XPL REST API Documentation

Release 1.0

I2SE GmbH

May 17, 2017

Contents

CHAPTER 1

XPL device's REST API documentation

Preamble

A single XPL device varies within the XPL device family in it's hardware configuration, i.e. there are various product variants available, for different usage scenarios and/or different mounting options.

However, all device variants feature a common device API to access the device's data and configure it to customer needs. This API is available as a RESTful HTTP service on every single device.

And this document describes the resources that make up this official API.

Device modelling

Before describing the API itself, it necessary to agree on a common wording and to understand some basic concepts which are implemented within the XPL device family. The following chapters describe these concepts and introduce some terms used within this document and on the web front-end of each device.

Physical channel

A physical channel is the base of the whole device, i.e. it is the representation of the screw terminals which form a usable unit. In other words, a *XPL Rail IO6* device has 6 physical channels. The numbering of this channels is fixed and represents the physical numbering. It is identical to the device printing/labels etc.

There exists two classes of physical channels: *serial* and *io*. As expected, the class *serial* is used for COM port like channels, e.g. RS-232 or CAN interfaces, whereas an *io* physical channel represents two/three screw terminals to connect a sensor and/or a relais and so on. These two classes are independently numbered from each other, because the devices printing/labeling will also start anew while enumerating this interfaces.

A physical channel can have multiple capabilities, e.g. an io channel can be configured as input or output, but there might be devices, which can not be switch-able and thus are limited to output only. Also, a serial physical channel is usually bound to its hardware driver (e.g. CAN vs. RS-485...).

Virtual channel

A virtual channel is a logical unit which is created in software. It requires at least one physical channel as input source, and can have multiple physical outputs. The main identification property is a unique virtual channel id, which is a simple positive integer number between 1 and 65535. By design, the virtual channel id zero is invalid.

Three different types of virtual channels exist:

- digital virtual channels
- analog virtual channels
- serial virtual channels

This three type exist independently from each other, that means a digital virtual channel with id 1 can exist in parallel to an analog virtual channel with id 1. Both channel does not interfere in this case and cannot be mixed.

A virtual channel can only have one value at the moment, therefore it must be given that only one input is active on each virtual channel for sane behaviour. This concept is only disregarded in case of serial virtual channels: since these channels does not really have a single value at any time, but are intended to transfer data streams, multiple physical channels can be combined into one serial virtual channel. In this case, the virtual channels act like a bus system: the input of each physical channel is transferred to all other physical channels attached. Note, that there is no attempt to ensure time or other synchronisations: as soon as one device recognises input data, it is sent out to the other devices. So it is also not possible to detect collisions as on a real physical UART interface. Upper layers must be prepared to handle both situations smoothly.

Another important point is, that this concept makes it very easy to build physical bus system converters, e.g. you could feed CAN data into a serial virtual channel and output it via a RS-485 physical channel. Please keep in mind, that no logical protocol conversation is made, so this example might make only limited sense, but demonstrates the capabilities best.

Put it all together

As stated above, only physical channels are always available on a device. The existence of virtual channels depends on the configuration of physical channels.

You can assign a virtual channel to a physical channel, and thus create this virtual channel. Then then physical channel acts as a value source for this channel or is driven by the virtual channel's value.

Virtual channel data is exchanged via HTTP over UDP multicast packets. When assigning virtual channel to a physical one, then you enable the generation and receiving of such multicast packets for this virtual (and thus physical) channel.

However, you are not required to assign a virtual channel to a physical one. In this case, the physical channel can not be controlled via multicast packets, nor generate multicast packet itself upon e.g. state changes.

Please also note, that the state of a virtual channel can only be queried from the device sourcing the virtual channel. This is important in the case, where packages were lost during network transmission, so it's not possible to query a stale virtual channel state, but only get the real, actual value from the source of information. However, you can always read back the state of an assigned physical channel.

Basics

Depending on the firmware of the device, it might support multiple API versions.

At the moment, there exists only one API version: version 1.0.

All data is sent and received as JSON.

Root entry point

The root entry point for the API within the XPL devices' URL space, is `http://<device>/api`.

The root entry point is used to obtain information about the API itself.

At the moment it is only possible to query the API version number. You have to issue a GET request to the root entry point to get this information:

```
$ curl http://<device>/api
```

You will receive a JSON response as follows:

```
{
  "version": "1.0"
}
```

RESTful URL mapping

Physical/virtual channels

Since our basic concept to access the device are channels, the both channel classes are reachable with `http://<device>/api/channel/physical` resp. `http://<device>/api/channel/virtual`.

So this results in the following URLs (for a device with 2 IO port, 1 CAN interface and 1 RS-232):

- `http://<device>/api/channel/physical`
- `http://<device>/api/channel/physical/io`
- `http://<device>/api/channel/physical/io/1`
- `http://<device>/api/channel/physical/io/2`
- `http://<device>/api/channel/physical/serial`
- `http://<device>/api/channel/physical/serial/1`
- `http://<device>/api/channel/physical/serial/2`

Or in other words:

`http://<device>/api/channel/physical/{physical_channel_class}/{physical_channel_id}`

The physical URL space is always available since it reflects the hardware of a device, thus providing a way to configure the hardware and to determine its capabilities. The user has always the possibility to access an dedicated individual object or to query 'all' objects within the requested class.

The available virtual URL space depends on the configuration of the device:

- `http://<device>/api/channel/virtual/{virtual_channel_type}/{virtual_channel_id}`

e.g.

- `http://<device>/api/channel/virtual/digital/1`
- `http://<device>/api/channel/virtual/analog/4711`
- `http://<device>/api/channel/virtual/serial/19`

Please note, that accessing e.g. `http://<device>/api/channel/virtual` includes all JSON objects which could also be accessed individually via their own URL.

Note: As mentioned before, this is not true for `http://<device>/api`, as this URL is handled special.

Device information

Beside the data about channels etc., there is another API entry point to obtain data about the XPL device itself. This information is read-only and can be queried by a GET request to the URL: `http://<device>/api/device`

The JSON response looks like:

```
{
    "product": "I2XPLR4-IO600",
    "modelname": "XPL Rail",
    "hardware_version": "1.0",
    "software_version": "0.12",
    "hostname": "xpltest2",
    "mac_address": "00:01:87:FF:FF:27",
    "mac_address_plc": "00:01:87:FF:FF:26",
    "serial": "0000004711",
    "uuid": "444d5314-1000-4400-9500-000187ffff27"
}
```

A details description of this JSON object is given in section *Device information*.

Powerline network status/configuration

As the XPL devices are powerline devices, it's possible to obtain some details of the powerline network from the device and to trigger some powerline related actions. This information is bundled in the ''/api/powerline" hierarchy, which consists of four sub-elements:

- `http://<device>/api/powerline/network`
- `http://<device>/api/powerline/local`
- `http://<device>/api/powerline/stations`
- `http://<device>/api/powerline/neighbors`

The ''network" property is an object which contains information about the powerline network itself. Details are explained in *Powerline Network Details*.

The ''local" property is a an object which contains detailed information about the current device's powerline controller, see *Powerline Local Device Details*.

The ''stations" property is a an list of detected remote powerline stations. Each list entry is a powerline station object, described in *Powerline Station Details*.

The ''neighbor" property is a an object which accumulates all XPL neighbors, i.e. other XPL devices found in the same powerline network. See *Neighbor Details*.

HTTP verbs

The RESTful approach strives to use appropriate HTTP verbs for each action. For a XPL device, this only make limited sense: we have a fixed number of physical channels which can be configured and/or their state can be queried. However, it is not possible to delete and or create such a physical channel.

It important to realize, that only physical channels can be configured. The properties of virtual channels, or the existence of virtual channels at all, is always the result of the configuration of a physical channel. While querying properties of virtual channels is a valid access pattern, a configuration of a virtual channel is not possible.

According to the REST paradigm, querying REST objects is done via HTTP GET requests. Modifying a REST object is normally done with PUT method. On a XPL device, PUT and POST methods are handled equally.

So, to configure a dedicated physical channel, issue a PUT request to its URL.

Example to configure physical io channel 2: `http://<device>/api/channel/physical/io/2`

```
PUT /api/channel/physical/io/2 HTTP/1.1
Content-Length: 165
Content-Type: application/json
Accept: application/json

{
    "label": "My Label 2",
    "type": "do",
    "mode": "normal",
    "virtual_channel": 17,
    "level": "inverted",
    "delay_on": 0,
    "delay_off": 0,
}
```

On success, you will receive a JSON object in the same manner, as a GET request would give you, already with the updated configuration. If an error occurs, you'll get a JSON error object, see below for details.

Parameters

When querying a RESTful object, it's possible to filter out and/or request properties of the object. For this, a **property class** has been assigned to some properties.

For the JSON objects that are part of the ''/api/channel" hierarchy, there exists the property classes *structual*, *config*, *state* and *caps* (for capabilities) at the moment.

The default object view includes the classes *config* and *state*, but hides *structural*. You get this default view when you do not supply any of the following, optional parameters.

Parameter	Value	Meaning
filter	*structural*, *config*, *state* or *caps*	This *hides* the properties of the given class.
unfilter	*structural*, *config*, *state* or *caps*	This forces the properties of the given class to appear.

So for example to only query the current states of all physical channels, issue the following GET request:

$ curl http://192.168.55.57/api/channel/physical?filter=config

Another use case could be to retrieve the whole physical channel configuration (configuration backup):

$ curl http://192.168.55.57/api/channel/physical?filter=state

For the JSON objects that are part of the ''/api/powerline/neighbors" hierarchy, a property class *details* has been defined which is not included by default. To include these properties, supply the optional ''unfilter" parameter as described above:

$ curl http://192.168.55.57/api/powerline/neighbors?unfilter=details

JSON property details

Physical channels

As stated above, a physical channel is of a dedicated **class** and has an unique **id**. These both properties/attributes uniquely identify this channel. The channel id is defined as a unsigned integer value in the range 1-65535, because

humans tend to number the natural way, e.g. starting with 1 and thus expecting the device label to number the first serial interface with 1, not as 0. So by definition, a physical channel of 0 is none-existent.

As already described above, the channel **class** has only two valid values: `io` and `serial`.

The possible physical channel **types** depend on the physical channel class:

- for class `serial`: this is one of the strings `rs485`, `rs422`, `rs232`, `can`, `mbus`, `wmbus`, `enocean`
- for class `io`: this is one of the strings `di`, `do`, `s0`, `ai`, `ao`

The possible physical channel **modes** depends on the physical channel class and type:

- for class `io`:
  - type `di`: `normal`, `flipflop`
  - type `do`: `normal`, `pulse`
  - type `ai`: `0-10 V`, `1-10 V`, `0-20 mA`, `4-20 mA`
  - type `ao`: `0-10 V`, `1-10 V`
- for class `serial`:

  This is a integer, where each bit reflects whether a given service is enabled for this physical serial channel.

The property **label** is common to all physical channels, all other properties depend on the channel type and the hardware capabilities of the individual channel. The following list is an overview of the actual available properties:

class **Property class**: structural

Availability: always

JSON datatype: string

Description: Channel class, that is `serial` or `io`.

id **Property class**: structural

Availability: always

JSON datatype: number

Description: Channel's unique ID (1-65535).

label **Property class**: config

Availability: always

JSON datatype: string

Description: User-defined name/label to associate the channel with, used e.g. in the web frontend. Up to 16 characters (bytes) can be stored.

enabled **Property class**: config

Availability: class `serial` or class `io`

JSON datatype: number

Description: Indicator whether this channel is used at all.

virtual_channel **Property class**: config

Availability: class `serial` or class `io`

JSON datatype: number

Description: The virtual channel id of the linked virtual channel; zero by default, which menas that no virtual channel is assigned.

type **Property class**: config

Availability: class `serial` or class `io`

JSON datatype: string

Description: Channel type (depends on `class`, see above).

supported_types **Property class**: caps

Availability: class `serial` or class `io`

JSON datatype: array

Description: This array contains a list of strings which reports the hardware capabilities of the corresponding channel. This depends on the XPL device variant. The strings in this list are valid strings for the `type` property.

mode **Property class**: config

Availability: class `serial` or class `io`

JSON datatype: string

Description: Operation mode (depends on `class` and `type`, see above).

pullup **Property class**: config

Availability: class `io` and type `di`

JSON datatype: number

Description: Disable/enable an internal pull-up resistor on this channel. At the moment, the only valid values are `0` or `1`.

level **Property class**: config

Availability: class `io` and ((type `di` and mode `normal`) or (type `do`))

JSON datatype: string

Description: Value `direct` means, that a HIGH level on the wire is mapped to logical 1 (aka *active high*); whereas `inverted` means the that HIGH level is mapped to logical 0 (aka *active low*).

edge **Property class**: config

Availability: class `io` and type `di` and mode `flipflop`

JSON datatype: string

Description: Edge of the signal to trigger: `falling` or `rising`.

delay_on **Property class**: config

Availability: class `io` and type `do`

JSON datatype: number

Description: Delay on time (in ms).

delay_off **Property class**: config

Availability: class `io` and type `do` and mode `normal`

JSON datatype: number

Description: Delay off time (in ms).

width **Property class**: config

Availability: class `io` and type `do` and mode `pulse`

JSON datatype: number

Description: Pulse width (in ms).

threshold Property class: config

Availability: class `io` and ((type `ai`) or (type `di` and pullup `0`))

JSON datatype: number

Description: Voltage level (normalized 16-bit value) to detect the input as logical 1.

pulses_per_unit Property class: config

Availability: class `io` and type `s0`

JSON datatype: number

Description: User-supplied value to calculate the current energy reading.

unit Property class: config (for class `io` and type `s0`), state else

Availability: class `io` and (type `s0` or type `ai` or type `ao`)

JSON datatype: string

Description: For a channel configured as S0 input, this is a user-supplied string up to 16 characters (bytes); for an channel configured as analog input, this is a fixed string `mA` or `V` depending on the physical capabilities/configuration of the channel.

value Property class: state (for class `io` and type `s0` additionally config)

Availability: class `io`

JSON datatype: number

Description: This is the current/actual value of this channel. For an analog or S0 channel, this is a floating point number which must be interpreted together with `unit`; for a digital channel, this can only have the values `0` or `1`.

Note: When configuring a S0 channel and both `pulse_counter` and `value` are contained within the request, then both values must correspond to each other, otherwise the request will fail.

normalized_value Property class: state

Availability: class `io` and (type `ai` or type `ao`)

JSON datatype: number

Description: This is the current/actual value of this channel, mapped into a 16-bit value, i.e. 0-65535. This way it is possible to interconnect different analog types.

pulse_counter Property class: state and config

Availability: class `io` and type `s0`

JSON datatype: number

Description: Contains the raw value of the internal impulse counter. Note: When configuring a S0 channel and both `pulse_counter` and `value` are contained within the request, then both values must correspond to each other, otherwise the request will fail.

baudrate Property class: config

Availability: class `serial`

JSON datatype: number

Description: Baudrate of the channel. It depends on the actual device, which baudrates are possible at all.

databits Property class: config

Availability: class `serial`

JSON datatype: number

Description: Count of databits of the channel. It depends on the actual device capabilities, which values are supported. At the moment, this can only be `7` or `8`.

parity **Property class**: config

Availability: class `serial`

JSON datatype: string

Description: Parity setting of the channel, that is `none`, `odd` or `even`. Note, that not all combinations with *databits* and/or *stopbits* might be possible, depending on the actual device capabilities.

stopbits **Property class**: config

Availability: class `serial`

JSON datatype: number

Description: Count of stop bits used at the channel. Note, that not all combinations with *databits* and/or *stopbits* might be possible, depending on the actual device capabilities. For example, for all current XPL devices, this is required to be `1`.

port **Property class**: config

Availability: class `serial`

JSON datatype: number

Description: Port number of TCP raw socket server or Telnet server bound to this channel.

idle_timeout **Property class**: config

Availability: class `serial`

JSON datatype: number

Description: Idle time after which a TCP/Telnet connection is terminated automatically.

flags **Property class**: config

Availability: class `serial`

JSON datatype: Array of strings

Description: Array which contains various flags of the physical serial channel:

- `sw_mode`: The operation mode (*type*) is software switchable (e.g. RS-232 vs. RS-485). Whether this is supported depends on the actual XPL device.
- `sw_ctrl_local`: The settings *baudrate*, *databits*, *parity* and *stopbits* can be configured via web frontend of the XPL device.

  Note: A configured Telnet server on this physical channel still negotiates RFC2217 in this case; however, requests to change the port settings are silently ignored. A client can detect this situation when requesting a change and still reading back the old settings afterwards.

- `sw_ctrl_remote`: Defaults for *baudrate*, *databits*, *parity* and *stopbits* can be configured via web frontend and take effect right after power on of the XPL or after reboot. But it is possible for a RFC2217-enabled client to switch these settings at run-time.

stats **Property class**: state

Availability: class `serial`

JSON datatype: Object

Description: Statistics counter of corresponding UART.

active_connection **Property class**: state

Availability: class `serial`

JSON datatype: Object

Description: This object is present only, when a client is connected to the corresponding channel server (e.g. Telnet server). Then it contains various information about the connected client.

Note: The physical channel class `serial` does not has any property *value* as there is no buffering and the data stream is considered as a transient state. That means, that it is not possible to read any actual data upon request, but only receive a notification when data is transferred.

Virtual channels

As stated above, a virtual channel has an unique **id**. The next important property/attribute is the channel **type**, which can be `digitial`, `analog`, or `serial`. (On database jargon, this is tuple (type, id) is the unique primary key.)

All other channel properties depend on the channel type as describe in the following list:

id **Property class**: structural

Availability: always

JSON datatype: number

Description: Channel's unique ID (1-65535).

type **Property class**: structural

Availability: always

JSON datatype: number

Description: Virtual channel type, i.e. `digital`, `analog` or `serial`.

value **Property class**: state

Availability: type `digital` or type `analog`

JSON datatype: number

Description: This is the current/actual value of this channel. See description for physical channel property *value* for details.

unit **Property class**: state

Availability: type `analog`

JSON datatype: string

Description: This is an inherited property of the physical channel which feeds this virtual channel.

normalized_value **Property class**: state

Availability: type `analog`

JSON datatype: string

Description: This is the current/actual value of this channel, normalized to an unsigned 16-bit value (0-65535).

stats **Property class**: state

Availability: type `serial`

JSON datatype: Object

Description: Statistics counter for the virtual serial channel.

Device information

The device information JSON object consists of some properties which describe details of the XPL device. For this object, no property classes are implemented.

This JSON object has the following read-only properties:

product JSON datatype: string

Description: Contains the product code of this device.

modelname JSON datatype: string

Description: Contains the modell name string of this device.

hardware_version JSON datatype: string

Description: Contains the hardware version string of this device.

software_version JSON datatype: string

Description: Contains the software version string of this device.

hostname JSON datatype: string

Description: Contains the software version string of this device.

mac_address JSON datatype: string

Description: Contains the MAC address of the main processor of this device.

mac_address_plc JSON datatype: string

Description: Contains the MAC address of the powerline processor of this device.

serial JSON datatype: string

Description: Contains the serial number of this device.

uuid JSON datatype: string

Description: Contains the devices's UUID. This UUID is generated based on a unique serial number of the embedded microcontroller and the MAC address of this device.

Powerline Network Details

The powerline network JSON object consists of some properties which describe details of the powerline network. For this object, no property classes are implemented.

This JSON object has the following properties:

nid JSON datatype: string

Availability: always

Description: Contains the hexadecimal representation of the powerline network identifier.

short_network_id JSON datatype: number

Availability: always

Description: Contains the short network id of powerline network.

cco JSON datatype: object

Availability: always

Description: Contains information about the current powerline's central coordinator. This object has the following properties itself:

mac_address JSON datatype: string

Availability: always

Description: Contains the MAC address of the current CCo.

tei JSON datatype: number

Availability: always

Description: Contains the terminal equipment number of the current CCo.

result JSON datatype: number

Availability: after remote pairing action

Description: Only present, when a remote pairing operation was triggered. Represents the result of this operation, i.e. it contains zero as long as the operation is not completed or was not successfully, see below.

While the properties above are read-only, this object allows to add a remote device via DAK (Device Access Key) to the powerline network. For this, issue a PUT request to this object and provide a JSON object consisting of a single 'dak' property which contains the DAK string of the device to add. Note, that a simple DAK string is converted XPL internally to its binary representation which is the common use-case. However, it's also possible to give a hexadecimal string representation of the DAK - in this case, it is used as is.

```
PUT /api/powerline/network HTTP/1.1
Content-Length: 38
Content-Type: application/json
Accept: application/json

{
    "dak": "ABCD-EFGH-IJKL-MNOP"
}
```

After this HTTP request, the XPL device will begin to perform the requested action by sending out a HomePlug AV packet to its powerline processor. Once this packet is sent, the powerline network JSON object will contain the ''result'' property. In other words, this property does not show up immediately, but it can take a short time (typically less than 1 s). The value of this property is zero at the beginning which means that the operation was not successfully. However, it may take some time until success is reported from lower protocol stack. In this case, the value of the property becomes 1. So it's recommended to issue the PUT request, wait some seconds (e.g. 30s) and then query the operation result with a GET request.

Powerline Local Device Details

This JSON object contains data of the XPL device's powerline controller.

mac_address JSON datatype: string

Description: Contains the hexadecimal representation of the powerline controller's MAC address.

tei JSON datatype: number

Description: Contains the terminal equipment number of the device within the powerline network.

chipset JSON datatype: string

Description: Contains the chipset name.

fw_version JSON datatype: string

Description: Contains the firmware version string of the powerline chipset.

usr JSON datatype: string

Description: Contains the user string of the powerline PIB.

mfg JSON datatype: string

Description: Contains the manufacturer string of the powerline PIB.

dak JSON datatype: string

Description: Contains the hexadecimal representation of the powerline controller's DAK (Device Access Key).

nmk JSON datatype: string

Description: Contains the hexadecimal representation of the powerline's network management keys.

is_cco: JSON datatype: number

Description: When the current XPL device has the CCo role of the powerline network, then this property is present and contains one. If not, then this property has the value zero and is obmitted.

The properties above are all read-only, except the ''nmk" property: issuing a PUT request to it allows to associate to another powerline network:

```
PUT /api/powerline/local HTTP/1.1
Content-Length: 41
Content-Type: application/json
Accept: application/json

{
    "nmk": "SecretPowerlineNetwork"
}
```

Note, that a simple NMK string is converted XPL internally to it's binary representation which is the usual use-case. However, it's also possible to give a hexadecimal string representation of the NMK - in this case, it is used as is:

```
PUT /api/powerline/local HTTP/1.1
Content-Length: 66
Content-Type: application/json
Accept: application/json

{
    "nmk": "B2:C5:1F:63:4E:43:A9:D4:B9:0F:DF:61:C4:ED:90:DD"
}
```

After this HTTP request, the XPL device will begin to perform the requested action by sending out a HomePlug AV packet to its powerline processor. Once this packet is sent, the powerline network JSON object will contain the ''result" property. In other words, this property does not show up immediately, but it can take a short time (typically less than 1 s). The value of this property is zero at the beginning which means that the operation was not successfully. However, it may take some time until success is reported from lower protocol stack. In this case, the value of the property becomes 1. So it's recommended to issue the PUT request, wait some seconds (e.g. 30s) and then query the operation result with a GET request.

This JSON object also allows further actions being performed on the XPL device. For this you have to issue a PUT request which consists of a single JSON object with a string property called ''action''. The possible actions are listed below:

Value	Action performed
factory_defaults	This resets the powerline chipset to its factory defaults.
randomize_nmk	This assigns a random network management key to the XPL device, or -in other words- leave the current powerline network.
pbsc	Performs Push Button Simple Connect. This is equivalent to physically pressing the Push Button at the front panel of the XPL device. See user manual for details.

Example: The following PUT request resets the powerline chipset to its factory defaults:

```
PUT /api/powerline/local HTTP/1.1
Content-Length: 38
Content-Type: application/json
Accept: application/json
```

```
{
    "action": "factory_defaults"
}
```

Powerline Station Details

This JSON object represents a powerline station within the current powerline network. Please note, that detail information of the other devices must be collected by querying these devices. This may take some time, and also not all devices of other manufacturers will report all requested information. Thus some properties might be missing for single stations.

This object is read-only, no actions can be performed on the list and/or list entries.

tei JSON datatype: number

Availability: always

Description: Contains the terminal equipment number of the station.

mac_address JSON datatype: string

Availability: always

Description: Contains the MAC address of the station.

avg_data_rate JSON datatype: object

Availability: always

Description: Contains average data rates as seen by the XPL device. This object has the following properties itself:

rx JSON datatype: number

Availability: always

Description: Average receive data rate (in Mbit).

tx JSON datatype: number

Availability: always

Description: Average transmit data rate (in Mbit).

chipset JSON datatype: string

Availability: optional

Description: Contains the chipset name of the station's powerline chipset.

fw_version JSON datatype: string

Availability: optional

Description: Contains the firmware version string of the station's powerline chipset.

usr JSON datatype: string

Availability: optional

Description: Contains the user string of the station's powerline PIB.

mfg JSON datatype: string

Availability: optional

Description: Contains the manufacturer string of the station's powerline PIB.

is_cco: JSON datatype: number

Availability: optional

Description: When this station has the CCo role of the powerline network, then this property is present and contains one. If not, then this property has the value zero and is omitted.

neighbor: JSON datatype: string

Availability: optional

Description: When this station is an XPL device and has a neighbor entry, then this property contains the MAC address of the corresponding neighbor list item.

Neighbor Details

This JSON object stores detail information about detected XPL neighbor devices, i.e. other XPL devices found in the same powerline network.

Such other XPL devices - called neighbor - are represented as key-value pair, where the name of the key is the MAC address of the neighbors main processor, and the value is a JSON object with detail information.

Such a neighbor JSON detail object has properties as explained below. Please note, that gathering the information from a neighbor consists of several steps so some details might not be available yet at the time of querying this object.

Please note, that properties of the class ‘’details” are only included when requested explicitely.

product Property class: -

JSON datatype: string

Availability: optional

Description: Contains the product code of this neighbor.

hardware_version Property class: details

JSON datatype: string

Availability: optional

Description: Contains the hardware version string of this neighbor.

software_version Property class: details

JSON datatype: string

Availability: optional

Description: Contains the software version string of this neighbor.

hostname Property class: -

JSON datatype: string

Availability: optional

Description: Contains the software version string of this device.

mac_address Property class: details

JSON datatype: string

Availability: always

Description: Contains the MAC address of the main processor of this neighbor.

serial Property class: -

JSON datatype: string

Availability: optional

Description: Contains the serial number of this neighbor.

ip_address Property class: -

JSON datatype: string

Availability: optional

Description: Contains the neighbor's current IPv4 address.

This JSON object is read-only, no actions can be performed on the object itself and/or sub-objects.

Error handling

When the client request is faulty or an internal error occured, then an error JSON response is generated and transmitted as result of the operation.

Usually, errors can only occur on PUT/POST requests. For GET requests, only HTTP errors "404 - Not found" are generated, because wrong request parameters etc. are silently ignored.

The structure of such a JSON error object returned is as follows:

```
{
  "version": "1.0",
  "result": 5
}
```

The following table lists the result codes and there meaning.

Result code	Description
0	No error occurred (this is normally not seen, as in this case real data is returned).
1	JSON parsing error.
2	JSON request too long/internal out of memory condition.
3	(reserved)
4	URL not recognized and/or invalid JSON data during POST/PUT request.
5	Requested action is temporary not available (internal flash is busy), re-issue the request after a few seconds.

Examples

cURL examples

The following examples uses curl which is available in nearly all Linux distributions. It also shows parts of the communication, e.g. the received HTTP headers in the response.

Query common XPL Rail device information

This example queries common device properties.

```
curl -v http://192.168.178.191/api/device
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> GET /api/device HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
```

```
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
< Content-Length: 288
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```


```
{
    "product" : "I2XPLR4-IO600",
    "modelname" : "XPL Rail",
    "hardware_version" : "1.0",
    "software_version" : "0.12",
    "vcs_version" : "r1132",
    "hostname" : "j-m-io600-2",
    "mac_address" : "00:01:87:0A:00:15",
    "mac_address_plc" : "00:01:87:0A:00:14",
    "serial" : "0000000815",
    "uuid" : "444d5314-1000-4b00-9400-0001870a0015"
}
```


Check configuration status of a physical channel

This example queries the current configuration of the physical channel 1.

```
curl -v http://192.168.178.191/api/channel/physical/io/1
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> GET /api/channel/physical/io/1 HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
< Content-Length: 73
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```


```
{
    "label" : "Channel 1",
    "supported_types" : ["di", "do", "ai", "s0"],
    "enabled" : 0
}
```

Configure physical channel 1 as Digitial Output

This example reconfigures the physical channel 1 as digital output.

```
curl -v -X PUT -H "Content-Type: application/json" -d '{"type":"do","mode":"normal
↪"}' http://192.168.178.191/api/channel/physical/io/1
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> PUT /api/channel/physical/io/1 HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
> Accept: */*
> Content-Type: application/json
> Content-Length: 29
>
* upload completely sent off: 29 out of 29 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
< Content-Length: 175
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```


```
{
    "label" : "Channel 1",
    "supported_types" : ["di", "do", "ai", "s0"],
    "type" : "do",
    "enabled" : 1,
    "mode" : "normal",
    "virtual_channel" : 0,
    "level" : "direct",
    "delay_on" : 0,
    "delay_off" : 0,
    "value" : 0
}
```


Switch channel 1 (Digitial Output) on

This example switches the physical digital channel on.

```
curl -v -X PUT -H "Content-Type: application/json" -d '{"value":1}' http://192.168.
↪178.191/api/channel/physical/io/1
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> PUT /api/channel/physical/io/1 HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
> Accept: */*
> Content-Type: application/json
> Content-Length: 11
>
```

```
* upload completely sent off: 11 out of 11 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
< Content-Length: 175
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```


```
{
    "label" : "Channel 1",
    "supported_types" : ["di", "do", "ai", "s0"],
    "type" : "do",
    "enabled" : 1,
    "mode" : "normal",
    "virtual_channel" : 0,
    "level" : "direct",
    "delay_on" : 0,
    "delay_off" : 0,
    "value" : 1
}
```


Configure physical channel 1 as Digitial Input

```
curl -v -X PUT -H "Content-Type: application/json" -d '{"type":"di","mode":"normal
↪"}' http://192.168.178.191/api/channel/physical/io/1
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> PUT /api/channel/physical/io/1 HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
> Accept: */*
> Content-Type: application/json
> Content-Length: 29
>
* upload completely sent off: 29 out of 29 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
< Content-Length: 173
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```


```
{
    "label" : "Channel 1",
    "supported_types" : ["di", "do", "ai", "s0"],
    "type" : "di",
    "enabled" : 1,
    "mode" : "normal",
```

```
    "pullup" : 0,
    "virtual_channel" : 0,
    "level" : "direct",
    "threshold" : 0,
    "value" : 0
}
```

Query state of physical channel 1 (Digitial Input)

In this example, the JSON response should only contain the actual state, but not configuration properties of the channel.

```
curl -v http://192.168.178.191/api/channel/physical/io/1?filter=config
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> GET /api/channel/physical/io/1?filter=config HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
< Content-Length: 11
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```

```
{
    "value" : 1
}
```

Query current readings of physical channel 1 (S0 Input)

In this example, the physical channel 1 is configured as S0 input and the query retrieves the current meter reading.

```
curl -v http://192.168.178.191/api/channel/physical/io/1
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> GET /api/channel/physical/io/1 HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
```

```
< Content-Length: 162
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```


```
{
    "label" : "Channel 1",
    "supported_types" : ["di", "do", "ai", "s0"],
    "type" : "s0",
    "enabled" : 1,
    "pulses_per_unit" : 1000,
    "unit" : "kWh",
    "pulse_counter" : 2173327,
    "value" : 2173.327
}
```


Query current configuration and state of all physical channels

```
curl -v http://192.168.178.191/api/channel/physical/io
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> GET /api/channel/physical/io HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
< Content-Length: 1056
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```


```
{
    "1" : {
        "label" : "Channel 1",
        "supported_types" : ["di", "do", "ai", "s0"],
        "type" : "s0",
        "enabled" : 1,
        "pulses_per_unit" : 1000,
        "unit" : "kWh",
        "pulse_counter" : 2173327,
        "value" : 2173.327
    },
    "2" : {
        "label" : "Channel 2",
        "supported_types" : ["di", "do", "ai", "s0"],
        "type" : "di",
        "enabled" : 1,
        "mode" : "normal",
        "pullup" : 1,
        "virtual_channel" : 0,
```

```
        "level" : "direct",
        "value" : 1
    },
    "3" : {
        "label" : "Channel 3",
        "supported_types" : ["di", "do", "ai", "s0"],
        "type" : "di",
        "enabled" : 1,
        "mode" : "normal",
        "pullup" : 0,
        "virtual_channel" : 0,
        "level" : "direct",
        "threshold" : 0,
        "value" : 0
    },
    "4" : {
        "label" : "Channel 4",
        "supported_types" : ["di", "do", "ai", "s0"],
        "type" : "do",
        "enabled" : 1,
        "mode" : "normal",
        "virtual_channel" : 0,
        "level" : "direct",
        "delay_on" : 0,
        "delay_off" : 0,
        "value" : 0
    },
    "5" : {
        "label" : "Channel 5",
        "supported_types" : ["di", "do", "ai", "s0"],
        "type" : "do",
        "enabled" : 1,
        "mode" : "normal",
        "virtual_channel" : 0,
        "level" : "direct",
        "delay_on" : 0,
        "delay_off" : 0,
        "value" : 0
    },
    "6" : {
        "label" : "Channel 6",
        "supported_types" : ["di", "do", "ai", "s0"],
        "type" : "ai",
        "enabled" : 1,
        "mode" : "0-10 V",
        "virtual_channel" : 0,
        "threshold" : 0,
        "normalized_value" : 0,
        "value" : 0.000,
        "unit" : "V"
    }
}
```


Query only current state of all physical channels

This is an example of an optimized query, where only current state properties are retrieved.

```
curl -v http://192.168.178.191/api/channel/physical/io?filter=config
```

```
* Hostname was NOT found in DNS cache
*   Trying 192.168.178.191...
```

```
* Connected to 192.168.178.191 (192.168.178.191) port 80 (#0)
> GET /api/channel/physical/io?filter=config HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 192.168.178.191
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: lwIP/1.4.1
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Methods: GET, POST, PUT, OPTIONS
< Access-Control-Allow-Headers: Content-Type
< Content-Length: 164
< Content-Type: application/json
< Pragma: no-cache
<
* Closing connection 0
```


```
{
    "1" : {
        "pulse_counter" : 2173327,
        "value" : 2173.327
    },
    "2" : {
        "value" : 1
    },
    "3" : {
        "value" : 0
    },
    "4" : {
        "value" : 0
    },
    "5" : {
        "value" : 0
    },
    "6" : {
        "normalized_value" : 0,
        "value" : 0.000,
        "unit" : "V"
    }
}
```


Python example

The following Python example relies on the Python requests library. It assumes that you already configured a given channel on your XPL Rail device as digitial output and allows to switch such a channel on or off. Intended as example, it does not do much error checking - this is left as an exercise to the reader ;-)

```
#!/usr/bin/env python
#
# Usage: xpl-on-off.py <ip> <channel> [on|off]
#
# Example: $ xpl-on-off.py 192.168.178.191 1 on
#
from __future__ import print_function
import sys
import json
import requests

if len(sys.argv) < 3:
```

```
    print("Usage: %s [-v] <ip> <channel> [on|off]" % (sys.argv[0]), file = sys.
↪stderr)
    sys.exit(2)

params = sys.argv[1:]

verbose = False
if params[0] == '-v':
    params.pop(0)
    verbose = True

host = params.pop(0)
path = "/api/channel/physical/io/%d" % (int(params.pop(0)))

on_off = params.pop(0).lower() == "on"
payload = { 'value': int(on_off) }

url = "http://" + host + path
headers = { 'Content-Type' : 'application/json' }
data = json.dumps(payload)

if verbose:
    print("URL: " + url, file = sys.stderr)
    print("JSON: " + data, file = sys.stderr)

r = requests.put(url, data = data, headers = headers)
try:
    r.raise_for_status()
except requests.exceptions.HTTPError:
    if verbose:
        print("ERROR", file = sys.stderr)
    sys.exit(1)

response = r.json()
if 'result' in response:
    if verbose:
        print("ERROR: %d" % (response['result']), file = sys.stderr)
    sys.exit(1)

if verbose:
    print("OK", file = sys.stderr)

sys.exit(0)
```

PHP example

The following example written in PHP uses a raw socket to connect to an XPL Rail. It assumes that you already configured a given channel on your XPL Rail device as digitial output and allows to switch such a channel on or off. Here too, there is no much error checking and you should really use some library.

```
<?php

    function xpl_on_off($host, $channel, $on_off)
    {
        $body = '{ "value": ' . strval(intval($on_off)) . ' }';

        $header  = "PUT /api/channel/physical/io/" . strval($channel) . " HTTP/1.
↪0\r\n";
        $header .= "Host: $host\r\n";
        $header .= "Content-Type: application/json\r\n";
        $header .= "Content-Length: " . strlen($body) . "\r\n";
```

```
        $header .= "\r\n";

        $fp = @fsockopen($host, 80);
        if ($fp === FALSE)
            return FALSE;

        $rv = fputs($fp, $header . $body);
        if ($rv === FALSE)
        {
            fclose($fp);
            return FALSE;
        }

        $response = "";
        while (!feof($fp))
            $response .= fgets($fp, 10 * 1024);
        fclose($fp);

        // search HTTP body
        $jsonstr = ltrim(strstr($response, "\r\n\r\n"));
        $json = json_decode($jsonstr, TRUE);

        // check for result properties, indicates error
        if (array_key_exists('result', $json))
            return $json['result'];

        // everthing is fine, return success to caller
        return TRUE;
    }

    xpl_on_off('192.168.178.191', 1, FALSE);
```

JavaScript/NodeJS example

The following example is written in JavaScript. You can embedd it within a custom web page, or use it in a NodeJS server environment. It also assumes that you already configured a given channel on your XPL Rail device as digitial output and allows to switch such a channel on or off.

```
#!/usr/bin/nodejs
/*
 * This JavaScript example shows how to build a JSON request to switch a
 * channel on an XPL Rail device on or off.
 * It can be used which NodeJS or within a browser environment. In a NodeJS
 * environment, install 'xhr2' library first, using:
 *  $ npm install xhr2
 * This library emulates the XMLHttpRequest object which is present in browser
 * environments by default.
 * For usage in browsers, you can comment out/remove also the code which is used
 * to pass the command line arguments to the functions.
 */

/* remove the following line when using this example within a browser environment
 ↪*/
var XMLHttpRequest = require('xhr2');

/**
 * A sample callback function which can be used with xpl_request.
 *
 * This callback is called when the request is finished or an error occurred.
 * @param {object} request - the XMLHttpRequest object of the request. which is
 ↪called after the request finished:
```

```
 * @param {object} error - Null if the request was successfull, an Error object otherwise.
 */
function example_callback(request, error)
{
    if (error)
        console.log(error.toString());
}

/**
 * Helper function to send a request to a given XPL Rail.
 *
 * @param {String} method - passed to the XMLHttpRequest object, i.e. 'GET' or 'PUT'
 * @param {String} url - URL on the XPL Rail to access
 * @param {Object} data - a JavaScript object containing data to send, is
 *                        converted to JSON string and send within the HTTP request
 * @param {Function} callback - callback function to be called on success/error
 * @param {Number} timeout - timeout in milliseconds for the request to complete
 * @return {bool} False on error, true otherwise.
 */
function xpl_request(method, url, data, callback, timeout)
{
    var r, r_timeout;

    /* sanitize and check parameters */
    method = method.toUpperCase();
    if (method != "GET" && method != "PUT")
        return false;

    /*
     * Because of a caching bug in IE, we must generate a unique URL for each
     * request, so we simply append a timestamp.
     */
    if (method == "GET")
    {
        /* look for existing url parameters and add required delimiter */
        if (url.search("?") >= 0)
            url += "&";
        else
            url += "?";

        url += "_ts=" + new Date().getTime();
    }

    /* create new request object */
    r = new XMLHttpRequest();

    /* register a callback in case request timeouts */
    r_timeout = setTimeout(function() {
        /* first abort the request */
        r.abort();

        /* then call user-defined callback with error information */
        callback(r, new Error("xpl_request: timeout occured"));
    }, timeout);

    /* register a callback to observer request progress */
    r.onreadystatechange = function() {
        if (r.readyState != 4)
            return;

        /* request finished, so cancel timeout */
```

```
        clearTimeout(r_timeout);

        /* check HTTP status code returned from server and call user-defined callback */
        if (r.status != 200)
            callback(r, new Error("xpl_request: server returned: " + r.status));
        else
            callback(r, null);
    }

    /* open url */
    r.open(method, url, true);

    /* if data is given convert to JSON string representation */
    if (data !== null)
    {
        r.setRequestHeader('Content-Type', 'application/json');
        data = JSON.stringify(data);
    }

    /* send out the request */
    r.send(data);
    return true;
}

/**
 * Helper function to switch a given channel (Digital Output) on a given XPL Rail.
 *
 * @param {String} host - hostname/ip address of the XPL Rail
 * @param {Number} channel - physical channel number to switch
 * @param {bool} on - True switches the channel on, False off.
 * @return {bool} False on error, true otherwise.
 */
function xpl_switch_channel(host, channel, on)
{
    var data = { 'value': on ? 1 : 0 };
    var url = "http://" + host + "/api/channel/physical/io/" + channel;

    return xpl_request("PUT", url, data, example_callback, 5000);
}

/* Remove the following lines when your run this example in a browser environment.
 * In a nodejs environment, it passes the command line arguments to our helper
 * functions.
 */
if (process.argv.length == 5)
{
    xpl_switch_channel(process.argv[2], process.argv[3], process.argv[4] == 'on');
}
else
{
    console.log("Usage: " + process.argv[2] + "<ip-address> <channel> on|off");
}
```

CHAPTER 2

Portal API documentation

Terminology

local network The ethernet / poweline network inside a users home.

local device The Internet of Things device that is installed in a local network.

portal A server in the internet that can be used to get the IPv4 address of the local device.

client A device (like a PC, Smartphone) that wants to connect to the local device.

Problem to address

Devices (local devices) in the local network should get their IPv4 address via DHCP. But if they cannot display this address and the DHCP server does not display it a user cannot access the device without trying addresses. A deterministic method to get the local device IP is needed.

This problem assumes that the device is installed in a typical IPv4 installation:

- the local network is connected to the internet
- between the internet and the local network NAT is used
- the local network has a DHCP server

Concept

registration

- the local device is installed in the local network
- on start it requests a DHCP address
- if this is successfull and a default gateway is provided:
  - connect to the portal JSON API
  - submit the following information

        * (indirectly) public internet IP address of the local network
        * local IP address
        * MAC address
        * device name (freely configurable on the device)
        * device type (product name)
- this is repeated every 15 minutes or whenever the local device address is changed
- the registration is stored on the portal server
- the registration data times out after 20 minutes

list

- a client connects to the portal (via JSON or human readable website) to request the available local devices and supplies the following information
  - (indirect) public internet IP address of the local network
- the server responds with all supplied information in the registration for this public ip address
  - for each registered device for this ip address
    * local IP address
    * MAC address
    * device name
    * device type

JSON interface

Registration

```
URL: http://DOMAIN/api/register/
Method POST
Version 1.0
```

This enables you to register your device to the portal. The key that is looked up in the internal database is your public IP address.

Body arguments

The POST request sends the following data:

- local IP address
- MAC Address
- Device Name
- Device Type

Body example

```
{
    "macaddress":"00:01:87:FF:FF:FF",
    "internalipaddress":"192.168.37.100",
    "devicename":"livingroom",
    "devicetype":"XXXXXXX"
}
```

Response

On success:

```
{
    "result":"success",
    "publicipaddress": "$THEPUBLICIPADDRESS"
}
```

On error:

```
{
   "result":"error",
   "description":"$ERRORDESCRIPTION"
}
```

List

```
URL: http://DOMAIN/api/list/
Method GET
Version 1.0
```

This enables you to get a list of local IPs of all your local devices.

Returns a list of all bridges on the local network and their internal IP addresses.

If there are no local devices on the external IP where the list request comes from then the system will return an empty list, json string: [].

Response data

The response contains the following data:

- local IP address
- MAC Address
- Device Name
- Device Type

Response example

If no device was found, or request failed due to server error:

```
[]
```

If some device are found:

```
[
    {
        "macaddress":"00:01:87:FF:FF:FE",
        "internalipaddress":"192.168.37.100",
        "devicename":"livingroom",
        "devicetype":"XXXXXXX"
    },
    {
        "macaddress":"00:01:87:FF:FF:FF",
        "internalipaddress":"192.168.37.103",
        "devicename":"garage",
        "devicetype":"YYYYYYY"
    }
]
```


HTML interface

The HTML interface can be used by humans to list their devices in the local network.

URL: http://DOMAIN/

The HTML data lists the following data for each local device on the same public IP as the requesting user:

- local IP address
- MAC Address
- Device Name
- Device Type