no size limitation, though. Furthermore each parameter can be read individually using the command `get_parameter`.

2.3.2 Device family (`device_family`)

The parameter `device_family` can be used to identify compatible device families. A single device family is defined as group of devices with identical functionality (regarding the Ethernet protocol). This identifier can be used to check if the connected device is compatible with the client application (e.g. DTM user interface).

Currently the following values are defined for `device_family`:

value	name	description
0	reserved	never used
1	OMDxxx-R2000	R2000 OMD UHD raw data devices with ultra-high resolution
2	OBDxxx-R2000	R2000 OBD detection devices with standard features
3	OMDxxx-R2000-HD	R2000 OMD HD raw data devices with high resolution
4	reserved	
5	OMDxxx-R2300	R2300 OMD multi-line scanner
6	OMDxxx-R2000-SD	R2000 OMD SD raw data devices with standard resolution
7	OMDxxx-R2300	R2300 OMD single-line scanner

2.3.3 User defined strings (`user_tag`, `user_notes`)

The parameters `user_tag` and `user_notes` are strings, that can be used by the user without restriction (except for a valid UTF-8 encoding – see definition of type `string` in section 2.1). The default value for `user_tag` is typically a short version of the product name (parameter `product`) while `user_notes` is empty per default.

2.4 Sensor capabilities

2.4.1 Parameter overview

The following static read-only parameters describe the sensor capabilities:

parameter name	type	unit	description	access
`feature_flags`	`array`		sensor feature flags (see below)	sRO
`emitter_type`	`uint`		type of light emitter used by the sensor (see below)	sRO
`radial_range_min`	`double`	m	min. measuring range (distance)	sRO
`radial_range_max`	`double`	m	max. measuring range (distance)	sRO
`radial_resolution`	`double`	m	*mathematical* resolution of distance values in scan data output	sRO
`angular_fov`	`double`	°	max. angular field of view	sRO
`angular_resolution`	`double`	°	*mathematical* resolution of angle values in scan data output	sRO
`scan_frequency_min`	`double`	Hz	min. supported scan rate (see section 2.6)	sRO
`scan_frequency_max`	`double`	Hz	max. supported scan rate (see section 2.6)	sRO
`sampling_rate_min`	`uint`	Hz	min. supported sampling rate (see section 2.6)	sRO
`sampling_rate_max`	`uint`	Hz	max. supported sampling rate (see section 2.6)	sRO
`max_connections`	`uint`		max. number of concurrent scan data channels (connections)	sRO

2.4.2 Device features (`feature_flags`)

The parameter `feature_flags` returns a JSON [9] encoded list of features available for the queried device. Currently the following features are defined:

feature name	description	reference	PFSDP version
`ethernet`	Ethernet interface		v1.00 or newer
`input_output_q1`	Digital switching input/output I/Q1	Chapter 7	v1.01 or newer
`input_output_q2`	Digital switching input/output I/Q2	Chapter 7	v1.01 or newer
`input_output_q3`	Digital switching input/output I/Q3	Chapter 7	v1.01 or newer
`input_output_q4`	Digital switching input/output I/Q4	Chapter 7	v1.01 or newer
`lens_contamination_monitor`	Sensor lens cover contamination monitor	Chapter 5	v1.03 or newer
`scan_data_filter`	Filter based processing of scan data	Chapter 4	v1.03 or newer

If a feature is available, its name is listed within the `feature_flags` array.

2.4.3 Emitter type (`emitter_type`)

The parameter `emitter_type` can be used to determine the type of light emitter (aka `transmitter`) used by the specific sensor. Currently the following emitter types are defined for R2000 devices:

type	description
0	undefined / reserved
1	red laser (660 nm)
2	infrared laser (905 nm)

2.5 Ethernet configuration

2.5.1 Parameter overview

The following parameters allow configuration changes of the Ethernet interface:

parameter	type	description	access	default
ip_mode	enum	IP address mode: static, dhcp, autoip	RW	autoip
ip_address	ipv4	static IP mode: sensor IP address	RW	10.0.10.9
subnet_mask	ipv4	static IP mode: subnet mask	RW	255.0.0.0
gateway	ipv4	static IP mode: gateway address	RW	0.0.0.0
ip_mode_current	enum	current IP address mode: static, dhcp, autoip	RO	autoip
ip_address_current	ipv4	current sensor IP address	RO	169.254.x.y
subnet_mask_current	ipv4	current subnet mask	RO	255.255.0.0
gateway_current	ipv4	current gateway address	RO	0.0.0.0
mac_address	string	sensor MAC address ("000D81xxxxxx")	sRO	–

The read-only parameters ip_mode_current, ip_address_current, subnet_mask_current and gateway_current provide access to the currently active IP configuration. This is especially useful when using automatic IP configuration via DHCP or AutoIP.

Please note:
Any changes to the Ethernet configuration (using set_parameter or reset_parameter) are applied after a system reboot only! The command reboot_device (see section 2.2.5) is available to initiate a reboot using the Ethernet protocol.

2.5.2 IP address mode (ip_mode)

The parameter ip_mode configures one of the following IP address modes:

IP mode	description
static	static IP configuration using ip_address, subnet_mask, gateway
autoip	automatic IP configuration using "Zero Configuration Networking" [15]
dhcp	automatic IP configuration using a DHCP server

Please note:
With automatic IP configuration using DHCP or AutoIP the parameters ip_address_current and subnet_mask_current might return the invalid IP address 0.0.0.0, if no valid IP address has been assigned to the sensor yet (e.g. if no DHCP server is found).

2.6 Measuring configuration

2.6.1 Parameter overview

The following (global) parameters are available for basic measurement configuration:

parameter name	type	unit	description	access	default
`operating_mode`	`enum`	–	mode of operation: `measure`, `emitter_off`	vRW	`measure`
`scan_frequency`	`double`	1 Hz	scan rate (10 Hz to 50 Hz)	RW	35 Hz
`scan_direction`	`enum`	–	direction of rotation: `cw` or `ccw`	RW	`ccw`
`samples_per_scan`	`uint`	samples	number of readings per scan	RW	`3600`
`scan_frequency_measured`	`double`	1 Hz	measured scan rate (current value)	RO	–

2.6.2 Mode of operation (`operating_mode`)

The parameter `operating_mode` controls the mode of operation of the sensor. Currently, the following modes are available:

operating mode	description
`measure`	Sensor is recording scan data
`emitter_off`	Emitter is disabled, no scan data is recorded
`transmitter_off`	*deprecated*, please use `emitter_off` instead

The mode `measure` is the normal mode of operation of the sensor and default after power-on. The mode `emitter_off` allows the user to deactivate the light emitter, e.g. to avoid interference with other optical devices. A mode switch from `measure` to `emitter_off` can only be performed, if no scan data connections are active, i.e. all handles have been released. While the operating mode is set to `emitter_off`, no new scan data connection handles can be requested (see section 3.2). This state is also signaled by the system status flag `scan_output_muted` (see section 2.8.2).

Please note:
The parameter `operating_mode` is a non-persistent parameter, i.e. it reverts to its default value after reboot, power cycle or factory reset.

PFSDP compatibility note:
The parameter value `transmitter_off` has been renamed to `emitter_off` with PFSDP version 1.03. The old name `transmitter_off` is still supported for write accesses to the parameter `operating_mode`, but read accesses return the new value. It is recommended to use the new name.

Example

Query: `http://<sensor IP address>/cmd/set_parameter?operating_mode=measure`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

2.6.3 Scan rate (`scan_frequency`, `scan_frequency_measured`)

The parameter `scan_frequency` defines the number of scans recorded per second (see section 3.1 for details). This is also called *scan rate*. For R2000 devices this value directly corresponds to the rotational speed of the sensor head. Valid values range from 10 Hz to 50 Hz with steps of 1 Hz (default is 35 Hz). Non-integer values are automatically rounded to integer values.

The parameter `scan_frequency_measured` reads back the actual scan rate resulting from the current rotational speed of the sensor head with a resolution of 0.1 Hz. It is a read-only parameter.

Example

Query: `http://<sensor IP address>/cmd/get_parameter?list=scan_frequency;scan_frequency_measured`

Reply:
```
{
    "scan_frequency":35,
    "scan_frequency_measured":34.900000,
    "error_code":0,
    "error_text":"success"
    }
```

2.6.4 Scan direction (`scan_direction`)

The parameter `scan_direction` defines the direction of rotation of the sensors head. User applications can choose between clockwise rotation (`cw`) or counter-clockwise rotation (`ccw`). Please refer to sections 3.1.1 and 3.1.2 on how these settings are related to the sensor coordinate system and the scan data output.

Example

Query: `http://<sensor IP address>/cmd/set_parameter?scan_direction=ccw`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

2.6.5 Scan resolution (`samples_per_scan`)

The parameter `samples_per_scan` defines the number of samples recorded within a scan (for details please refer to section 3.1). This value implicitly also defines the *scan resolution*, i.e. the angular step between two subsequent measurements.

R2000 devices support a number of discrete values for this parameter. Table 2.1 lists all available setting for R2000 UHD, HD and SD devices. Requesting any other number of samples per scan results into an error message.

Please note:
The number of `samples_per_scan` multiplied by the `scan_frequency` gives the *sampling rate*, i.e. the number of measurements per second (this is also called *pulse repetition rate* in laser-safety terminology).
The sensor supports sampling rates between `sampling_rate_min` and `sampling_rate_max` (see section 2.4). Thus the number of `samples_per_scan` indirectly also limits the maximum value for the parameter `scan_frequency` (and vice versa). Therefore table 2.1 denotes the maximum scan rate as well.

PFSDP compatibility note:
The values 1680, 2100 and 2800 for `samples_per_scan` are supported by devices with PFSDP version 1.02 or newer.

Example

Query: `http://<sensor IP address>/cmd/set_parameter?samples_per_scan=3600`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

samples per scan	scan resolution	scan rate (max)	sampling rate (max)
25 200	0.014°	10 Hz	252 kHz
16 800	0.021°	15 Hz	252 kHz
12 600	0.029°	20 Hz	252 kHz
10 080	0.036°	25 Hz	252 kHz
8400	0.043°	30 Hz	252 kHz
7200	0.050°	35 Hz	252 kHz
6300	0.057°	40 Hz	252 kHz
5600	0.064°	45 Hz	252 kHz
5040	0.071°	50 Hz	252 kHz
4200	0.086°	50 Hz	210 kHz
3600	0.100°	50 Hz	180 kHz
3150	0.114°	50 Hz	158 kHz
2800	0.129°	50 Hz	140 kHz
2520	0.143°	50 Hz	126 kHz
2400	0.150°	50 Hz	120 kHz
2100	0.171°	50 Hz	105 kHz
1800	0.200°	50 Hz	90 kHz
1680	0.214°	50 Hz	84 kHz
1440	0.250°	50 Hz	72 kHz
1200	0.300°	50 Hz	60 kHz
900	0.400°	50 Hz	45 kHz
800	0.450°	50 Hz	40 kHz
720	0.500°	50 Hz	36 kHz
600	0.600°	50 Hz	30 kHz
480	0.750°	50 Hz	24 kHz
450	0.800°	50 Hz	23 kHz
400	0.900°	50 Hz	20 kHz
360	1.000°	50 Hz	18 kHz
240	1.500°	50 Hz	12 kHz
180	2.000°	50 Hz	9 kHz
144	2.500°	50 Hz	7 kHz
120	3.000°	50 Hz	6 kHz
90	4.000°	50 Hz	5 kHz
72	5.000°	50 Hz	4 kHz

(a) R2000 UHD devices

samples per scan	scan resolution	scan rate (max)	sampling rate (max)
8400	0.043°	10 Hz	84 kHz
7200	0.050°	11 Hz	80 kHz
6300	0.057°	13 Hz	82 kHz
5600	0.064°	15 Hz	84 kHz
5040	0.071°	16 Hz	81 kHz
4200	0.086°	20 Hz	84 kHz
3600	0.100°	23 Hz	83 kHz
3150	0.114°	26 Hz	82 kHz
2800	0.129°	30 Hz	84 kHz
2520	0.143°	33 Hz	84 kHz
2400	0.150°	35 Hz	84 kHz
2100	0.171°	40 Hz	84 kHz
1800	0.200°	46 Hz	83 kHz
1680	0.214°	50 Hz	84 kHz
1440	0.250°	50 Hz	72 kHz
1200	0.300°	50 Hz	60 kHz
900	0.400°	50 Hz	45 kHz
800	0.450°	50 Hz	40 kHz
720	0.500°	50 Hz	36 kHz
600	0.600°	50 Hz	30 kHz
480	0.750°	50 Hz	24 kHz
450	0.800°	50 Hz	23 kHz
400	0.900°	50 Hz	20 kHz
360	1.000°	50 Hz	18 kHz
240	1.500°	50 Hz	12 kHz
180	2.000°	50 Hz	9 kHz
144	2.500°	50 Hz	8 kHz
120	3.000°	50 Hz	6 kHz
90	4.000°	50 Hz	5 kHz
72	5.000°	50 Hz	4 kHz

(b) R2000 HD devices

samples per scan	scan resolution	scan rate (max)	sampling rate (max)
7200	0.050°	10 Hz	72 kHz
6300	0.057°	11 Hz	70 kHz
5600	0.064°	12 Hz	68 kHz
5040	0.071°	14 Hz	71 kHz
4200	0.086°	17 Hz	72 kHz
3600	0.100°	20 Hz	72 kHz
3150	0.114°	22 Hz	70 kHz
2800	0.129°	25 Hz	70 kHz
2520	0.143°	28 Hz	71 kHz
2400	0.150°	30 Hz	72 kHz
2100	0.171°	30 Hz	63 kHz
1800	0.200°	30 Hz	54 kHz
1680	0.214°	30 Hz	50 kHz
1440	0.250°	30 Hz	43 kHz
1200	0.300°	30 Hz	36 kHz
900	0.400°	30 Hz	27 kHz
800	0.450°	30 Hz	24 kHz
720	0.500°	30 Hz	22 kHz
600	0.600°	30 Hz	18 kHz
480	0.750°	30 Hz	14 kHz
450	0.800°	30 Hz	14 kHz
400	0.900°	30 Hz	12 kHz
360	1.000°	30 Hz	11 kHz
240	1.500°	30 Hz	7 kHz
180	2.000°	30 Hz	5 kHz
144	2.500°	30 Hz	4 kHz
120	3.000°	30 Hz	4 kHz
90	4.000°	30 Hz	3 kHz
72	5.000°	30 Hz	2 kHz

(c) R2000 SD devices

Table 2.1: List of valid values for parameter `samples_per_scan`

2.7 HMI / Display configuration

2.7.1 Parameter overview

This section lists all (global) parameters that are available to configure the sensors human machine interface (HMI) consisting of the heads LED display and two push-buttons.

parameter name	type	description	access	default
`hmi_display_mode`	`enum`	normal operation display mode	RW	`static_logo`
`hmi_language`	`enum`	display language: `english`, `german`	RW	`english`
`hmi_button_lock`	`bool`	lock HMI buttons: `on` / `off`	RW	`off`
`hmi_parameter_lock`	`bool`	set HMI to read-only: `on` / `off`	RW	`off`
`locator_indication`	`bool`	LED locator indication: `on` / `off`	vRW	`off`
display mode parameters (see chapter 6)				
`hmi_static_logo`	`binary`	bitmap image for mode `static_logo`	RW	P+F logo
`hmi_static_text_1`	`string`	text line 1 for mode `static_text` (max. 30 chars)	RW	`"Pepperl+Fuchs"`
`hmi_static_text_2`	`string`	text line 2 for mode `static_text` (max. 30 chars)	RW	`"R2000"`
`hmi_application_bitmap`	`binary`	bitmap image for mode `application_bitmap`	vRW	<empty>
`hmi_application_text_1`	`string`	text line 1 for `application_text` (max. 30 chars)	vRW	<empty>
`hmi_application_text_2`	`string`	text line 2 for `application_text` (max. 30 chars)	vRW	<empty>

The display mode parameters allow client application to directly access the HMI LED display. Please refer to chapter 6 for a detailed description on using these functionalities.

2.7.2 HMI display mode (hmi_display_mode)

The parameter hmi_display_mode controls the content of the HMI LED display during normal operation, i.e. while neither warnings nor errors are displayed and the user did not activate the HMI menu. Depending on the device family, the following display modes are available:

Display mode	Device family	Description
off	all	Display is blank.
static_logo	all	Show a static logo.
static_text	all	Show two lines of static text.
bargraph_distance	OMDxxx-R2000	Show visualization of measured distances.
bargraph_echo	OMDxxx-R2000	Show visualization of measured echo values.
bargraph_reflector	OMDxxx-R2000	Show visualization of high echo targets.
application_bitmap	OMDxxx-R2000	Show an application-provided bitmap.
application_text	OMDxxx-R2000	Show two lines of application-provided text.

The setting of hmi_display_mode is stored into non-volatile memory, i.e. it is preserved during a power cycle.

2.7.3 HMI display language (hmi_language)

The parameter hmi_language controls the language of text messages (menu, warnings, errors) shown by the HMI LED display. Currently the setting english and german are available. The current setting is stored into non-volatile memory, i.e. it is preserved during a power cycle.

2.7.4 HMI button lock (hmi_button_lock)

The boolean parameter hmi_button_lock allows to disable the HMI buttons on the sensors front. If set to on any push of a button is ignored. This enables client applications to deny users access to the HMI menu of the sensor.

Please note:
Locking the buttons also prevents access to read-only information like the current Ethernet configuration. If this is not intended consider using the parameter hmi_parameter_lock instead.

2.7.5 HMI parameter lock (hmi_parameter_lock)

Protocol version 1.01 adds the boolean parameter hmi_parameter_lock which allows to disable parameter changes via the HMI display menu of the sensor. This enables client applications to prevent users from changing sensor parameters while retaining the possibility to determine current settings for parameters available from the HMI menu (e.g. current Ethernet configuration).

2.7.6 Locator indication (locator_indication)

The parameter locator_indication temporarily activates a distinctive flashing pattern for the Power and Q2 LEDs. This function can be used to identify a specific R2000 device if multiple devices are installed.

Please note:
The locator indication function is non-persistent, i.e. it is automatically disabled after reboot, power cycle or factory reset.

2.8 System status

2.8.1 Parameter overview

The following (read only) parameters can be accessed to get status information from the sensor.

parameter name	type	unit	description	access
status information				
`status_flags`	`bitfield`	–	sensor status flags (see section 2.8.2)	RO
`load_indication`	`uint`	%	current system load (0 % to 100 %)	RO
time information				
`system_time_raw`	`ntp64`	–	raw system time (see section 3.1.5)	RO
`up_time`	`uint`	min	time since power-on	RO
`power_cycles`	`uint`	–	number of power cycles	RO
`operation_time`	`uint`	min	overall operating time	RO
`operation_time_scaled`	`uint`	min	overall operating time scaled by temperature	RO
operating conditions				
`temperature_current`	`int`	°C	current operating temperature	RO
`temperature_min`	`int`	°C	minimum lifetime operating temperature (power-up update delay 15min)	RO
`temperature_max`	`int`	°C	maximum lifetime operating temperature (power-up update delay 15min)	RO

Example

Query: `http://<sensor IP address>/cmd/get_parameter?list=up_time;power_cycles`

Reply:
```
{
    "up_time":44,
    "power_cycles":22,
    "error_code":0,
    "error_text":"success"
    }
```

2.8.2 System status flags (`status_flags`)

The read-only parameter `status_flags` (see section 2.8) provides an array of system status flags:

bit	flag name	description
Generic		
0	`initialization`	System is initializing, valid scan data not available yet
2	`scan_output_muted`	Scan data output is muted by current system configuration (see section 2.6.2)
3	`unstable_rotation`	Measured scan rate does not match set value
Warnings		
8	`device_warning`	Accumulative flag – set if device displays any warning
9	`lens_contamination_warning`	LCM warning threshold triggered for at least one sector (see chapter 5)
10	`low_temperature_warning`	Current internal temperature below warning threshold
11	`high_temperature_warning`	Current internal temperature above warning threshold
12	`device_overload`	Overload warning – sensor CPU overload is imminent
Errors		
16	`device_error`	Accumulative flag – set if device displays any error
17	`lens_contamination_error`	LCM error threshold triggered for at least one sector (see chapter 5)
18	`low_temperature_error`	Current internal temperature below error threshold
19	`high_temperature_error`	Current internal temperature above error threshold
20	`device_overload`	Overload error – sensor CPU is in overload state
Defects		
30	`device_defect`	Accumulative flag – set if device detected an unrecoverable defect

System status flags are similar to scan data header status flags (see section 3.4.3) but provide up-to-date information on the current device status (not associated to specific scan data).

Please note:
All flags not listed in the above table are reserved and should be ignored.

2.8.3 System load indication (`load_indication`)

The status variable `load_indication` gives a rough indication of the current CPU load of the sensor:

- **0 % – System is idle**
  The system is idle, if the HMI Display is disabled (`hmi_display_mode == off`) and there is no active scan data output running (neither TCP nor UDP).
- **100 % – System is overloaded**
  The system may go into overload if too many clients are requesting scan data via active TCP/UDP connections. In this case nominal operation of the R2000 cannot be guaranteed! Please reduce system load by disabling HMI display, reducing number of TCP/UDP connections or reducing scan resolution.

3 Scan data output using TCP or UDP

3.1 Principles of scan data acquisition

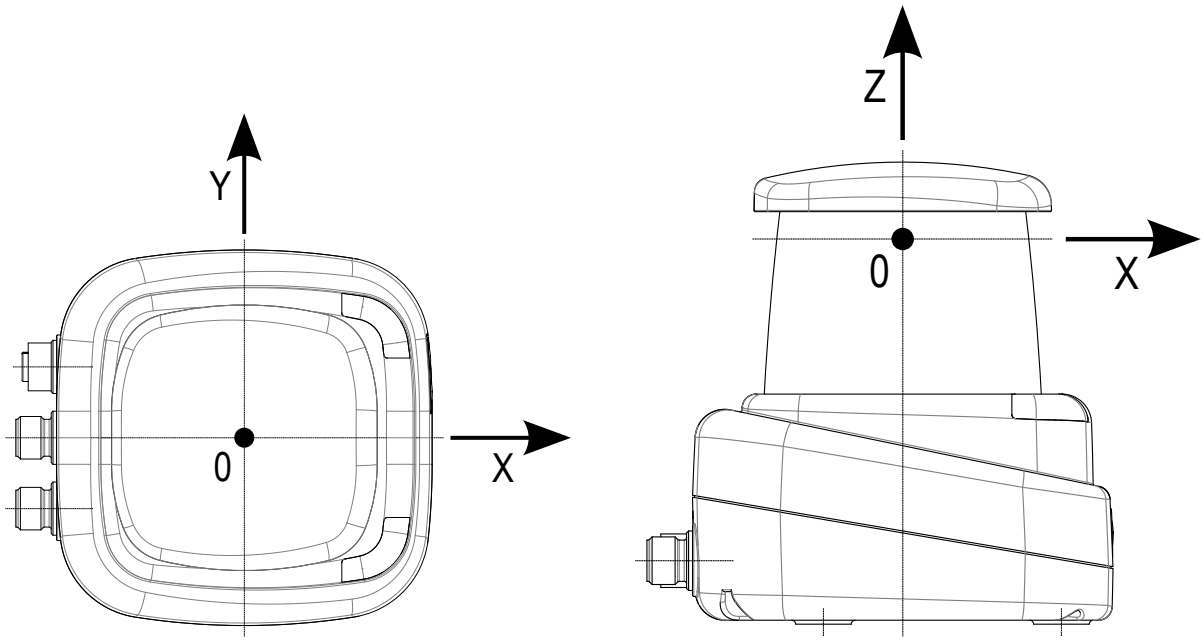
The R2000 is a *laser scanner* designed to periodically measure distances within a full 360° field of view while rotating with a constant rate as defined by the parameter `scan_frequency` (see section 2.6). The measurements are aggregated into *scans*. A single scan corresponds to one revolution of the sensor head, and yields a sequence of *scan points* (also called *samples*). The number of scan points within a scan is defined by the parameter `samples_per_scan` (see section 2.6).

Each scan point is comprised of a distance value for a corresponding angle as well as an echo amplitude. However, since measurements are performed with a uniform angular resolution (depending on the parameter `samples_per_scan`), the actual scan data output typically just gives distance and amplitude data for each sample. The corresponding angular reading can be reconstructed by adding up the angular increments from the starting angle of the scan. The output format of scan data depends on the scan data packet type used – please refer to section 3.4 for further details.

The following subsections describe various basic concepts of the scan data representation used by the R2000.

3.1.1 Sensor coordinate system

The sensor coordinate system is defined as right-handed Cartesian coordinate system. Figure 3.1 shows this coordinate system for the top view and one side view of the sensor: The *origin* is located at the point of intersection of the axis of rotation and the axis of the laser beam. The *X-axis* points to the sensor front (with status LEDs). The *Y-axis* is located perpendicular to the X-axis and parallel to the base-plate of the sensor (pointing upwards in fig. 3.1a). The *Z-axis* is collinear to the axis of rotation (pointing upwards in fig. 3.1b).



(a) R2000 top view (b) R2000 side view

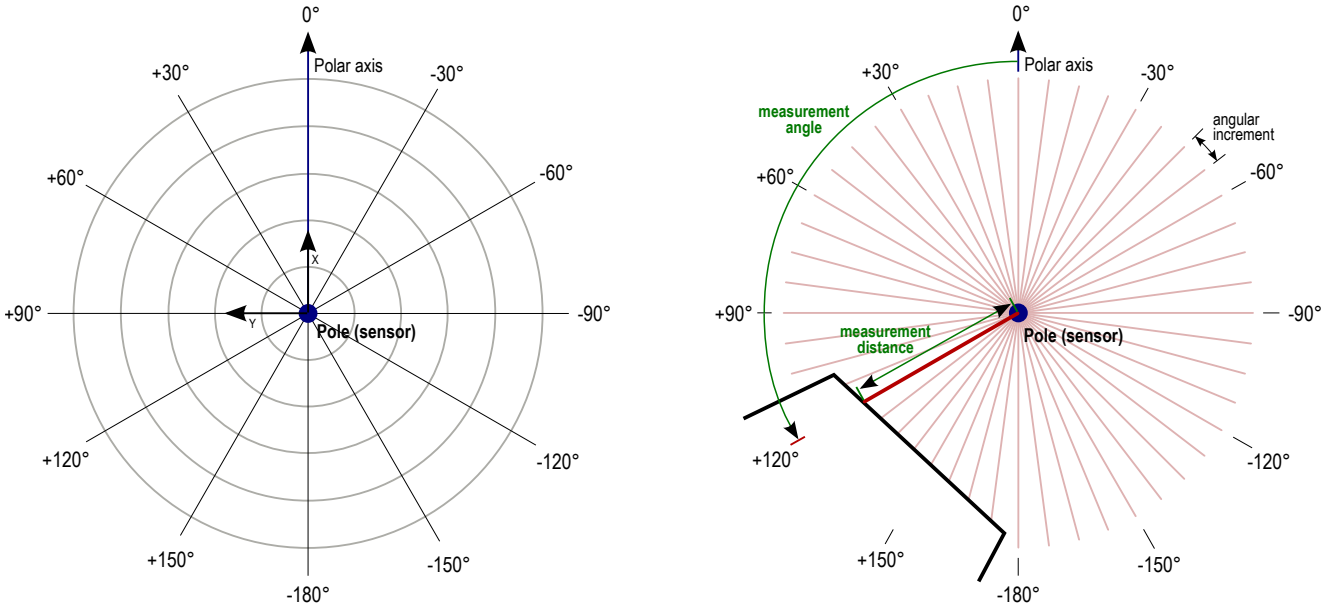
Figure 3.1: Sensor coordinate system

3.1.2 Scan data coordinate system

The plane formed by the X-axis and the Y-axis of the sensor coordinate system is called *scan plane*. All measurements of the laser scanner are recorded within this plane. Scan data acquisition is performed sequentially in the direction of head rotation around the origin of the scan plane. Therefore scan data is typically represented within a polar coordinate system

(see fig. 3.2a). The pole of the coordinate system is defined by the axis of rotation (Z-axis of the sensor coordinate system). The reference for angle measurements (polar axis) is equivalent to the X-axis of the sensor coordinate system (pointing upwards in fig. 3.2a).

During nominal operation scan points are continuously recorded using a uniform *angular increment* and direction of rotation. Both, angular increment and direction of rotation, can be configured by global device parameters (see section 2.6). By default the laser scanner rotates in mathematically positive direction. This direction is called *counter-clockwise* (abbreviated *ccw*) – the angular increment between two subsequent scan points has a positive value. The opposite direction is accordingly called *clockwise* (abbreviated *cw*) – the angular increment between two subsequent scan points has a negative value.



(a) Polar coordinates of laser scan

(b) Single scan point coordinates

Figure 3.2: Scan data coordinate system

Figure 3.2b shows a (simplified) example of a laser scan with a small number of samples. The *measurement angle* of a single scan point (angular coordinate) is calculated within the scan plane with reference to the polar axis. The measurement distance (radial coordinate) is determined by the distance from the center of rotation (pole) to the object hit by the laser beam. Angular coordinates within the 360° field of view are specified with a value range of $[-180°; +180°[$ including −180° but excluding +180°.

3.1.3 Distance readings

Distance readings are typically output as integer value as defined by the scan data packet type (see section 3.4). In case of invalid measurements (e.g. no echo detected or distance out of range) the distance reading is set to an error substitution value: the biggest representable integer value for a distance value (e.g. 0xFFFFFFFF for an `uint32` typed distance value).

Please note:
The measurement resolution and measurement range are limited by the physical capabilities of the sensor as listed in the sensor data-sheet. This information is also available by means of the read-only variables `radial_resolution`, `radial_range_min` and `radial_range_max` (see section 2.4).

3.1.4 Echo amplitude readings

For each measurement of the sensor optional amplitude data is available to the client. R2000 amplitude data is output as dimensionless linear value with a fixed resolution of 12 bit.

On principle, amplitude data can deliver an estimate of the *relative* reflectivity of an object only. Measured amplitude depends on the surface properties of the target object (its *absolute* reflectivity), its distance to the sensor, the angle of incidence of the sensors laser beam on the target surface, etc. – therefore a direct comparison of amplitude data is only viable for object surfaces under similar observation conditions.

Please note:
Please note that amplitude data is not calibrated. Thus amplitude data of different sensor devices may not be identical even under similar observation conditions!

The least significant values of the 12 bit amplitude data are reserved for the following special values:

value	name	description
0	no echo	receiver detected no echo
1	blinding	receiver overloaded due to excessive echo amplitude
2	error	unable to measure echo amplitude
3	reserved	internal (should not occur during normal operation)
4	reserved	internal (should not occur during normal operation)
5	reserved	internal (should not occur during normal operation)
6	weak echo	detected echo too weak for a valid measurement
7-31	reserved	reserved for internal use
>31	amplitude	measured echo amplitude value

All values in the range of 7 to 31 are reserved for internal use. The smallest amplitude value for a *valid* measurement is 32.

3.1.5 Timestamps

The R2000 devices record *raw* timestamps while scan data is being captured. They are generated by an internal system clock that starts counting from zero at power-on. Its resolution is better than 1 ms and its drift is below 100 ppm. The timestamps are stored in 64bit NTP timestamp format (see section 2.1.8 for details). Raw time is always incrementing without any discontinuities or overflows.

For synchronisation with another clock source, the client application can access the raw *system* time via the device parameter `system_time_raw` (see section 2.8). When `system_time_raw` is read using `get_parameter` the device will return the raw system time for the point in time, when the command has been received. Please note that both sending the request for a timestamp and receiving the reply with the timestamp are affected by the non-deterministic HTTP transmission delay.

A typical approach for synchronising the sensor system time with a client clock would be:

1. Send a `get_parameter` command for reading `system_time_raw` to the device.
   Record the client time $t^C_{request}$ for the point in time when the command has been sent.
2. Wait for the reply for `get_parameter` which provides the sensor system time t^S_{sync}.
   Record the client time t^C_{reply} for the point in time when the reply has been received.
3. Assuming a symmetric delay for the HTTP transmission of request and reply, calculate the client time t^C_{sync} that corresponds to t^S_{sync}:

$$t^C_{sync} = t^C_{request} + \frac{(t^C_{reply} - t^C_{request})}{2}$$

Alternatively an electrical output signal can be used for time synchronization (see section 7.3.3).

3.2 Principles of scan data output

3.2.1 Introduction

In order to receive scan data from the laser scanner the client application needs to establish a *scan data connection* to the sensor. Basically the laser scanner supports two different types of data channels: TCP and UDP. TCP channels provide a reliable and error proof channel for transmission of scan data at the cost of potentially unpredictable latency. In contrast, UDP channels allow data transmission with minimum latency at the expense of potential unrecoverable data corruption or data loss. Both TCP and UDP data channels are managed using the HTTP command interface.

For typical applications the following steps are necessary to use scan data output:

1. Set up global configuration of the scanner (see chapter 2), if necessary
2. Establish a data channel to the sensor (see sections 3.3.1 and 3.3.2)
3. Configure scan data output (see section 3.3.6), if necessary
4. Start scan data transmission (see section 3.3.4)
5. Receive scan data from the device (see section 3.4)
6. Stop scan data transmission (see section 3.3.5)
7. Terminate the data channel to the sensor (see section 3.3.3)

Section 3.3 covers the required commands for managing scan data output in detail.

3.2.2 Scan data connection handles

The PFSDP protocol supports parallel scan data connections to multiple clients. In order to configure and control these connections individually, each connection is identified by a unique connection *handle*. A handle is defined as random alpha-numeric string of maximal 16 characters. The sensor ensures that each handle is used for only one active scan data connection. Applications should not make any further assumption regarding the structure of a handle as implementation details might change with new firmware versions (see also below).

Compatibility to handle implementation of R2000 firmware v1.0x

Connection handles have been specified as *random* alpha-numeric string since the first version of the Ethernet communication protocol. Unfortunately, R2000 firmware versions prior to v1.20 implemented a rather systematic handle generation algorithm, that caused subsequent handles to receive linear increasing signatures. With firmware v1.20 this implementation has been updated to a more random handle generation pattern.

3.2.3 Scan data connection watchdog

By default each scan data connection (identified by a handle) features a watchdog timer. If a data channel is not used within a defined time period the associated scan data output will be stopped, the data channel will be closed and the data channel handle will be invalidated by the sensor. This way the device can free up precious resources for new scan data connections that would be otherwise permanently blocked by "zombie" connections.

In order to prevent a watchdog timeout, the client needs to feed the watchdog on regular basis. This can be done with the command `feed_watchdog` (see section 3.3.8) or using "in-line" watchdog feeds for TCP scan data connections (see section 3.3.9). Each call resets the watchdog timer.

The watchdog timeout period can be configured by the client application individually for each scan data connection using the parameters `watchdog` and `watchdogtimeout` of the commands `request_handle_udp` (see section 3.3.1) , `request_handle_tcp` (see section 3.3.2) and `set_scanoutput_config` (see section 3.3.6). The parameter `watchdogtimeout` specifies the timeout period within the range of 1 s up to 500 s. The parameter `watchdog` enables (value `on`) or disables (value `off`) the watchdog. Per default the watchdog is enabled with a timeout period of 60 s.

Please note:
Although the watchdog timeout period (`watchdogtimeout`) can be specified with a resolution of 1 ms, the effective internal resolution used by current firmware versions is about 10 s. Client software should not rely on shorter reaction times.

Please note:
Enabling the watchdog (`watchdog`) or changing the watchdog timeout period (`watchdogtimeout`) using the command `set_scanoutput_config` implicitly feeds the watchdog (i.e. the watchdog timeout is reset).

3.2.4 Scan data output customization

A client may customize some properties of how scan data is output over a scan data channel. These configuration settings are specific to a single data connection identified by the unique connection handle. The settings can be set while initiating a new scan data connection using `request_handle_tcp` (see section 3.3.2) and `request_handle_udp`(see section 3.3.1), or can be changed for an existing scan data connection using `set_scanoutput_config` (see section 3.3.6).

Selecting a start angle

A client may define a (virtual) start angle for scan data output using the parameter `start_angle` (see section 3.3.6). All scan points recorded *before* this start angle (in scan direction) are discarded. The first scan point (index 0) of a scan has a angle which is equal to or behind the given start angle. The parameter `start_angle` does not control the angular value at which scan points are *recorded*. It only defines a criterion, which scan points should be *output* for a specific scan data connection.

The value of `start_angle` refers to the (polar) measurement angle of the scan data coordinate system (see section 3.1.2). By default `start_angle` is set to the beginning of the sensors angular field of view at –180.0° (i.e. `start_angle` $= -1800000$) for `ccw` rotation. Subsequent scan points (index $(n+1)$) within the scan data stream are ordered according to the direction of rotation of the measuring beam.

Limiting the number of scan points

The parameter `max_num_points_scan` allows to limit the number of scan points that are *output* over a scan data connection. In contrast to the global parameter `samples_per_scan`, which controls how many samples per scan are *recorded* by the sensor, the setting of `max_num_points_scan` affects the number of scan points *output* for a specific scan data connection only.

If `max_num_points_scan` is set to a value below `samples_per_scan`, the client application receives less scan points than the sensor records. In combination with the parameter `start_angle` this allows client application to obtain only a segment (sector) of a scan instead of all recorded scan points. Figure 3.3 visualizes such a setup. This can be very useful to reduce data traffic if the full field of view of the sensor is not needed.

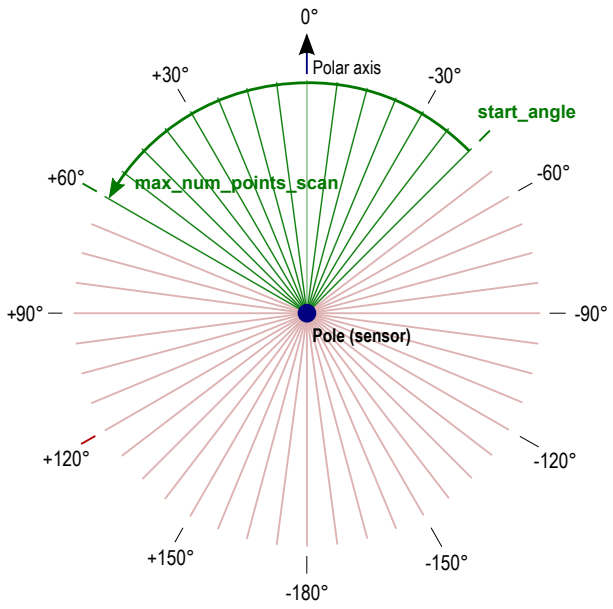


Figure 3.3: Restriction of scan output to a segment

Please note:
Setting `max_num_points_scan` to a value above `samples_per_scan` will have no effect. The resulting scan output will not contain any additional (dummy) samples.

PFSDP compatibility note:
The parameter `max_num_points_scan` is available on devices with PFSDP version 1.01 or newer.

Limiting the number of scans

The parameter `skip_scans` allows to reduce the *output* rate of scans over a scan data connection in comparison to the *recording* rate of scans as defined by the global parameter `scan_frequency`. The setting of `skip_scans` affects the *output* of scans for a specific scan data connection only (see section 3.3.6).

This option is useful for applications that require a high scan rate in order to reduce the motion blur effect in scans recorded in a dynamic environment (e.g. with moving object) but do not need every scan at such a high rate.

The decimation of scan output is transparent to the receiving client. The entry `scan_number` in the scan data header (see section 3.4.2) counts transmitted scans and ignores skipped scans. Therefore a client receiver handle the configuration `scan_frequency`=50 with `skip_scans`=4 the same way as the configuration `scan_frequency`=10 with `skip_scans`=0. In both cases scans are received at a rate of 10 Hz.

PFSDP compatibility note:
The parameter `skip_scans` is available on devices with PFSDP version 1.03 or newer.

Additional scan data packet checksum

The parameter `packet_crc` allows a client to activate an additional 32 bit checksum calculation for each scan data packet. The checksum is calculated over the whole packet including the packet header, the packet payload and any padding. If enabled, the resulting 32 bit CRC value is appended at the very end of each scan data packet (field `packet_crc` – see section 3.4.1). Client application implementations should use the offset (`packet_size` − 4) to access this value.

An additional scan data checksum is usually only required for applications with exceptional requirements regarding data integrity. For typical applications data integrity is already ensured by the checksums of the underlying TCP or UDP transport layer as well as the Ethernet data link layer. Due to performance considerations it is recommended to enable the additional scan data checksum only if it is required by the specific application.

The PFSDP packet checksum calculation can be configured by setting `packet_crc` to one of the following values:

value	description
`none`	Checksum calculation is disabled. The field `packet_crc` is not present in scan data packets.
`CRC32C`	Checksum is calculated using the CRC-32C algorithm: `width=32 poly=0x1edc6f41 init=0xffffffff refin=true refout=true xorout=0xffffffff`

The checksum is calculated byte-by-byte in memory byte-order. To verify the packet checksum on the client side it is recommended to use a CRC library supporting the CRC-32C algorithm in order to avoid implementation issues.

Example: The CRC-32C checksum of the buffer {`0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08`} should be `0x46891F81`. [10]

Please note:
If packet checksum calculation is disabled the field `packet_crc` is not present in scan data packets (see section 3.4.1).

PFSDP compatibility note:
The parameter `packet_crc` is available on devices with PFSDP version 1.04 or newer.

3.2.5 Using multiple concurrent scan data connections

As pointed out before, the scan data protocol is designed to support multiple concurrent scan data connections. However, CPU resources are limited for current R2000 devices. It depends on the measuring configuration of the sensor (see section 2.6) how many concurrent connections can be operated without adverse effects due to system overload.

It is the responsibility of the client application software to ensure that the system load resulting from concurrent data channels can be handled by the sensor (see section 1.1). Neither a successful request of a connection handle (see sections 3.3.1 and 3.3.2) nor a successful connection establishment guarantee that the requested amount of data can be continuously provided by the sensor in real time. System load of the device can be monitored by reading the system status variable `load_indication` – see section 2.8.

The maximum number of connections is limited to the value of `max_connections` (see section 2.4).

Please note:
A single client data channel can be handled by the sensor without any restrictions.

3.3 Commands for managing scan data output

The subsequent sections describe all commands.

3.3.1 `request_handle_udp` – request for an UDP-based scan data channel

The command `request_handle_udp` is used to request a handle for an UDP-based scan data transmission from the sensor to the client. If successful the sensor will send scan data to the client using the target IP address and UDP port specified at the handle request. Figure 3.4 gives an overview on the communication between sensor and client when using an UDP-based channel for scan data output.

Command arguments

The command `request_handle_udp` accepts the following arguments:

argument name	type	unit	description	default
`address`	`ipv4`	–	required: target IP address of the client	–
`port`	`uint`	–	required: target port for UDP data channel (client side)	–
`watchdog`	`bool`	`on` / `off`	optional: enable or disable connection watchdog	`on`
`watchdogtimeout`	`uint`	1 ms	optional: connection watchdog timeout period	60 000 ms
`packet_type`	`enum`	–	optional: scan data packet type: `A`, `B`, `C`, (see section 3.4)	`A`
`packet_crc`	`enum`	–	optional: enable additional checksum for scan data packets	`none`
`start_angle`	`int`	0.0001 °	optional: angle of first scan point for scan data output	`-1800000`
`max_num_points_scan`	`uint`	samples	optional: limit number of points in scan data output	`0` (unlimited)
`skip_scans`	`uint`	scans	optional: reduce scan output rate to every (n+1)th scan	`0` (unlimited)

The optional arguments of `request_handle_udp` facilitate an adequate initial configuration of the scan data output, which can be later modified using the command `set_scanoutput_config`. Please refer to section 3.3.6 for a detailed description of these optional arguments.

Command return values

- `handle` – unique (random) alpha-numeric string as identifier (handle) for the new UDP data channel

During a valid command call the scanner creates a new UDP channel to the client using the specified target IP address and port number. In case of an error the returned value for `handle` is invalid and `error_code` / `error_text` return details regarding the cause of the negative response (see section 1.2.6).

Please note:
The scanner will refuse a request to create a new UDP channel if the maximum number of concurrent client connections (TCP or UDP) is exceeded (see `max_connections` in section 2.4.1).

Please note:
Since an UDP scan data connection is established from the sensor to the client (“incoming connection”) it is prone to be blocked by firewall software. Please ensure that your firewall settings allow incoming UDP connections from the sensor IP address to your client application.

Please note:
Applications should not make any assumption regarding the structure of a handle. Handles should be treated as random alpha-numeric string of max. 16 characters.

Command example

Query: `http://<sensor IP address>/cmd/request_handle_udp?address=192.168.10.20&port=54321&packet_type=C`

Reply:
```
{
    "handle":"s10",
    "error_code":0,
    "error_text":"success"
    }
```

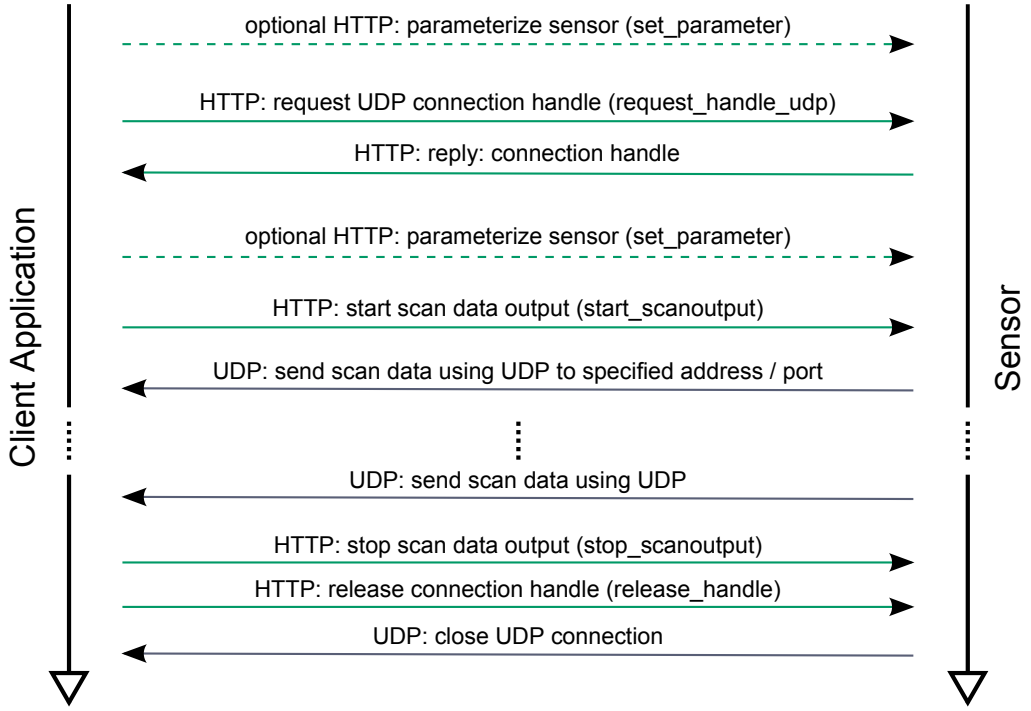


Figure 3.4: Timeline: scan data transmission using UDP

3.3.2 `request_handle_tcp` – request for a TCP-based scan data channel

The command `request_handle_tcp` is used to request a handle for a TCP-based scan data transmission from the sensor to the client. If successful, the client is allowed to create a new TCP connection to the sensor in order to receive scan data. Figure 3.5 gives an overview on the communication between sensor and client when using an TCP-based channel for scan data output.

Command arguments

The command `request_handle_tcp` accepts the following arguments:

argument name	type	unit	description	default
`address`	`ipv4`	–	optional: IP address of the client	(see below)
`port`	`uint`	–	optional: desired port for client connection (sensor side)	(see below)
`watchdog`	`bool`	`on` / `off`	optional: enable or disable connection watchdog	`on`
`watchdogtimeout`	`uint`	1 ms	optional: connection watchdog timeout period	60 000 ms
`packet_type`	`enum`	–	optional: scan data packet type: `A`, `B`, `C`, (see section 3.4)	`A`
`packet_crc`	`enum`	–	optional: enable additional checksum for scan data packets	`none`
`start_angle`	`int`	0.0001 °	optional: angle of first scan point for scan data output	`-1800000`
`max_num_points_scan`	`uint`	samples	optional: limit number of points in scan data output	`0` (unlimited)
`skip_scans`	`uint`	scans	optional: reduce scan output rate to every (n+1)th scan	`0` (unlimited)

The optional arguments of `request_handle_tcp` facilitate an adequate initial configuration of the scan data output, which can be later modified using the command `set_scanoutput_config`. Please refer to section 3.3.6 for a detailed description of these optional arguments.

Command return values

- `handle` – unique (random) alpha-numeric string as identifier (handle) for the new TCP data channel
- `port` – port number for the new TCP data channel

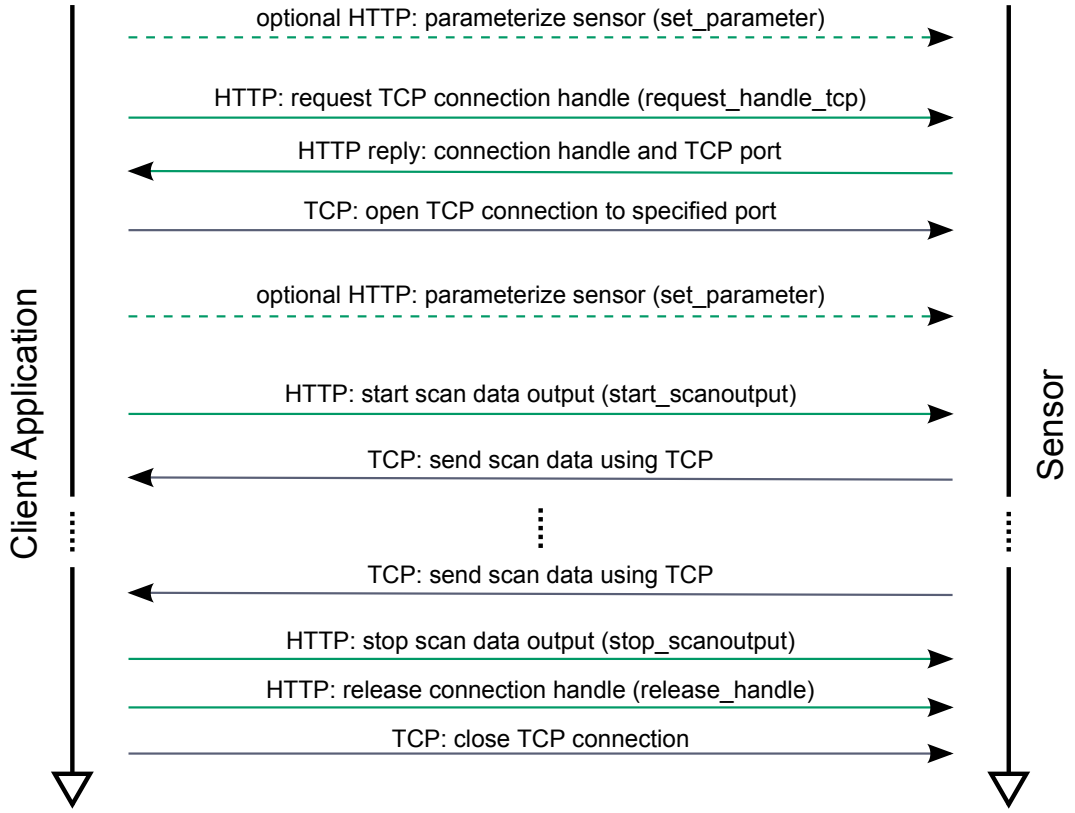


Figure 3.5: Timeline: scan data transmission using TCP

On success the sensor returns a TCP port number in `port`, which is now open for a client data connection. Note, that each port accepts a single TCP connection only! If the argument `address` had been specified upon calling `request_handle_tcp`, the scanner accepts an IP connection originating from the given IP address only (using the returned `handle`).

Using the argument `port` the client can try to reserve a specific port for its TCP connection. If this port is already in use, `request_handle_tcp` returns an error. If the argument `port` is not specified the sensor autonomously selects an ephemeral port within the range 32768 – 61000.

A call to `request_handle_tcp` might fail, e.g. if the maximum number of concurrent client connections is reached. In case of an error the returned values for `handle` and `port` are invalid and `error_code` / `error_text` provide error details (see section 1.2.6).

Please note:
It is recommended to use automatic port selection by the sensor instead of requesting a specific port number using the command argument `port`. A fixed port might be blocked by other applications or a previous connection.

Please note:
Applications should not make any assumption regarding the structure of a handle. Handles should be treated as random alpha-numeric string of max. 16 characters.

Please note:
The scanner will refuse a request to create a new TCP channel if the maximum number of concurrent client connections (TCP or UDP) is exceeded (see `max_connections` in section 2.4.1).

Command example

Query: `http://<sensor IP address>/cmd/request_handle_tcp?packet_type=A&watchdogtimeout=1000&start_angle=0`

Reply:
```
{
    "port":39731,
    "handle":"s22",
    "error_code":0,
    "error_text":"success"
    }
```

3.3.3 `release_handle` – release a data channel handle

Using the command `release_handle` the client can release a data channel handle. Any active scan data output using this handle will be stopped immediately. An associated UDP-based data channel is closed by the sensor itself. An associated TCP-based data channel should be closed by the client.

Command arguments

argument name	type	description
`handle`	`string`	handle for scan data channel (max. 16 chars) (required argument – always specified first)

Command example

Query: `http://<sensor IP address>/cmd/release_handle?handle=s22`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

3.3.4 `start_scanoutput` – initiate output of scan data

The command `start_scanoutput` starts the transmission of scan data for the data channel specified by the given handle. When started, the sensor will begin sending scan data to the client using an established UDP or TCP channel with the given handle – see section 3.3.1 and section 3.3.2. (Re-)starting a scan data transmission also resets the counters for scan number and scan packet number in the scan data header (see section 3.4.2). Scan data output always starts at the beginning of a new scan (with scan number 0 and scan packet number 1).

Command arguments

argument name	type	description
`handle`	`string`	handle for scan data channel (max. 16 chars) (required argument – always specified first)

Command example

Query: `http://<sensor IP address>/cmd/start_scanoutput?handle=s22`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

3.3.5 `stop_scanoutput` – terminate output of scan data

The command `stop_scanoutput` stops the transmission of scan data for the data channel specified by the given handle. Scan data output stops immediately after the current scan data packet – not necessarily at the end of a full scan.

Please note:
TCP clients might still receive several scan data packets after sending `stop_scanoutput`, due to the TCP stack data queue.

Command arguments

argument name	type	description
`handle`	`string`	handle for scan data channel (max. 16 chars) (required argument – always specified first)

Command example

Query: `http://<sensor IP address>/cmd/stop_scanoutput?handle=s22`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

3.3.6 `set_scanoutput_config` – reconfigure scan data output

Using the command `set_scanoutput_config` the client can parametrize scan data output separately for each active scan data output channel. All command arguments solely apply to the *output* of scan data. Customization of (global) parameters referring to the recording of measurements (scan data) is done by use of the command `set_parameter` (see section 2.6).

Command arguments

argument name	type	unit	description	default
`handle`	`string`	–	handle for scan data channel (max. 16 chars) (required argument – always specified first)	–
`watchdog`	`bool`	`on` / `off`	optional: enable or disable connection watchdog	`on`
`watchdogtimeout`	`uint`	1 ms	optional: connection watchdog timeout period	60 000 ms
`packet_type`	`enum`	–	optional: scan data packet type: `A`, `B`, `C`, (see section 3.4)	`A`
`packet_crc`	`enum`	–	optional: enable additional checksum for scan data packets	`none`
`start_angle`	`int`	0.0001 °	optional: angle of first scan point for scan data output	`-1800000`
`max_num_points_scan`	`uint`	samples	optional: limit number of points in scan data output	`0` (unlimited)
`skip_scans`	`uint`	scans	optional: reduce scan output rate to every (n+1)th scan	`0` (unlimited)

It is recommended (but not required) to stop sensor data output while using `set_scanoutput_config`. In case scan data output is active, the point in time when modified configuration settings are applied to the running data stream is non-deterministic. After the new settings are applied, scan data output is suspended until the start of an new scan (skipping scan data packets in-between). If the client application depends on a deterministic switching behavior, it should stop scan data transmission first using `stop_scanoutput`, change settings using `set_scanoutput_config` and finally restart the data stream with `start_scanoutput`.

Parameter `start_angle`

The user can control the angle for the first scan point of a scan by means of the parameter `start_angle`. The range of valid values is [`-1800000`; `+1800000`[including `-1800000` (–180 °) but excluding `+1800000` (+180 °). The specified value does not have to match the configured angular resolution for scan data acquisition (see section 2.6) – the sensor will start scan data output with the first scan point whose recording angle is equal to or following behind the specified angle in direction of rotation.

Please note:
The command `get_scanoutput_config` (see section 3.3.7) will return the exact user specified value, while the entry “absolute angle of first scan point” within the scan data packet header (see section 3.4.2) will specify the exact value of the first scan point actually used.

Command example

Query: `http://<sensor IP address>/cmd/set_scanoutput_config?handle=s22&packet_type=B&start_angle=-900000`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

Parameter `max_num_points_scan`

This parameter allows to limit the number of samples that are output for each scan. In combination with the parameter `start_angle` a client application can reduce scan data output to a single region of interest (*sector*). Please refer to section 3.2.4 for further details.

The parameter is specified as unsigned integer (`uint`) and accepts any non-negative number. The value `0` is recognized as special case for 'no limitation', i.e. the sensor outputs always all points of scan. This is also the default value.

Parameter `skip_scans`

This parameter allows an application to receive only every (n+1)th recorded scan. All other scans are not transmitted to the client thus reducing communication load significantly. Please refer to section 3.2.4 for further details.

The parameter is specified as unsigned integer (`uint`) and accepts any non-negative number. The default setting is to output all scans recorded (`skip_scans` set to value `0`).

3.3.7 `get_scanoutput_config` – read scan data output configuration

The command `get_scanoutput_config` returns the current scan data output configuration for a specified scan data output channel (UDP or TCP).

Command arguments

argument name	type	description
`handle`	`string`	handle for scan data channel (max. 16 chars) (required argument – always specified first)
`list`	`string`	semicolon separated list of parameter names (optional)

If the argument `list` is not specified the command will return the current value of all available configuration parameters (see section 3.3.6).

Command example

Query: `http://<sensor IP address>/cmd/get_scanoutput_config?handle=s22`

Reply:
```
{
    "address":"0.0.0.0",
    "port":39050,
    "watchdog":"on",
    "watchdogtimeout":60000,
    "packet_type":"A",
    "start_angle":-1800000,
    "error_code":0,
    "error_text":"success"
    }
```

3.3.8 `feed_watchdog` – feed connection watchdog

The command `feed_watchdog` feeds the connection watchdog, i.e. each call of this command resets the watchdog timer. Please refer to section 3.2.3 for a detailed description of the connection watchdog mechanism.

Command arguments

argument name	type	description
`handle`	`string`	handle for scan data channel (max. 16 chars) (required argument – always specified first)

Command example

Query: `http://<sensor IP address>/cmd/feed_watchdog?handle=s36924971`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

Please note:
Enabling the watchdog (`watchdog`) or changing the watchdog timeout period (`watchdogtimeout`) using the command `set_scanoutput_config` (see section 3.3.6) implicitly feeds the watchdog as well.

3.3.9 TCP in-line watchdog feeds

A TCP "in-line" watchdog feed uses the backward channel of an existing TCP scan data connection (see section 3.3.2). It allows to feed the connection watchdog without imposing an HTTP connection for every feed action as required by the `feed_watchdog` command (see section 3.3.8).

Feed sequence

In order to feed the watchdog of an existing TCP scan data connection the following byte sequence needs to be send from the client application to the sensor:

`0x66 0x65 0x65 0x64 0x77 0x64 0x67 0x04`

This 8-byte sequence represents the ASCII string `feedwdg<eot>`, which is recognized by the sensor. The sensor does not send any confirmation whether the watchdog request has been processed. However, since the TCP connection ensures an error free transmission this confirmation is not needed anyway.

Please note:
Due to limitations of the sensor firmware client applications should not send in-line watchdog feed requests more often than once per second (maximum feed rate of 1 Hz).

Please note:
Enabling the watchdog (`watchdog`) or changing the watchdog timeout period (`watchdogtimeout`) using the command `set_scanoutput_config` (see section 3.3.6) implicitly feeds the watchdog as well.

PFSDP compatibility note:
TCP in-line watchdog feeds are available on devices with PFSDP version 1.01 or newer.

3.4 Transmission of scan data

Scan data is always transmitted within packets. A complete scan is usually transmitted using multiple scan data packets (see section 1.1 for basic design considerations). Each packet comprises of a generic header, a scan data specific header and the actual scan data.

A new scan will always start with a new scan data packet, i.e. the first sample of a new scan will always appear as first sample of a new packet. Each scan data packet is transmitted as soon as the required data is available. This *streaming* approach allows a client application to start processing scan data with minimal delay – eliminating the need to wait until the full scan is recorded and transmitted to the client completely.

Multiple scan data packet types are defined to output different sets of scan data information efficiently. These packet types follow a standard structure – differing in the bulk scan data only. Within bulk scan data each scan point is represented by a structure containing the favored amount of data (distance, amplitude, etc.). The following sections describe scan data packets in detail.

3.4.1 Basic packet structure

Each data packet has the following basic structure:

type	name	description
`uint16`	`magic`	magic byte (0xa25c) marking the beginning of a packet
`uint16`	`packet_type`	type of scan data packet (low-byte: payload type, high-byte: header type)
`uint32`	`packet_size`	overall size of this packet in bytes (header, payload, checksum)
`uint16`	`header_size`	size of header in bytes (i.e. offset to payload data)
`...`	`...`	packet type specific additional header information
`uint8[]`	`header_padding`	0-3 bytes padding (to align the header size to a 32bit boundary)
`...`	`payload_data`	packet type specific payload data
`uint8[]`	`payload_padding`	0-3 bytes padding (to align the payload size to a 32bit boundary)
`uint32`	`packet_crc` (optional)	optional checksum of whole packet (except this field) **Note:** This field is only present if packet checksums are enabled (see section 3.2.4)

Please note:
Although the structure of the packet usually appears to be fixed, it is highly recommended that client applications always evaluate the entries for packet size and header size since they may change due to future extensions.

The magic byte at the beginning of the packet header is designed to be used as synchronization mark within a continuous data stream. It can be ignored if synchronization is not needed.

The starting address of payload data is always aligned to a 32bit address boundary by using padding bytes within the header (`header_padding`). Additionally, the overall size of the packet is always aligned to 32bit boundary. Depending on the scan data packet type there might be additional padding bytes (`payload_padding`) at the end of the packet. Currently, this is only the case for scan data packets of type B (see section 3.4.5) containing an odd number of points.

3.4.2 Typical structure of a scan data header

A scan data packet contains a scan data header with information on the scan and the scan data itself. The scan data header is designed in ways that each scan data packet can be processed independent of other scan data packets belonging to the same scan.

A typical scan data header has the following structure:

type	name	description
uint16	magic	magic byte (0xa25c) marking the beginning of a packet
uint16	packet_type	type of scan data packet
uint32	packet_size	overall size of this packet in bytes (header, payload, checksum)
uint16	header_size	size of header in bytes (i.e. offset to payload data)
uint16	scan_number	sequence number for scan (counting transmitted scans, starting with 0, overflows)
uint16	packet_number	sequence number for packet (counting packets of a particular scan, starting with 1)
ntp64	timestamp_raw	raw timestamp of first scan point in this packet (see section 3.1.5)
uint64	reserved	reserved field
uint32	status_flags	scan status flags (see section 3.4.3)
uint32	scan_frequency	currently configured scan rate (0.001 Hz)
uint16	num_points_scan	number of scan points (samples) within complete scan (depending on configured FOV)
uint16	num_points_packet	number of scan points within this packet
uint16	first_index	index of first scan point within this packet
int32	first_angle	absolute angle of first scan point in this packet (0.0001 °)
int32	angular_increment	delta angle between two scan points (0.0001 °) (CCW rotation: positive increment, CW rotation: negative increment)
uint32	iq_input	bit field for switching input state (see section 7.3.2) (all bits zero for devices without switching I/Q)
uint32	iq_overload	bit field for switching output overload warning (see section 7.3.2) (all bits zero for devices without switching I/Q)
ntp64	iq_timestamp_raw	raw timestamp for status of switching I/Q (see section 3.1.5)
uint64	reserved	reserved field
uint8[]	header_padding	0-3 bytes padding (to align the header size to a 32bit boundary)
...	scandata	packet type specific scan data
uint8[]	payload_padding	0-3 bytes padding (to align the payload size to a 32bit boundary)
uint32	packet_crc (optional)	optional checksum of whole packet (except this field) **Note:** This field is only present if packet checksums are enabled (see section 3.2.4)

Please note:
The field `num_points_scan` states the total number of scan points *output* for each recorded scan. It is always equal to either `samples_per_scan` or `max_num_points_scan`, whichever is smaller for the specific scan data connection. Please refer to section 3.2.4 for more details on this matter.

Please note:
The field `scan_frequency` states the current *target value* for the scan acquisition rate as defined by the global parameter `scan_frequency` (see section 2.6.3). The *instantaneous value* of the angular velocity of the measuring beam can be estimated from the timestamps of the current and subsequent scan data packets.

Please note:
Angular values specified with a resolution of 0.0001 ° are usually prone to rounding errors due to the decimal range of values. They are part of the header for convenience only. Subsequent calculations requiring precise angular values should calculate an exact angle for each scan point by reference to its index number, the configured angular increment and the configured start angle of the scan:

CCW rotation: $$exact\_angle_{scanpoint} = start\_angle_{scan} + index_{scanpoint} * \frac{angular\_fov}{num\_points\_scan}$$

CW rotation: $$exact\_angle_{scanpoint} = start\_angle_{scan} - index_{scanpoint} * \frac{angular\_fov}{num\_points\_scan}$$

3.4.3 Scan data header status flags

Scan data header status flags are similar to system status flags (see section 2.8.2) but provide status information specific to the scan data of a scan data packet. Each scan data header contains an `uint32` entry `status_flags` (see section 3.4.2) comprised of the following flags:

bit	flag name	description
Informational		
0	`scan_data_info`	Accumulative flag – set if any informational flag (bits `1..7`) is set
1	`new_settings`	System settings for scan data acquisition changed during recording of this packet. This flag is triggered by write accesses to global parameters affecting the measuring configuration (see section 2.6) which can be done by any client. Changes to connection-specific parameters (see section 3.3.6) do not trigger this flag!
2	`invalid_data`	Consistency of scan data is not guaranteed for this packet.
3	`unstable_rotation`	Measured scan rate did not match set value while recording this scan data packet.
4	`skipped_packets`	Preceding scan data packets have been skipped due to connection issues, changes to scan data acquisition settings or scan data inconsistencies.
Warnings		
8	`device_warning`	Accumulative flag – set if any warning flag (bits `9..15`) is set
9	`lens_contamination_warning`	LCM warning threshold triggered for at least one sector (see chapter 5)
10	`low_temperature_warning`	Current internal temperature below warning threshold
11	`high_temperature_warning`	Current internal temperature above warning threshold
12	`device_overload`	Overload warning – sensor CPU overload is imminent
Errors		
16	`device_error`	Accumulative flag – set if any error flag (bits `17..23`) is set
17	`lens_contamination_error`	LCM error threshold triggered for at least one sector (see chapter 5)
18	`low_temperature_error`	Current internal temperature below error threshold
19	`high_temperature_error`	Current internal temperature above error threshold
20	`device_overload`	Overload error – sensor CPU is in overload state
Defects		
30	`device_defect`	Accumulative flag – set if device detected an unrecoverable defect

Please note:
All flags not listed in the above table are reserved and should be ignored.

3.4.4 Scan data packet type A – distance only

Scan data packets of type A have the following structure:

type	name	description
packet header		
uint16	magic	magic byte (0xa25c) marking the beginning of a packet
uint16	packet_type	type of scan data packet: 0x0041 (ASCII character 'A')
uint32	packet_size	overall size of this packet in bytes (header, payload, checksum)
uint16	header_size	size of header in bytes (i.e. offset to payload data)
uint16	scan_number	sequence number for scan (counting transmitted scans, starting with 0, overflows)
uint16	packet_number	sequence number for packet (counting packets of a particular scan, starting with 1)
ntp64	timestamp_raw	raw timestamp of first scan point in this packet (see section 3.1.5)
uint64	reserved	reserved field
uint32	status_flags	scan status flags (see section 3.4.3)
uint32	scan_frequency	currently configured scan rate (0.001 Hz)
uint16	num_points_scan	number of scan points (samples) within complete scan (depending on configured FOV)
uint16	num_points_packet	number of scan points within this packet
uint16	first_index	index of first scan point within this packet
int32	first_angle	absolute angle of first scan point in this packet (0.0001°)
int32	angular_increment	delta angle between two scan points (0.0001°) (CCW rotation: positive increment, CW rotation: negative increment)
uint32	iq_input	bit field for switching input state (see section 7.3.2) (all bits zero for devices without switching I/Q)
uint32	iq_overload	bit field for switching output overload warning (see section 7.3.2) (all bits zero for devices without switching I/Q)
ntp64	iq_timestamp_raw	raw timestamp for status of switching I/Q (see section 3.1.5)
uint64	reserved	reserved field
uint8[]	header_padding	0-3 bytes padding (to align the header size to a 32bit boundary)
scan point data		
uint32	distance	measured distance (in mm) Invalid measurements return 0xFFFFFFFF.
	...	
packet checksum		
uint32	packet_crc (optional)	optional checksum of whole packet (except this field) **Note:** This field is only present if packet checksums are enabled (see section 3.2.4)

Please note:
The field num_points_scan states the total number of scan points *output* for each recorded scan. It is always equal to either samples_per_scan or max_num_points_scan, whichever is smaller for the specific scan data connection. Please refer to section 3.2.4 for more details on this matter.

Please note:
The field scan_frequency states the current *target value* for the scan acquisition rate as defined by the global parameter scan_frequency (see section 2.6.3). The *instantaneous value* of the angular velocity of the measuring beam can be estimated from the timestamps of the current and subsequent scan data packets.

3.4.5 Scan data packet type B – distance and amplitude

Scan data packets of type B have the following structure:

type	name	description
packet header		
uint16	magic	magic byte (0xa25c) marking the beginning of a packet
uint16	packet_type	type of scan data packet: 0x0042 (ASCII character 'B')
uint32	packet_size	overall size of this packet in bytes (header, payload, checksum)
uint16	header_size	size of header in bytes (i.e. offset to payload data)
uint16	scan_number	sequence number for scan (counting transmitted scans, starting with 0, overflows)
uint16	packet_number	sequence number for packet (counting packets of a particular scan, starting with 1)
ntp64	timestamp_raw	raw timestamp of first scan point in this packet (see section 3.1.5)
uint64	reserved	reserved field
uint32	status_flags	scan status flags (see section 3.4.3)
uint32	scan_frequency	currently configured scan rate (0.001 Hz)
uint16	num_points_scan	number of scan points (samples) within complete scan (depending on configured FOV)
uint16	num_points_packet	number of scan points within this packet
uint16	first_index	index of first scan point within this packet
int32	first_angle	absolute angle of first scan point in this packet (0.0001°)
int32	angular_increment	delta angle between two scan points (0.0001°) (CCW rotation: positive increment, CW rotation: negative increment)
uint32	iq_input	bit field for switching input state (see section 7.3.2) (all bits zero for devices without switching I/Q)
uint32	iq_overload	bit field for switching output overload warning (see section 7.3.2) (all bits zero for devices without switching I/Q)
ntp64	iq_timestamp_raw	raw timestamp for status of switching I/Q (see section 3.1.5)
uint64	reserved	reserved field
uint8[]	header_padding	0-3 bytes padding (to align the header size to a 32bit boundary)
scan point data		
uint32	distance	measured distance (in mm) Invalid measurements return 0xFFFFFFFF.
uint16	amplitude	measured amplitude (padded 12bit value – most significant bits are zero) Please see section 3.1.4 for a description of amplitude data values.
	...	
padding		
uint8[]	payload_padding	0 or 2 bytes padding (to align the payload size to a 32bit boundary)
packet checksum		
uint32	packet_crc (optional)	optional checksum of whole packet (except this field) **Note:** This field is only present if packet checksums are enabled (see section 3.2.4)

Please note:
The field num_points_scan states the total number of scan points *output* for each recorded scan. It is always equal to either samples_per_scan or max_num_points_scan, whichever is smaller for the specific scan data connection. Please refer to section 3.2.4 for more details on this matter.

Please note:
The field scan_frequency states the current *target value* for the scan acquisition rate as defined by the global parameter scan_frequency (see section 2.6.3). The *instantaneous value* of the angular velocity of the measuring beam can be estimated from the timestamps of the current and subsequent scan data packets.

3.4.6 Scan data packet type C – distance and amplitude (compact)

Scan data packets of type C have the following structure:

type	name	description
packet header		
uint16	magic	magic byte (0xa25c) marking the beginning of a packet
uint16	packet_type	type of scan data packet: 0x0043 (ASCII character 'C')
uint32	packet_size	overall size of this packet in bytes (header, payload, checksum)
uint16	header_size	size of header in bytes (i.e. offset to payload data)
uint16	scan_number	sequence number for scan (counting transmitted scans, starting with 0, overflows)
uint16	packet_number	sequence number for packet (counting packets of a particular scan, starting with 1)
ntp64	timestamp_raw	raw timestamp of first scan point in this packet (see section 3.1.5)
uint64	reserved	reserved field
uint32	status_flags	scan status flags (see section 3.4.3)
uint32	scan_frequency	currently configured scan rate (0.001 Hz)
uint16	num_points_scan	number of scan points (samples) within complete scan (depending on configured FOV)
uint16	num_points_packet	number of scan points within this packet
uint16	first_index	index of first scan point within this packet
int32	first_angle	absolute angle of first scan point in this packet (0.0001°)
int32	angular_increment	delta angle between two scan points (0.0001°) (CCW rotation: positive increment, CW rotation: negative increment)
uint32	iq_input	bit field for switching input state (see section 7.3.2) (all bits zero for devices without switching I/Q)
uint32	iq_overload	bit field for switching output overload warning (see section 7.3.2) (all bits zero for devices without switching I/Q)
ntp64	iq_timestamp_raw	raw timestamp for status of switching I/Q (see section 3.1.5)
uint64	reserved	reserved field
uint8[]	header_padding	0-3 bytes padding (to align the header size to a 32bit boundary)
scan point data		
uint20	distance	measured distance (in mm) – maximum representable value is 1 km Invalid measurements return 0xFFFFF.
uint12	amplitude	measured amplitude Please see section 3.1.4 for a description of amplitude data values.
	...	
packet checksum		
uint32	packet_crc (optional)	optional checksum of whole packet (except this field) **Note:** This field is only present if packet checksums are enabled (see section 3.2.4)

Scan data packets of type C differ from type B in the binary size of the values `distance` and `amplitude` only. For type C these values are encoded as bit fields within a `uint32` type.

Please note:
The field `num_points_scan` states the total number of scan points *output* for each recorded scan. It is always equal to either `samples_per_scan` or `max_num_points_scan`, whichever is smaller for the specific scan data connection. Please refer to section 3.2.4 for more details on this matter.

Please note:
The field `scan_frequency` states the current *target value* for the scan acquisition rate as defined by the global parameter `scan_frequency` (see section 2.6.3). The *instantaneous value* of the angular velocity of the measuring beam can be estimated from the timestamps of the current and subsequent scan data packets.

3.5 Data transmission using TCP

The TCP/IP-based scan data output provides a reliable and error proof channel for transmitting the stream of scan data packets. Communication partners have no control on how scan data packets are wrapped into one or more Ethernet frames (TCP segments), though. For this reason there is no 1:1 mapping between PFSDP scan data packets and Ethernet frames on the transport layer. A single Ethernet frame can contain (partial) data from more than one scan data packet. Furthermore, there is no simple rule on how the client TCP stack provides received data to the client application.

Please note:
If output of scan data is slowed down due to delayed or missing TCP acknowledgements from a client, high load on the scanner (e.g. concurrent requests from many clients) or other network congestion, the scanner may decide to skip transmission of scan data packets or complete scans in order to avoid increasing latency and excessive memory usage. It will never transmit only partial scan data packets. Additionally, skipped packets are signaled with the flag `skipped_packets` in the scan data header of the next scan data packet (see section 3.4.3).

3.6 Data transmission using UDP

The UDP/IP-based scan data output provides a low latency channel for scan data transmission. Each scan data packet is send as separate UDP message (datagram) using (at least) one Ethernet frame. In case an UDP message (scan data packet) is lost during transmission, no error correction is provided. Corrupted scan data packets are discarded. The client application can make use of all successfully received scan data packets though, since every scan data packet incorporates a full scan data header which allows to process the contained scan data separately.

Please note:
The sensor uses a special real time (RT) task for UDP scan data output in order to minimize latency. This RT task is currently available for a single UDP client connection only. Additional (parallel) UDP client connections are handled by non-RT tasks and might show inferior time behavior.

4 Filter-based scan data processing

4.1 Introduction to scan data filtering

Many typical customer applications cannot take advantage of the high angular resolution of the R2000 devices due to the large amount of data that need to be processed on the client side when running at the maximum sampling rate. The introduction of scan data filtering adds an option for in-device pre-processing, reducing the amount of scan data output while still utilizing the high scan resolution.

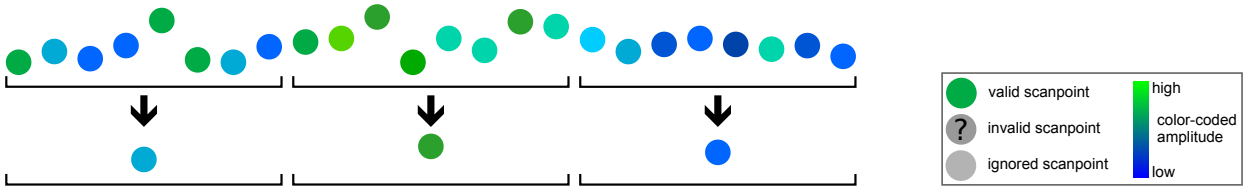
The basic idea of scan data filtering is to combine a configurable number of N adjacent scan points (input values) into a single resulting scan point (output value) using one of various predefined algorithms. A filter algorithm calculates both a distance value and an amplitude value from the input data. The resulting scan point is placed at the center of the processing window for both angular value and timestamp value. All operations are performed in the sensor coordinate system (see section 3.1.1).

Please note:
Scan data filtering is applied *globally*, i.e. its settings affect *all clients*. It should be treated similar to the (global) measuring configuration (see section 2.6).

Scan data filtering can be considered as transparent to a client application. On protocol level there is no difference between a scan recorded with a lower resolution and a scan recorded with a high resolution and scan data filtering enabled. However, the latter provides a potentially higher signal quality. For example an application using 3150 points/scan at 10 Hz may instead also use a scan resolution of 25200 points/scan at 10 Hz with an 8:1 decimation filtering enabled. With both configurations, scan data output has an effective sampling rate of 31.5 kHz.

4.1.1 Block-wise processing

Filter algorithms with block-wise processing calculate a single output value for N input values. After processing the input values the input window is shifted by N values, i.e. each input value is processed only once. Thus the number of points in the resulting scan is reduced by a *decimation* factor of $1:N$ compared to the input scan, i.e. the output scan contains only $1/N$ scan points and has a N times coarser scan resolution (with a constant angular increment).



Above figure shows an example for the decimation process of 24 input scan points with a window size of 8 points. Each scan point is represented by a circle with a color-encoded echo amplitude (blue: low echo, green: high echo). The filtered result contains only 3 output scan points – on for each group of input values (8:1 decimation).

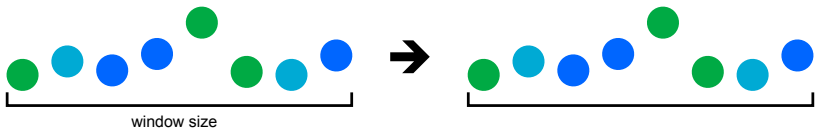
PFSDP compatibility note:
Block-wise scan data filtering requires a device with PFSDP version 1.03 or newer. Furthermore the device must support the device feature `scan_data_filter` – please refer to section 2.4 for details on sensor capabilities.

4.2 Filter algorithms

This section describes the available algorithms for scan data filtering, selectable by the global parameter `filter_type`. All parameters are discussed in detail in section 4.3.

4.2.1 No filter (pass-through)

Per default no filtering is performed on sensor data. All recorded scan points are passed-through to the client without change. This behavior is identical to devices that do not support scan data filtering (e.g. older firmware releases).

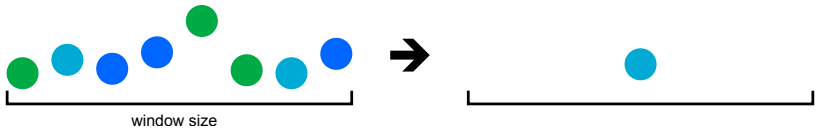


Related configuration parameters: –

4.2.2 Average filter

The *average* filter calculates a simple arithmetic average (distance and amplitude) of all scan data points within the configured window size (`filter_width`).

For block-wise processing the result is a single output scan point replacing the complete group of input scan points:

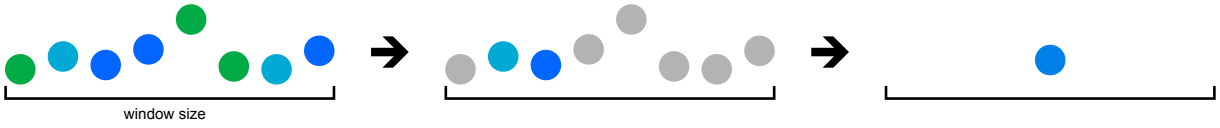


Related configuration parameters: `filter_width`, `filter_error_handling`

4.2.3 Median filter

The *median* filter calculates a median value from all scan data points within the configured window size (`filter_width`). For this purpose, first all scan points are (virtually) sorted by their distance value. For *odd* window sizes the middle scan point is selected as output sample. For *even* window sizes the two middle points are selected and the output sample is calculated as arithmetical average (for both distance and amplitude) of these points.

For block-wise processing the resulting scan point replaces the group of input scan points:

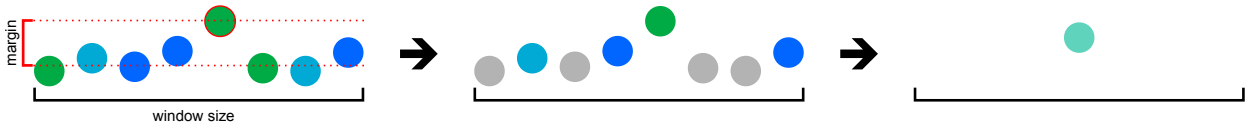


Related configuration parameters: `filter_width`, `filter_error_handling`

4.2.4 Maximum filter

The *maximum* filter is a more complex filter operation. It calculates the arithmetic average from a subset of scan points within the configured filter window (`filter_width`). Scan points are selected by first determining the scan point with the maximum distance within the current filter window. Then all scan points within this window are eliminated, whose distance value falls below the maximum distance value less a threshold value (`filter_maximum_margin`). The remaining points are used to calculate an arithmetic average for both distance and amplitude.

For block-wise processing the resulting scan point replaces the group of input scan points:

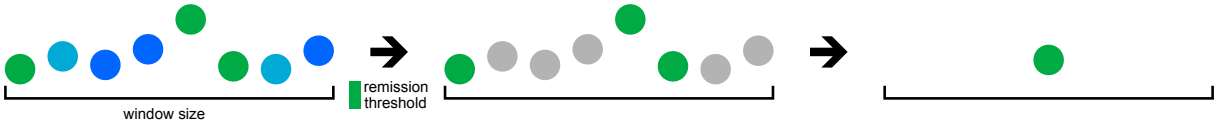


Related configuration parameters: `filter_width`, `filter_error_handling`, `filter_maximum_margin`

4.2.5 Remission filter

The *remission* filter calculates a simple arithmetic average (distance and amplitude) from a subset of scan points within the configured window size (`filter_width`). Scan points are selected by comparing their individual echo amplitude to a threshold value (`filter_remission_threshold`). Only scan points with an amplitude above the threshold are used to calculate a single average.

For block-wise processing the resulting scan point replaces the group of input scan points:



Related configuration parameters: `filter_width`, `filter_error_handling`, `filter_remission_threshold`

4.3 Filter configuration

Scan data filtering is configured globally using the commands for sensor parametrization (see section 2.2). This section gives an overview on the available settings.

4.3.1 Parameter overview

The following (global) parameters are available for configuration of scan data filtering:

parameter name	type	unit	description	access	default
`filter_type`	`enum`	–	algorithm for filtering (see section 4.3.2 for details)	RW	`none`
`filter_width`	`uint`	samples	window size for filtering (see section 4.3.3 for details)	RW	`4`
`filter_error_handling`	`enum`	–	strategy for filtering invalid values (see section 4.3.4 for details)	RW	`tolerant`
`filter_maximum_margin`	`uint`	1 mm	margin for filter type `maximum` (see section 4.3.5 for details)	RW	100 mm
`filter_remission_threshold`	`enum`	–	threshold for filter type `remission` (see section 4.3.6 for details)	RW	`reflector_std`

4.3.2 Filter types (filter_type)

The parameter filter_type selects the filtering algorithm that is applied globally to all scan data recorded by the sensor. Currently, the following algorithms are available (see section 4.2 for details):

filter type	description
none	Filtering disabled. Output all recorded samples (pass through).
average	Calculate arithmetic average from N raw samples (see section 4.2.2).
median	Calculate median from N raw samples (see section 4.2.3).
maximum	Filter raw samples by distance and calculate average (see section 4.2.4).
remission	Filter raw samples by remission and calculate average (see section 4.2.5).

Example

Query: `http://<sensor IP address>/cmd/set_parameter?filter_type=average`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

4.3.3 Filter width (filter_width)

The parameter filter_width controls the window size of the filter algorithm applied to all recorded scan data. It defines the number of recorded samples (scan data points) that are processed to produce a (single) filtered output sample. All filter algorithms available on R2000 devices are applied block-wise, i.e. the amount of output data is reduced by the ratio filter_width:1.

R2000 devices currently support the following window sizes: 2, 4, 8, 16

Please note:
The filtered output scan point is always placed at the center of the filter window for both angular value and timestamp value (see section 4.1).

Example

Query: `http://<sensor IP address>/cmd/set_parameter?filter_width=4`

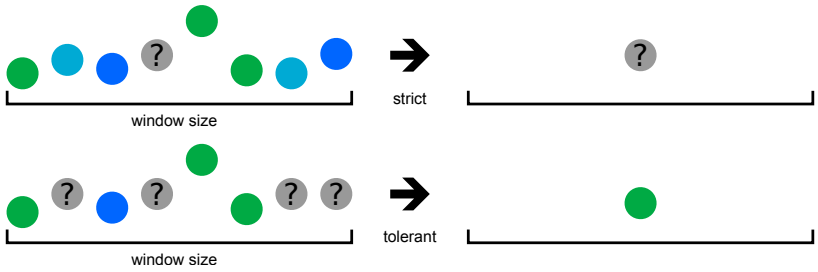
Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

4.3.4 Filter error handling (filter_error_handling)

The parameter filter_error_handling specifies how the filter algorithm is handling invalid measurement values within the group of scan data points as configured by filter_width.

parameter value	description
strict	Result is invalid, if any scan data point of the group is invalid.
tolerant	Result is valid, if at least one scan data point of the group is valid.

The following pictures illustrate this behavior:



Example

Query: `http://<sensor IP address>/cmd/set_parameter?filter_error_handling=tolerant`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

4.3.5 Maximum filter margin (`filter_maximum_margin`)

The parameter `filter_maximum_margin` is evaluated by the *maximum filter* algorithm (see section 4.2.4). It defines the allowed distance of a scan point to the maximum distance value within the group of scan data points. The parameter has a resolution of 1 mm and accepts values in the range from 0 mm up to 65 535 mm.

Example

Query: `http://<sensor IP address>/cmd/set_parameter?filter_maximum_margin=220`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

4.3.6 Remission filter threshold (`filter_remission_threshold`)

The parameter `filter_remission_threshold` controls the threshold for the *remission filter* algorithm (see section 4.2.5). The parameter can be set to one of several pre-defined thresholds representing the remission of typical target surfaces. All scan points with a remission below the configured threshold are filtered (marked as invalid). The following table lists the available parameter values:

threshold	type of target used as reference for filtering
`diffuse_low`	Natural, non-black targets (e.g. gray surfaces)
`diffuse_high`	Natural, bright targets (e.g. white surfaces)
`reflector_min`	Very small reflectors or very bright natural targets (e.g. metal surfaces)
`reflector_low`	Rather small reflectors or reflective natural surface (e.g. polished surfaces)
`reflector_std`	All typical reflectors
`reflector_high`	Larger reflectors
`reflector_max`	Large reflectors

Example

Query: `http://<sensor IP address>/cmd/set_parameter?filter_remission_threshold=diffuse_high`

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

5 Lens contamination monitor (LCM)

This chapter describes the capabilities and configuration of the *lens contamination monitor* (LCM).

PFSDP compatibility note:
The lens contamination monitor (LCM) is available on devices with PFSDP version 1.03 or newer, if the corresponding feature flag `lens_contamination_monitor` (see section 2.4.1) is set.

5.1 LCM introduction

The LCM continuously monitors the contamination of the sensors lens cover. The lens cover is segmented into 12 *sectors* – each covering a 30° field of view. The LCM sectors are numbered in counter-clockwise orientation starting with *Sector 0* at –180° at the back of the sensor. Figure 5.1 illustrates the mapping of the LCM sectors.

The lens contamination monitor (LCM) evaluates the contamination of each sector separately and compares it to a warning threshold and an error threshold. In case of a positive test result either a warning flag or an error flag is set for the respective sector. The evaluation can be enabled or disabled for each sector individually. Furthermore the sensitivity and reaction time of the LCM can be configured globally. The following sections provide details on the LCM configuration.

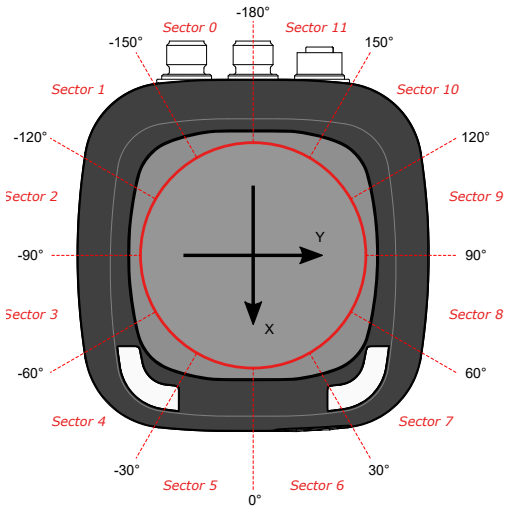


Figure 5.1: LCM sectors

5.2 LCM configuration

5.2.1 Parameter overview

The following (global) parameters are available for configuration of the lens contamination monitor:

parameter name	type	unit	description	access	default
`lcm_detection_sensitivity`	`enum`	–	sensitivity of lens contamination detection (`disabled`, `low`, `medium`, `high`)	RW	`disabled`
`lcm_detection_period`	`uint`	1 ms	reaction time of LCM to lens contamination	RW	5000 ms
`lcm_sector_enable`	`enum`	–	array of flags to enable / disable LCM sectors (`on` or `off`)	RW	`on` (all)
`lcm_sector_warn_flags`	`bitfield`	–	bit field with warning state of LCM sectors	RO	0
`lcm_sector_error_flags`	`bitfield`	–	bit field with error state of LCM sectors	RO	0

5.2.2 LCM detection sensitivity (`lcm_detection_sensitivity`)

The parameter `lcm_detection_sensitivity` controls the detection sensitivity of the lens contamination monitor:

mode	description
`disabled`	lens contamination is not detected (default)
`low`	only severe lens contamination is detected
`medium`	moderate lens contamination is detected
`high`	slight lens contamination is detected

The sensitivity should be adjusted according to the environmental conditions and functional requirements of a specific application.

5.2.3 LCM detection periodic (lcm_detection_period)

The parameter lcm_detection_period controls the reaction time of lens contamination monitor, i.e. how fast it detects and signals a contamination. The reaction time is specified in milliseconds [ms]. The default value is 5000 (5 s).

Please note:
A low value for lcm_detection_period means a faster reaction to contamination but might also cause spurious contamination warnings due to temporary disturbances like floating dust.

5.2.4 LCM sector configuration (lcm_sector_enable)

The parameter lcm_sector_enable allows to limit the lens contamination monitoring to specific sectors of the sensors 360° field of view. Please refer to section 5.1 for a definition of LCM sectors. The sector evaluation is defined by an array of flags (one flag per sector), which can be set to on or off. Per default all sectors are enable, i.e. the LCM operates on the whole field of view.

Example

Disable lens contamination detector for the backside of the sensor (sectors 0-2 and 9-11):

Query: http://<sensor IP>/cmd/set_parameter?lcm_sector_enable=off,off,off,on,on,on,on,on,on,off,off,off

Reply:
```
{
    "error_code":0,
    "error_text":"success"
    }
```

5.2.5 LCM status flags (lcm_sector_warn_flags, lcm_sector_error_flags)

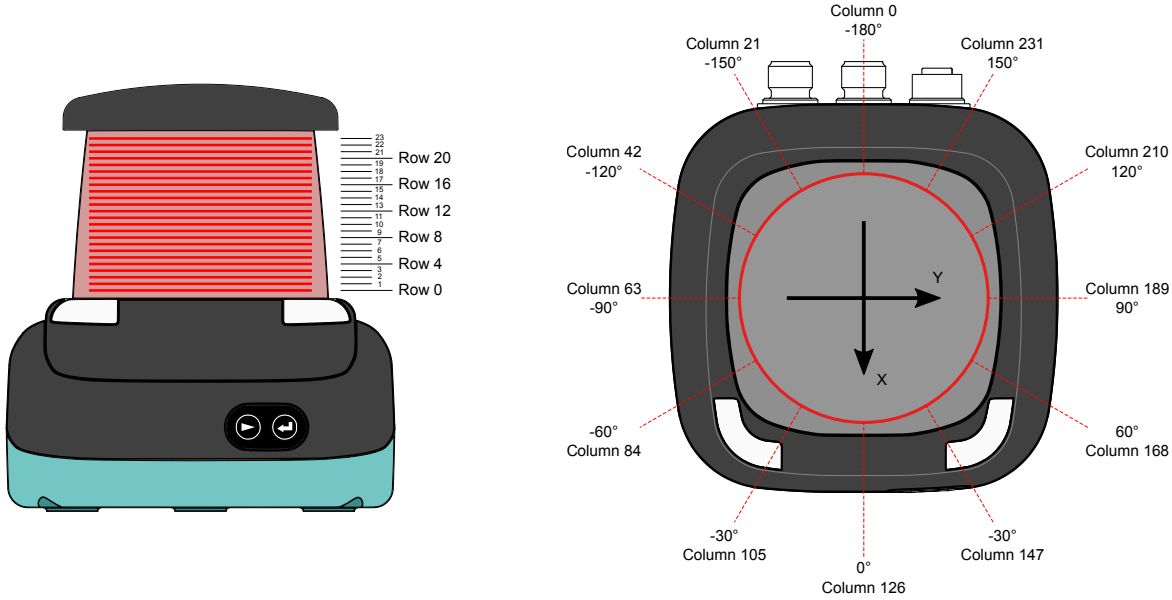
The read-only parameters lcm_sector_warn_flags and lcm_sector_error_flags provide detailed information on the current lens contamination state for each LCM sector. Contamination is reported with two levels of severity – either as *warning* or as *error*. If at least on sector reports a contamination warning in lcm_sector_warn_flags, also the system status flag lens_contamination_warning is set (see section 2.8.2). Accordingly, if at least on sector reporting a contamination error in lcm_sector_error_flags, also the system status flag lens_contamination_error is set (see section 2.8.2).

6 Working with the HMI LED display

This chapter features a detailed description of the HMI LED display of the R2000 and instructions on using it for displaying application-specific information.

6.1 Technical overview

The R2000 device family features a multi-function HMI LED display. The display is created by an array of 24 LEDs that are mounted on one edge of the rotating sensor head, making use of the so-called *Persistence of Vision (POV)* characteristic of the human eye. By rapid updates of the LED array during the rotation of the sensor head a virtual raster graphic with 24 rows and 252 columns is created. For best readability scan rates above 35 Hz are recommended (see parameter `scan_frequency` in section 2.6).



(a) Rows of LED display

(b) Columns of LED display

Figure 6.1: HMI display coordinate system

Figure 6.1 shows the positioning of the LED display with respect to the sensor coordinate system (as defined in section 3.1.1). Information on the LED display is usually shown centered at 0 ° on the front of the sensor (column 126). The display area starts at column 0 on the back of the sensor at –180 °. The columns are arranged in mathematical positive order up to column 251 at approximately +178.6 °. The transition from column 251 to column 0 on the sensors back is seamless, enabling a usable field of view of full 360 °. When rendering the display content the sensor firmware takes the sensors direction of rotation into account. The client application does not need to consider the current value of parameter `scan_direction` (section 2.6) when preparing content for the HMI LED display.

Figure 6.2 shows a two-dimensional representation of the display pixel layout (2D bitmap). This is a simplified view since the curvature and 360 ° wrap-around nature of the real display are not shown. The physical display area covers approximately 48 mm in height and 170 mm in width. This results into a horizontal pixel density (resolution) of about 38 dpi and a vertical pixel density of about 12 dpi. The three times higher horizontal resolution implies that three horizontal pixels need to be combined in order to show a single square 'pixel' on the HMI LED display.

Application developers can utilize the LED display for showing custom text messages or custom bitmap images. The following sections describe these use-cases in detail.

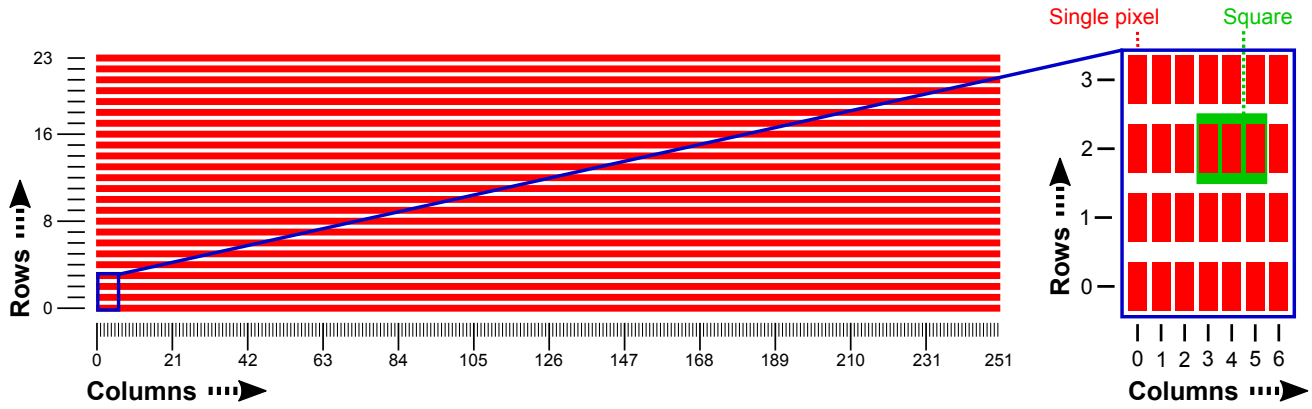


Figure 6.2: HMI display pixel layout (252 x 24 pixels)

6.2 Displaying custom text messages

Using short text messages is the easiest way of showing custom information on the R2000 HMI LED display. The sensor supports two different modes for displaying text: A mode for static text messages that are preserved even after a power-cycle and a mode for rather volatile text messages that are updated by the client application more frequently.

6.2.1 Overview

The text display features two independent lines of text – one in the upper half of the display and one in the lower half. Text is displayed with a fixed width font of 8 pixel height (using 8 of 24 display rows). Each text line is limited to a length of max. 30 characters and shown horizontally centered at the front of the sensor. The display supports a selection of typical special characters (e.g. umlauts) out of the UTF-8 code range. Unsupported special characters are replaced by a question mark character (`'?'`).

6.2.2 Static text messages (static text)

The display mode `static_text` allows client applications to display up to two lines of static text on the HMI LED display. The application text lines are stored in the parameters `hmi_static_text_1` and `hmi_static_text_2` within non-volatile memory, i.e. the content is not lost during a power-cycle. The display mode is especially suited for displaying rarely updated information, e.g. an identification string for the sensor. The parameters `hmi_static_text_1` and `hmi_static_text_2` are reset on request only (e.g. when loading the factory defaults). The default content reads `'Pepperl+Fuchs'` and `'R2000'`.

Steps for displaying static text messages are:

1. Write text for the upper display line to `hmi_static_text_1` using `set_parameter`.
2. Write text for the lower display line to `hmi_static_text_2` using `set_parameter`.
3. Enable the static text display by setting parameter `hmi_display_mode` to `static_text` using `set_parameter`.

Please note:
Since each write access to `hmi_display_mode`, `hmi_static_text_1` and `hmi_static_text_2` triggers a write access to non-volatile memory with a limited number of write-cycles, it is strongly recommended to write these parameters only if necessary.

Command example

Query for selecting the display mode `static_text`:
`http://<IP address>/cmd/set_parameter?hmi_display_mode=static_text`

Query for setting the displayed text to *'Hello World!'*:
`http://<IP address>/cmd/set_parameter?hmi_static_text_1=Hello&hmi_static_text_2=World!`

6.2.3 Dynamic text messages (application text)

The display mode `application_text` enables client applications to display up to two lines of custom text on the HMI LED display. The application text lines are stored in the parameters `hmi_application_text_1` and `hmi_application_text_2` using volatile memory only, i.e. the content is lost during reset. Therefore, this display mode is especially suited for displaying frequently updated information, e.g. status information of the client application processing the sensors scan data. Per default (e.g. after power-on) `hmi_application_text_1` and `hmi_application_text_2` are empty.

Steps for displaying application text messages are:

1. Enable the application text display by setting parameter `hmi_display_mode` to `application_text` using `set_parameter`.
2. Write text for the upper display line to `hmi_application_text_1` using `set_parameter`.
3. Write text for the lower display line to `hmi_application_text_2` using `set_parameter`.

Please note:
Since each write access to `hmi_display_mode` triggers a write access to non-volatile memory with a limited number of write-cycles, it is strongly recommended to write it only to select the display mode. For subsequent content updates write to `hmi_application_text_1` and `hmi_application_text_2` only.

Command example

Query for selecting the display mode `application_text`:
`http://<IP address>/cmd/set_parameter?hmi_display_mode=application_text`

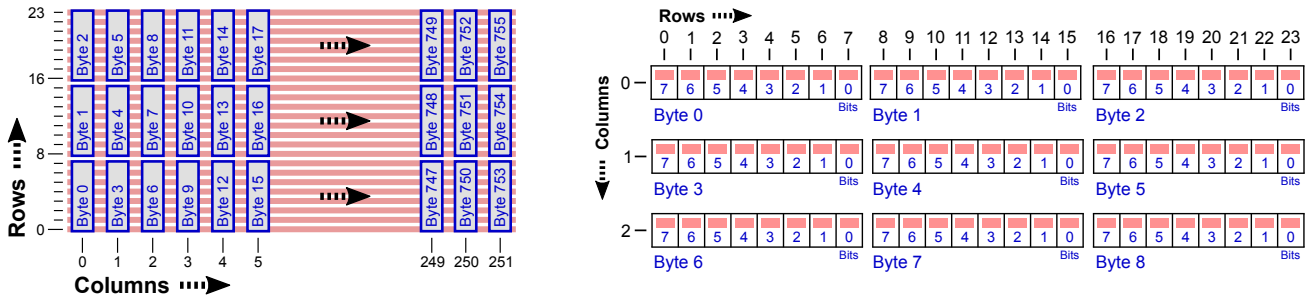
Query for setting the displayed text to *'My status message'*:
`http://<IP address>/cmd/set_parameter?hmi_application_text_1=My status&hmi_application_text_2=message`

6.3 Displaying custom bitmaps

Alternatively to simple text messages (as described in section 6.2) the sensor allows client application to display custom bitmaps on the HMI LED display. This gives maximum flexibility regarding the displayed content, but requires more complex preparations by the client firmware. Similar to the display modes for text messages the sensor provides a mode for static bitmaps (logos) that are preserved even after a power-cycle and a mode for rather dynamic graphics that are updated by the client application more frequently.

6.3.1 Overview

Section 6.1 did already cover various details on the technical implementation of the LED display. This section concentrates on how display pixels are mapped into a binary frame buffer and how this data is transferred to the sensor.



(a) Mapping of bitmap to bytes

(b) Mapping of pixels to bytes

Figure 6.3: Byte mapping of HMI display pixels

Figure 6.3 shows the mapping of display pixels to a binary frame buffer. Each pixel of the LED display (see fig. 6.2) is represented by a single bit within this frame buffer. The 6048 pixels ($24 * 252 = 6048$) of a complete display image map into a frame buffer of 756 B ($6048/8 = 756$). The mapping starts at row 0 of column 0 in the lower left corner of the 2D display image: Pixel $(0;0)$ maps into byte 0 bit 7, pixel $(0;1)$ maps into byte 0 bit 6 and so on. The last pixel $(0;23)$ of the first column maps into byte 3 bit 0. The first pixel $(1;0)$ of the second column maps into byte 4 bit 7. This mapping scheme

continues column by column until the very last pixel $(251; 23)$ in the upper right corner of the 2D display image fills bit 0 of byte 755.

Bitmap access to the HMI LED display always updates the complete display frame buffer. The necessary data is stored in binary form within the display parameters `hmi_static_logo` and `hmi_application_bitmap` – see subsequent sections for details. Writing to these parameters with `set_parameter` requires the binary content to be encoded as `base64url` string for the command URI (see sections 1.2.1 and 1.2.2). Reading the parameters returns a `base64` encoded string as part of the *JSON* encoded (see section 1.2.3). Please refer to section 2.1 for a more detailed description of binary parameter types.

6.3.2 Static bitmaps

The parameter `hmi_static_logo` allows to customize the graphic shown by the HMI display mode `static_logo`. The static bitmap is stored into non-volatile memory, i.e. the content is not lost during a power-cycle. Therefore, this display mode is especially suited for displaying rarely updated information, e.g. a custom company logo. The parameter `hmi_static_logo` is reset on request only (e.g. when loading the factory defaults). The default value shows a Pepperl+Fuchs logo.

Steps for customizing the static logo are:

1. Write a custom bitmap to the parameter `hmi_static_logo` using the command `set_parameter`.
2. Display the bitmap by setting parameter `hmi_display_mode` to the value `static_logo` using `set_parameter`.

Please note:
Since each write access to `hmi_display_mode` and `hmi_static_logo` triggers a write access to non-volatile memory with a limited number of write-cycles, it is strongly recommended to write these parameters only if necessary.

6.3.3 Application bitmaps

The display mode `application_bitmap` enables client applications to display a custom bitmap image on the HMI LED display. The bitmap is stored in the parameter `hmi_application_bitmap` using volatile memory only, i.e. the content is lost during reset. Therefore, this display mode is especially suited for displaying frequently updated information, e.g. status graphics of the client application processing the sensors scan data. Per default (e.g. after power-on) `hmi_application_bitmap` is empty.

Steps for displaying an application bitmap are:

1. Write an application bitmap to the parameter `hmi_application_bitmap` using the command `set_parameter`.
2. Display the application bitmap by setting parameter `hmi_display_mode` to the value `application_bitmap` using the command `set_parameter`.

Please note:
Since each write access to `hmi_display_mode` triggers a write access to non-volatile memory with a limited number of write-cycles, it is strongly recommended to write it only to select the display mode. For subsequent content updates write to `hmi_application_bitmap` only.

Please note:
The interface to the HMI LED display is currently not designed for real-time updates. It is recommended to update the application bitmap not more often than once per second (update rate 1 Hz). In case of faster updates the behavior of the display is undefined.

6.3.4 Converting graphics for the HMI display

To convert an existing graphic to the HMI LED display the following steps are recommended:

1. Stretch the graphic by factor 3 in horizontal direction to compensate for the asymmetrical display resolution.
2. Trim the image to an aspect ratio of $2 : 21$ (vertical:horizontal). Keep in mind that the image is shown on a 360 ° surface, so repeating the image for different view angles might be a good idea. A tried and trusted approach is to trim the image to a $2 : 7$ aspect ration and then repeat this image three times in horizontal direction.
3. Down-scale the image to a resolution of 24 x 252 pixels.
4. Reduce the image to a black-and-white graphics with only two colors.

5. If necessary, manually optimize the resulting low-resolution graphic by removing artifacts from the conversion process.
6. Save the image to a common graphics file format with support for monochrome images (2 bits per pixel).
7. Convert the image file to binary format using a common image conversion tool like Image Magick. The following command uses the tool `convert` from *Image Magick* to convert a bitmap into the correct raw binary data:

   ```
   > convert input.bmp -rotate 90 -negate GRAY:output.bin
   ```

   The resulting binary file (`'output.bin'` in the above example) should have a size of exactly 756 B.
8. Finally, store the binary data in either `hmi_static_logo` or `hmi_application_bitmap`. The data needs to be encoded as `base64url` string [8] with a length of exactly 1008 B.

It is highly recommended to use vector graphics as source for creating bitmaps for the HMI LED display. This way many conversion artefacts can be avoided resulting in higher image quality.

7 Switching input/output channels I/Qn

This chapter describes the configuration and usage of the sensors switching input/output channels.

7.1 Introduction

Many R2000 devices are equipped with I/Q channels that can be used as either digital input or digital output. The presence of an I/Q channels is indicated by the system feature flags (see section 2.4).

7.2 Commands for I/Q channel parametrization

7.2.1 `list_iq_parameters` – list I/Q parameters

The command `list_iq_parameters` is similar to the generic `list_parameters` command (see section 2.2.1) but returns all parameters related to the switching input/output channels I/Qn.

Example

Query: `http://<sensor IP address>/cmd/list_iq_parameters`

Reply:
```
{
    "iq_parameters":[
    "iq_global_enable",
    "iq_input",
    "iq_output",
    "iq_overload",
    "iq1_mode",
    "iq2_mode",
    "iq3_mode",
    "iq4_mode",
    "iq1_polarity",
    "iq2_polarity",
    "iq3_polarity",
    "iq4_polarity",
    "iq1_off_delay",
    "iq2_off_delay",
    "iq3_off_delay",
    "iq4_off_delay"
    ],
    "error_code":0,
    "error_text":"success"
    }
```

7.2.2 `get_iq_parameter` – read a I/Q parameter

The command `get_iq_parameter` is similar to the generic `get_parameter` command (see section 2.2.2) but operates on parameters related to the switching input/output channels I/Qn. The command returns the current value of one or more parameters:

```
http://<sensor IP address>/cmd/get_iq_parameter?list=<param1>;<param2>
```

Command arguments

- `list` – semicolon separated list of parameter names (optional)

If the argument `list` is not specified the command will return the current value of all available parameters.

Example

Query: `http://<sensor IP address>/cmd/get_iq_parameter?list=iq_input;iq1_mode`

Reply:
```
{
"iq_input":1,
"iq1_mode":"input_high_z",
"error_code":0,
"error_text":"success"
}
```

7.2.3 `set_iq_parameter` – change an I/Q parameter

The command `set_iq_parameter` is similar to the generic `set_parameter` command (see section 2.2.3) but operates on parameters related to the switching input/output channels I/Qn. Using the command `set_iq_parameter` the value of any write-accessible I/Q parameter can be modified:

`http://<sensor IP address>/cmd/set_iq_parameter?<param1>=<value>&<param2>=<value>`

Command arguments

- `<param1> = <value>` – new <value> for parameter <param1>
- `<param2> = <value>` – new <value> for parameter <param2>
- ...

Please note:
The command `set_iq_parameter` returns an error message, if any parameter specified as command argument is unknown or a read-only parameter. The return values `error_code` and `error_text` have appropriate values in this case (see section 1.2.6).

Please note:
All I/Q channel configuration parameters are non-persistent, i.e. they return to their default values on every power cycle. Therefore, user applications need to configure these settings on every start.

Example

Query: `http://<sensor IP address>/cmd/set_iq_parameter?iq1_mode=output_push_pull`

Reply:
```
{
"error_code":0,
"error_text":"success"
}
```

7.3 Parameters for I/Q channel configuration

This section provides information on all available parameters for configuring the switching input/output channels of the device. This applies to both, electrical and logical configuration.

Please note:
All I/Q channel configuration parameters are non-persistent, i.e. they return to their default values on every power cycle. Therefore, user applications need to configure these settings on every start.

7.3.1 Electrical configuration of I/Q channels

This section describes the parameters for the *electrical configuration* of the switching input/output channels I/Qn. For each channel the following parameters are defined (the list applies to channel I/Q1):

parameter name	type	description	access	default
`iq_global_enable`	`bool`	I/Q global enable switch for all I/Q channels (`on`, `off`)	vRW	`off`
`iq<n>_source`	`enum`	I/Q signal source (`iq_output`, `timesync`)	vRW	`iq_output`
`iq<n>_mode`	`enum`	I/Q channel operation mode (`disabled`, `input_high_z`, `output_push_pull` `output_n_switching`, `output_p_switching`)	vRW	`disabled`
`iq<n>_polarity`	`enum`	I/Q channel polarity (`active_high`, `active_low`)	vRW	`active_high`
`iq<n>_off_delay`	`uint`	I/Q channel pulse extension (ms)	vRW	0 ms

Please note:
The channel-specific settings for polarity, operation mode and off-delay are defined for each I/Q channel present in the device, e.g. `iq1_polarity` for channel I/Q1, `iq2_mode` for channel I/Q2, ... Any I/Q channel not present in a specific device (as indicated by the system feature flags – see section 2.4) has no associated parameter (e.g. access to `iq8_mode` fails if I/Q8 is not present).

I/Q channels global enable (`iq_global_enable`)

The parameter `iq_global_enable` acts as global enable switch for all I/Q channels. It is non-persistent and defaults to `off` at system startup. While set to `off` all I/Q channels are disabled – regardless of their individual configuration. This way the user can set up and change I/Q configurations while avoiding switching artefacts at the output pins.

I/Q channel signal source (`iq<n>_source`)

The parameter `iq<n>_source` selects which signal source takes control of the *output* state of an I/Q channel. The following signal sources are available:

source	description
`iq_output`	output state is defined by user variable `iq_output` (see section 7.3.2)
`timesync`	output signal for time synchronization (see section 7.3.3)

The signal source can be configured for each I/Q channel separately (`iq1_source` for channel I/Q1, `iq2_source` for channel I/Q2, ...).

Please note:
The available signal source might be depending on the specific I/Q channel. On current R2000 devices the following options are available:

I/Q channel	available sources
`iq1_source`	`iq_output`
`iq2_source`	`iq_output`, `timesync`
`iq3_source`	`iq_output`
`iq4_source`	`iq_output`

I/Q channel operation mode (iq<n>_mode)

For each I/Q channel an operation mode can be configured using the parameter iq<n>_mode (iq1_mode for channel I/Q1, iq2_mode for channel I/Q2, ...). The following operation modes are available:

operation mode	description
disabled	no function (high impedance)
input_high_z	input with high impedance
output_push_pull	output with P and N channel
output_p_switching	output with P channel only
output_n_switching	output with N channel only

I/Q channel polarity (iq<n>_polarity)

The polarity of each I/Q channel can be individually configured using the parameter iq<n>_polarity. It defines the translation between the logic level and the electric level of the I/Q channel for all operating modes (both input and output).

The following table gives an overview on this logic level to electric level translation for all output modes:

logic level	polarity	electric level (output_push_pull)	electric level (output_p_switching)	electric level (output_n_switching)
0 (inactive)	active_high	LOW	HI-Z	LOW
1 (active)		HIGH	HIGH	HI-Z
0 (inactive)	active_low	HIGH	HIGH	HI-Z
1 (active)		LOW	HI-Z	LOW

The following table gives an overview on the electric level to logic level translation for all input modes:

electric level	polarity	logic level (input_high_z)
LOW	active_high	0 (inactive)
HIGH		1 (active)
HI-Z		? (application-dependent)
LOW	active_low	1 (active)
HIGH		0 (inactive)
HI-Z		? (application-dependent)

I/Q channel pulse extension (iq<n>_off_delay)

Each switching output channel provides a programmable pulse extension which guarantees a minimum duration for an active output signal. More precisely, in case of an *off* transition (change from active to inactive state) of a bit in iq_output, the active state of the corresponding output channel is extended for the configured delay time T_{off}. If the bit in iq_output is set back to 1 before T_{off} expires, the intermediate 0 state will be suppressed at the output pin.

Please note:
An *on* transition (inactive to active state change) does not trigger this functionality. This change is applied to the output pin immediately.

Furthermore, the I/Q pulse extension functionality (as configured by iq<n>_off_delay) is affected by the following special cases:

- The pulse extension is *not* applied, if the I/Q channels are disabled by means of iq_global_enable (off state).
- All active pulse extensions are aborted, when iq_global_enable is switched off (transition from on to off) – globally disabling all I/Q channels.
- Changes to iq_output do *not* trigger a pulse extension for any I/Q channel that is operating in 'input' or 'disabled' mode (see iq<n>_mode in section 7.3.1).
- If the electrical configuration of an I/Q channel (section 7.3.1) is changed from mode 'output' to either 'input' or 'disabled' then this change takes effect immediately – aborting any currently active pulse extension.

7.3.2 Logical state of I/Q channels

The following parameters define the *logical state* (active or inactive) of the switching input/output channels I/Qn. These settings are independent of the electrical configuration of each channel (see section 7.3.1).

parameter name	type	description	access	default
`iq_input`	`bitfield`	bit field with state of switching inputs (0 – inactive, 1 – active) Bit 0 – I/Q1, Bit 1 – I/Q2, Bit 2 – I/Q3, Bit 3 – I/Q4, . . .	RO	0
`iq_output`	`bitfield`	bit field with logic state of switching outputs (0 – inactive, 1 – active) Bit 0 – I/Q1, Bit 1 – I/Q2, Bit 2 – I/Q3, Bit 3 – I/Q4, . . .	vRW	0
`iq_overload`	`bitfield`	bit field with status of switching outputs (0 – normal, 1 – overload) Bit 0 – I/Q1, Bit 1 – I/Q2, Bit 2 – I/Q3, Bit 3 – I/Q4, . . .	RO	0

I/Q input state (`iq_input`)

The current state of all digital switching I/Q channels can be read using the I/Q status variable `iq_input`. Each bit represents an individual I/Q channel (up to 32 channels). This works for I/Q channel operating as input or output (see section 7.3.1). Disabled and non-present I/Q channels are always read as 0 in the corresponding bit.

Please note:
`iq_input` returns 0 for all channels if I/Q channels are globally disabled via `iq_global_enable` (see section 7.3.1).

I/Q output state (`iq_output`)

The variable `iq_output` controls the logic state of all I/Q channels that are configured to an output operation mode (`iq<n>_mode`) and to the `iq_output` signal source (`iq<n>_source`). If a different signal source is selected or if an I/Q channel is configured as input, disabled or not present at all, the corresponding bit in `iq_output` is ignored.

I/Q overload state (`iq_overload`)

The variable `iq_overload` signals an overload condition at any I/Q channel configured to an output operation mode. If an I/Q channel is configured as input, disabled or not present the corresponding bit is always read as 0.

7.3.3 I/Q output signal for raw timestamp synchronization

To enable a low-level synchronization of the sensors raw system time (see section 3.1.5) with the system time of an external client, the sensor can generate a periodic synchronization pulse at selected I/Q pins. The synchronization signal can be configured using the following parameters:

parameter name	type	description	access	default
`iq_timesync_interval`	`uint`	Interval for generating a `timesync` pulse (ms)	RW	4000 ms

The synchronization pulse will be generated each time the raw system timestamp `system_time_raw` reaches an integer multiple of the configured interval `iq_timesync_interval`.

Please note:
Although the parameter `iq_timesync_interval` is specified with the unit 1 ms it currently accepts only values with a resolution of 1 s, i.e. integral multiples of 1000 ms.

To enable the synchronization signal generation on an I/Q channel, the following settings need to be applied:

1. Set I/Q mode to an output mode using the parameter `iq<n>_mode` (see section 7.3.1).
2. Set I/Q polarity to an appropriate value using the parameter `iq<n>_polarity` (see section 7.3.1).
3. Set I/Q pulse extension to an appropriate value using the parameter `iq<n>_off_delay` (see section 7.3.1).
4. Set I/Q source to `timesync` using the parameter `iq<n>_source` (see section 7.3.1).
5. Select the period of the `timesync` signal using the parameter `iq_timesync_interval` (see above).
6. Enable all I/Q channels using the parameter `iq_global_enable` (see section 7.3.1).

Please note:
If `iq<n>_off_delay` is set to 0 ms the pulse length of the synchronization signal is implementation-specific. It is highly recommended to configure `iq<n>_off_delay` to a non-zero value.

Please note:
The maximum pulse length of the synchronization signal is 500 ms. Larger values of `iq<n>_off_delay` are internally capped to 500 ms.

Please note:
The I/Q timestamp synchronization signal is currently available on I/Q2 only.

Please note:
All I/Q channel configuration parameters are non-persistent, i.e. they return to their default values on every power cycle. Therefore, user applications need to configure these settings on every start.

8 Advanced topics

This chapter covers various advanced topics about using R2000 devices in more complex applications.

8.1 Device discovery using SSDP

The R2000 provides support for the *Simple Service Discovery Protocol (SSDP)* [14] in order to discover any R2000 devices and their associated IP address within the Ethernet network. SSDP uses UDP multicast messages to query SSDP aware devices.

In order to discover all R2000 devices, the following steps need to be performed:

1. Send a SSDP search request.
2. Process SSDP replies from devices.
3. Read a SSDP device description from each device for additional information.

The following sections describe each step in detail.

8.1.1 SSDP search request

The first step of the SSDP device discovery is to issue a search request on the local network. For this purpose an UDP listener needs to be opened on the local UDP port 1900. Then an UDP datagram with the following content needs to be sent to the UDP multicast address 239.255.255.250 at port 1900:

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
ST: urn:pepperl-fuchs-com:device:R2000:1
MAN: "ssdp:discover"
MX: 1
```

The specified URN addresses R2000 devices only. Other SSDP aware devices on the network will ignore this request.

Please note:
On a client PC with multiple network adapters, the SSDP search request needs to be performed on each network adapter.

8.1.2 SSDP device reply

The second step of the discovery procedure requires the client application to wait for replies to the above search request using the created UDP listener. Each R2000 device on the local network will answer the search request with a message similar to this example:

```
HTTP/1.1 200 OK
LOCATION: http://10.0.10.9/ssdp.xml
SERVER: pfda/1.0 UPnP/1.0 R2000/1.0
CACHE-CONTROL: max-age=1800
EXT:
ST: urn:pepperl-fuchs-com:device:R2000:1
USN: uuid:7df9a5ed-07f6-45e1-ac55-333340704340::urn:pepperl-fuchs-com:device:R2000:1
```

This reply contains two important pieces of information:

- The line `LOCATION:` contains the IP address of the device within an URL pointing to a more detailed SSDP device description (see next section).
- The line `USN:` contains an unique identifier (uuid) for this specific device. This uuid allows to identify this R2000 device even if its IP address changes.

8.1.3 SSDP device description

The final step of the SSDP discovery procedure is to obtain the XML based device description. This step can be skipped, if no detailed information on the discovered devices are needed. R2000 devices provide a `ssdp.xml` file at the URL from the `LOCATION` field of the SSDP device reply (see previous section):

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <device>
    <deviceType>urn:pepperl-fuchs-com:device:R2000:1</deviceType>
    <friendlyName>OMD10M-R2000-B23-V1V1D (#40000007343704)</friendlyName>
    <manufacturer>Pepperl+Fuchs</manufacturer>
    <manufacturerURL>http://www.pepperl-fuchs.com</manufacturerURL>
    <modelDescription>2D Laser Scanner</modelDescription>
    <modelName>OMD10M-R2000-B23-V1V1D</modelName>
    <modelNumber>232934</modelNumber>
    <serialNumber>40000007343704</serialNumber>
    <UDN>uuid:7df9a5ed-07f6-45e1-ac55-333340704340</UDN>
    <serviceList>
      <service>
        <serviceType>urn:pepperl-fuchs-com:service:none:1</serviceType>
        <serviceId>urn:pepperl-fuchs-com:serviceId:none</serviceId>
        <controlURL>/cmd/</controlURL>
        <eventSubURL></eventSubURL>
        <SCPDURL>/service.xml</SCPDURL>
      </service>
    </serviceList>
    <iconList>
      <icon>
        <id>0</id>
        <mimetype>image/png</mimetype>
        <width>48</width>
        <height>48</height>
        <depth>24</depth>
        <url>/device.png</url>
      </icon>
    </iconList>
    <X_pfsdpVersionMajor>1</X_pfsdpVersionMajor>
    <X_pfsdpVersionMinor>2</X_pfsdpVersionMinor>
    <X_pfsdpDeviceFamily>1</X_pfsdpDeviceFamily>
    <X_dtmAccessPath>/dtm/</X_dtmAccessPath>
  </device>
</root>
```

The standard SSDP XML device description contains already various useful fields:

- `manufacturer` – vendor name of the device (see parameter `vendor` in section 2.3)
- `modelName` – product name of the device (see parameter `product` in section 2.3)
- `modelNumber` – part number of the device (see parameter `part` in section 2.3)
- `serialNumber` – serial number of the device (see parameter `serial` in section 2.3)

R2000 devices additionally provide the following non-standard items with PFSDP specific information:

- `X_pfsdpVersionMajor` – major PFSDP protocol revision (see `version_major` in section 1.2.7)
- `X_pfsdpVersionMinor` – minor PFSDP protocol revision (see `version_minor` in section 1.2.7)
- `X_pfsdpDeviceFamily` – PFSDP device family (see `device_family` in section 2.3)

A Troubleshooting the Ethernet communication

This chapter contains some basic suggestions for troubleshooting issues concerning the R2000 Ethernet communication.

A.1 Checking the Ethernet setup

In case of communication problems, first ensure a working Ethernet connection between PC and sensor. Please consider the following steps:

- **Sensor IP configuration**
  Check the current IP configuration of the sensor in the HMI menu ”Ethernet Info” (see user manual). If necessary, change the configuration in the ”Ethernet Setup” menu and reboot the device to apply the changes. Now verify the IP configuration in the ”Ethernet Info” menu.
- **Ethernet connection**
  Use the network utility *ping* to verify the network connection between sensor and PC. The sensor will reply to all ping requests it receives. If *ping* does not receive any replies, re-check the IP configuration of your client PC and the sensor. Make sure the IP addresses of both devices are within the same subnet.
- **Electrical connection**
  In case of connectivity problems, check the link status and link speed of the sensor, the client PC and any network infrastructure device (router, switch, etc.) in-between to rule out electric connection issues. For maximum reliability, try to use a direct cable-based Ethernet connection between sensor and PC. The sensor supports Auto-MDIX – a cross-over Ethernet cable is not required.

A.2 Debugging using a web browser

If basic network connectivity has been established, verify that the HTTP command interface is operational with a standard web browser. Please consider these steps:

- **Proxy settings**
  Make sure that no proxy is used when accessing the sensor. In the browser settings, either completely disable any proxy or add a proxy exception for the sensor IP address.
- **HTTP access**
  Try to access the sensor via the following URL:

  ```
  http://<sensor IP address>/cmd/protocol_info
  ```

  This command should return some basic protocol information (see section 1.2.7). If this is not the case, re-check your proxy settings and Ethernet setup (see above).
- **HTTP commands**
  You can test the syntax and effect of any HTTP command used in your application software just by sending the command from a web browser. The web browser will display the response received from the sensor – making it easy to review any potential error messages. Furthermore, after changing sensor settings with the `set_parameter` command (see section 2.2.3), it might be helpful to read back all parameters using the command `get_parameter` (see section 2.2.2).

A.3 Debugging using Wireshark

For complex communication issues it is highly recommended to use the free network traffic analysis tool *Wireshark* [13] to sniff and record the Ethernet communication between the client software and the R2000 sensor.

For example, this can be very helpful for:

- Checking the content of HTTP messages and the corresponding replies
- Checking order and time behavior of HTTP commands
- Checking time behavior of scan data output (TCP or UDP)

In case you contact your sensor support representative about a specific communication issue, it is highly recommended to have a Wireshark log file (`.pcap`) at hand for examination by the technical support organisation.

B Protocol version history

B.1 Protocol version 1.04

Minor protocol extension (backward-compatible to protocol versions 1.00, 1.01, 1.02 and 1.03)

Notable extensions:

- Section 3.3.6: Added new parameter `packet_crc` for scan data connections.

Note: This protocol version is implemented by R2000 firmware v1.60 or newer.

B.2 Protocol version 1.03

Major protocol extensions (backward-compatible to protocol versions 1.00, 1.01 and 1.02)

Significant extensions:

- Chapter 4: Added various options for scan data filtering.
- Chapter 5: Added various options for lens contamination monitoring.

Notable extensions:

- Section 2.6.5: Added scan resolutions 2520 and 3150 to R2000 UHD and HD devices (Table 2.1).
- Section 3.3.6: Added new parameter `skip_scans` for scan data connections.
- Section 3.4.2: Added fields `iq_timestamp_raw` and `iq_timestamp_sync` in scan data header.

Notable changes:

- Section 2.4.1: Removed irrelevant sensor-capability parameters `max_scan_sectors` and `max_data_regions`.
- Section 2.6.2: Renamed value `transmitter_off` to `emitter_off`.
- Section 3.2.2: Removed deprecated parameter `deprecated_handle_generation`.

Note: This protocol version is implemented by R2000 firmware v1.50 or newer.

B.3 Protocol version 1.02

Minor protocol extensions (backward-compatible to protocol versions 1.00 and 1.01)

Notable extensions:

- Section 2.4: Added new informational parameter `emitter_type`.
- Section 2.6.5: Added scan resolutions 1680, 2100 and 2800 to R2000 UHD and HD devices (Table 2.1).

Note: This protocol version is implemented by R2000 firmware v1.21 or newer.

B.4 Protocol version 1.01

Protocol enhancements (backward-compatible to protocol version v1.00)

Significant extensions:

- Section 2.7: Added access to bitmap shown by display mode `static_logo`
- Section 2.7: Added display mode `static_text` to show static custom text
- Section 2.7: Added display mode `application_bitmap` to show a dynamic custom bitmap
- Section 2.7: Added display mode `application_text` to show a dynamic custom text
- Section 3.2.4: Added option `max_num_points_scan` to limit number of points per scan for scan data output
- Section 3.3.9: Added mechanism for TCP in-line watchdog feeds
- Chapter 7: Added commands and parameters for switching input/output channels

Notable extensions:

- Section 2.2.6: Added new command `factory_reset` to perform a complete factory reset.
- Section 2.3: Added device family for HD devices (OMDxxx-R2000-HD)
- Section 2.4: Added capability values `scan_frequency_min` and `scan_frequency_max`
- Section 2.4: Added capability values `sampling_rate_min` and `sampling_rate_max`
- Section 2.6.2: Added parameter `operating_mode` to control mode of operation (e.g. disable emitter)
- Section 2.7: Added new parameter `hmi_parameter_lock` for setting the HMI menu to read-only.
- Section 3.4.2: Redefined field `output_status` to `iq_input` (all scan data packet headers)
- Section 3.4.2: Redefined field `field_status` to `iq_overload` (all scan data packet headers)

Note: This protocol version is implemented by R2000 firmware v1.20 or newer.

B.5 Protocol version 1.00

First public release.

Note: This protocol version is implemented by R2000 firmware v1.00 or newer.

C Document change history

C.1 Release 2024-05 (protocol version 1.04)

Minor document update:

- Replaced term *scan frequency* with term *scan rate* in whole document.
- Section 2.1: Added description of `array` type
- Section 2.5.1: Fixed incorrect default values for `ip_mode`, `ip_address_current` and `subnet_mask_current`
- Sections 3.1.5 and 3.4: Removed references to synchronized timestamps (not available on R2000 devices)
- Section 5.2: Added detailed description for LCM configuration parameters
- Various minor textual and cosmetic updates

C.2 Release 2020-05 (protocol version 1.04)

Document update for OMD SD device release:

- Section 2.3.2: Added device family definition for R2000 OMD SD devices
- Section 2.4.2: Fixed name of I/Q feature flags (`input_output_qN` instead of `input_output_iqN`)
- Section 2.5.1: Fixed documentation of subnet mask default (`255.0.0.0` instead of `255.255.255.0`)
- Section 2.6.5: Updated table 2.1 for UHD devices (50 Hz maximum scan frequency)
- Section 2.6.5: Added OMD SD devices to table 2.1
- Section 3.4.3: Fixed documentation for accumulative status flags
  (accumulation does not cover the accumulated flags themselves)
- Various minor textual and cosmetic updates

C.3 Release 2019-07 (protocol version 1.04)

Document update for LCM feature release:

- Section 2.8.2: Added documentation of LCM status flags
- Section 3.4.3: Added documentation of LCM status flags
- Chapter 5: Publish chapter on lens contamination monitor (LCM)
- Various minor textual and cosmetic updates

C.4 Release 2018-10 (protocol version 1.04)

Document update for protocol version 1.04:

- Section 3.2.4: Added description of option `packet_crc` for an additional scan packet checksum.
- Various minor textual and cosmetic updates

C.5 Release 2017-11 (protocol version 1.03)

Document update for protocol version 1.03:

- Section 2.4.1: Clarified description of `radial_resolution` and `angular_resolution`.
- Section 2.4.1: Corrected type of `scan_frequency_min` and `scan_frequency_max` (`double` instead of `int`)
- Section 2.6.5: Updated table 2.1 to reflect new maximum scan frequency of 100 Hz.
- Section 2.8.2: Updated description of system status flags.
- Section 3.2.4: Added description of option `skip_scans` to reduce the scan *output* frequency.
- Sections 3.4 to 3.6: Updated and clarified details of scan data transmission.
- Section 3.4.1: Documented potential padding at the end of a scan data packet (`payload_padding`).
- Section 3.4.3: Added LCM status flags and updated description of flags.
- Chapter 4: Added chapter about integrated scan data filtering.
- Section 6.3.3: Corrected steps for displaying an application bitmap:
  Parameter `hmi_display_mode` needs to be set to `application_bitmap` instead of `static_logo`.
- Chapter 5: Added chapter on monitoring lens contamination.
- Chapter 7: Added chapter on switching input/output channels.
- Section 7.3.3: Added description of new timestamp synchronization signal.
- Various textual and cosmetic updates

C.6 Release 2016-03 (protocol version 1.02)

Minor documentation update for protocol version 1.02:

- Section 2.6.5: Added list of available scan resolutions for R2000 HD devices to table 2.1.
- Section 3.4.3: Updated description of flag `new_settings`.
- Section 8.1: Added section on device discovery using SSDP.
- Various textual and cosmetic updates

C.7 Release 2015-04 (protocol version 1.01)

Document update for protocol version 1.01:

- Added descriptions for all protocol extensions listed in appendix B.4
- Section 1.2.2: Moved description of parameter types to separate section (section 2.1)
- Section 2.1: Extended description for various parameter types (`bitfield`, `string`, `IPv4`, `ntp64`, `binary`)
- Section 2.8.2: Added missing description of temperature warning and error flags in system status flags
- Section 3.2: Separated description of basic scan data output mechanisms from description of commands (section 3.3)
- Section 3.2.2: Added separate section for details on connection handles and backwards compatibility
- Section 3.2.3: Added separate section for details on the connection watchdog mechanism
- Section 3.2.4: Added section on configuration options for scan data output
- Section 3.2.5: Added separate section on performance considerations for concurrent scan data connections
- Section 3.4.3: Added missing description of temperature warning and error flags in scan data header status flags
- Chapter 6: Added chapter about client application access to the HMI LED display
- Appendix B: Separated protocol history and document history
- Various textual and cosmetic updates

C.8 Release 2013-08 (protocol version 1.00)

First public release.

Index for commands and parameters

This index provides a quick reference for all commands and parameters defined by this communication protocol.

References

[1] RFC-791: Internet Protocol Specification
http://tools.ietf.org/html/rfc791

[2] RFC-1305: Network Time Protocol (Version 3)
http://tools.ietf.org/html/rfc1305

[3] RFC-2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types
http://tools.ietf.org/html/rfc2046

[4] RFC-2069: An Extension to HTTP : Digest Access Authentication
http://tools.ietf.org/html/rfc2069

[5] RFC-2616: Hypertext Transfer Protocol – HTTP/1.1
http://tools.ietf.org/html/rfc2616

[6] RFC-3629: UTF-8, a transformation format of ISO 10646
http://tools.ietf.org/html/rfc3629

[7] RFC-3986: Uniform Resource Identifier (URI)
http://tools.ietf.org/html/rfc3986

[8] RFC-4648: The Base16, Base32, and Base64 Data Encodings
http://tools.ietf.org/html/rfc4648

[9] RFC-7159: The JavaScript Object Notation (JSON) Data Interchange Format
http://tools.ietf.org/html/rfc7159

[10] Online CRC32 calculation of an example buffer
http://crccalc.com/?crc=0x010x020x030x040x050x060x070x08&method=crc32&datatype=hex

[11] Libwww: the W3C Protocol Library
http://www.w3.org/Library/

[12] libcurl: free and easy-to-use client-side URL transfer library
http://curl.haxx.se/libcurl/

[13] Wireshark: free network protocol analyzer for Unix and Windows
http://www.wireshark.org/

[14] Simple Service Discovery Protocol (Draft v1.03)
https://tools.ietf.org/html/draft-cai-ssdp-v1-03

[15] Zero Configuration Networking (Zeroconf)
http://www.zeroconf.org/

FACTORY AUTOMATION – SENSING YOUR NEEDS

Worldwide Headquarters
Pepperl+Fuchs GmbH · Mannheim · Germany
E-mail: fa-info@pepperl-fuchs.com

USA Headquarters
Pepperl+Fuchs Inc. · Twinsburg, OH · USA
E-mail: fa-info@us.pepperl-fuchs.com

Asia Pacific Headquarters
Pepperl+Fuchs Pte Ltd · Singapore
Company Registration No. 199003130E
E-mail: fa-info@sg.pepperl-fuchs.com

#74998 clean – 2024-05-16 13:25

www.pepperl-fuchs.com

PEPPERL+FUCHS
SENSING YOUR NEEDS