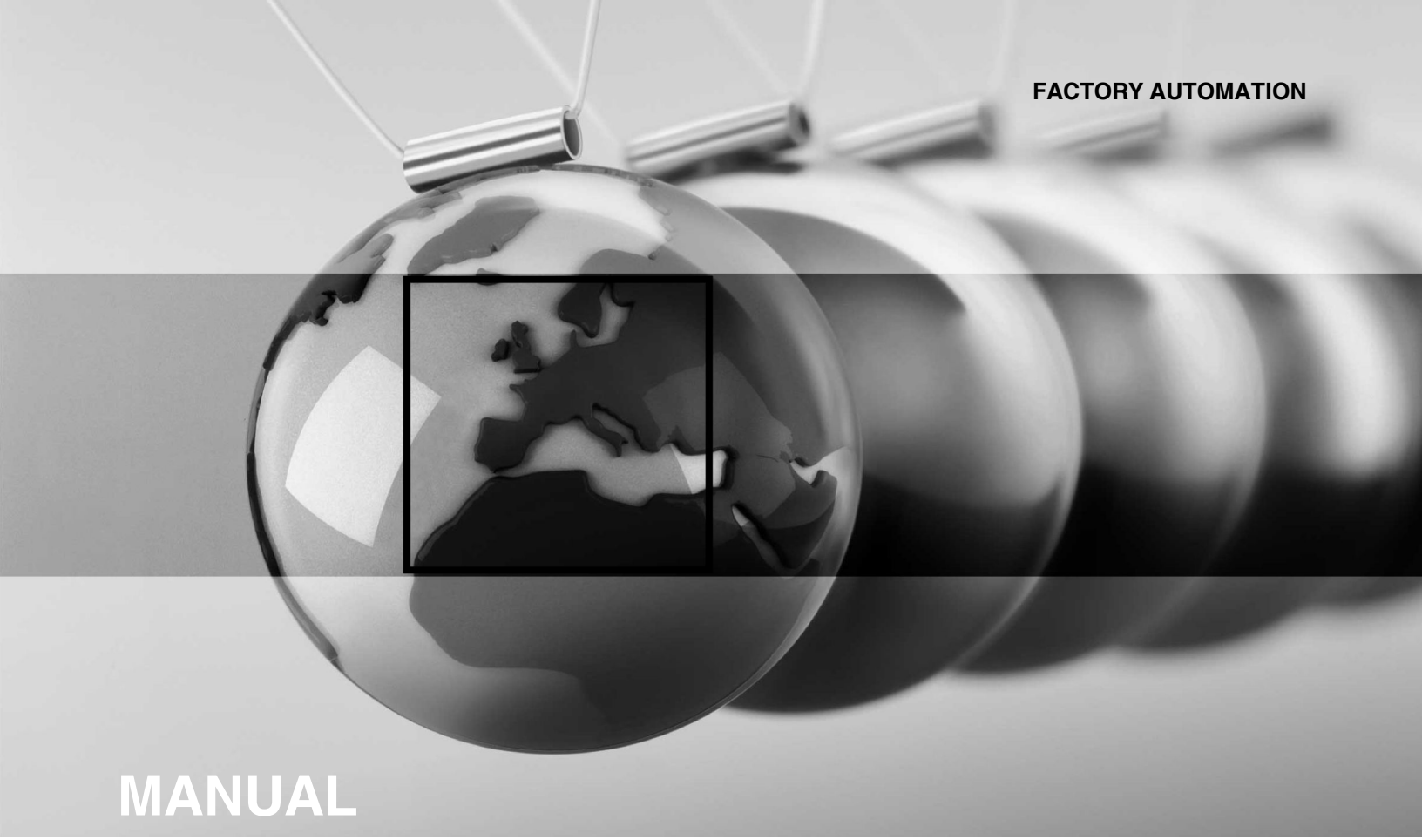
FACTORY AUTOMATION



MANUAL

IO-Link Master

ICE2 and ICE3 Models

REST API Addendum

EtherNet/IP™

IO-Link

PROFINET®

PEPPERL+FUCHS

Table of Contents

2024-10

1. REST API - HTTP API

The REST API is available starting with application base 1.5.37 for either PROFINET IO or EtherNet/IP.

In addition to providing a browser-based user-interface, there is an HTTP-based interface designed for use by external programs such as PortVision DX. The general goal was to follow RESTful design principles as much as is practical considering the underlying functionality and typical use cases.

- http://en.wikipedia.org/wiki/Representational state transfer
- http://developer.ibm.com/articles/ws-restful

1.1. Authentication

Authentication is handled via standard HTTP protocol methods using the same username/password settings as those used by the browser-based UI.

The *user* and *operator* usernames have read-only access to all configuration, status, firmware, and log information. The *admin* username has full access to read/write/clear/set-to-default configuration and status information, all actions, and firmware upload/update operations.

1.2. Paths

The API URL paths all start with **`/api`** and are organized into nine trees:

```
/api/config
/api/status
/api/logs
/api/action
/api/firmware
/api/iodd
/api/isdu
/api/datastorage
/api/security
```

1.3. Configuration

Accessing/manipulating configuration data is done via the **/api/config** namespace which contains three different sub-trees:

/ api/config/data	Use to read/write configuration data itself.
/ api/config/clear	Use to reset configuration data to factory default values.
/ api/config/verify	Use to verify proposed configuration data without writing.
/ api/config/directory	Use to obtain meta-information about the configuration data tree structure.

1.3.1. Configuration Data Read/Write

Configuration data values are accessed by using the **data** namespace:

```
/api/config/data[/<sub-tree>]
```

The <subtree> element is optional. If no sub-tree is present in the request, the entire configuration data tree is accessed. Example URLs:

```
http://<host>/api/config/data
http://<host>/api/config/data/network
http://<host>/api/config/data/network/hostname
```

The following HTTP requests are implemented for the configuration data namespace:

GET	Returns the specified configuration data tree as a JSON file.
PUT	Writes the supplied JSON data to the specified configuration data tree. The structure of the supplied JSON file must match the existing configuration data structure exactly. The JSON data sent with the request must not contain any extra fields and must not omit any fields. If the supplied JSON tree does not match, or if any of the data values are invalid, the request will be rejected and an error status and message will be returned.
POST	Writes the supplied JSON data to the specified configuration data tree. The supplied data may be sparse (it may omit fields that are to be left unchanged). If any of the supplied data values are invalid, the entire request will be rejected and an error status/message will be returned. If the supplied data contains extra fields, they will be discarded, the remaining data (if it is valid) will be written, and a warning message will be returned.
DELETE	Causes the specified data tree to be reset to default values. [Identical to sending a **GET** request in the **clear** namespace.]

1.3.2. Configuration Reset

The **clear** namespace can be used reset the specified data to default values.

```
/api/config/clear[/<sub-tree>]
```

A **GET** request will reset the specified data to factory settings. This is identical to sending a **DELETE** command in the **data** namespace.

1.3.3. Configuration Verify

The **verify** namespace can be used to do a verify-only **PUT** or **POST** request.

```
/api/config/verify[/<sub-tree>]
```

The **PUT** and **POST** actions/replies are the same as for the **/api/config/data** resource name above, except that after the tree/data is checked, no data is actually written.

1.3.4. Configuration Directory

A JSON representation of the device's configuration data tree structure can be obtained by sending a **GET** request to the following resource path.

```
/ api/ config/ directory [/<depth>] [/<sub-tree>]
```

If present, the sub-tree element specifies the root of sub-tree to be returned. If it is not present, the directory will begin at the root of the configuration data tree.

If the depth element is present, it will be a decimal number specifying the depth of the tree to be returned. No depth value or a value of 0 will return the entire tree. A depth value of 1 will return a single-level list of elements that are the immediate children of the specified sub-tree.

Examples:

/api/ config/ directory	Returns the entire configuration data tree structure.
/ api/config/ directory/ 2	Returns a two-level list of the top-level elements in the configuration data tree and their immediate children.
/ api/config/ directory/ network	Returns the entire configuration data under **network**.
/ api/config/ directory/1/network	Returns a single-level directory of the elements immediately under **network**.

1.4. Status and Diagnostics

Status and diagnostic data are accessed in a manner similar to configuration data using the **/api/status** namespace which contains the following namespaces:

/ api/status/ data	Use to read or clear status and diagnostic data.
/api/status/clear	Use to clear status/diagnostic data.
/api/status/ directory	Use to obtain meta-information about the status data tree structure.

1.4.1. Status and Diagnostics Data

Status and diagnostics data values are accessed by using the **data** namespace:

```
/api/status/data[/<sub-tree>]
```

The <subtree> element is optional. Info sub-tree is present in the request, the entire status/diagnostics data tree is accessed. Example URLs:

```
/api/status/data
/api/status/data/system
/api/status/data/system/MacAddress
```

For clear operations exactly one top level tree must be specified:

DELETE	/api/status/data/system
DELETE	/api/status/clear/iolink
GET	/api/status/clear/system
GET	/api/status/clear/iolink

The following HTTP request is implemented for the **status data** namespace:

GET	Returns the specified status/diagnostics data tree as a JSON file.
DELETE	Resets/clears any resettable status/diagnostic values in the specified top-level tree. Only a single top-level tree is supported (the <sub-tree> element is required and can only contain single name. Not all status/diagnostic values are resettable. Values reflecting current-state will not be altered, but counters, error messages, sticky flags, etc. will be reset/cleared. [Identical to using the **GET** command within the **clear** namespace.]

1.4.2. Status and Diagnostics Clear

Status and diagnostic data can be reset/cleared using the **clear** namespace:

```
/api/status/clear/<subsystem>
```

Sending a **GET** request to the **clear** namespace will reset/clear status/diagnostic values within the specified subsystem. This is identical to send the **DELETE** command within the **data** namespace. The clear operation can only be applied to a single top-level tree.

1.4.3. Status and Diagnostics Directory

A JSON representation of the device's status data tree structure can be obtained by sending a **GET** request to the following resource path.

```
/api/config/directory [/<depth>][/<sub-tree>]
```

For additional details see *Configuration Directory* on Page 4.

1.5. Log Files

System log files (e.g. syslog, dmesg, etc.) can be accessed using the /**api/logs** namespace which is organized into the following sub-trees:

/api/logs/file/ <filename>	Use to read/clear system log files.
/api/logs/clear/ <filename>	Use to clear system log files.
/api/logs/directory	Use to obtain a list of available log files.

1.5.1. Log File Access

The actual log files are accessed using the **`file`** namespace:

/api/logs/file/<filename>

The <filename> element is required, and must be one of the available log files (e.g. syslog, dmesg, eventlog, ps, top). The following HTTP request types are implemented:

`GET`	Returns the specified log file as an ASCII text file. The file format varies depending on which file is requested.
`DELETE`	Clears the specified log file. This operation is not implemented for some files (e.g. ps and dmesg), and will have no effect when specified for such files. [Identical to using a **`GET`** command within the **`clear`** namespace.]

1.5.2. Log File Clear

Log file(s) may be cleared by using the **`clear`** namespace:

```
/api/logs/clear/<filename>
```

Sending a **`GET`** request to the clear namespace will reset/clear the specified file. The <filename> element is required and must be one of the supported filenames.

1.5.3. Log File Directory

A list of available log files can be obtained using the **`directory`** namespace:

```
/api/logs/directory
```

The following HTTP requests are implemented:

`GET`	Returns a JSON array containing the names of available log files.

1.6. IODD Files

The **iodd** namespace provides access to the areas of the filesystem that contain IO-Link IODD files and associated data. These files exist into two separate areas: **config** and **std**.

The **config** iodd file area is empty when units are shipped from the factory and contains files (and derived data) loaded by the user. The con.fig iodd file area also contains JSON and language files generated by the Web UI code from the IODD .xml files found in the **std**iodd file area.

The **std** iodd area contains the IODD files defined by the IO-Link standard which are shipped as part of the application base image. It should generally be considered read-only.

The paths for these two areas are:

```
/api/iodd/config
/api/iodd/std
```

Each of these two paths can be treated much like a file system containing a tree of files/directories.

1.6.1. Config IODD Area

At the top level of the **config** area are the following:

- Numerical directories. Each of the numerical directory names is the decmial vendor id of one or more IODD files that have been loaded by the user. Within each of the vendor id directories, there is another level of numerical directories whose names correspond to the decimal device ids for the loaded IODD files. Within each device id directory are the uploaded IODD XML file, the graphical image files, a **config.json** file generated from the .xml file, and PHP language dictionaries generated from uploaded IODD language files.
- The **ioddfile.json** file. This is a catalog file listing some basic characteristics about each of the user-loaded IODD files found under the numeric directories mentioned above.
- The **language** directory. This directory contains PHP language dictionaries generated from the **std** IODD files.
- The JSON files generated from each of the standard IODD files found in the **std** area.

Note that though .xml files can be displayed by the Web UI, the webui only uses the JSON and PHP files during routine operation. The XML files are only parsed once when they are uploaded.

1.6.2. STD IODD Area

The **std** iodd area contains only the standard IO-Link .xml files. For example:

```
/api/iodd/std
|
|-- IODD-StandardDefinitions1.0.1-de.xml
|-- IODD-StandardDefinitions1.0.1-fr.xml
|-- IODD-StandardDefinitions1.0.1.xml
|-- IODD-StandardDefinitions1.1-de.xml
|-- IODD-StandardDefinitions1.1-fr.xml
|-- IODD-StandardDefinitions1.1.xml
|-- IODD-StandardUnitDefinitions1.0.1-de.xml
|-- IODD-StandardUnitDefinitions1.0.1-fr.xml
|-- IODD-StandardUnitDefinitions1.0.1.xml
|-- IODD-StandardUnitDefinitions1.1-de.xml
|-- IODD-StandardUnitDefinitions1.1-fr.xml
`-- IODD-StandardUnitDefinitions1.1.xml
```

These files should normally be considered as read-only and should be updated as part of the application base. These files are parsed on an as-needed basis, and corresponding JSON and PHP files are generated in the **config** area.

1.6.3. Operations

The following operations are defined for the iodd areas:

Directory	An HTTP **GET** request for a path that ends in /. dir will return a JSON array listing for the directory/file specified by the request path preceding the trailing **/ .dir.**
Recursive Directory	An HTTP **GET** request for a path that ends in /. rdir will return a recursive JSON array listing for the directory/file specified by the request path preceding the trailing **/ . rdir.** An **rdir** request on a file will behave the same as a **dir** request on a file.
Size	An HTTP **GET** request for a path that ends in **/.size** will return a single line of ascii text containing a decimal number representing the disk usage (in units of K bytes) for the file or directory specified by the request path preceding the trailing **/.size.** The size of a directory will include disk space used by all contents under that directory.
Get File	An HTTP **GET** request for a path (excluding those with special suffixes described above) that specifies an existing file will return the requested file. The response content-type will be set according to the filename suffix if it is recognized (e.g. **application/json, application/xml, image/png, image/gif**, etc.).
Get Archive	An HTTP **GET** request for a path that specifies a directory will return an archive of the contents of the specified directory. The type of archive will be determined by the request's HTTP Accept: header. The currently supported values are: application/zip, application/x-tar, application/x-tar-gz. If no Accept: header is found in the request, or if it has a value of*/*, then application/zip will be assumed.
Put Archive	An HTTP **PUT** or **POST** request with a content-type of **application/zip, application/x-tar**, or **application/x-tar-gz** will create a directory (if needed) with the specified path (creating parent directories as needed). The request data content will be treated as an archive and will be uncompressed/unpacked within the specified directory. Existing files will be overwritten as needed.
Put File	An HTTP **PUT** or **POST** request with a content-type other than the those listed above will create a file with the specified path (creating parent directories as needed) and the request data will be written to that file. Any existing file with that path will be overwritten.
Remove	An HTTP **DELETE** request will remove the file or directory (and all contents) specified by the specified path.

Note: *Using the above API it is possible to create and delete files/directories named . dir, .rdir, and .size. Directory listings will show such files/directories. Such files will be returned as part of a get archive request on a parent directory. But, it will not be possible to retrieve such files directly since a GET request on such a path will be interpreted a request for meta-information about the parent path.*

1.6.4. Curl Examples

Here is an example showing how to back up the user-loaded IODD files as a zip file (default format for **GET** is a zip archive):

```
$  curl  http://10.0.0.99/api/iodd/config  >user-iodd.zip
```

Writing those files back to the IO-Link master device requires that you specify a content-type:

```
$ curl -H Expect: -H Content-Type:application/zip -T user-iodd.zip http://10.0.0.99/
api/iodd/config
```

If you want to get the files in a format other than a zip archive, you must specify an Accept: header in the **GET** request:

```
$  curl  -H Accept:application/x-tar-gz  http://10.0.0.99/api/iodd/config  >user-
iodd.tar.gz
```

When writing them back, you must again specify the format:

```
$  curl -H Expect:  -H Content-Type:application/x-tar-gz  -T user-iodd.tar.gz  http:/
/10.0.0.99/api/iodd/config
```

1.6.5. Path Restrictions

Although the iodd server code has been designed to prevent any special treatment or shell evaluation of any characters/strings found in file paths, the following characters are not permitted because they can cause security issues when interpreted by a shell:

- tilde:~
- backslash: \
- star:*
- dollar: $
- parent directory strings: ../or/..

1.7. Actions

The **action** namespace can be used to perform a variety of miscellaneous operations on the device. All require admin privileges.

```
/api/action/reboot
```

Sending the data string 1 with a **PUT** request will cause the device to reboot after replying to the message.

```
/ api/action/ identify
```

Sending the data string on or off, with a PUT request will turn identify (flash LED) mode on or off. A **GET** request will return the data string on or off.

1.8. Firmware

The **firmware** namespace can be used to list, install, or update device firmware. There are two categories of firmware: images and packages. Each has its own namespace:

```
/api/firmware/image
/api/firmware/package
```

1.8.1. Images

An **image** is a block of opaque binary data - usually with a ulmage header to allow identification and integrity checking. An image is copied (either with or without the ulmage header) directly into a raw NANO flash partition. It could be a file system image, a kernel+ rootfs image, U-Boot executable image, bootstrap executable image, etc.

The following image paths are supported:

api/firmware/image/ directory	A **GET** request will return a text file containing a list of flash partitions and version numbers of installed images
/api/firmware/image/<partition>	A **PUT** request will install the accompanying file in the specified partition. The <partition> specification can be a partition device name such as **mtd3** or it can be a partition label such as **U-Boot-Code** or **ulmage-Primary**.
/api/firmware/image/<partition>	A **DELETE** request will erase the specified flash partition.

Here is an example of using the **curl** utility to update the **OS/rootfs** partition image:

```
curl -H Expect: -T system-1.00.uimage http://10.0.0.99/api/firmware/image/uimage-Primary
```

The **-H Expect**: option is required to tell curl to send the data file without waiting for the web server to send a 100-continue response after receiving the initial part of the post.

A similar command can be used to update the application base:

```
curl -H Expect: -T application-base-eip-1.4.2.uimage http://10.0.0.99/api/firmware/image/apps
```

A **force** option can be appended to the path to override restrictions based on model number, vendor, etc:

```
curl -H Expect: -T system-1.00.uimage http://10.0.0.99/api/firmware/image/uimage-Primary/force
```

or

```
curl -H Expect: -T application-base-eip-1.4.2.uimage http://10.0.0.99/api/firmware/image/apps/force
```

1.8.2. Packages

A **package** is a bundle of files for use with the **ipkg** package management utility (a derivative of Debian's dpkg). The following package paths are supported:

/ api/firmware/ package/ directory	A **GET** request will return a text file containing a list of installed packages and their versions.
/ api/firmware/package	A **PUT** request will install/update the accompanying .ipk package file.
/api/firmware/package/ <package-name>	A **DELETE** request will uninstall the named package.

Here is an example of using the **curl** utility to update/install a package:

```
curl -H Expect: -T iolink-driver 1.2.1.1 arm.ipk http://10.0.0.99/api/firmware/package
```

A **force** option can be appended to the path to override restrictions based on model number, version number, vendor, etc:

```
curl -H Expect: -T iolink-driver 1.2.1.1 arm.ipk http://10.0.0.99/api/firmware/package/force
```

1.9. ISDU

The **isdu** namespace can be used to perform IOLink ISDU read and write operations on the devices. Requests are sent as a JSON data array to the path below:

```
/api/isdu/request
```

The response will either be an HTTP error and associated mesage text if the request was not recognized as valid JSON, or a JSON array in the case where the request was valid JSON.

1.9.1. Request Format

The request must consist of a single JSON array. Each element in the array is a JSON object containing a single read or write request. Required fields for both read and write request objects:

req	The **req** field must be a string with value of either **read** or **write**.
port	The **port** field must be an integer ranging from 0-3 for a four-port IO-Link master unit or 0-7 for an eight-port IO-Link master unit.
index	The **index** field is the ISDU index and is an integer from 0-65535. Different IO-Link devices implement different sets of indexes. In general, only index 0 and index 1 are guaranteed to work for all IO-Link devices. Optional field for both read and write request objects.
subindex	The **subindex** field is an optional integer value. If none is provided, a subindex of 0 will be used in the request sent to the IO-Link device. Required field for write request objects.
data	The **data** field is a string containing one or more white-space delimited hexadecimal byte values.

1.9.2. Example Requests

The example below shows a request array containing a number of read and write requests:

```
  'req': 'read',
  'port': 0,
  'index': 0
},
{
  'req': 'read',
  'port': 0,
  'index': 0,
  'subindex': 3
},
{
  'req': 'read',
  'port': 0,
  'index': 24
},
{
  'req': 'write',
  'port': 0,
  'index': 24,
  'data' : '31 32 33 34 35 36 37'
},
{
  'req': 'read',
  'port': 0,
  'index': 24
},
{
  'req': 'write',
  'port': 0,
  'index': 24,
  'subindex': 4,
  'data': '44'
},
{
  'req': 'read',
  'port': 0,
  'index': 24
```

1.9.3. Response Format

The response consists of a JSON array containing a response object for each request object that was present in the request array.

If any of the request objects contained invalid data or was missing a required field, then the entire array of requests is rejected and none of the requests will be executed. The response objects corresponding to erroneous request objects contain a single `status` field containing an error message. The response objects corresponding to valid request objects are empty.

```
[
  {
    'req': 'read',
    'port': 0,
    'index': 0
  },
  {
    'req': 'clear',
    'port': 0,
    'index': 0,
    'subindex': 3
  },
]
```

It generates the following response array:

```
[
    {
    },
    {
      "status": "Missing or invalid 'req' value"
    }
]
```

The first, valid, read request was not executed, so there is no error message or response status/data. The follow request contains two valid requests:

```
    'req': 'read',
    'port': 0,
    'index': 0
  },
  {
    req': 'read',
    'port': 0,
    'index': 3
  },
```

It generates a response that looks like this:

```
    "req": "read",
    "port": 0,
    "index": 0,
    "subindex": 0,
    "code": 16,
    "status": "OK",
    "data": "00 28 19 21 11 41 00 01 36 00 01 as 00 00 00 00"
  },
  {
    "req": "read",
    "port": 0,
    "index": 3,
    "subindex": 0,
    "code": 36,
    "status": "OK",
    "data": "00 00 00 00 00 32 ff ff fa 2d 00 0c 01 00 18 01 00 3c 01 00 3d 01 02 26 01 02 45 01 02 46 01 07 d0
```

Request objects that were executed will have req, port, index, subindex, code, and status fields. Read requests may also have a data field. If the request was successful, the code field will be an integer telling how many bytes were read or written, and the status field will be the string OK.

If the request object was executed but failed, then the code field will contain a negative number and the status field will contain an error message. Responses to failed read requests will not contain a data field.

For example, both of the read request object below are valid, and get executed, but one is rejected by the device and fails:

```
[
  {
    'req': 'read',
    'port': 0,
    'index': 0
  },
  {
    'req': 'read',
    'port': 0,
    'index': 33
  },
]
```

Response:


```
[
    {
      "req": "read",
      "port": 0,
      "index": 0,
      "subindex": 0,
      "code": 16,
      "status": "OK",
      "data": "00 28 19 21 11 41 00 01 36 00 01 a8 00 00 00 00"
    },
    {
      "req": "read",
      "port": 0,
      "index": 33,
      "subindex": 0,
      "code": -2122317807,
      "status": "status=0x01: protocol error 0x80: device application error 0x11 -- index invalid"
    }
]
```


Likewise, write responses will contain status and code fields, but no data field.


```
[
  {
    'req': 'write',
    'port': 0,
    'index': 24,
    'data': '41 42 43 44 45 46 47'
  },
  {
    'req': 'read',
    'port': 0,
    'index': 24
  },
  {
    'req': 'write',
    'port': 0,
    'index': 24,
    'subindex': 3,
    'data': '55'
  },
]
```

Response:


```
[
    {
      "req": "write",
      "port": 0,
      "index": 24,
      "subindex": 0,
      "code": 7,
      "status": "OK"
    },
    {
      "req": "read",
      "port": 0,
      "index": 24,
      "subindex": 0,
      "code": 7,
      "status": "OK",
      "data": "41 42 43 44 45 46 47"
    },
    {
      "req": "write",
      "port": 0,
      "index": 24,
      "subindex": 3,
      "code": -2122317806,
      "status": "status-0x01: protocol error 0x80: device application error 0x12 -- subindex invalid"
    }
]
```


1.10. Data Storage

The **`datastorage`** namespace can be used to read, write, and delete the files used by the IOLink Data Storage subsystem for storage of IO-Link device configuration data. The path used is

`/api/datastorage/data`

Operations are the same as for IODD files.

HTTP **`GET`** on **`/api/datastorage/data`** will return an archive (by default a zip file) containing all of the data storage files.] There are typically one to eight files which are named port1 through port N. Each file will contain an opaque blob of binary data sized from a few tens of bytes to a several hundred bytes. There are typically no subdirectories.

1.11. Security

The **`security`** tree can be used to write or delete the various certificate and key files used by protocols that support encryption and authentication.It can not be used to read certificates or keys from the IO-Link master.

File	Format	Description
web/cert_key	PEM cert+key	Server certificate and privite key used by the web server
opcua/server_cert	PEM/DER cert	Server certificate used by OPC UA server
opcua/server_key	PEM/DER key	Private key for OPC UA server certificate above
opcua/client_auth_cert1	PEM/DER cert	Certificate used by OPC UA server to authenticate client connections and sessions
opcua/client_auth_cert2	PEM/DER cert	Certificate used by OPC UA server to authenticate client connections and sessions
mqtt/client_cert	PEM cert	Certificate used by MQTI client to authenticate itself
mqtt/client_key	PEM key	Certificate used by MQTI client to authenticate itself
mqtt/server_auth_cert	PEM cert	Certificate used to authenticate MQTI server

These files are write/delete only.

They can be written via a **`PUT`** or **`POST`** to the file namespace:

```
PUT  /api/security/file/opcua/server_cert
POST /api/security/file/opcua/server_cert
```

They can be deleted via a **`DELETE`** request on the **`.file`** namespace, or via a **`GET`** request on the **`clear`** namespace.

```
DELETE /api/security/file/opcua/server_cert
GET    /api/security/clear/opcua/server_cert
```

A JSON array containing the currently supported paths can be read via a **`GET`** request to the **`directory`** namespace:

```
GET   /api/security/directory
```

1.12. Summary of Operations

The table below summarizes the available namespaces and HTTP operations.

Namespace	HTTP Requests
/ api/config/data[/ <sub-tree>]	GET,PUT,POST,DELETE
/ api/config/clear[/sub-tree>]	GET
/ api/config/verify	PUT, POST
/ api/config/directory[/ <depth>][/ <sub-tree>]	GET
/ api/status/data[/<sub-tree>]	GET,DELETE
/ api/status/ clear[/ <tree>]	GET
/ api/status/directory[/ <depth>][/ <sub-tree>]	GET
/ api/logs/file/ <filename>	GET,DELETE
/ api/logs/ directory	GET
/ api/action/reset	PUT
/ api/action/identify	GET,PUT
/ api/firmware/image/directory	GET
/ api/firmware/image/ <partition>	PUT, DELETE
/ api/firmware/package/directory	GET
/ api/firmware/package	PUT
/ api/firmware/package/ <package>	DELETE
/ api/iodd/ config	GET,PUT,POST,DELETE
/ api/iodd/std	GET,PUT,POST,DELETE
/ api/isdu/request	PUT,POST
/ api/datastorage/data	GET,PUT,POST,DELETE
/ api/security/file	PUT,POST,DELETE
/ api/security/clear	GET
/ api/security/directory	GET

FACTORY AUTOMATION – SENSING YOUR NEEDS

Worldwide Headquarters
Pepperl+Fuchs Group
68307 Mannheim · Germany
Tel. +49 621 776-0
E-mail: info@de.pepperl-fuchs.com

USA Headquarters
Pepperl+Fuchs Inc.
Twinsburg, Ohio 44087 · USA
Tel. +1 330 4253555
E-mail: sales@us.pepperl-fuchs.com

Asia Pacific Headquarters
Pepperl+Fuchs Pte Ltd.
Company Registration No. 199003130E
Singapore 139942
Tel. +65 67799091
E-mail: sales@sg.pepperl-fuchs.com

www.pepperl-fuchs.com

PEPPERL+FUCHS
SENSING YOUR NEEDS

TDOCT-9571

2024-10