Miele Professional IP Profile

Technical Specification

CONFIDENTIAL COPY

Contact: Dr. Nils Langhammer, EC/EKE/DPS
Adrian Klingbeil, BIP/RD

Projektbezeichnung	/E-EC/Miele@home/Komponenten
Projektleiter	flaube
Verantwortlich	
Zuletzt geändert	22.09.2021 08:15:39 CEST
Bearbeitungszustand	in Bearbeitung
Dokumenten ID	744705
Dokumentablage	
Template Version	Version 0.4

Inhalt

1
Introduction

1.1

This document describes the technical specification of the Miele Professional IP Profile. The main goal of the Miele IP Profile is to provide a generic, self-descriptive and low-complexity way to interface with Miele professional appliances. The Miele Professional IP Profile is similar to the Miele@home IP Profile for household appliances. It provides a high performance interface that is used by MDU for standard diagnosis, maintanance and validation and also for EndOfLine testing. Besides that it enables Miele Professional appliances to connect to the Miele Cloud Service or to integrate them in laundry service scenarios.

The Miele IP Profile is a so-called RESTful architecture that is used to interact with Miele appliances.

In general, two different frameworks exist for Miele Professional appliances.

Miele Professional IP Profile - Basic
The Basic variant of the Miele IP Profile is derived from to the domestic IP Profile according to [Miele_IP_Profile_Core_Framework]. Details are given in chapter 10.

Miele Professional IP Profile - Extended
The extended (or standard) variant of the Miele IP Profile provides additional services and enhanced security methods.

Different mandatory and optional services are supported. This specification covers the mandatory services for Prof. appliances.

The mandatory services are
- Ident (methods to read the ident label of the Miele *Device*)
- State (representation of the current device state)
- profService (standardized interface for customer service related data)
- profSession (contains User-Login and Device-Pairing sessions handling)

Several optional services are defined that are application-specific. The optional standardized services are covered in [Miele_Professional_IP_Profile_Services].

1.2
Participants and Primary Contacts

1.2.1

Name	Function	Role
Dr. Nils Langhammer	GTE/DPS	Editor/Reviewer, primary contact for general communication aspects
Adrian Klingbeil	BIP/RD	Editor/Reviewer, primary

		contact for professional services, PCS topics, security and session management
Frank Heutger	PS/PML	Reviewer
Dr. Ernst Juhnke	GTZ/ICA	Reviewer
Matthias Klocke	GTZ/ICA	Reviewer
Norbert Conrads	BMP/CSB	Reviewer
Stefan Wöstemeyer	GTZ/CSD	Reviewer
Bernd Mayregger	GTZ/CSD	Reviewer
Jan Mohwinkel	LEP/QM	Reviewer
Holger Müller	GTZ/ICS	Reviewer
Dr. Matthias Köckerling	BIP/RD	Reviewer

1.3

Definitions/Terminology

1.3.1

This subsection contains the definitions of several terms and definitions.

Device: A Miele Professional appliance that resides on a *Host*.

Host: This is an IP addressable entity within a LAN that contains 0..* *Devices*.

Client: An external tool or frontend, which wants to connect or is already connected to the Device.

LAN: This is an abbreviation of local area network.

Miele PCS: Miele Professional Common Software. It's a framework which provides common base functionality for application, webUI, displayUI including session and security issues.

Service: An enclosure or well-defined set of functionality or content.

TLS: This is an abbreviation of transport layer security.

1.4

Relevant Standards and Documents

1.4.1

[RFC4346] TLS 1.1

[RFC5246] TLS 1.2

[RFC6455] The Websocket Protocol (December 2011)

[RFC2616] Hypertext Transfer Protocol – HTTP/1.1

[RFC2818] HTTP Over TLS

[RFC5280] Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

[RFC7231] Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content

[RFC3986] Uniform Resource Identifier (URI): Generic Syntax

[RFC6750] The OAuth 2.0 Authorization Framework: Bearer Token Usage

[Fielding] Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures (April 2013)

[Miele_IP_Profile_Core_Framework] Integrity ID 485504

[Miele_IP_Profile_Services] Integrity ID 486261

[Miele_IP_Profile_Cloud_Connection_MCSv2] Integrity ID 486421

[Miele_Professional_IP_Profile_Services] Integrity ID 782624

[draft-oberstet-hybi-tavendo-wamp-02] The Web Application Messaging Protocol (current version 2)

2

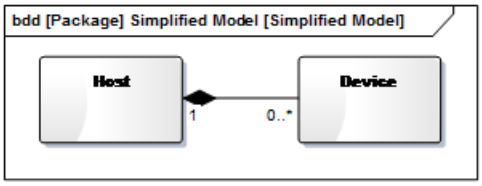
System Architecture

2.1

Fundamental System Topology

2.1.1

The fundamental system topology of a *Host* is shown in the following diagram.



A *Host* contains 0..* *Devices*. A *Device* offers multiple services (the services aspect is not shown in this image).

2.2

Miele Professional IP Profile Protocol Stack

2.2.1

The Miele Professional IP Profile is based on a RESTful architecture. The resulting protocol stack contains several protocols that are used to realize different applications and services.

The following figure shows an abstract and simplified view of the corresponding protocol stack. Furthermore, it shows the parts that are standardized and covered by this specification.

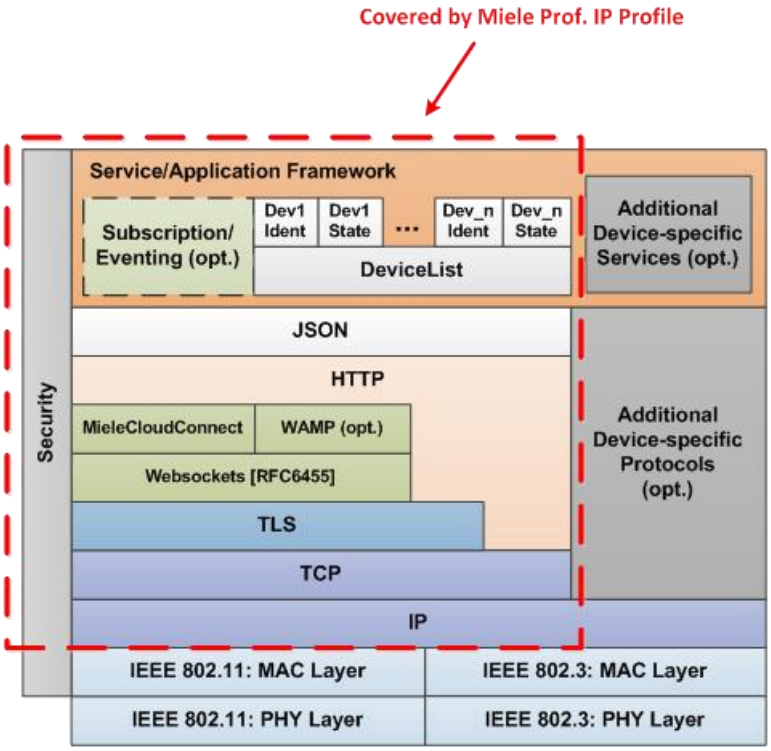


Figure: Fundamental Miele Professional IP Profile protocol stack

The functionalities of the individual communication layers are described within the following chapters of this document. The *Service/Application Framework* contains several mandatory and optional services of a Miele *Device*. The layer denoted as *DeviceList* is an intermediate layer that is used to represent multiple *Devices* on a single *Host*. This can be used by a gateway in order to map *Devices* from other technologies to the Miele Professional IP profile.

All applications, the corresponding *Service Discovery* and all implementations of this standard must fulfill the following requirements:

Prior to any messages being sent to a Device, it is assumed that service discovery has been employed or that services are known a-priori in order to determine service support on other Devices.

2.3

REST Basics

2.3.1

The Miele Prof. IP profile uses a RESTful architecture. Thus, it is based on the HTTP protocol according to [RFC2616]. The following sections describe the supported HTTP verbs, return codes and header fields.

2.3.2

Supported HTTP Verbs

2.3.2.1

The Miele Professional IP Profile supports the following HTTP verbs. The usage is

GET The GET Operation is used to receive a resource. A GET is safe. That means it does not change a resource.

PUT The PUT Operation is used to modify or change a resource.

POST The POST Operation is used to generate a resource. Furthermore, it is used for the transmission of commands that cannot be mapped to a single resource.

DELETE The DELETE Operation is used to delete a resource.

2.3.3

Supported HTTP Return Codes

2.3.3.1

Return codes indicate whether the command was successfully transmitted to the *Host*. In addition to that they describe, if the request has been successfully interpreted. The usage of the return codes complies with the definition given in [RFC2616].

Regarding the Miele Prof. IP-Profile, they do not give information on the results. The results of the requests are returned in the response payload

Code		Description
101	Switching Protocols	This is used by the *Host* in order to indicate that it is willing to comply with the client's request to switch the protocol. This is used for the Websocket cloud connection (see chapter 6).
200	OK	The request has succeeded. The response contains the corresponding payload that depends on the request.
201	Created	The request has been fulfilled and resulted in a new resource being created. The host returns a *Location* header field that contains the address of the new resource.
202	Accepted	The request has been accepted for processing, but the processing has not been completed. This status is returned, if the request requires additional time.
204	No Content	The request has been fulfilled but no entity body is returned.
307	Temporary Redirect	The requested resource is temporarily available under a different location that is given in the *Location* header. The client should try to use the original address in future requests.
400	Bad Request	The request could not be understood due to malformed syntax. The client should not repeat the request without modifications.
401	Unauthorized	The request requires user authentication.
403	Forbidden	The request has been understood but it is not fulfilled as the client does not have the corresponding rights. This is described in chapter 5.
404	Not Found	The address given in the request does not exist.
405	Method Not Allowed	The method specified in the Request-Line is not allowed for the resource identified by the Request-URI. The response contains the *Allow* header field that describes the supported HTTP verbs. *Example:* *Allow: GET, PUT*
406	Not Acceptable	This is used to indicate that a server is unable to generate a response according to the requirements sent in the *Accept* header field (e.g. unsupported version number).
409	Conflict	The 409 (Conflict) status code indicates that the request could not be completed due to a conflict with the current state of the target resource.
411	Length Required	This return code is may be returned by a *Host*, if a PUT or POST operation is performed without a Content-Length header field.
429	Too Many Requests	This return code may be returned by a *Device*
500	Internal Server Error	This is used to indicate that the *Host* has an internal error.
501	Not Implemented	This response is generated, if the client tries to use an unsupported HTTP method or verb.
503	Service Unavailable	The server is currently unable to handle the request

		due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay. If known, the length of the delay MAY be indicated in a Retry-After header. If no Retry-After is given, the client SHOULD handle the response as it would for a 500 response.
504	Gateway Timeout	This return code is used by the *Host*, if it acts as a gateway for a different technology. The usage of this response code indicates that the *Host* did not receive a timely response.

Table: Supported HTTP Status Codes

2.3.4

Supported HTTP Header Fields

2.3.4.1

The following table contains an overview on the HTTP/1.1 header fields that must be supported by all *Hosts*.
Note: A Host can support all header fields according to [RFC2616], this is only the minimum mandatory subset.

Field Name	Request	Response	Comment
Host	X		Mandatory according to HTTP/1.1. It can be either set to the <*Host Name*> or to an IP-address/port combination according to HTTP/1.1.
Accept	X		Mandatory.
Content-type	X	X	Always present in responses that have an entity body. In addition to that, it is required in PUT/POST operations.
Content-length	X	X	Always present in responses that have an entity body. In addition to that, it is required in PUT/POST operations. It can be omitted for Transfer-Encoding chunked.
Date	X	X	Given in UTC

Table: Mandatory HTTP Header Fields

2.3.4.2

Behaviour upon receipt of an unsupported Header Field

2.3.4.2.1

If a *Host* receives an unsupported/unknown header field and/or HTTP verb, the following procedure is applied:

1. The *Host* checks, if the request can be executed, if the header field is ignored.

2. In case it can be executed, the *Host* ignores the unsupported/unknown value(s) and returns the regular result that would have been generated in absence of the unsupported/unknown value(s)

3. In case the request cannot be executed, the host responds with a status 501 (“Not Implemented”)

2.3.5

HTTP Request-Pipelining

2.3.5.1

HTTP Request-Pipelining can be used in order to reduce latencies.

The maximum supported number of pipelined requests should be 3. If a smaller number is only possible for a certain implementation this shall be negotiated with the editors of this document.

2.3.6

API Versioning and Media Type

2.3.6.1

The *Media Type* is used for versioning of the REST-API. It is returned in the Accept and Content-Type Header-fields. Thus, client and server are able to perform content negotiation based on the values given in these fields.

The *Media Type* always has the following form:

<*Media Type*>:= application/vnd.miele.v[*version_number*]+json; charset=utf-8

The [*version_number*] is set to the supported version of the Miele Professional IP specification. For this version of the specification it equals

[*version_number*] := 1

Thus, the complete *Media Type* for this specification is

***<Media Type>*:= application/vnd.miele.v1+json; charset=utf-8**

Note: This Media Type has been officially registered by the IANA.

In the case a product follows this specification but is not Miele branded, it is not allowed to use the official (IANA) Miele Media Type. Such implementations MUST use "application/json; charset=utf-8".

2.4

General Resource Structure

2.4.1

All resources are organized in a tree structure according to the general principles of a RESTful architecture. URLs and URIs are used to address the resources. All *Hosts* employ the same structure.

<table>
<tr><td rowspan="5">/</td><td><object:Root></td><td></td><td></td><td></td></tr>
<tr><td rowspan="4">Devices/</td><td><object:DeviceList></td><td></td><td></td></tr>
<tr><td rowspan="3"><id:Device>/</td><td><object:DeviceCapabilities></td><td></td></tr>
<tr><td rowspan="2"><id:Service>/</td><td rowspan="2"><..service-spec..></td></tr>
<tr></tr>
</table>

Table General resource structure

Starting point of the REST-API is <*object:Root*>. The next level contains the <*object:DeviceList*>. The <*object:DeviceList*> contains the representation of the *Devices* that can be accessed via this *Host*. Details are given in chapter 3. There can be additional optional resources located on <object:Root>.

The second level <*object:DeviceCapabilities*> contains information about the services that are supported by a certain *Device*. The URL is /Devices/<*id:Device*>/. Finally, the subsequent levels contain service or application specific information.

The following table contains the exemplary resource structure of a *Host* with the ID 1234567890. This *Host* supports the mandatory services *Ident*, *State*, *profErrorList, profErrorLog*, *profOperatingData* and the optional services *profProcessData and profDisinfect*.

/			
/	Devices/		
/	Devices/	1234567890/	
/	Devices/	1234567890/	Ident/
/	Devices/	1234567890/	State/
/	Devices/	1234567890/	profErrorList/
/	Devices/	1234567890/	profErrorLog/
/	Devices/	1234567890/	profOperatingData/
/	Devices/	1234567890/	profSession/
/	Devices/	1234567890/	profProcessData/
/	Devices/	1234567890/	profDisinfect/

Table: Resource structure of an exemplary Host

2.5

General Data Structures

2.5.1

This section contains a description of the general json object types.

2.5.2

<attribute:ReturnCode>

2.5.2.1

The <*attribute:ReturnCode*> represents a return code. The structure is as follows:

<*attribute:ReturnCode*> := Success | Failure

(Example: Success)

2.5.3

<attribute:bool>

2.5.3.1

The <*attribute:bool*> represents a boolean state. The structure is as follows:

<*attribute:bool*> := true | false

(Example: true)

2.5.4

<command:http>

2.5.4.1

The <*command:http*> represents a single http verb. The structure is as follows

<*command:http*> := GET | PUT | POST | DELETE

(Example: GET)

2.5.5

<attribute:url>

2.5.5.1

The <*attribute:url*> represents an URL. The structure is as follows

<*attribute:url*> := http[s]://<Host name>[:<port>]/[<path>]

(Example: https://hostname:443/callback/)

The usage of the trailing slash is optional. Therefore, the URLs "/test" and "/test/" are treated equally by all *Hosts*.

"href"-URLs can be absolute (leading slash /<name>) or relative (without leading slash <name>).

This is not valid for URLs that contain an ?-Operater, e.g. *data?idx1=2*, in this case the trailing / is not supported.

3

Host Structure and Service Discovery

3.1

General Functionality

3.1.1

The Service-Discovery is performed in several steps:

1. It is assumed that the IP address of the *Host* is known or it is resolved via the <*Host Name*>.
2. GET the <*object:Root*> from the *Host*.
3. GET <*object:DeviceList*> from <host>/Devices/
4. GET the <*object:DeviceCapabilities*> of all *Devices*
5. GET the relevant data <*objects*> of the services that have been discovered during step 4).

3.1.2

Host Name

3.1.2.1

The <*Host Name*> is an identifier for the Miele Professional *Host*. The <*Host Name*> is limited to a maximum length of 63 bytes.

Regarding the usage of the *Host* header field in HTTP, please refer to the statement in chapter 2.3.

The default <*Host Name*> is generated automatically from the MAC-address.
The ASCII representation of the MAC-address is used to generate the <*Host Name*>. The <*Host Name*> has the following form:

<*Host Name*>:= “Miele-“ +<*ASCII representation of MAC address without colons*>

Example 1: <*Host Name*> of a device with a 6-byte MAC address (D3:BE:09:C6:D1:F0):

“Miele-D3BE09C6D1F0”

Example 2: <*Host Name*> of a device with an 8-byte MAC address (D1:D0:D3:BE:09:C6:D1:F0):

“Miele-D1D0D3BE09C6D1F0”

The default <*Host Name*> can be changed by the customer later.
This aspect is device specific and it is not covered by this specification. E.g. an implementer of this document *may* decide to implement such a change in the UI.

3.2

GET <object:Root>

3.2.1

The <*object:Root*> object is returned, if a GET request is performed on the root URL of a *Host* with the *Accept* field set to *application/vnd.miele.v1+json*.

URL	http://<host name>/
HTTP Request	GET / HTTP/1.1 Host: <*host name*> Accept: application/vnd.miele.v1+json
HTTP Request Payload	*empty*

HTTP Response Header	HTTP/1.1 200 OK Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>

HTTP Response Payload :=
 	
<*object:Root*>
	{
	“Devices”: <*object:DeviceListLink*>,
	[“Subscriptions”: <*object:SubscriptionListLink*>,] (optional)
	“Host”: <host name according to 3.1.2>,
	“Info” : “<*additional information as txt, can be empty*>”
	}

3.2.2

<object:DeviceListLink>

3.2.2.1

The <*object:DeviceListLink*> contains the relative link to the *Device* list of a Miele Prof. device. Currently, the link always points to “/Devices/”.

<*object:DeviceListLink*> := {“href” : “Devices/”}

3.2.3

<object:Root>.Host

3.2.3.1

The *Host* field contains the hostname. The datatype is string.

3.2.4

<object:Root>.Info

3.2.4.1

The *Info* field contains unstructured data formatted as TXT. The datatype is string. It can be used to provide additional information (e.g. for debugging, extended versioning, etc.). A client implementation MUST NOT rely on information returned in this field.

3.2.5

<object:SubscriptionListLink> (optional)

3.2.5.1

The <*object:SubscriptionListLink*> contains the relative link to the subscription and eventing service. It is only present, if the Miele Prof. *Host* supports the optional subscription and eventing mechanism according to chapter 7.3.

<*object:SubscriptionListLink*> := {“href”: “Subscriptions/”}

3.2.6

Example: Root directory of a host

3.2.6.1

This example shows the root directory of a generic *Host* with the *Host* name Miele-D3BE09C6D1F0. The information is retrieved by issuing a GET to “http://Miele-D3BE09C6D1F0/”.

```
GET / HTTP/1.1
Host: Miele-D3BE09C6D1F0
Accept: application/vnd.miele.v1+json
```

The *Host* replies with its root object. In this example, the Host supports the subscription and eventing mechanism.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
“Devices”: {“href”: “Devices/”},
“Subscriptions”: {“href”: “Subscriptions/”},
“Host”: “Miele-D3BE09C6D1F0”
“Info”: “Miele xxxxxxx v1.0.1.2”
}
```

3.3

GET <object:DeviceList>

3.3.1

The <*object:DeviceList*> object is returned, if a GET request is performed on the URL that is returned in <*object:DeviceListLink*> with the *Accept* field set to *application/vnd.miele.v1+json*.

URL	http://<host name>/Devices/
HTTP Request	GET /Devices/ HTTP/1.1 Host: <*host name*> Accept: application/vnd.miele.v1+json
HTTP Request Payload	*empty*
HTTP Response Header	HTTP/1.1 200 OK Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>

HTTP Response Payload :=
 	
<*object:DeviceList*>
	{
	<*id:Device*>: <*object:Device*>,
	<*id:Device*>: <*object:Device*>,
	…
	}

The <*object:DeviceList*> contains the links to the underlying *Devices*. The list can contain a single or multiple <*object:Device*> pairs.

3.3.2

<id:Device>

3.3.2.1

The <*id:Device*> tag contains a unique ID for a Miele *Device*. It equals the <*object:IdentLabel*>*.FabNumber* (see chapter 4). The structure is always as follows:

<*id:Device*> := <unique 12-byte ID>

(Example: 123456789101)

If <*object:IdentLabel*>*.FabNumber* is empty, not existing or invalid (a valid *FabNumber* contains always ASCII encoded numbers), the MAC address of the device communication module shall be used in ASCII representation with “mac-“ before the address.

(Example: mac-f830c992dd00ffff)

3.3.3

<object:Device>

3.3.3.1

The *<object:Device>* contains information about an underlying *Device*.

<object:Device> :=
{
<id:DeviceFieldName>: *<id:DeviceFieldValue>*,
<id:DeviceFieldName>: *<id:DeviceFieldValue>*,
… }

3.3.4

<object:Device> properties

3.3.4.1

<id:Device FieldName>	Datatype	Read(R) Write(W)	Description
href	string	R	Relative link to the *Device* capabilities
Group	string	R	ASCII string set to fixed value “Professional”

Table: Fields used in the <object:Device>

3.3.4.2

<object:Device>.href

3.3.4.2.1

The *href* field contains the ASCII encoded relative link to the *Device* capabilities (see chapter 3.4) of the device. The data type is string and the value is set to “<id:Device>/”. This property of the *<object:Device>* is read-only and cannot be modified.

3.3.4.3

<object:Device>.Group

3.3.4.3.1
The *Group* field contains the ASCII encoded fixed value “Professional”. The data type is string. This property of the <object:Device> is read-only and cannot be modified.

Example 1: “Group”: “Professional”

3.3.5

Example 1: Device List with a standalone device

3.3.5.1
This example shows the *Device* list of a host with the name “HostNameDev". The information is retrieved by issuing a GET to “http://HostNameDev/Devices/”.

```
GET /Devices/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Host* replies with the *Device* list that contains only a single *Device*.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
“123456789101”: {
            “href”:”123456789101/”,
            “Group”: “Professional”
             }
}
```

3.3.6

Example 2: Device list with multiple devices

3.3.6.1
This example shows the *Device* list of a *Host* with multiple *Devices* with the name “HostNameDev". The information is retrieved by issuing a GET to “http://HostNameDev/Devices/”.

```
GET /Devices/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Host* replies with the *Device* list that contains two *Devices*.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
“123456789101”: {
            “href”:”123456789101/”,
            “Group”: “Professional”
           },
“423456739121”: {
            “href”:”423456739121/”,
            “Group”: “Professional”
           }
}
```

3.3.7

Example 3: Device List with a standalone device without fabrication number

3.3.7.1

This example shows the *Device* list of a *Host* with the name “HostNameDev". The information is retrieved by issuing a GET to “http://HostNameDev/Devices/”.

```
GET /Devices/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Host* replies with the *Device* list that contains only a single *Device*.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
“mac-f830c992dd00ffff”: {
            “href”:”mac-f830c992dd00ffff/”,
            “Group”: “Professional”
             }
}
```

3.4

GET <object:DeviceCapabilities>

3.4.1

The <*object:DeviceCapabilities*> object is returned, if a GET request is performed on the URL that is given in the <*object:Device*>.href with the *Accept* field set to *application/vnd.miele.v1+json*.

URL	http://<host name>/Devices/<*id:Device*>/
HTTP Request	GET /Devices/<id:Device>/ HTTP/1.1 Host: <*host name*> Accept: application/vnd.miele.v1+json
HTTP Request Payload	*empty*
HTTP Response Header	HTTP/1.1 200 OK Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>

HTTP Response Payload :=
 	
<*object:DeviceCapabilities*>
	{
	<*id:Service*>: <*object:ServiceLink*>,
	<*id:Service*>: <*object:ServiceLink*>,
	…
	}

The <*object:DeviceCapabilities*> contains the links to the underlying services of a *Device*. The list can contain multiple <*id:Service*>: <*object:ServiceLink*> pairs.

3.4.2

<id:Service>

3.4.2.1

The <*id:Service*> field name contains the ASCII encoded name of a single service. The data type is string. The possible <*id:Service*> values for the mandatory services can be found in chapter 4.

Example: “State”

The professional specific services always use a prepended “prof” in their name.

Example: “profDisinfect”

3.4.3

<object:ServiceLink>

3.4.3.1

The <*object:ServiceLink*> contains the relative link to a service of a Miele *Device*. The link always points to "<*id:service*>/".

<*object:ServiceLink*> := {"href" : "<id:service>/"}

3.4.4

Example: Device Capabilities of a standalone device

3.4.4.1

This example shows the *Device* capabilities of a *Device* with the ID "123456789101". This device resides on the *Host* with the name "HostNameDev". The *Device* supports the *Ident*, *State* and *profDisinfect* services.

The information is retrieved by issuing a GET to "http://HostNameDev/Devices/123456789101/".

```
GET /Devices/123456789101/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Host* replies with the *Device* capabilities list that contains the relative links to the supported services of the *Device*.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
  "Ident": {"href": "Ident/"},
  "State": {"href": "State/"},
  "profService": {"href": "profService/"},
"profSession": {"href": "profSession/"}
}
```

The *Ident* and *State* resource MUST always point to "href": "Ident/" and "href": "State/". It is NOT ALLOWED to re-direct them to another URL.

The other resources are allowed to return any URL on this Host. An application must follow this URL including all additional attributes.

4
Mandatory Services

4.1
Miele Professional *Devices* support a different number of services and applications. The services can be separated into mandatory and optional services. The number of mandatory and optional services depends on the *Device* type.

All URLs that are returned with a *href* field MUST be set to a fixed value for a certain SWID. They MUST NOT change dynamically.

<id:Service>	Mandatory	Optional	Description
Ident	X		View and modify general information about the *Device* (e.g. the device type, device name, etc.)
State	X		View and modify the current *Device* state
profService	X		Customer service relevant service
profSession	X		User-Login and Device-Pairing sessions handling

Table: Mandatory Services

4.2

Service: Device Ident

4.2.1
The *Device* ident service is used to view and modify general information of a Miele *Device*. The ident can be retrieved by issuing a GET request to the URL of the *Device* ident service. Furthermore, it is possible to modify selected parameters of the *Device* ident (e.g. the *Device* name) with the PUT operator.

4.2.2

GET <object:DeviceIdent>

4.2.2.1

The *<object: DeviceIdent>* object is returned, if a GET request is performed on the corresponding URL of the service with the *Accept* field set to *application/vnd.miele.v1+json*.

URL	**http://<host name>/Devices/*<id:Device>*/Ident/**
HTTP Request	GET /Devices/<id:Device>/Ident/ HTTP/1.1 Host: <*host name*> Accept: application/vnd.miele.v1+json
HTTP Request Payload	*empty*
HTTP Response Header	HTTP/1.1 200 OK Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>

HTTP Response Payload :=
 	
<*object:DeviceIdent*>
	{
	<*id:DeviceIdentFieldName*>: <*id:DeviceIdentFieldValue*>,
	<*id:DeviceIdentFieldName*>: <*id:DeviceIdentFieldValue*>,
	…
	“DeviceIdentLabel”: <*object:IdentLabel*>,
	[“XKMIdentLabel”: <*object:IdentLabel*>]
	}

The *<object:DeviceIdent>* contains information about the *Device* type, the *Device* ident label and the XKM ident label. Thus, it contains several fields, whereas every field is a single key value pair.

4.2.2.2

<object:IdentLabel>

4.2.2.2.1

The <*object:IdentLabel*> contains the ident label of a *Device* or an XKM module.

<*object:IdentLabel*> :=

{
  <*id:IdentLabelName*>: <*id:IdentLabelValue*>,
  <*id:IdentLabelName*>: <*id:IdentLabelValue*>,
  …
}

4.2.3

PUT <object:DeviceIdent>

4.2.3.1

The usage of PUT is only possible for the customer service to write the ident of the device (e.g. if necessary according to external requirements).

The writeable field values of <*object: DeviceIdent*> can be modified with a PUT request in order to change the *Device* state. One or multiple fields can be modified at once. The HTTP response contains information about the results of all operations.

URL	**http://<host name>/Devices/<*id:Device*>/Ident/**
HTTP Request	PUT /Devices/<id:Device>/Ident/ HTTP/1.1 Host: <host name> Content-Type: application/vnd.miele.v1+json Content-Length: <*decimal number of octets*>

HTTP Request Payload	{
	<*id:DeviceIdentFieldName*>: <*id:DeviceIdentFieldValue*>,
	<*id:DeviceIdentFieldName*>: <*id:DeviceIdentFieldValue*>,
	…
	}

HTTP Response Header	HTTP/1.1 200 OK Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>
HTTP Response Payload	[{"<*attribute:ReturnCode*>": {<*id:DeviceIdentFieldName*>: <*id:DeviceIdentFieldValue*>}}, {"<*attribute:ReturnCode*>": {<*id:DeviceIdentFieldName*>: <*id:DeviceIdentFieldValue*>}}, …

	]

Example: Set *DeviceName* to “Machine ABCD”

Perform a PUT to “http://<host name>/Devices/<id:Device>/Ident/” with the request payload
{
“DeviceName”: “Machine ABCD”
}

If the operation is successful, the response payload contains the results
{
“Success”: {“DeviceName”: “Machine ABCD”}
}

If the operation fails, the response payload contains the unmodified value
{
“Failure“: {“DeviceName”: “PreviousValue”}
}

The same happens, if one tries to modify a field that is read-only.

4.2.4

<object:DeviceIdent> properties

4.2.4.1

<id:DeviceIdent FieldName>	Datatype	Read(R) Write(W)	Description
DeviceType	uint16	R	Miele definition of the *Device* type
DeviceName	string	R/W	Devicename as UTF-8 encoded string
ProtocolVersion	uint8	R	Internal protocol version of the *Device*.
DeviceIdentLabel	<*object:IdentLabel*>	R / (W)	Ident label of the device
XKMIdentLabel	<*object:IdentLabel*>	R / (W)	Ident label of the communication module (optional)
pCommDate	string	R	Commissioning date (ISO8601) of the *Device.*
Optional			
pSystemLanguage	string	R	Country and language of the *Device*.
pRevisionList	array	R	Revisions of all electronics

Table: Fields used in the <object:DeviceIdent>

4.2.4.2

<object:DeviceIdent>.DeviceType

4.2.4.2.1

The *DeviceType* field of the *Device* state object equals the Miele Definition of the *Device* type. The datatype is unsigned 16b Integer. This property of the device ident is read-only and cannot be modified.

The following table shows the supported values for Professional *Devices*.

If a new device is developed according to this specification that does not fit into the types listed into the table, the project team shall contact the editor that is named in chapter 1.2. Necessary changes will then be appended to this specification. This process ensures compatibility to the involved system blocks (Miele Cloud Service, etc.)

Miele DeviceType	Short	Description German	Description English
3		Waschmaschine Semi-Professional	washing machine semi-professional
4		Wäschetrockner Semi-Professional	tumble dryer semi-professional
5		Waschmaschine Professional	washing machine professional
6		Wäschetrockner Professional	tumble dryer professional
8		Geschirrspüler Semi-Professional	dishwasher semi-professional
9		Geschirrspüler Professional	dishwasher professional
44		Reinigungs- und Desin-fektionsautomat	
51			SemiProf Washer (DEPRECATED)
52			SemiProf Dryer (DEPRECATED)
53			Profi-M Washer (DEPRECATED)
54			Profi-M Dryer (DEPRECATED)
60	RDG Labor	Reinigungs- und Desin-fektionsgerät Labor	
61	RDG Medizin	Reinigungs- und Desin-fektionsgerät Medizin	
62	ETD		Endo Thermal Disinfector
63	TTS		Table Top Sterilizer

64	DGS	Dampf-Groß-Sterilisator	
65	CTWA	Container- und Trolley-Waschanlage	
66	DGD	Dampf-Groß-Desinfektor	

Table: Miele Professional *Device* types and IDs

4.2.4.3

<object:DeviceIdent>.DeviceName

4.2.4.3.1

The *DeviceName* field contains the UTF-8 encoded device name. The DeviceName attribute has a maximum length of 61 Byte (20 characters+length). The data type is string. This property of the device ident is read- and writeable and can thus be modified. It can be set to a custom name.

4.2.4.4

<object:DeviceIdent>.ProtocolVersion

4.2.4.4.1

The ProtocolVersion field contains information about the internal Miele protocol version of the *Device*. The data type is unsigned 8b Integer. The property of the ident is read-only and cannot be modified. The following table contains the possible values:

Value	Description
0	No standardized Miele protocol
1	UART (Opcode85)
2	MBus DOP
3	MBus DOP2
4	HDR DOP2
200	DBUS Cooling (OEM GTO)
201	ToD Cooling (OEM GTO)
202	USB DOP2
203	LHB TOD2 Cooling (OEM GTO)
Remaining	Reserved

Table: ProtocolVersion values

4.2.4.5

<object:DeviceIdent>. pCommDate

4.2.4.5.1
The *pCommDate* field of the *Device* ident object contains the timestamp of the *Device* commissioning. The datatype is string. This property of the *Device* ident is read-only and cannot be modified. If no commissioning date has been set, the field is set to "" (empty string).

Example: “pCommDate”: “2013-10-12“

4.2.4.6

<object:DeviceIdent>. pSystemLanguage (optional)

4.2.4.6.1
The *pSystemLanguage* field contains the country and language setting of the *Device*. The data type is string. This property of the *Device* ident is read-only and cannot be modified.
Example: Country Germany, language German

“pSystemLanguage”: “de-DE”

4.2.4.7

<object:DeviceIdent>.pRevisionList (optional)

4.2.4.7.1
The *pRevisionList* field contains information about the revisions of all electronics and their software versions.
This field is optional and thus it may be omitted by the Device. The SWIDs in DeviceIdentLabel.SWIDs must be filled with information returned in *pRevisionList*.

The data type is array<object>. This property of the ident is read-only and cannot be modified.

The structure is
“pRevisionList”: [
{“Name”: “HMI”, “Instance”: 1, “Type”: 1, “Version”: “V2.1_B0”, “Bootloader”: “V1.2”, “MatNumber”: “123456”, “SerialNumber”: “12345”},
{“Name”: “SLT”, “Instance”: 1, “Type”: 2, “Version”: “V1.7_B0” , “Bootloader”: “V1.2”, “SWID”: “3435”},
….
]

Name: Describes the name of the electronic
Instance: Describes the instance number (used for identical hardware parts) starting with 1
Type: Contains the type
1: Hardware,
2: Software
Version: Contains the version of the firmware
Bootloader: Contains the version of the bootloader
MatNumber (optional): Contains the material number (only existing for Type=Hardware)
SerialNumber (optional): Contains the serial number that is also printed on the barcode (only existing for Type=Hardware)
SWID (optional): Contains the official software ID (only existing for Type=Software)

4.2.5

<object:IdentLabel> properties

4.2.5.1

The <object:IdentLabel> contains the ident label of a Miele *Device* or a communication module.
If the field/object cannot be filled for a certain device (e.g. no FabIndex known, the corresponding fields shall be set to empty "" values.

<id:DeviceIdent FieldName>	Datatype	Read(R) Write(W)	Description
Version	string	R	ASCII string version of the *Device* label ("E")
FabNumber	string	R / (W)	ASCII string FABRIKATIONSNUMMER
FabIndex	string	R / (W)	ASCII string FABRIKATIONSINDEX
TechType	string	R / (W)	ASCII string TECHNISCHER_TYP
MatNumber	string	R / (W)	ASCII string MATERIALNUMMER
SWIDs	array<uint16>	R	Array containing the software IDs.
ReleaseVersion (optional)	string	R	ASCII string release version. This field only exits for XKMIdentLabel
ReleaseSuffix	string	W	ASCII string that can be used to modify the ReleaseSuffix. This field only exists for XKMIdentLabel.

Table: Fields used in the <object:IdentLabel>

4.2.5.2

<object:IdentLabel>.Version

4.2.5.2.1

The *Version* field contains the ASCII encoded version of the device label. The data type is string and currently the value is set to fixed value “E”. This property of the ident label is read-only and cannot be modified.

4.2.5.3

<object:IdentLabel>.FabNumber

4.2.5.3.1
The *FabNumber* field contains the ASCII encoded FABRIKATIONSNUMMER of the *Device* or communication module. The data type is string.

Example: “FabNumber”: “012345678911”

4.2.5.4

<object:IdentLabel>.FabIndex

4.2.5.4.1
The *FabIndex* field contains the ASCII encoded FABRIKATIONSINDEX of the device or module. The data type is string.

Example: “FabIndex”: “01”

If no *FabIndex* is given for the Device, the field is set to fixed value “FabIndex”: “”.

4.2.5.5

<object:IdentLabel>.TechType

4.2.5.5.1
The *TechType* field contains the ASCII encoded TECHNISCHER_TYP of the *Device* or module. The data type is string.

Example: “TechType”: “CVA6000”

If no *TechType* is given for the Device, the field is set to fixed value “Techtype”: “”.

4.2.5.6

<object:IdentLabel>.MatNumber

4.2.5.6.1
The *MatNumber* field contains the ASCII encoded MATERIALNUMMER of the device or module. The data type is string.

Example: "MatNumber": "07654321"

If no *MatNumber* is given for the Device, the field is set to fixed value "MatNumber": "".

4.2.5.7

<object:IdentLabel>.SWIDs

4.2.5.7.1
The *SWIDs* field contains the software IDs of the *Device* or communication module. The software IDs are returned in an array that contains none or multiple unsigned 16b Integer values. The number of software IDs is device specific. This property of the *Device* state is read-only and cannot be modified.

Example: "SWIDs": [1234, 5678, 1122, 3344]

The usage of this field is mandatory. If the field *pRevisionList* is used, the SWIDs are taken from the corresponding fields *SWID* in *pRevisionList* .

4.2.5.8

<object:IdentLabel>.ReleaseVersion

4.2.5.8.1
The *ReleaseVersion* field contains the release version of the communication module. This field is optional, and it is only present if the <*object:IdentLabel*> represents an "*XMKIdentLabel*". The data type is string and the property is read-only and canot be modified.

Example: "ReleaseVersion": "00.51"

4.2.5.9

<object:IdentLabel>.ReleaseSuffix

4.2.5.9.1

Note. This not yet implemented (new feature)!

The *ReleaseSuffix* field can be used to modify the suffix of the release version of the communication module. This field is optional, and it is only present if the <*object:IdentLabel*> represents an "*XMKIdentLabel*". The data type is string and the property is **write-only**, it can only be modified via the *Miele Cloud Service* connection and locally with a user with admin rights.

It is used to switch the communication module between productive and quality development paths remotely. This functionality is similar to the OIF that is used to write the *ReleaseVersion* field.

Only a fixed set of 3 values can be send with a PUT operation:

PUT "ReleaseSuffix": "QEC"
PUT "ReleaseSuffix": "QXX"
PUT "ReleaseSuffix": "PXX"

other values are not evaluated by the communication module.

If the field is written, the suffix within the field *ReleaseVersion* is updated according to the written value. The field ReleaseSuffix is not returned via a GET or sent via an event.

Example: "ReleaseSuffix": "QXX" (switch the module to the current Q development path, then a remote update can be performed)

4.2.5.10

<object:IdentLabel>.SWBuildTime

4.2.5.10.1

Note. This not yet implemented (new feature)!

The *SWBuildTime* is used to reflect the time when the SW has been built for this module.

This field is optional and is only present if the <*object:IdentLabel*> represents an "*XMKIdentLabel*". Its property is **read-only**. It is readable via the *Miele Cloud Service* connection.

The data type is array, which contains six integer fields to represent date and time: [d,m,yyyy,h,m,s]

Example:
If the SW has been build on Feb, 29th 2020 at 9:06:02 o'clock,
the SWBuildTime array reflects the time as follows [29, 2, 2020, 9, 6, 2].

4.2.6

GET single field/value property from <object:DeviceIdent> (optional)

4.2.6.1

Note: This feature is optional and may not be supported by all Hosts!

Instead of performing a GET in order to receive the entire *<object:DeviceIdent>*, it is also possible to receive single fields/properties of the object.

Furthermore, it is also possible to receive the entire *<object:IdentLabel>* of the *DeviceIdentLabel* or parts of it.

Example 1: If a GET is performed on

http://<host name>/Devices/<id:Device>/Ident/DeviceType/

the *DeviceType* field is returned

```
{
  “DeviceType”: 17
}
```

Example 2: If a GET is performed on

http://<host name>/Devices/<id:Device>/Ident/DeviceIdentLabel/

the *<object:IdentLabel>* for the *DeviceIdentLabel* is returned

```
{
  “Version”: “E”,
  “FabNumber”: “012345678911”,
  “FabIndex”: “01”,
  “TechType”: “CVA6000”,
  “MatNumber”: “07654321”,
  “SWIDs”: [1234, 5678, 1122, 3344]
}
```

4.2.7

PUT single field/value property from <object:DeviceIdent> (optional)

4.2.7.1

Note: This feature is optional and may not be supported by all Hosts!

Instead of performing a PUT on the entire *<object:DeviceIdent>*, it is also possible to modify single fields/properties of the object.

Example: A PUT can performed on

http://<host name>/Devices/<id:Device>/Ident/DeviceName/

with the request payload

```
{
    “DeviceName”: “MyNewDevice”
}
```

in order to change the device name. In this example, the result is identical to a put operation to

http://<host name>/Devices/<id:Device>/Ident/

with the request payload

```
{
    “DeviceName”: “MyNewDevice”
}
```

The difference is that only properties of the resources that are given in the URL can be modified.

4.2.8
Write Ident

4.2.8.1
The *DeviceIdent* can be written by the customer service technician. The local diagnosis according to chapter 8 must have been successfully activated and the MDU must be connected.

The MDU can write the *DeviceIdent* via a PUT operation

```
PUT /Devices/<id>/Ident/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Content-Type: application/vnd.miele.v1+json
Content-Length: <decimal number of octets>

{
```

```
        “DeviceName“: “Name given by customer“,
        “DeviceIdentLabel”: {
                “Version”: “E”,
                “FabNumber”: “012345678911”,
                “FabIndex”: “”,
                “TechType”: “PS5662v”,
                “MatNumber”: “07654321”
        }
}
```

The *Version* is not written, but it is validated by the *Device*. If it is different from “E”, the request is discarded.

It is only possible to write the entire *DeviceIdentLabel*. It is not possible to write single parts.

Note: DeviceName and DeviceIdentLabel can be written seperately.

The *Device* checks the information. If everything is successfully written, the device responds with a 200.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
        { “Success”: { “DeviceName“: “Name given by customer“ }},
        { “Success”: { “DeviceIdentLabel”:
                                { “Version”: “E”, “FabNumber”: “012345678911”, “FabIndex”:
        “”,
                                  “TechType”: “PS5662v”, “MatNumber”: “07654321” } }
        }
]
```

The *DeviceName* can be set by the customer. The *DeviceIdent* can only be written with dedicated MDU user rights.

4.2.9

Example: GET device ident of a generic device

4.2.9.1

This example shows the ident of a *Device* with the ID “123456789101”. This *Device* resides on the *Host* with the name “HostNameDev". The information is retrieved by issuing a GET to “http://HostNameDev/Devices/123456789101/Ident/”.

```
GET /Devices/123456789101/Ident/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* replies with its ident.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        “DeviceType”: 17,
        “DeviceName”: “My Device Name”,
        “pSystemLanguage”: “de-DE”,
        “ProtocolVersion”: 2,
        “pCommDate”: “2013-10-12“,
        “DeviceIdentLabel”: {
                “Version”: “E”,
                 “FabNumber”: “012345678911”,
                 “FabIndex”: “”,
                 “TechType”: “PS5662v”,
                 “MatNumber”: “07654321”,
                 “SWIDs”: [1234, 5678, 1122, 3344, 3435]
        },
        “pRevisionList”: [
                { “Name”: “HMI”, “Instance”: 1, “Type”: 1, “Version”: “V2.1_B0”, “Boot-
                loader”: “V1.2”,
                  “MatNumber”: “123456”, “SerialNumber”: “12345” },
                { “Name”: “SLT”,  “Instance”: 1, “Type”: 2, “Version”: “V1.7_B0” , “Boot-
                loader”: “V1.2”,
                  “SWID”: “3435” },
                  ….
          ]
}
```

4.3

Service: Device State

4.3.1

The *Device* state service is used to view and modify the state of a Miele *Device*. The current state can be retrieved by issuing a GET request to the URL of the state service. Furthermore, it is possible to modify several parameters of the *Device* state with the PUT operator.

4.3.2

GET <object:DeviceState>

4.3.2.1

The *<object: DeviceState>* object is returned, if a GET request is performed on the corresponding URL of the service with the *Accept* field set to *application/vnd.miele.v1+json*.

URL	**http://<host name>/Devices/<*id:Device*>/State/**
HTTP Request	GET /Devices/<id:Device>/State/ HTTP/1.1 Host: <*host name*> Accept: application/vnd.miele.v1+json
HTTP Request Payload	*empty*
HTTP Response Header	HTTP/1.1 200 OK Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>

HTTP Response Payload :=
 	
<*object:DeviceState*>
	{
	  <*id:DeviceStateFieldName*>: <*id:DeviceStateFieldValue*>,
	  <*id:DeviceStateFieldName*>: <*id:DeviceStateFieldValue*>,
	  …
	}

The *<object:DeviceState>* provides an image of the current *Device* state. Thus, the Miele *Device* maps the contents of the device internal data objects to this corresponding data object. Thus, it contains several fields, whereas every field is a single key value pair.

Professional specific fieldnames start with a prepended “p”, e.g. “pSystemTime**”.**

4.3.3

PUT <object:DeviceState> (optional)

4.3.3.1

The usage of PUT is optional and can be deactivated (e.g. if necessary according to external requirements). The PUT is only possible for optional resources. The mandatory fields are read-only.

The writeable field values of *<object: DeviceState>* can be modified with a PUT request in order to change the *Device* state. One or multiple fields can be modified at once. The HTTP response contains information about the results of all operations.

URL	**http://<host name>/Devices/<*id:Device*>/State/**

HTTP Request	PUT /Devices/<id:Device>/State/ HTTP/1.1 Host: <*host name*> Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>

HTTP Request Payload	{
	<*id:DeviceStateFieldName*>: <*id:DeviceStateFieldValue*>,
	<*id:DeviceStateFieldName*>: <*id:DeviceStateFieldValue*>,
	…
	}

HTTP Response Header	HTTP/1.1 200 OK Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>
HTTP Response Payload	[{"<*attribute:ReturnCode*>": {<*id:DeviceStateFieldName*>: <*id:DeviceStateFieldValue*>}}, {"<*attribute:ReturnCode*>": {<*id:DeviceStateFieldName*>: <*id:DeviceStateFieldValue*>}}, …]

Example: Perform a PUT to "http://<host name>/Devices/<id:Device>/State/" with the request payload

```
{
	"…some writeable field name….": [3, 25]
}
```

If the operation is successful, the response payload contains the results

```
[
	{"Success": {"…some writeable field name….": [3, 25]}}
]
```

If one operation fails, the response payload contains the unmodified value

```
[
	{"Failure": {"…some writeable field name….": [1, 10]}}
]
```

The same happens, if one tries to modify a field that is read-only.

4.3.4

<object:DeviceState> properties

4.3.4.1

The professional-specific properties start with a “p“.

<id:DeviceState FieldName>	Datatype	Read(R) Write(W)	Description
Status	uint8	R	Main *Device* state
ProgramID	uint16	R	Program ID
ProgramPhase	uint16	R	Current program phase
pRemainingTime	string	R	Relative remaining time as ISO8601 time difference
pElapsedTime	string	R	ElapsedTime as ISO8601 time difference
pSystemTime	string	R	Current system time stamp
pStartTime	string	R	Time stamp of the start time of the program
pEndTime	string	R	Time stamp of the estimated end time of the program
pLastNotificationTime	string	R	Time stamp of the last notification
Optional fields			
pExtended	object	R/(W)	All optional fields are grouped in the field pExtended. (W) means, if optional field is marked as 'W'

Table: Fields used in the <object:DeviceState>

If a field is not valid, or if it cannot be filled during a certain Device state, it shall be set to empty/0 (e.g. “” or 0).

If the <object:DeviceState> is used in eventing scenarios (e.g. for the *Miele Cloud Connection*, etc.), all fields that represent times (*pRemainingTime*, *pElapsedTime*, *pSystemTime*, *pStartTime*, *pEndTime*) must only trigger one event every minute! If the *Status* or *ProgramID* changes the event is triggered immediately.

4.3.4.2

<object:DeviceState>.Status

4.3.4.2.1

eThe *Status* field of the device state object describes the main *Device* state. The datatype is unsigned 8b Integer. This property of the *Device* state is read-only and cannot be modified.

The following table shows the supported values.

Enumeration	Value	Description
Reserved	0	Reserved
OFF	1	Appliance in off state
STAND-BY	2	Appliance in stand-by
PROGRAMMED	3	Appliance already programmed
PROGRAMMED WAITING TO START	4	Appliance already programmed and ready to start
RUNNING	5	Appliance is running
PAUSE	6	Appliance is in pause
END PROGRAMMED	7	Appliance end programmed task
FAILURE	8	Appliance is in a failure state
PROGRAMME INTERRUPTED	9	The appliance programmed tasks have been interrupted
IDLE	10	Appliance is in idle state
RINSE HOLD	11	Appliance rinse hold
SERVICE	12	Appliance in service state
SUPERFREEZING	13	Appliance in superfreezing state
SUPERCOOLING	14	Appliance in supercooling state
SUPERHEATING	15	Appliance in superheating state
MANUALCONTROL	16	Appliance is in manual program control state (e.g. manual external control on Benchmark machines)
WATERDRAIN	17	Appliance in water drain state
Reserved	16...30	Reserved
BOOT	31	Appliance is in booting state
SAFE STATE	32	Appliance is in safe state
SHUTDOWN	33	Appliance is shutting down
UPDATE	34	Appliance is in update state
SYSTEST	35	Appliance is in systest state
Reserved	36...63	Reserved
Non standardized	64…127	Non standardized
DEFAULT	144	Default – proprietary
LOCKED	145	Locked – proprietary
SUPERCOOLING_ SUPERFREEZING	146	Supercooling_Superfreezing - proprietary
NOT_CONNECTED	255	No connection to this appliance
Proprietary	128…255	Proprietary

Table: Status field values

4.3.4.3

<object:DeviceState>.ProgramID

4.3.4.3.1

The *ProgramID* field of the *Device* state object equals the id of the program that is currently selected. The datatype is unsigned 16b Integer. This property of the *Device* state is read-only and cannot be modified.

Enumeration	Value(s)
Own program or no operation	0
Standardized program ID for this device type	1…65535

Table: ProgramID field values

4.3.4.4

<object:DeviceState>.ProgramPhase

4.3.4.4.1
The *ProgramPhase* field of the *Device* state object describes the current program phase. The value is *Device* specific. The datatype is unsigned 16b Integer. This property of the *Device* state is read-only and cannot be modified.
This field is only valid, if the process is running. The default value is 0.

4.3.4.5

<object:DeviceState>.pRemainingTime

4.3.4.5.1
The *pRemainingTime* equals the relative remaining time given as a ISO8601 duration. The array values are only updated during the RUNNING state. The value is “” (empty string) during the other states. This property of the *Device* state is read-only and cannot be modified.

Example: pRemainingTime of 12 hours, 30 minutes and 17 seconds

“pRemainingTime”: “PT12H30M17S“

4.3.4.6

<object:DeviceState>.pElapsedTime

4.3.4.6.1
The *pElapsedTime* contains the elapsed time since the program start. The value is incremented according to the following rules:

- if the state changes from any other state value to 0x05 “Running”
- the value is incremented in the states 0x05 “Running” and 0x06 “Paused”
- the incrementing is stopped, and the value is not incremented in the state 0x07 “End” , 0x08 “Failure” or 0x09 “Program interrupted”
- the value is reset to “” if the state changes from “Running”, “Paused”, “End”, “Failure” to any other state

The default value is “”.

The value is only valid in states Running (5), Paused (6), End (7), Failure (8) and Program interrupted (9). A client shall only evaluate the field in these states.

Example: ElapsedTime of 12 hours, 30 minutes and 17 seconds

“pElapsedTime”: “PT12H30M17S“

4.3.4.7

<object:DeviceState>.pSystemTime

4.3.4.7.1
The *pSystemTime* field of the *Device* state object contains the current system timestamp. The datatype is string and the format is ISO 8601. This property of the *Device* state is read-only and cannot be modified.

Example: “pSystemTime”: “2014-10-22T14:57:40+02:00“

4.3.4.8

<object:DeviceState>.pStartTime

4.3.4.8.1
The *pStartTime* field of the *Device* state object contains the timestamp of the program start. This field is also set, if the *Device* is in *Status*==4 (PROGAMMED WAITING TO START). The datatype is string. This property of the *Device* state is read-only and cannot be modified. The default value is empty string “”.

Example: “pStartTime”: “2014-10-22T14:00:40+02:00“

Logic in EKXX communication modules:

if (Status changes from any (except RINSE_HOLD(11) or PAUSE(6)) to -> 5 RUNNING)
pStartTime is set to pSystemTime when status changed *->5

if (Status == 4 PROGRAMMED_WAITING_TO_START)
  pStartTime = CurrentTime + StartTimeRelative
else if (Status == 5 || Status == 6 || Status == 7 || Status == 8 || Status == 9 || Status == 11)

  pStartTime is the timestamp saved when the program was started (see trigger before)
else
  pStartTime = "" (empty)

4.3.4.9

<object:DeviceState>.pEndTime

4.3.4.9.1
The *pEndTime* field of the *Device* state object contains the timestamp of the estimated end time of the running program and the end time of the last process, if the *Device* is in State 7 (End Programmed). The datatype is string. This property of the *Device* state is read-only and cannot be modified. This field is only valid, if the process is running (5), paused (6), end programmed (7), failure (8) or interrupted (9). The value shall be ignored by all clients in all other states. The default value is empty string “”.

Example: “pEndTime”: “2014-10-22T15:30:40+02:00“

Logic in EKXX communication modules:

if (Status -> 7 END PROGRAMMED || Status -> 8 FAILURE || Status -> 9 INTERRUPTED)
pEndTime equals pSystemTime (saved timestamp) when (Status changed 5->7) || (Status changed 5->8) || (Status changed 5->9 || Status changed 6->7) || Status changed 6->8) || Status changed 6->9) || Status changed 11->7) || Status changed 11->8) || Status changed 11->9))

else if (Status == 5 RUNNING || Status == 6 PAUSED)
  pEndTime = pSystemTime + pRemainingTime
else
  pEndTime = ""

4.3.4.10

<object:DeviceState>.pLastNotificationTime (optional)

4.3.4.10.1
The *pLastNotificationTime* field of the *Device* state object contains the timestamp of the last notification (in the meaning of *profNotification*). The notification itself must be read via the *profNotification* service (Details are given in chapter 5). This field is only valid, if the *profNotification* service is available.

The datatype is string. This property of the Device state is read-only and cannot be modified.

Example: “pLastNotificationTime”: “2014-10-22T13:30:40.123+02:00"

4.3.4.11

<object:DeviceState>.ProcessUID (optional)

4.3.4.11.1
The *ProcessUID* field of the *Device* state object contains a random number that is assigned by the communication module, if a new program is started. This field is not persistently saved. This field is only valid, if the *Device* is in Status != OFF.

The datatype is string. This property of the *Device* state is read-only and cannot be modified.

Example: “ProcessUID”: “0000000.....2112312...."

default "ProcessUID":"00000000000000000000000000000000000000000000000000000000000000000000000000000000"

initial value: "00000000000000000000000000000000000000000000000000000000000000000000000000000000"

datatype: 32Byte - value as string

Condition for creation:
If ProcessUID == "00000000000000000000000000000000000000000000000000000000000000000000000000000000"

AND

When State/Status changes from any other state to RUNNING or PROGRAMMED_WAITING_TO_START!

Condition for reset to initial value (Process/Program ends):

When State/Status changes from

- RUNNING(5)
- PAUSE(6)
- END_PROGRAMMED(7)
- FAILURE(8)
- PROGRAM_INTERRUPTED(9)
- RINSE_HOLD(11)
- OFF(1)
- PROGRAMMED_WAITING_TO_START(4)
- MANUALCONTROL(16)

to any other Status (except the ones from the list).

4.3.4.12

<object:DeviceState>.pExtended

4.3.4.12.1

The *pExtended* field of the device state object contains all additional information. The content is *Device* specific and the datatype is object. This property of the *Device* state is read-only and cannot be modified.

The number of fields can change during a running program. That means it is possible that different fields are returned at the start and at the end of a program.

4.3.4.12.2
pExtended for Prof LMD Device

4.3.4.12.2.1
The following table describes the *pExtended* fields for Prof LMD *Devices*

<FieldName>	Description
pPDSyncState	The pPDSyncState shows the synchronization state of XKM LMD module with the device for the saved protocol data. The XKM itself does not save any protocols after power down. Datatype is enum. The default value is 0, means not synchronized. pPDSyncState:

	0 = UNSYNCHRONIZED (sync not started) 1 = SYNCHRONIZING (sync in progress - during synchronization of the saved protocols) 2 = SYNCHRONIZED (everything is synchronized - except the current active process) 3 = SYNC_TIMEOUT (currently NOT USED) 4 = SYNC_ERROR (any kind of failure during synchronization - sync could not be completed)
pPDChargeID	Current Charge ID of the running process. Only set if one protocol is active. Default value: ""
pPDProgramPhase	Current program phase, if program is running.
Temperature	Temperature array of the process in Celsius multiplied with the factor 100. The valid temperature range is between -273.15°C and 327.67°C. The precision equals 0.01°C. An empty array indicates that the measured value is invalid/not available. If a current temperature is not used/existing in the current device state of the *Device*, its corresponding value in the CurrentTemperature array is set to -32768. Example 1: Temperature with two values (40°C and 75°C) "Temperature": [4000, 7500]
A0Value	A0 value: uint16 value
Conductivity	Conductivity value uint16 value unit: µS/cm
CleaningPressure	Cleaning pressure value uint16 value unit: mBar

Table: Fields used in the pExtended for Prof LMD Devices

Example: pExtended for Prof LMD device

```
"pExtended": {
  "pPDSyncState": 1,
  "pPDChargeID": "12345678",
  "pPDProgramPhase": 23,
  "Temperature": [0,0],
  "A0Value": 0,
  "Conductivity": 0,
  "CleaningPressure": 0
}
```

4.3.4.12.3
pExtended for Professional Laundry Devices

4.3.4.12.3.1
The following table describes the *pExtended* fields for Professional Laundry *Devices*

<FieldName>	Datatype	Read(R) Write(W)	Description
DoorOpen	bool	R	true: Door is open false: Door is closed
DeviceLocked	bool	R	Get DeviceLocked State for professional devices. Get reports last state set in the device!
SpinningSpeed (optional)	u16	R	SpinningSpeed is split into CurrentSpinningSpeed and TargetSpinningSpeed
TargetSpinningSpeed	u16	R	TargetSpinning speed of a washer or washer/dryer Example: "TargetSpinningSpeed": 1600
CurrentSpinningSpeed	u16	R	CurrentSpinning speed of a washer or washer/dryer Example: "CurrentSpinningSpeed": 1500
Temperatue (optional)	array<s16>	R	In Modulo unused. Temperature is split into CurrentTempertature and TargetTemperature.

TargetTemperature	array<int16>	R	Target temperatures of the process in Celsius multiplied with the factor 100. The valid temperature range is between -273.15°C and 327.67°C. The precision equals 0.01°C. An empty array indicates that the measured value is invalid/not available. If a target temperature is not used/existing in the current device state of the *Device*, its corresponding value in the TargetTemperature array is set to -32768. Example 1: TargetTemperature with two values (40°C and 75°C) "TargetTemperature": [4000, 7500]
CurrentTemperature	array<int16>	R	Current temperatures of the process in Celsius multiplied with the factor 100. The valid temperature range is between -273.15°C and 327.67°C. The precision equals 0.01°C. An empty array indicates that the measured value is invalid/not available. If a current temperature is not used/existing in the current device state of the *Device*, its corresponding value in the CurrentTemperature array is set to -32768. Example 1: CurrentTemperature with two values (40°C and 75°C) "CurrentTemperature": [4000, 7500]

WaitingPayment	bool	R	true: User has selected a program and started at the machine. Payment signal must be set in order to start. false: Device is not waiting for payment signal (user can change program, program can be running, etc.).
pProgramphase	Object	R	Is only for semi-professional devices. In the Pro-professional devices it will return "0".
Extras	Object	R	Device dependent list with selected extras (bool)
AutoDosing (optional)	Object	R	Selected options (bool) for dosing
ProgramId	u16	R	If the device is switched off, the value is empty.
ProgramName	String	R	If the device is switched off, the value is empty
BlockId	u16	R	If the device is switched off, the value is empty
BlockName	String	R	If the device is switched off, the value is empty
StepId	u16	R	If the device is switched off, the value is empty
StepName	String	R	If the device is switched off, the value is empty
CurrentLoadWeight	s32	R	The current Load weight in the machin. (In Kilogramm). It is the value which is shown on the UI of the machine.
SINominalLoad	s32	R	Maximal Load based on the selected program [Gramm] 0x80000000 == NOT_USED, 0x80000001 == NOT_ACTIVE
SISetLoad	s32	R	Current load of the device entered by the user.[Gramm] 0x80000000 == NOT_USED, 0x80000001 == NOT_ACTIVE
SICurrentLoad	s32	R	Current load of the device, determined by the device. [Gramm] 0x80000000 == NOT_USED, 0x80000001 == NOT_ACTIVE

*Example: pExtended*for a Professional Laundry Device

"pExtended": {
"DoorOpen": false,
"TargetSpinningSpeed": 1600,
"CurrentSpinningSpeed": 1500,
"TargetTemperature": [600],
"CurrentTemperature": [590],
WaitingPayment: true

```
"Extras": {
    "Quick":"true",
    "Single":"false,
    "Eco":"false",
    "Intensive":"false",
    "PreRinse""true"
  }
}
```

4.3.4.12.3.2
Extras and Autodosing Fields

4.3.4.12.3.2.1

REST Service Modified: 10.09.2019	Objects Modified: 10.09.2019
WASCHING MACHINE pExtended	
/Devices/<DEVICE_ID>/State/pExtended/Extras/	
/Devices/<DEVICE_ID>/State/pExtended/Extras/Quick	GLOBAL_PS_Context.ContextParaWM.Extras.Quick
/Devices/<DEVICE_ID>/State/pExtended/Extras/Single	GLOBAL_PS_Context.ContextParaWM.Extras.Single
/Devices/<DEVICE_ID>/State/pExtended/Extras/WaterPlus	GLOBAL_PS_Context.ContextParaWM.Extras.WaterPlus
/Devices/<DEVICE_ID>/State/pExtended/Extras/RinsingPlus	GLOBAL_PS_Context.ContextParaWM.Extras.RinsingPlus
/Devices/<DEVICE_ID>/State/pExtended/Extras/PreWash	GLOBAL_PS_Context.ContextParaWM.Extras.PreWash
/Devices/<DEVICE_ID>/State/pExtended/Extras/Soak	GLOBAL_PS_Context.ContextParaWM.Extras.Soak
/Devices/<DEVICE_ID>/State/pExtended/Extras/RinseHold	GLOBAL_PS_Context.ContextParaWM.Extras.RinseHold
/Devices/<DEVICE_ID>/State/pExtended/Extras/ExtraQuiet	GLOBAL_PS_Context.ContextParaWM.Extras.ExtraQuiet
/Devices/<DEVICE_ID>/State/pExtended/Extras/SteamSmoothing	GLOBAL_PS_Context.ContextParaWM.Extras.SteamSmoothing
/Devices/<DEVICE_ID>/State/pExtended/Extras/PreRinse	GLOBAL_PS_Context.ContextParaWM.Extras.PreRinse
/Devices/<DEVICE_ID>/State/pExtended/Extras/Microfibre	GLOBAL_PS_Context.ContextParaWM.Extras.Microfibre
/Devices/<DEVICE_ID>/State/pExtended/Extras/Gentle	GLOBAL_PS_Context.ContextParaWM.Extras.Gentle
/Devices/<DEVICE_ID>/State/pExtended/Extras/AllergoWash	GLOBAL_PS_Context.ContextParaWM.Extras.AllergoWash
/Devices/<DEVICE_ID>/State/pExtended/Extras/Eco	GLOBAL_PS_Context.ContextParaWM.Extras.Eco
/Devices/<DEVICE_ID>/State/pExtended/Extras/Intensive	GLOBAL_PS_Context.ContextParaWM.Extras.Intensive
/Devices/<DEVICE_ID>/State/pExtended/Extras/StarchHold	GLOBAL_PS_Context.ContextParaWM.Extras.StarchHold
/Devices/<DEVICE_ID>/State/pExtended/CurrentTemperature	GLOBAL_PS_Context.ContextParaWM.Temperatures.CurrentValue
/Devices/<DEVICE_ID>/State/pExtended/TargetTemperature	GLOBAL_PS_Context.ContextParaWM.Temperatures. ?????
/Devices/<DEVICE_ID>/State/pExtended/CurrentSpinningSpeed	GLOBAL_PS_Context.ContextParaWM.
/Devices/<DEVICE_ID>/State/pExtended/TargetSpinningSpeed	GLOBAL_PS_Context.ContextParaWM.
Tumble Dryer pExtended	
/Devices/<DEVICE_ID>/State/pExtended/Extras/	
/Devices/<DEVICE_ID>/State/pExtended/Extras/Gentle	GLOBAL_PS_Context.ContextParaTD.Extras.Gentle
/Devices/<DEVICE_ID>/State/pExtended/Extras/Short	GLOBAL_PS_Context.ContextParaTD.Extras.Short
/Devices/<DEVICE_ID>/State/pExtended/Extras/Refresh	GLOBAL_PS_Context.ContextParaTD.Extras.Refresh
/Devices/<DEVICE_ID>/State/pExtended/Extras/Hygiene	GLOBAL_PS_Context.ContextParaTD.Extras.Hygiene
/Devices/<DEVICE_ID>/State/pExtended/Extras/SingleDry	GLOBAL_PS_Context.ContextParaTD.Extras.SingleDry
/Devices/<DEVICE_ID>/State/pExtended/Extras/Eco	GLOBAL_PS_Context.ContextParaTD.Extras.Eco
/Devices/<DEVICE_ID>/State/pExtended/Extras/AntiCrease	GLOBAL_PS_Context.ContextParaTD.Extras.AntiCrease
/Devices/<DEVICE_ID>/State/pExtended/Extras/GentleSimple	GLOBAL_PS_Context.ContextParaTD.Extras.GentleSimple
/Devices/<DEVICE_ID>/State/pExtended/Extras/ResidualMoisture	GLOBAL_PS_Context.ContextParaTD.ResidualMoisture.CurrentValue
/Devices/<DEVICE_ID>/State/pExtended/AutoDosing/	GLOBAL_PS_Context.ContextParaWM.AutoDosing
/Devices/<DEVICE_ID>/State/pExtended/AutoDosing/Container	GLOBAL_PS_Context.ContextParaWM.AutoDosing.Container
/Devices/<DEVICE_ID>/State/pExtended/AutoDosing/NoLaundryDetergent	GLOBAL_PS_Context.ContextParaWM.AutoDosing.NoLaundryDetergent
/Devices/<DEVICE_ID>/State/pExtended/AutoDosing/NoFabricConditioner	GLOBAL_PS_Context.ContextParaWM.AutoDosing.NoFabricConditioner
/Devices/<DEVICE_ID>/State/pExtended/AutoDosing/NoAdditive	GLOBAL_PS_Context.ContextParaWM.AutoDosing.NoAdditive

4.3.4.12.4
pExtended for a ZSVA Device

4.3.4.12.4.1
The following table describes the *pExtended* fields for ZSVA *Devices*

<FieldName>	Datatype	Read(R) Write(W)	Description
CycleNo	integer	R	No. of the cycle
LotNo	string	R	Lot number, default is empty string “” (empty string shall be set if not existing in the *Device*)
InitiatedBy	string	R	Level and ID
CurrentUser	string	R	Current user
DoorLoadingSide	uint8	R	ENUM 0: Default 1: Tight 2: Closed 3: InBetween 4: Open 5: Error
DoorUnLoadingSide	uint8	R	ENUM 0: Default 1: Tight 2: Closed 3: InBetween 4: Open 5: Error
LastMaintenance	string	R	Date of the last maintenance e.g. “2014-10-31”
LastPerformanceTest	string	R	Date of the last performance test e.g. “2014-12-11” (OPTIONAL)
CycleResult	uint8	R	ENUM 0: Default 1: Successful 2: NOT Successful (FAILED)
A0_F0	integer	R	A0/F0 value
CycleCancelledBy	string	R	Level and ID

Table: Fields used in the pExtended for ZSVA Devices

*Example: pExtended*for a ZSVA device

```
“pExtended”: {
  “CycleNo”: 54321,
  “LotNo”: “”,
   “InitiatedBy”: “…”,
   “CurrentUser”: “…”,
```

"DoorLoadingSide": 2,
"DoorUnLoadingSide": 2,
"LastMaintenance": "2014-06-13",
"LastPerformanceTest": "2014-10-23",
"ProcessResult": 0,
"A0_F0": "…",
"CycleCancelledBy": "…"
}

4.3.5

GET single field/value property from <object:DeviceState>

4.3.5.1

Instead of performing a GET in order to receive the entire *<object:DeviceState>*, it is also possible to receive single fields/properties of the object.

Example: If a GET is performed on

http://<host name>/Devices/<id:Device>/State/Status/

the *Status* field is returned

{
"Status": 17
}

4.3.6

Example: GET device state of a generic Device

4.3.6.1

This example shows the *Device* state of a device with the ID "123456789101". This *Device* resides on the *Host* "HostNameDev. The information is retrieved by issuing a GET to "http://HostNameDev/ Devices/123456789101/State/".

```
GET /Devices/123456789101/State/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* replies with the current *Device* status.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        “Status”: 5,
        “ProgramID”: 43,
        “ProgramPhase”: 15,
        “pRemainingTime”: "PT12H30M17S",
        “pElapsedTime”: "PT12H30M17S",
        “pSystemTime”: “2014-10-22T14:00:00+02:00”,
        “pStartTime“: “2014-10-22T13:48:00+02:00”,
        “pEndTime“: “2014-10-22T17:20:00+02:00“,
        “pLastNotificationTime”: “2014-10-22T13:30:40.123+02:00“,
        “pExtended”: {
                “CycleNo”: 54321,
                “LotNo”: “”,
                 “InitiatedBy”: “…”,
                 “CurrentUser”: “…”,
                 “DoorLoadingSide”: 2,
                 “DoorUnLoadingSide”: 2,
                 “LastMaintenance”: “2014-06-13”,
                 “LastPerformanceTest”: “2014-10-23”,
                 “CycleResult”: 0,
                “A0_F0”: “…”,
                “CycleCancelledBy”: “…”
        }
}
```

4.4
Service: profSession

4.4.1

The resource /.../profSession/ handles the session management and contains at least the user-login and -logout resources and may be extended by the device pairing resources by some *Devices*.

```
GET /…/profSession/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* provides a link where at least the information about the version can be found.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>
```

```
{
        “version” : 100
}
```

Devices requiring a *Device Pairing* provide a link where additional resources can be found (**other *Devices* that do not support the *Device Pairing* MUST NOT return the “*device_pairing_<method>*” resources!**).

Note: Device in Device Pairing is defined from the point of view of the Client. This specification uses Device as a reserved word in the meaning of the glossary.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        “version” : 200,
        “device_pairing_register” : { “href” : “/../profDisinfect/sessions/devices” },
        “device_pairing_login” : { “href” : “/../profDisinfect/sessions/devicetoken” },
        “user_write_access” : { “href” : “/../profDisinfect/sessions/users/writeaccess” },
        “user_login” : { “href” : “/../profDisinfect/sessions/login” },
        “user_logout” : { “href” : “/../profDisinfect/sessions/users” }
}
```

The following table describes the resources, especially as some of them are optional (although the service is mandatory).

<Resource-Name>	Mandatory (y/n)	Description
version	Y	Versioning information about the pairing methods. The following values are supported: 1: User Login only: version 100 2: User Login and Device Pairing: version 200
device_pairing_register	N	Only *Devices* requiring a Device Pairing have to provide this resource, other MUST NOT.
device_pairing_login	N	Only *Devices* requiring a Device Pairing have to provide this resource, other MUST NOT.
user_write_access	N	Only Devices supporting a write access protection provide this resource, other MUST NOT
user_login	N	Only *Devices* that have a flexible user login URL must provide this resource
user_logout	N	Only *Devices* that have a flexbile user logout URL must provide this resource

The service *profSession* is based on tokens. A token is used to transfer some (decoded) information between the server and the client, e.g. if a service (call) requires authorization, a token has to be provided.

Tokens always have a limited life span and can only be used while they are still valid. At any time, (session) tokens may also expire due to other reasons like a client logout or a backend abort due to inactivity. The validity can't be determined by the client in any case. But clients can for example choose to invalidate their session prior to the expiration date by actively telling the backend.

This specification does not handle deeper details as they may vary.

4.4.2
Version 100

4.4.2.1
Login

4.4.2.1.1
A Client can login into the *Device* by providing the user *<loginname>* and *<password>*.

The client can perform the following command in order to login

```
POST /…/profSession
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Content-Type: application/vnd.miele.v1+json
Content-Length: …

{
"LoginName": "Admin",
"Password": "the current password"
}
```

If there are wrong credentials, the call returns <-- 403 (Forbidden). Also if there is no user created it will turn 401 (Unauthorized).
Moreover the limit for wrong passwords entries are 3. After that the response code 429 (Too many Requests) will be send. This causes that the user is blocked for 15 minutes. Each try after the third increases the blocking 15 minutes up and the limit are 6 tries. At the end the user can be blocked maximum to 90 minutes (6 wrong password tries). A workaround for that is to reset the complete XKM but that would mean all the current settings get lost.

Otherwise with correct credentials there will be a 200 OK return. It will include the SessionID (Bear Token) and the SessionID will timeout after 10 minutes.

```
HTTP/1.1 200 OK
```

```
Content-Type: application/vnd.miele.v1+json
Content-Length: …

{
"SessionId":
"0000650471314B9CBF4265A6D98CFFB459521B52E8A7553BA63C68CDCD76F4BB91",
"Timeout": 600
}
```

The following table describes the fields that are returned by this service.

<FieldName>	Datatype	Read(R) Write(W)	Description
SessionId	string	R	Authorized session token (user is logged in), it has to be used as header property *Authorization: Bearer <token>* for any service which requires an authentication in all subsequent calls.
Timeout	integer	R	Token timeout in seconds. If the *SessionId* has not been used for the time returned in *Timeout*, the *SessionId* is discarded and the user must login again in order to get a new token. Therefore, the *SessionId* is automatically refreshed everytime it is used. The default value returned is 600s (10min).

4.4.2.2
Logout

4.4.2.2.1
A client can be logged with the following call

```
DELETE /…/profSession HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Authorization: Bearer <SessionId>
```

After a successful call, the *SessionId* becomes invalid.

```
HTTP/1.1 204 No Content
```

4.4.2.3
Password Change

4.4.2.3.1
A Client can change the password with the following request

```
POST /…/profSession
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Content-Type: application/vnd.miele.v1+json
Content-Length: …

{
"LoginName": "Admin",
"Password": "the current password"
"PasswordNew": "the new password"
}
```

If there are wrong credentials, the call returns
<-- 403

If the credentials are correct, the *Device* returns

```
HTTP/1.1 204 No Content
```

4.4.2.4
Security: Rate Limiting

4.4.2.4.1
The *Device* applies a rate limiting functionality for the login and password change methods.

A separate failure counter is used for every user. The following logic is applied

1. Every time the login or password change fails due to a wrong password for a certain *LoginName*, the failure counter for this *LoginName* is increased by 1.

2. If the failure counter exceeds 3 the login and password change functionality is DISABLED for this *LoginName*. So, this *LoginName* cannot be used anymore.
    - *Note: Existing Sessions remain valid.*
3. Every 15min the failure counter is decreased by 1.
4. The failure counter has a maximum of 6. If another failure occurs, it stays at 6.

4.4.2.5
Default Admin User

4.4.2.5.1
The *Device* contains a default Admin user that can be used to add and configure additional users. If the standard user did not set a password a 401 is returned instead of 403.

The username is "Admin" and the password is "Admin".

4.4.2.6
WhoAmI

4.4.2.6.1
The following URL can be called, if a sessions exists in order to get information about the user that uses the session:

```
GET /…/profSession/whoami/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

Response (user logged in):

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length:

{
    "ID":101,
    "LoginName":"Admin",
"Type":1,
    "Description":"admin description",
    "Roles":[1, 2, 101, 102]
}
```

Response (user not logged in):

HTTP/1.1 404 Not found
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length:

4.4.3
Version 110

4.4.3.1
Version 110 is very similar to version 100. The only difference is that the Admin user must login first and initially change the password. This provides a higher level of security.

The default password is "" (empty) in this case.

Note: The password change is mandatory in order to use any other service!

4.4.3.2
Password Policy

4.4.3.2.1
◦It must be at least eight characters long – the longer the better

◦It must be made up of uppercase and lowercase letters, special characters and digits (at least 3 of the 4 criteria)

The following set of special characters should be supported:
" !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"

See: https://www.owasp.org/index.php/Password_special_characters

◦Dictionary entries or names of family members/pets and their dates of birth, etc., may not be used

◦Keyboard patterns (e.g. asdf, 1234) must be avoided

◦Adding numbers or special characters to the end of a word is not recommended (e.g. Miele123)

4.4.4
Version 200

4.4.4.1
Device Pairing (optional)

4.4.4.1.1
Note: The URL depends on the value returned in <service:profSession>.href. In the following examples, prof-Disinfect has been returned. It could have also returned profSession.

Device Pairing Register
This scenario must be supported by any *Client* in the case "device_pairing_register" is returned.

It is strictly recommended to limit the count of *Device Pairing Register* requests, to e.g. 10.

User story:
If a *Client* connects to a *Device* for the first time (*Device* does not know about this *Client*), a pairing request has to be registered.

A *Client* can be registered by providing a *<devicename>* (unique id), a *<devicedescription>* and a *<devicelevel> with:*

- DEVICE_NAME: a device-Id generated by the client (e.g. MEI, Apple-Id) with a **length of at least 6 chars**.. If the device-Id is already existing on the backend, the client has to regenerate the device-Id.
- DEVICE_DESCRIPTION: a human readable description (e.g. "Adrian's Laptop").
- DEVICE_LEVEL (optional, for PCS mandatory): this is an integer, which specifies the device level (or device class).
  The following table shows the supported values (any further must be added here):

Enumeration	Value	Description	Notice
n/a	**0-99**	*reserved*	0-99 Reserved
GUI	**100**	Display GUI	100-199 Common Devices
GUI2	101	Display GUI #2	
WebUI	102	Web Browser	
PDSW	103	Prozess-Data-Software	
Dash	104	Dashboard (ZSVA)	
Smart	105	Smart Devices like a pad	
GLT	106	Central building control („Gebäude-Leit-Technik")	
Cloud	**200**	Miele Cloud (reserved - not used)	200-299 Miele Devices
MDU	201	Miele MDU Tool	
BandEnde	202	Miele "Bandende"	
EndoC	**300**	BIP/RD (already used)	300-399 OEM Partners

```
POST /…/profDisinfect/sessions/devices?DEVICE_NAME=<devname>&DEVICE_DESCRIPTION=<devdescr>&
DEVICE_LEVEL=<devlevel> HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* registers the *Client* and replies with

```
HTTP/1.1 201 Created
Content-Type: application/vnd.miele.v1+json; charset=utf-8
```

If the DEVICE_NAME is rejected by the machine because already existing, the call returns
<-- 429

If there are too many device pairing requests or device pairings, the call returns
<-- 409

Device Pairing Login
This scenario must be supported by any *Client* in the case "device_pairing_login" is returned.
The *Client* has to execute a device login with this pairing-pin for each (re)connect to the *Device*.

In any case, the length of a *PAIRING_PIN* has to be at least 6 chars!

It is strictly recommended to limit the *Device Pairing Login* attempts. After 3 failed attempts, the registered device (*DEVICE_NAME*) shall be implicitly removed.

User story 1:
An authorized user agrees to this *Client* (pairing request) and assigns a pin for this *Client* on the *Device*. As long as a succeeded pairing registration is not deleted on the *Device*, the pairing-pin stays typically valid.

User story 2:
A few Clients (from device level Enumeration above) don't need a manually interaction to get the pairing-pin, e.g. the MDU tool. In this case a pairing-pin is calculated (inner brackets are processed first):

PAIRING_PIN := HEX(HMAC_SHA256(KEY_SECRET, CONTENT))

GEN_PAIRING_PIN	**Explanation**
HEX	the HMAC result is converted into a lower case hex-string HEX-Result Example: "e3b0c44298fc1c14b... "
HMAC_SHA256	Standard IT algorithm (reference: rfc4634)
KEY_SECRET	A secret, e.g. a UUID as string UUID-Example: "5d935b43-4416-4054-b9ac-53d2523c150a" 1. It is known by very few persons which work on the Device or MDU. 2. It is strictly prohibited to email or publish it. 3. It must not be stored without further encrypting on persistent flash

	4. it is only allowed to decrypt it in memory
CONTENT	The CONTENT is a concatenation of values, separated by a '#' with: **FabNumber** (mandatory) as string, according to the chapter *Service: Device Ident* *Example:* "12345678" **Further Fields** (optional, may vary) <*aValue*> Example: CONTENT := "12345678" + '#' + "<val_*1*>" + '#' + "<val_*n*>"

Example:
PAIRING_PIN := *HEX(HMAC_SHA256("5d935b43-4416-4054-b9ac-53d2523c150a", "12345678#some-val"))*

results in
PAIRING_PIN := *d106514f5305e856b6cc173ce37468e56a4968fcb3c44eb6287dd67b71826dbf*

Any Case:
The token can be acquired by providing the <*devicename*> (unique id) and the <*pairingpin*>.

```
GET /…/profDisinfect/sessions/devicetoken?DEVICE_NAME=<devname>&PAIRING_PIN
=<pairingpin> HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* replies with a new session token (device is paired now) and the optional field for the session timeout.

Info: PCS doesn't support the optional field for the session timeout as the exact time, a token becomes invalid, can be found by decoding the (JWT) token with base64.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        "SESSION_TOKEN": "<dp_sessiontoken>",
        "SESSION_TIMEOUT": 12345
}
```

If the DEVICE_NAME is unknown by the backend, the call returns
<-- 400

If the PAIRING_PIN is rejected by the machine, the call returns
<-- 400

Further error codes
<-- 409 (PCS might also use this error code in future)

The following table describes the fields that are returned by this service

<FieldName>	Datatype	Read(R) Write(W)	Description
SESSION_TOKEN	string	R	Session token (for a paired device), it has to be used as header property *Authorization: Bearer <token>* for e.g. a user login.
SESSION_TIMEOUT	integer	R	Token timeout in seconds (optional).

4.4.4.1.2
Version 200 - Device Pairing

4.4.4.1.2.1
Version 200 of the device pairing calculates the PAIRING_PIN from

CONTENT := <FabNr>#<DEVICE_NAME used in POST for pairing request>

For example:

CONTENT := 12345678#MDU

4.4.4.2
User Session Handling (mandatory)

4.4.4.2.1
Note: The URL depends on the value returned in <service:profSession>.href. In the following examples, profDisinfect has been returned. It could have also returned profSession.

User Login
For *Devices* requiring a *Device Pairing*, a user login is only possible after a successful *Device Pairing* has been completed.

A *Clients* has to provide the header property *Authorization* (token from “device_pairing_login”) in the case a *Device* requires a *Device Pairing*. More details to the header property *Authorization* can be found in [RFC6750].

A user/*Client* can login into the *Device* by providing the user <*loginname*> and <*pin*> (password) referenced by the "user_login" resource. There may be other authentication backends such as "rfid", which are irrelevant for remote logins.

The length of a *PIN* has to be at least for the profSession *version* >= 200 (with device pairing)

1. very limited access (e.g. an equipment operator): at least 4 chars
2. limited access (it depends on the *Access Control List* for a user): at least 6 chars
3. full access (e.g. service technician): at least 10 chars

```
POST /…/profDisinfect/sessions/login?LOGIN_NAME=<loginname>&PIN=<pin> HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Authorization: Bearer <dp_sessiontoken>
```

Note: In case the Device does not require the optional Device Pairing, the Authorization header is not transmitted by the user/Client.

You may also **set a new password** by additionally providing the optional NEW_PIN=<*new_pin*>. In this case the <*pin*> (password) will be accepted only for setting the <*new_pin*> (new password); you have to login again.

```
POST /…/profDisinfect/sessions/login?LOGIN_NAME=<loginname>&PIN=<pin>&NEW_PIN=<new_pin> HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Authorization: Bearer <dp_sessiontoken>
```

Note: In case the Device does not require the optional Device Pairing, the Authorization header is not transmitted by the user/Client.

The *Device* replies with the authorized session token (user is logged in); optional fields may be supported.

Info: PCS doesn't support the optional SESSION_TIMEOUT field for the session timeout as the exact time. PCS uses standardized JWT token that contain additional information about the timeout and expiration.

```
HTTP/1.1 201 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        "SESSION_TOKEN": "<u_sessiontoken>",
        "SESSION_TIMEOUT": 12345,
        "FIRST_NAME": "firstname",
        "LAST_NAME": "lastname"
}
```

If there are wrong credentials, the call returns
<-- 400

The password is expired, you have to set a new password. The expired <*pin*> (password) must still be validated by the device. If it is not possible or the <*pin*> (password) doesn't match 400 will have to be returned by the device (this is a security issue).

<-- 401

If the user is locked, the call returns

<-- 409

The following table describes the fields that are returned by this service.

<FieldName>	Datatype	Read(R) Write(W)	Description
SESSION_TOKEN	string	R	Authorized session token (user is logged in), it has to be used as header property *Authorization: Bearer <token>* for any service which requires an authentication in all subsequent calls.
SESSION_TIMEOUT	integer	R	Token timeout in seconds (optional). If the token has not been used for the time returned in SESSION_TIMEOUT, the token is discarded and the user must login again in order to get a new token. Therefore, the token is automatically refreshed everytime it is used. The default value returned is 600s (10min).
FIRST_NAME	string	R	First name of the logged in user (optional). This field is informational and not needed for futher session logic.
LAST_NAME	string	R	Last name of the logged in user (optional). This field is informational and not needed for futher session logic.
(additional-fields)	any-type	R	The content may contain additional fields (optional). This fields are informational and not needed for futher session logic.

User Log out

A user/*Client* can be logged out via the "user_logout" resource.

The header property *Authorization* is required - with <*token*> returned from the "user_login" resource.

```
DELETE /…/profDisinfect/sessions/users HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Authorization: Bearer <u_sessiontoken>
```

After a successful call, the session-<*token*> becomes invalid.

```
HTTP/1.1 204 No Content
```

If the log out is rejected by the *Device*, the call returns
<-- 400 Bad Request

4.4.4.3
User Write Access (optional)

4.4.4.3.1
Note: The URL depends on the value returned in <service:profSession>.href. In the following examples, prof-Disinfect has been returned. It could have also returned profSession.

User Write Access
This scenario must be supported by any *Client* in the case "user_write_access" is returned.

User story:
Some devices don't allow a write access for most of the ressources even if a used is logged in – typically it may be a medical device. In this case a logged in user on a client has to ask for write access permission. To avoid a logical concurrency, only a single write access is allowed at once (this should be the default assumption of any client).

To ask for the user write access, clients have to call:

```
POST /…/profDisinfect/sessions/users/writeaccess HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Authorization: Bearer <u_sessiontoken>
```

The *Device* replies with

```
HTTP/1.1 201 Created
Content-Type: application/vnd.miele.v1+json; charset=utf-8
```

If the 201 has been received, the client has to poll the current state of the user write access. The write access is granted via an confirmation on the Device UI.
Invalid content (there isn't a content), the call will return
<-- 400

If the request is rejected (e.g. as user already has got write access rights, the session token is invalid or another client already has got write access rights), the call will return
<-- 409

To ask for the current state of the user write access, clients have to call:

```
GET /…/profDisinfect/sessions/users/writeaccess HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Authorization: Bearer <u_sessiontoken>
```

The *Device* replies the state of the write access request

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        "STRG.GLOBAL.ZUSTAND.ClientMitSchreibzugriffStatus": 1
}
```

<FieldName>	Datatype	Read(R) Write(W)	Description
STRG.GLOBAL.ZUSTAND .ClientMitSchreibzugriffStatus (this field name will be changed)	int	R	0 = write access request has been rejected or released for a valid reason, e.g. internal state 1 = write access request is beeing processed. On the gui screen (internal display), a qualified user has to • confirm the write access request (results in *2*) • reject it the write access request (results in *0*) It is strictly recommended, to support this gui screen, othercase the implementation of the host will automatically have to change implicitly from *1* to *2* (without user confirmation). 2 = write access is possible (client has got it). It is recommended, to lock the gui in this case (to prevent a multi master access) 3 = write access request is possible 4 = write access request is currently not possible - please try again later
(additional-fields)	any-type	R	The content may contain additional fields (optional).

To release user write access, clients have to call:

```
DELETE /…/profDisinfect/sessions/users/writeaccess HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Content-Length: <decimal number of octets>
Authorization: Bearer <u_sessiontoken>
```

The *Device* replies

```
HTTP/1.1 204 No Content
```

The session will not be ended for that reason (but still may be released in the meaning of expired token at any time).
The client may ask for the current state of the user write access again now as described above.
Invalid content (there isn't a content), the call will return
<-- 400
The client is currently not the owner of the user write access, the call will return
<-- 401

4.5
Customer Service REST Services

4.5.1

New Version (Compatible with Domestic)

4.5.1.1

For most Miele *Devices* customer service diagnosis is performed via the DOPx, Oc85 or DOP2 protocol.

Some newer *Devices* offer services that are used by the Miele customer service in order to perform a REST-based diagnosis of a *Device*. This chapter describes the optional services. They can only be used, if they are returned via /Devices/<id>/.

All services in this and all subchapters can only be accessed via the customer service Soft-AP, via an authorized user and via the *Miele Cloud Service*.

The list of all supported service can be read via

```
GET /…/profService/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* responds

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
"Mode": 1,
"ErrorList": {"href": "ErrorList/"},
"XKM": {"href": "XKM"},
"OperatingData": {"href": "OperatingData/"},
"PowerSupply": {"href": "PowerSupply/"},
"ComponentState": [0, 0, 0, 0, 1, 1,],
"SensorState": [1, 255, 255, 1, 128],
"CounterLogData": [0, 2, ,2, 0, 1, 2, 7, 0],
"TemperatureLogData": {"href": "TemperatureLogData/"}
}
```

The following table describes the fields that are returned by this service

<FieldName>	Datatype	Read(R) Write(W)	Description
Mode (optional)	Integer	R/W	Current mode 0: Normal mode

			1: Service mode The field is only writable via Customer Service Soft-AP!
ErrorList	object	R	Link for the error list service of the appliance
OperatingData	object	R	Link for the error list service of the appliance
PowerSupply (optional)	object	R	Link for the power supply information service of the *Device*, only present if supported by the *Device*
XKM	object	R	This returns informations about the XKM module
ComponentState (optional)	array[int]	R	Array that contains information about the *Device* specific component state(s), only present if supported by the *Device*
SensorState (optional)	array[int]	R	Array that contains information about the *Device* specific sensor value(s), only present if supported by the *Device*
CounterLogData (optional)	array[int]	R/W	Array that contains information about the *Device* specific counter log value(s), only present if supported by the *Device*
TemperatureLogData (optional)	object	R	Link for the temperature log data service of the *Device*, only present if supported by the *Device*

4.5.1.2
DELETE CounterLogData

4.5.1.2.1
The *CounterLogData* can be DELETED (reset to 0) via

```
DELETE /…/profService/CounterLogData/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* replies with

```
HTTP/1.1 204 No Content
```

4.5.1.3
Service: ErrorList

4.5.1.3.1
The resource /Devices/<id>/ErrorList/ contains the standardized error list of the *Device*.

```
GET /…/profService/ErrorList/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* provides a link where the relevant information can be found.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        "LastReset": "2014-10-20T07:54:214",
        "Active": [1, 3, 100],
        "1": { "href": "1/"},
        "2": { "href": "2/"},
        "3": { "href": "3/"},
        "10": { "href": "10/"},
        "20": { "href": "20/"},
        "30": { "href": "30/"},
        "100": { "href": "100/"}
        ...
}
```

The following table describes the fields that are returned by this resource.

<FieldName>	Datatype	Read(R) Write(W)	Description
LastReset (optional)	string (timestamp)	R	Shows when the error list has been deleted the last time. *LastReset* is "" (empty), if never resetted. If system clock is not set and errors are deleted this value is "2012-01-01T00:02:00"

Active	array[integer]	R	Array that contains the number of all active errors
<id>	object	R	Official Miele error ID for the availab-le errors

4.5.1.3.2
Read Error Details

4.5.1.3.2.1
Details of a single error can be requested via

```
GET /…/profService/ErrorList/<code> HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The Device returns the details for the error given in <*code*> parameter.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>


{
        "Code": 10,
        "ShortText": "",
        "Count": 12,
        "LastOccurence": "2014-10-20T07:54:214",
        "LastOccurenceOperatingMinutesDevice": 15200
}
```

The following table describes the fields that are returned by this service

<FieldName>	Datatype	Read(R) Write(W)	Description
Code	Integer	R	Code of the er-ror according to the standardized

			error list
ShortText	String	R	Short text of the error (this field is optional) It is always “” (empty) at the moment.
Count	Integer	R	Numer of occurrence since last reset of the error list
LastOccurence	string (timestamp)	R (see comment below)	Timestamp for last occurrence. Default Value: 0000
LastOccurenceOperatingMinutesDevice	integer	R (see comment below)	Appliance operating minutes for last occurrence Default Value: Value from the last successful received data package

Note: The Device fills either LastOccurence OR LastOccurenceOperatingMinutesDevice, as only one parameter can be provided by the Device! In this case the second parameter contains the default value.

4.5.1.3.3
Delete Error List

4.5.1.3.3.1
The error list can be DELETED (reset to 0) via

```
DELETE /…/profService/ErrorList/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* sets the *LastReset* field to the current time and date. Then, it deletes the entire error list (reset *Count* to 0, *LastOccurence* to NULL and *LastOccurenceOperatingMinutesDevice* to 0 for every error). Finally, it replies with

```
HTTP/1.1 204 No Content
```

This is only possible via customer service soft-AP.

4.5.1.4
Service: OperatingData

4.5.1.4.1
The resource /.../OperatingData/ contains the operating data of the *Device*.

```
GET /…/profService/OperatingData/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* provides the operating data (*Electronic* is typically the main controller board is meant here)

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        “OperatingHoursElectronic”: 123456,
        “OperatingHoursPreviousElectronic”: 1593482,
        “OperatingHoursDevice”: 1716938
}
```

The following table describes the fields that are returned by this service

<FieldName>	Datatype	Read(R) Write(W)	Description
OperatingHoursElectronic	integer	R	Operating hours of the electronic in [minutes].
OperatingHoursDevice	integer	R	OperatinHoursElectronic + OperatingHoursPreviousElectronic Equals operating hours of the device in [minutes]
OperatingHoursPreviousElectronic	integer	R/W	Summation of operating hours of all electronics prior replacement in [minutes]

Table: Fields used in the profOperatingData service

The MDU can write the *OperatingHoursPreviousElectronic* with a

```
PUT /…/profService/OperatingData/OperatingHoursPreviousElectronic HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Content-Type: application/vnd.miele.v1+json; charset=utf-8
```

```
Content-Length: <decimal number of octets>

{
        “OperatingHoursPreviousElectronic”: 1593482
}
```

This is only possible via customer service Soft-AP.

If this has been performed successfully, the *Device* responds

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
        {“Success”: {“OperatingHoursPreviousElectronic”: 1593482}}
]
```

It is also valid to just return the 204 without content (not writing "Success").

4.5.1.5
Power Supply

4.5.1.5.1

```
GET /…/profService/PowerSupply/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
"Voltage": 233,
"Frequency": 50
}
```

The following table describes the fields that are returned by this service

<FieldName>	Datatype	Read(R) Write(W)	Description
Voltage	integer	R	Mains voltage in [Volt]

Frequency	integer	R	Mains frequency in [Hz]

4.5.1.6
Mode Switch (optional)

4.5.1.6.1
The MDU can switch the appliance between normal operation mode and customer service mode. This is only possible in customer service Soft-AP.

The MDU can switch the mode via a PUT operation

```
PUT /Devices/<id>/profService/Mode HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Content-Type: application/vnd.miele.v1+json
Content-Length: <decimal number of octets>

{

      "Mode": 1
}
```

The following two modes are supported

Mode	Description
0	NormalOperation
1	ServiceMode

The *Device* checks the information. If the switch is successful, the device responds with a 200.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
      { "Success": { "Mode": "1" }}
]
```

The MDU can also read the current mode via a GET operation

```
GET /Devices/<id>/profService/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Content-Length: <decimal number of octets>
```

If everything is successful, the *Device* responds with 200 and the current value

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        "Mode": "1"
}
```

4.5.2
Old Version - Deprecated

4.5.2.1
Service: profErrorList

4.5.2.1.1

The resource /.../profErrorList/ contains the standardized error list of the *Device*.

```
GET /…/profErrorList/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The Device provides a link where the relevant information can be found.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        “list”:          { “href”: “/…/profDisinfect/errors“ },
        “active”  :    { “href”: “/…/profDisinfect/errors?active=1” },
        “info”:          { “href”: “/…/profDisinfect/errors/info” }
}
```

Note: This is just an example URL. In general, any URL on this Host could be returned (e.g. ‘errors’). An application must follow this URL including all additional attributes.

Read list of Errors
The error list of all supported errors can be requested via the "list"-tagged URL. The list contains an array with all errors supported by a *Device*.

```
GET /…/profDisinfect/errors HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* provides a list with all supported errors.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
        { "Code": 10},
        { "Code": 12},
        { "Code": 20},
        { "Code": 30},
        { "Code": 50},
        …
]
```

Read list of active Errors

The error list of all active errors can be requested via the "active"-tagged URL. The list contains an array with all errors that have occurred at least once since the last deletion of the error list.

```
GET /…/profDisinfect/errors?active=1 HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* provides a list with all errors that have a *"Count"* > 0.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
        { "Code": 10},
        { "Code": 20},
        { "Code": 30}
]
```

Read Error Details

Details of a single error ("list" or "active") can be requested via

```
GET /…/profDisinfect/errors/error?Code=<code> HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The Device returns the details for the error given in <*code*> parameter.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
        {
            "Code": 10,
            "ShortText": "…",
            "Count": 12,
            "LastOccurence": "2014-10-20T07:54:21",
            "LastOccurenceOperatingMinutesDevice": 15200,
            "pExtended": { … }
        }
]
```

The following table describes the fields that are returned by this service

<FieldName>	Datatype	Read(R) Write(W)	Description
Code	Integer	R	Code of the error according to the standardized error list
ShortText	String	R	Short text of the error (this field is optional)
Count	Integer	R	Numer of occurrence since last reset of the error list
LastOccurence	string (timestamp)	R	Timestamp for last occurrence
LastOccurenceOperatingMinutesDevice	integer	R	Appliance operating minutes for last occurrence
LastReset	string (timestamp)	R	(OPTIONAL) Last time this error has been resetted while it has been active.
pExtended	object	R	Object for additional information (optional)

Table: Fields used in the profErrorList service

Read Error Info

The error Info can be requested via the "info"-tagged URL. This Resource contains additional infos related to the /.../profErrorList/ Service.

```
GET /…/profDisinfect/errors/info HTTP/1.1
```

```
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The Device returns a resource with additional infos.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
        { “LastReset”: “2014-10-20T07:54:21” }
]
```

The following table describes the fields that are returned by this resource.

<FieldName>	Datatype	Read(R) Write(W)	Description
LastReset	string (ti-mestamp)	R	Shows when the error list has been deleted the last time. *LastReset* is NULL, if never resetted.

DELETE Error List

The error list can be DELETED (reset to 0) via

```
DELETE /…/profDisinfect/errors HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* sets the *LastReset* field to the current time and date. Then, it deletes the entire error list (reset *Count* to 0, *LastOccurence* to NULL and *LastOccurenceOperatingMinutesDevice* to 0 for every error). Finally, it replies with

```
HTTP/1.1 204 No Content
```

Read Details of all active Errors (optional)

Details of all active errors can be requested via

```
GET /…/profDisinfect/errors/error?active=1&details=1 HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The Device returns the details for all active errors.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
```

```
        {
            "Code": 10,
            "ShortText": "…",
            "Count": 12,
            "LastOccurence": "2014-10-20T07:54:21",
            "LastOccurenceOperatingMinutesDevice": 15200,
            "pExtended": { … }
        },
        {
            "Code": 13,
            …
        },
        …
]
```

4.5.2.2

Service: profErrorLog

4.5.2.2.1

The resource /.../profErrorLog/ contains the standardized error log of the *Device*. The error log should contain at least the last 500 errors. It shall be implemented like a ring buffer. If a new entry is added and the list is full, the oldest entry is removed.

```
GET /…/profErrorLog/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The Device provides a link where the relevant information can be found.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        "errorLogs" : { "href" : "/../profDisinfect/errorLogs" }
}
```

Note: This is just an example URL. In general, any URL on this Host could be returned. An application must follow this URL including all additional attributes.

```
GET /…/profDisinfect/errorLogs[?MinTime=<timestamp>] HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The optional parameter *MinTime* can be used (extends the URL parameter list) to set a timestamp since when the error log shall be provided, e.g. "&MinTime=2014-10-20-T07:54:24+02:00"

If *MinTime* is missing, the entire error log is provided.
A request for a single entrys is not supported for this service.

The *Device* provides the error log

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
        {
            “Time”: “2014-10-20T07:54:21+02:00”,
            “OperatingMinutesDevice”: 15200,
            “Code”: 10,
            “ProgramID”: 10,
            “ProgramPhase”: 15,
            “ProgramStep”: 3,
            “pExtended”: { …. }
        },
        { …. },
        { …. }
]
```

The following table describes the fields that are returned by this service

<FieldName>	Datatype	Read(R) Write(W)	Description
Time	Timestamp (string)	R	Timestamp of occurrence
OperatingMinutesDevice	integer	R	Total operating minutes of the Device when occured
Code	integer	R	ID of the error
ProgramID	integer	R	ID of the program that was running when the log entry was created, value is 0 if error occurs outside of a program
ProgramPhase	integer	R	ProgramPhase that was present when the log entry was created, value is 0 if error occurs outside of a program
ProgramStep	integer	R	ProgramStep when the log entry was created, value is 0 if error occurs outside of a program
pExtended	object	R	Object for additional information (optional). The data is a nested JSON object. For example, this field could contain a short

			description of the error.

Table: Fields used in the profErrorLog service

4.5.2.3

Service: profOperatingData

4.5.2.3.1

The resource /.../profOperatingData/ contains the operating data of the *Device*.

```
GET /…/profOperatingData/ HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The Device provides a link where the relevant information can be found.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        “data” : { “href” : “/../profDisinfect/operatingData“ }
}
```

Note: This is just an example URL. In general, any URL on this Host could be returned. An application must follow this URL including all additional attributes.

The operating data is returned if a GET is sent to the corresponding resource.

```
GET /…/profDisinfect/operatingData HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
```

The *Device* provides the operating data (*Electronic* is typically the main controller board is meant here)

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        “OperatingMinutesElectronic”: 123456,
        “OperatingMinutesPreviousElectronic”: 1593482,
        “OperatingMinutesDevice”: 1716938,
        “pExtended”: { … }
}
```

The following table describes the fields that are returned by this service

<FieldName>	Datatype	Read(R) Write(W)	Description
OperatingMinutesElectronic	integer	R/W (only in end-of-line)	Operating minutes of the electronic. Write is only possible, if the *Device* supports end-of-line tests via IP (OPTIONAL). The field is read-only during normal operation after end-of-line.
OperatingMinutesDevice	integer	R	OperatingMinutesElectronic + OperatingMinutesPreviousElectronic Equals operating minutes of the device
OperatingMinutesPreviousElectronic	integer	R/W	Summation of operating minutes of all electronics prior replacement
pExtended	object	R	Object for additional information (optional). This is a nested JSON object.

Table: Fields used in the profOperatingData service

The MDU can write the *OperatingMinutesPreviousElectronic* with a

```
PUT /…/profDisinfect/operatingData HTTP/1.1
Host: HostNameDev
Accept: application/vnd.miele.v1+json
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

{
        “OperatingMinutesPreviousElectronic”: 1593482
}
```

If this has been performed successfully, the *Device* responds

```
HTTP/1.1 200 OK
Content-Type: application/vnd.miele.v1+json; charset=utf-8
Content-Length: <decimal number of octets>

[
        {“Success”: {“OperatingMinutesPreviousElectronic”: 1593482}}
]
```

It is also valid to just return the 204 without content (not writing "Success").

5
Optional Services, Protocols and Interfaces

5.1

Miele Professional *Devices* support a device specific number of optional services. Every professional-specific service starts with the prefix **prof**.

The optional services are specified in [Miele_Professional_IP_Profile_Services].

6
Security and Authentication

6.1

This chapter describes the security and authentication for communication that is NOT between the *Miele Cloud Service* and Miele Prof. *Host*. Aspects regarding the *Miele Cloud Connection* security are covered in the next chapter.

In addition to general security topics, this specification covers two security modes:
1. *Miele **Normal** Security Mode* (usage of *User Pairing* without *Device Pairing*)
2. *Miele **Strict** Security Model* (driven by PCS: combination of *Device Pairing* and *User Pairing*)

Every Miele professional *Device* has to realize at least the *Miele Normal Security Mode* to become compliant with this specification.

Miele medical *Devices* are seriously recommended to adopt the *Miele Strict Security Mode*.

6.2
HTTPs and HTTP

6.2.1

The communication between a *Host* and a client application **should** be performed via HTTPs in the LAN.

If there is a requirement for a certain application that plain HTTP shall be used, this is also allowed. However, it must be noted that these *Hosts* have an increased vulnerability and reduced communication security/privacy! If a new *Host* shall be developed according to this specification that uses plain HTTP, this must be negotiated with the editors of this document.

Note: Miele PCS devices do not support plain HTTP.

For the normal case, HTTPs is used. Here, only TLS 1.1 or 1.2 are allowed. SSL 3.0 or TLS 1.0 are not allowed due to known security holes. If there is a future update to the TLS specification, the changes shall be evaluated and this specification shall also be updated.

Regarding TLS, only secure cypher suites shall be used (e.g. no RC4). The following table is a proposal with appreciated cypher suites. However, it is possible to support additional cypher suites, if required. Furthermore, the use of cipher suites that do use RSA for Key Exchange i.e. TLS_RSA_* is no longer appreciated.

This table shall be reviewed and evaluated in a fixed time interval (at least every 3 months) in order to check if there are some new/known security bugs. If a certain cypher suite is considered broken, the impacts shall be immediately reviewed.

Appreciated TLS cypher suites	Supported Libraries		Supported Miele Backends		Supported Browser		
	ARM Mbed TLS	TI CC3200	MCSv2	MCS Prof	IE 11	Mozilla	Chrome
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	X				X	X	X
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	X				X	X	X
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	X		X	X	X	X	X
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	X		X	X	X	X	
Depreciated cypher suites (not allowed for new projects)							
TLS_RSA_WITH_AES_256_CBC_SHA	X	X	X	X	X		
TLS_RSA_WITH_AES_256_CBC_SHA256	X	X	X	X	X	X	X
Depreciated cypher suites (usage forbidden)							
SSL_RSA_WITH_RC4_128_SHA		X					
SSL_RSA_WITH_RC4_128_SHA		X					
SSL_RSA_WITH_RC4_128_MD5		X					
TLS_ECDHE_RSA_WITH_RC4_128_SHA	X	X			X		

Table: Exemplary cypher suites

If a certain cypher suite is considered broken, the impacts shall be immediately reviewed.

At the moment, the Miele Cloud offers TLS 1.0/1.1 in addition to TLS 1.2. If possible, TLS 1.2 shall be used by a client (APP or *Host*). TLS 1.0/1.1 might be deactivated in the future.

It is recommended that both, APP and *Host* perform DNS host name verification.

6.3
Certificate Handling

6.3.1
Miele Prof. IP *Hosts* use a server certificate in order to prove its identity against a client application, if they communicate via HTTPs. The format is X.509 according to [RFC5280].

The minimum keysize of RSA 2048 is mandatory and must be supported by every *Host*. It is appreciated, if a *Host* uses longer key sizes.

The following concept is used in every *Host*, if no customer specific certificate is used

1. Initially after the first startup, the *Host* generates a self-signed certificate with a validity of 2 years.
2. If possible, the certificate should contain the electronic device label in the following fields
    1. O/OU/CN and the v3 extension subjectAltName is used
    2. The next subsection contains an example of a certificate configuration and generation process.

3. The corresponding private key must be stored encrypted (at least with AES-256) on the *Host*. The key used for the encryption could be generated from a hardware part that is different on every *Host* (or a random key could be generated). It MUST NOT be possible to read/retrieve this symmetric key from the *Host* (e.g. an ID from the silicon chips could be used that is not available outside the control).
4. The private key of the certificate is decrypted from the *Host* after startup, and it should ideally held only in RAM if technically possible. It MUST NOT be possible to read/retrieve it from an external application via external interfaces.
5. A user or an external application must accept (trust) the self-signed certificate initially, if it connects with the *Host*. The external application MUST fulfill the following requirements:
    1. Show the certificate to the user and ask him to validate before adding it to the trust chain of the application.
    2. Inform the user about the validity of the certificate. This information shall be accessible in the application anytime.
6. If the certificate expires or if the *Host* is reset to factory defaults, the *Host* deletes the old certificate and generates a new self-signed one.
7. All external applications MUST fulfil the following requirements if a new certificate has been used on an existing Host
    1. If the current date indicates that the old certificate has been expired, it shall inform the user that he must trust the new certificate. The following steps are identical to 4).
    2. If the old certificate should have been still valid, it shall inform the user with a noticeable WARNING that the certificate has been changed and that he must approve this. The following steps are identical to 4)
        1. If the external application is a web browser it will show that there is a new untrusted certificate. The following steps are identical to 4)
8. (Optional, not mandatory but strongly recommended): The *Device* UI can provide a functionality to show the *Host* certificate on the *Device* UI. Then, the user can validate it in the external application and on the *Device* UI.

Usage of a customer specific certificate

The self-signed certificate can be changed by the customer. A common use-case would be that the customer changes the default certificate against a certificate that is generated by the customer PKI.

This certificate can be uploaded via the profFileTransfer Service. After it is uploaded the communication module **MUST** be power cycled by the user in order to resychronize all external clients (close all existing sessions, etc.), otherwise the old certificate is kept until the next power cycle/reboot.

6.3.2
Self-Signed Certificate Generation

6.3.2.1

This chapter describes the standardized format for self-signed certificates that might be used with *Miele Professional Hosts or others.*

The format is described with the help of an OpenSSL example.

The fields are filled according to the following rules:

Mandantory

O := <Vendor> (e.g. Miele)
OU := <VendorOU> (e.g. Miele Professional)
CN := <Product Name> OR <Vendor>-<TechnicalType>-<FabNr> (e.g. Miele-...-....)

Optional
DNS.1 := <Vendor>-<mac-addr>.local. (e.g. Miele-.....local.)

The configuration file ***selfsigned.conf*** for OpenSSL has the following content (fields O,OU,CN,DNS.1 are filled exemplarily):

```
[ req ]
distinguished_name     = req_distinguished_name
prompt                 = no
# add extentions to the cert
x509_extensions      = v3_ca


[ req_distinguished_name ]
O                      = Miele & Cie. KG
OU                      = Miele Professional
CN                      = Miele-PS5662v-123456789

[ v3_ca ]
subjectAltName = @alt_names
keyUsage = critical, digitalSignature, keyEncipherment


[alt_names]
DNS.1 = Miele-001D63FFFE0313BA.local.
```

Note: The fields
subjectAltName = @alt_names
[alt_names]
DNS.1 = Miele-001D63FFFE0313BA.local.
are optional. It is recommended to support them for future compatibility.

The certificate is generated with OpenSSL with the following commands:

```
# Create private key for rootCA
genrsa -out selfsigned-privkey.pem 2048

# Create self-signed certificate for rootCA
req -new -x509 -days 730 -sha256 -key selfsigned-privkey.pem -out selfsigned.pem -
config selfsigned.conf

# Print the rootCA certificate
x509 -in selfsigned.pem -text -noout

# Create DER versions of the rootCA certificate
```

```
x509 -inform PEM -outform DER -in selfsigned.pem -out selfsigned.der
```

6.4
Authentication and Authorisation

6.4.1
Authentication is application- and device-specific. Therefore, an application may decide to use HTTP Basic Authentication via HTTPs or a token-based approach (e.g. OAuth2, etc.).

Authorization for all resources MUST be handled via an access-control-list (ACL). Therefore, a *Host* must contain a list that describes which resource can be accessed by whom.

For example, it is possible that the service profProcessData is readable without any authorization but the service profProcessStatistics can only be accessed by a certain user.

In the ACL, the Miele Cloud Service MUST be a dedicated role. So, it is possible to read Ident and State service via the Cloud. The Cloud is authenticated during the TLS handshake via its certificate. Details can be found in [Miele_IP_Profile_Cloud_Connection_MCSv2].

Note: A proposal or details reagarding possible implementations of this required can be given by the editors of this document.

If a new device is developed according to this specification, the project team shall provide its proposed ACL for review. It will be reviewed from the persons named in chapter 1.2. After it is approved, it will be appended to this specification. This process ensures compatibility to the involved system blocks (Miele Cloud Service, etc.)

6.5
Normal Security Mode

6.5.1
For the normal security mode a *Device* has to:
- adopt all already mentioned security issues in this specification so far
- provide the mandatory *profSession* Service (without device pairing)

The *profSession* Service returns a (session) token. If (typically always) a service call requires authorization, this token has to be provided in the header field *Authorization* by the *Client*.

Example:

```
<HTTP-Method> /../profDisinfect/some/call  HTTP/1.1
```

Host: HostNameDev
Accept: application/vnd.miele.v1+json
Authorization: Bearer *<sessiontoken>*

6.6
Strict Security Mode

6.6.1
This chapter is PCS driven and complements the *Miele Normal Security Mode.*

Miele PCS-based medical devices use a *Strict(er) Security Model*, that's why PCS assumes some preconditions. The related topics are covered in the following chapters. Everybody is welcome to adopt the strict security model for more security on his *Device*.

Almost all services require an authentication. A Device can, as well as a user, authenticate itself using a token.

Especially the Device Pairing must be supported by any *Client* before a user-login will be allowed. More details can be found in the chapter *Device Pairing* below.

6.6.2
HTTPs / WebSocket Secure (WSS)

6.6.2.1
With the reference to the chapter above, Miele PCS allowes only HTTPs (TLSv1.2 in default), all other ports are blocked.

6.6.3
Session Management

6.6.3.1
Miele PCS uses a JSON Web Token (JWT) for the Session Management which is encrypted with a random secret. Any HTTPs connection to Miele PCS uses the session management.

6.6.4
Device Pairing

6.6.4.1

Any connected *Client* (even a *Device's* internal display) must be paired in a Miele PCS based *Device* for almost all resources. This guarantees that only allowed *Clients* are able to connect the *Device* (the PCS framework). After the pairing succeeded, a user login is possible.
As a *Client*, tokens can be received by providing a client's unique identifier and a given pin-code (password).

For new *Clients*, the *Device* (backend) first has to know about it, so a previously authorized third-person may allow the access and provide us with the required pin-code.

For a few cases, there is a simplified mechanism for getting a device session-token, e.g. for the MDU tool. In these cases, the unique device identifier and pins are predefined and a client can directly get their token by using the predefined data.
The device pairing is defined in the mandatory service *profSession.*

6.6.5
User Login

6.6.5.1

Any method is assigned to dedicated user rights. Any user (or client) is also assigned to dedicated user rights. When a user is logged in, he is allowed to execute only corresponding methods. The assigment is statically declared and can't be changed. After user log in succeeded, any allowed function is possible.
The user login is defined in the mandatory service *profSession.*

6.6.6
Device Levels

6.6.6.1

The optional parameter denotes the assigned *device level* (or device class) to a client.
Only the stated device levels are allowed to execute the corresponding function.
The *device level* is defined in the mandatory service profSession.

6.6.7
URI White Listing

6.6.7.1

The Miele PCS component WebServer supports white listing for known URIs. Any other requested URI will be rejected with Error Code 404.

6.6.8
ACL: ZSVA-devices

6.6.8.1
ZSVA devices use the following Access Control List:

Ressource	Access Control
<object:Root> /	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/Ident/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/State/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profErrorList/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profErrorLog/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profOperatingData/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profSession/	Special case: See chapter 4.7
/Devices/<id:Device>/profProgram/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profSensor/	Cloud: Readable User (or Client): depends on dedicated User Rights

	and Device Level
/Devices/<id:Device>/profProcessData/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profNotification/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profProcessStatistics/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profFileTransfer/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level
/Devices/<id:Device>/profDisinfect/	Cloud: Readable User (or Client): depends on dedicated User Rights and Device Level

6.7
General Security Best Practices

6.7.1
General security enhancements and best practices that shall be applied are

- Enforcement of strong passwords
- Removing unnecessary accounts
- Blocking unnecessary ports
- Restriction of version and software information
- Restricting access rights for processes
- Input validation
- Secure password storage
- Minimization of active modules and programs
- Using ASLR and PIE
- Supplying security patches
- Integrity protection against faulty hardware
- Continuous penetration tests
- Code and system review with external 3rd party security experts

7
Miele Cloud Connection

7.1
The *Miele Cloud Connection* is used for remote scenarios, e.g. remote service. The interface between a Miele Professional IP Device and the *Miele Cloud Service* is derived from the interface of Miele household appliances.

The protocol and all interfaces are specified in [Miele_IP_Profile_Cloud_Connection_MCSv2].

The main difference between household and professional appliances is the registration and pairing process. The pairing process in professional scenario involves a *Client-App* which uses interfaces needed for the configuration of the cloud connection. The corresponding services are specified in [Miele_IP_Profile_Cloud_Connection_MCSv2].

7.2
Miele Cloud Service Professional Host Names

7.2.1

PROD-Environment				
DEV				
Domain	**Global Name**	**EU Name**		
.professional.miele-iot.com	dispatch	dispatch-eu		
.professional.miele-iot.com	ntp	ntp-eu		
.professional.miele-iot.com	registration	registration-eu		
.professional.miele-iot.com	rest	rest-eu		
.professional.miele-iot.com	service	service-eu		
.professional.miele-iot.com	update	update-eu		
.professional.miele-iot.com	updatecontrol	updatecontrol-eu		
.professional.miele-iot.com	websocket	websocket-eu		
.professional.miele-iot.com	analytics-kibana	analytics-kibana-eu		
.professional.miele-iot.com	api	api-eu		
.professional.miele-iot.com	grafana	grafana-eu		

.professional.miele-iot.com	kibana	kibana-eu		
QS-Environment				
DEV				
Domain	**Global Name**	**EU Name**		
.qs.professional.miele-iot.com	dispatch	dispatch-eu		
.qs.professional.miele-iot.com	ntp	ntp-eu		
.qs.professional.miele-iot.com	registration	registration-eu		
.qs.professional.miele-iot.com	rest	rest-eu		
.qs.professional.miele-iot.com	service	service-eu		
.qs.professional.miele-iot.com	update	update-eu		
.qs.professional.miele-iot.com	updatecontrol	updatecontrol-eu		
.qs.professional.miele-iot.com	websocket	websocket-eu		
.qs.professional.miele-iot.com	analytics-kibana	analytics-kibana-eu		
.qs.professional.miele-iot.com	api	api-eu		
.qs.professional.miele-iot.com	grafana	grafana-eu		
.qs.professional.miele-iot.com	kibana	kibana-eu		
DEV-Environment				
DEV				
Domain	**Global Name**	**EU Name**		
.dev.professional.miele-iot.com	dispatch	dispatch-eu		
.dev.professional.miele-iot.com	ntp	ntp-eu		
.dev.professional.miele-iot.com	registration	registration-eu		
.dev.professional.miele-iot.com	rest	rest-eu		
.dev.professional.miele-iot.com	service	service-eu		
.dev.professional.miele-iot.com	update	update-eu		
.dev.professional.miele-iot.com	updatecontrol	updatecontrol-eu		
.dev.professional.miele-	websocket	websocket-eu		

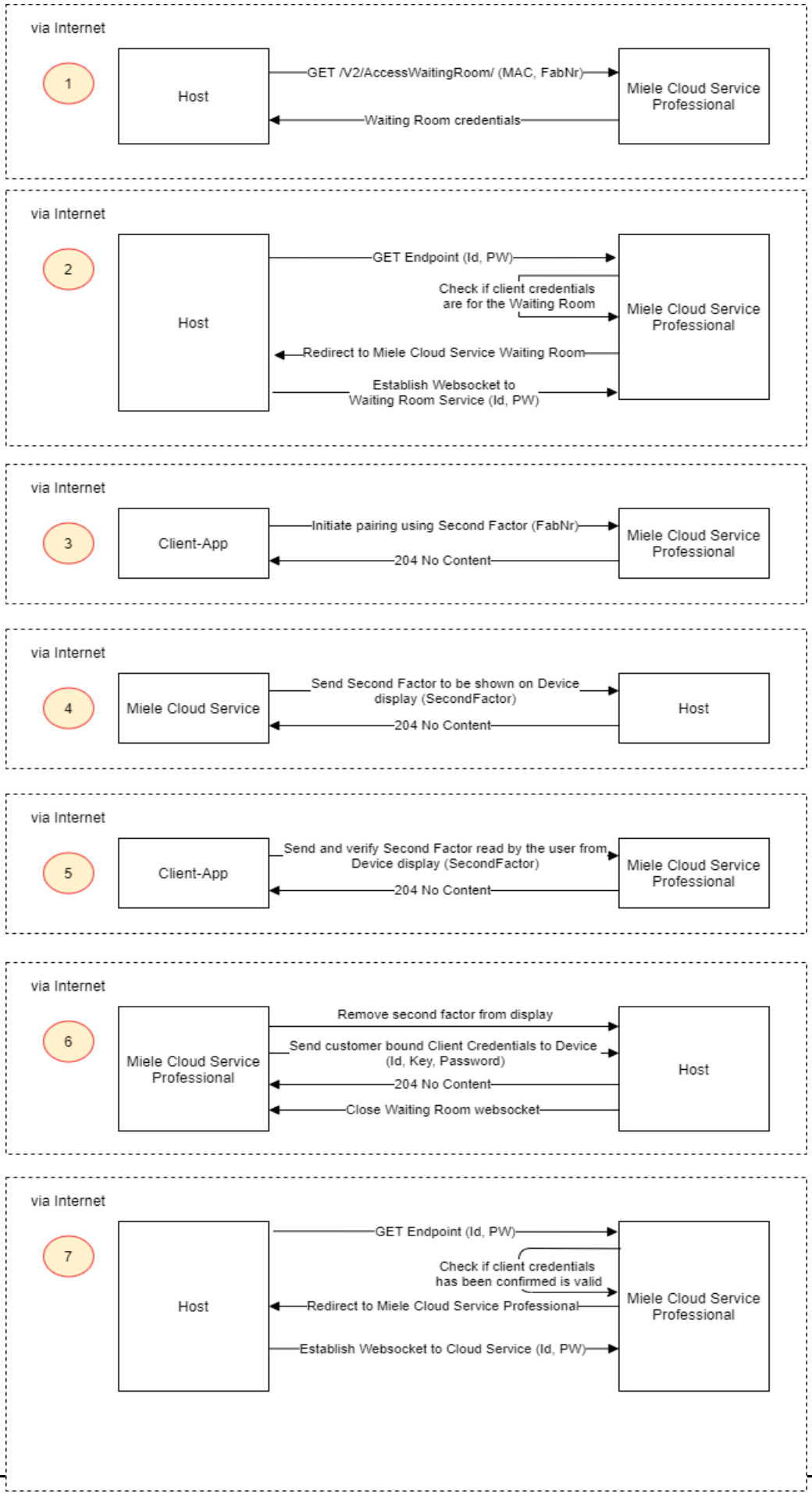
iot.com				
.dev.professional.miele-iot.com	analytics-kibana	analytics-kibana-eu		
.dev.professional.miele-iot.com	api	api-eu		
.dev.professional.miele-iot.com	grafana	grafana-eu		
.dev.professional.miele-iot.com	kibana	kibana-eu		

Remote-Update CDN (Host from which container files are downloaded)
There is not separation between PROD, QS and DEV.
The Host is

remoteupdate.professional.miele-iot.com

7.3
Registration and Pairing Process Overview

7.3.1
The registration and pairing process of a professional appliance follows the sequence flow in the following diagram:


via Internet
1
Host
GET /V2/AccessWaitingRoom/ (MAC, FabNr)
Waiting Room credentials
Miele Cloud Service Professional
via Internet
2
Host
GET Endpoint (Id, PW)
Check if client credentials are for the Waiting Room
Redirect to Miele Cloud Service Waiting Room
Establish Websocket to Waiting Room Service (Id, PW)
Miele Cloud Service Professional
via Internet
3
Client-App
Initiate pairing using Second Factor (FabNr)
204 No Content
Miele Cloud Service Professional
via Internet
4
Miele Cloud Service
Send Second Factor to be shown on Device display (SecondFactor)
204 No Content
Host
via Internet
5
Client-App
Send and verify Second Factor read by the user from Device display (SecondFactor)
204 No Content
Miele Cloud Service Professional
via Internet
6
Miele Cloud Service Professional
Remove second factor from display
Send customer bound Client Credentials to Device (Id, Key, Password)
204 No Content
Close Waiting Room websocket
Host
via Internet
7
Host
GET Endpoint (Id, PW)
Check if client credentials has been confirmed is valid
Redirect to Miele Cloud Service Professional
Establish Websocket to Cloud Service (Id, PW)
Miele Cloud Service Professional

7.4
Registration and connection to the Waiting Room

7.4.1
A professional appliance connects to the *Miele Cloud Service* automatically as soon as the internet connection is available. This happens even the appliance isn't paired so far. For not paired appliances a virtual, so called *Waiting Room,* is provided in the *Professional Miele Cloud Service* where the appliances establish the web socket connection to. The connection to the *Waiting Room* allows publishing of anononymous device information for analytics use cases and enables to initiate pairing to a customer account using arbitrary authentication mechanisms without limitation for operating a *Client-App* in the customer HAN.

The *Miele Cloude Service* ensures that an appliance connected to the *Waiting Room* is accessed from outside like a *Client-App* only for pairing purposes.

In the case of not existing cloud credentials (e.g. factory reset) a professional appliance performs automatically a registration process to the *Waiting Room* in order to acquire waiting room credentials (step 1 in the flow diagram). Those credentials are handled by the device as common *Cloud Credentials* the same as for Miele household appliances. Thus after acquiring access to the *Waiting Room* the credentials are used to establish a websocket connection to the *Miele Cloud Service* in the common way. The single difference is that the appliance will be redirected and connects to the *Waiting Room* instead to the *Cloud Service* (step 2 in the flow diagram).

7.4.2
GET AccessWaitingRoom

7.4.2.1
A Miele professional appliance acquires access to the *Miele Cloud Service Waiting Room* using this interface. In order to secure the *Waiting Room* to be accessed only by Miele appliances the professional XKM module uses the *Client Certificate Authentication* mechanism described in [RFC5246] with a dedicated certificate
stored on the module. The *Miele Cloud Service* verifies the provided certificate in SSL/TLS handshake and refuses the connection with a fatal alert in the case of a mistrust.

URL	**https://<HostMieleCloudProfessional>/V2/AccessWaitingRoom/**
HTTP Request	GET /V2/AccessWaitingRoom/ HTTP/1.1 Host: <*HostMieleCloudProfessional*> Accept: application/vnd.miele.v1+json Authorization: Basic Base64(user:password)
HTTP Request Payload	*empty*

Authentication	Authentication has to be done on two layers: 1) On SSL/TLS layer the client authenticates using a client certificate stored on each professional communication XKM module including the corresponding private key. 2) On HTTP layer the basic authentication is used the same as in domestic case to transfer the MAC and fabrication number to *Miele Cloud Service*. The basic credentials are encoded as: **User:** [MAC-Address]_PROF **Password:** ASCII-HEX(SHA256 ([FabNumber]))
HTTP Response Header	HTTP/1.1 200 OK Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <*decimal number of octets*>

HTTP Response Payload	{
	“CloudId”: “ABC234542abcd12345678654dhFDtE2z”,
	“CloudPassword”: “tE2zABC234542abcd12345678654dhFD”,
	“CloudKey”:“000102030405060708090A0B0C0D0E0F101112131415161718191A1B…”
	}

Response status codes	200 OK 404 Not Found 500 Internal Server Error (try again later)

7.5
Device pairing using Second Factor on the display

7.5.1
The pairing process for a professional appliance to a customer account is designed to support multiple mechanisms. In the first phase the pairing using a *Client-App* and a *Second Factor* shown on the appliance display is provided. This mechanism is described in this section with specification of the required interfaces on the XKM.

The *Second Factor* is a 8 digits number displayed on the appliance display to verify that a customer which is authenticated in the *Client-App* has physical access to the appliance and therefore authorized to pair it.

The pairing process of a professional appliance is initiated from the *Client-App* by a request to the *Miele Cloud Service* (step 3 in the flow diagram). The appliance is identified by its fabrication number with the expectation that the appliance is connected to the *Waiting Room*. The *Miele Cloud Service* will generate a random *Second Factor* and memorize it for later verification. The generated *Second Factor* is transmitted to the XKM using the service *POST /Security/Cloud/Pairing/SecondFactorOnDisplay/* in order to display the number on the display (step 4 in the flow diagram). In case of any further instructions the displayed *Second Factor* MUST be removed from the display after a timeout of 300 seconds. The pairing process completes

when the customer enters the read number from display into the *Client-App* which transmits the input to the *Miele Cloud Service* for verification (step 5 in the flow diagram). In the case the input value matches the expected *Second Factor* the *Miele Cloud Service* instructs the XKM using the service DELETE */Security/Cloud/Pairing/SecondFactorOnDisplay/* to remove the *Second Factor* from display. Afterwards the *Miele Cloud Service* transmits new *Cloud Credentials* to the XKM using the service *PUT /Security/Cloud/Credentials/* which indicates to close the current web socket connection to the *Waiting Room* (step 6 in the flow diagram). The reconnect to *Miele Cloud Service* using the new credentials (step 7 in the flow diagram) will ensure a connection to the full qualified *Cloud Service* in context of the paired customer account.

7.5.2
POST /Security/Cloud/Pairing/SecondFactorOnDisplay/

7.5.2.1
This service is only available from the Miele Cloud Service. In the HAN scenario the service responses with a 403 Forbidden status.

URL	https://<HostName>/Security/Cloud/Pairing/SecondFactorOnDisplay/
HTTP Request	POST /Security/Cloud/Pairing/SecondFactorOnDisplay/ HTTP/1.1 Host: <HostName> Accept: application/vnd.miele.v1+json Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <decimal number of octets>

HTTP Request Payload	{
	"SecondFactor": "<8-digits-number>"
	}

HTTP Response Header	HTTP/1.1 204 No Content
HTTP Response Payload	*empty*
Response status codes	204 No Content 403 Forbidden

7.5.3
DELETE /Security/Cloud/Pairing/SecondFactorOnDisplay/

7.5.3.1

This service is only available from the Miele Cloud Service. In the HAN scenario the service responses with a 403 Forbidden status.

URL	https://<HostName>/Security/Cloud/Pairing/SecondFactorOnDisplay/
HTTP Request	DELETE /Security/Cloud/Pairing/SecondFactorOnDisplay/ HTTP/1.1 Host: <HostName> Accept: application/vnd.miele.v1+json
HTTP Request Payload	*empty*
HTTP Response Header	HTTP/1.1 204 No Content
HTTP Response Payload	*empty*
Response status codes	204 No Content 403 Forbidden

7.5.4
POST /Security/Cloud/Credentials/

7.5.4.1
This service is only available from the Miele Cloud Service. In the HAN scenario the service responses with a 403 Forbidden status.

After responding to the request the XKM must close the current web socket connection and reconnect with the new credentials to the Miele Cloude Service.

URL	https://<HostName>/Security/Cloud/Credentials/
HTTP Request	POST /Security/Cloud/Credentials/ HTTP/1.1 Host: <HostName> Accept: application/vnd.miele.v1+json Content-Type: application/vnd.miele.v1+json; charset=utf-8 Content-Length: <decimal number of octets>

HTTP Request Payload	{
	“CloudId”: “ABC234542abcd12345678654dhFDtE2z”,
	“CloudPassword”: “tE2zABC234542abcd12345678654dhFD”,
	“CloudKey”:“000102030405060708090A0B0C0D0E0F101112131415161718191A1B…”
	}

HTTP Response Header	HTTP/1.1 204 No Content
HTTP Response Payload	*empty*
Response status codes	204 No Content 403 Forbidden

8
Customer Service IP Diagnosis

8.1
LAN: MDU Local Connection Establishment

8.1.1
Local diagnosis is performed between the Miele Professional *Device* and the MDU via a point-to-point Ethernet connection or a WiFi-connection.

The following description covers the point-to-point Ethernet connection.

Regarding the realization of the point-to-point connection, two possibilities exist
- Case 1: The *Host* has ONE Ethernet interface
  - In normal operation mode, the Device is connected to the customer network via this Ethernet interface. For diagnosis, the single Ethernet interface is reconfigured.
- Case 2: The *Host* has ONE or MORE Ethernet interfaces
  - In this case, the *Host* has a dedicated Ethernet interface for the point-to-point connection with the MDU.

8.1.2
Case 1: MDU Connection Establishment for Devices with ONE Ethernet interface

8.1.2.1

Step	Activities
1	The technician arrives at the *Host*.
2	The technician disconnects the Ethernet interface from the customer network
3	The *Host* recognizes that the Ethernet interface is disconnected

	(link is down).
4	The technician connects his laptop directly to the *Host* via Ethernet (point-to-point)
5	The laptop recognizes that no DHCP server is available and sets an IP address 192.168.10.XYZ (XYZ is not allowed to be in the range .90-.99).
6	The technician identifies himself on the *Device* UI (this may have been done also prior to this point).
7	The technician activates local diagnosis via the *Device* UI. (The Device MAY optionally require an additional authorization from the technician, e.g. “please reenter your password”). If the *Device* detects a DHCP server, it is most likely connected to an infrastructure. In this case it shows a message (e.g. “Local diagnosis cannot be activated within an infrastructure”) and it aborts the process.
8	The *Host* saves the current configuration and it reconfigures the Ethernet interface for local diagnosis and sets the IP 192.168.10.90.
9	If the *Host* is rebooted, the point-to-point diagnosis mode is left and the previous configuration is restored. (*This is valid for every subsequent step.*)
10	The MDU sends a GET to https://192.168.10.90/Devices/ on port 443 **without** authorization, due to MDU requirements. • If the certificate check fails, the MDU should request a user action from the MDU user (“Invalid certificate, click here to proceed anyway”). It should show the user the certificate details according the description in chapter 6.3

11	The *Host* replies with the *Devices* list.
	
	For example
	{
	“123456789101”: {
	“href”:”123456789101/”,
	“Group”: “Professional”
	}
	}

12	The MDU sends a GET to https://192.168.10.90/Devices/<FabNr>/ **without** authorization and evaluates the location of <service:profSession>.
13	The MDU sends a GET to https://192.168.10.90/Devices/<FabNr>/<sevice:profSession>.href/ **without** authorization. Then, it evaluates the version information and the returned href locations. It must follow the links returned via href in the next steps. (*Note: They may point to other device specific locations.*)
14	Security must be performed according to chapter 6. **Case 1) Devices with high security (version 200) use a two-stage authentication** 1. Device pairing For PCS, the device pairing must be performed according to the description in chapter 6.5, with the difference that the

	device pairing key will be auto generated (due to MDU useability no manuel input for the pin is necessary). 2. Login with username and password **<u>Case 2) Devices with normal security (version 100)</u>** 1. Login with username and password
15	The MDU is logged in and sends a GET to https://192.168.10.90/Devices/<FabNr>/Ident/
16	The MDU reads the *SWIDs*, performs additional checks and loads the corresponding DUDs.
17	The subsequent steps from the MDU are performed according to the description in the DUD.

8.1.3

<u>Case 2: MDU Connection Establishment for Devices with TWO or MORE Ethernet interfaces</u>

8.1.3.1

Step	Activities
1	The technician arrives at the *Host*.
2	*(OPTIONAL – RECOMMENDED)* The technician identifies himself on the *Device* UI (this may have been done also prior to this point).
3	*(OPTIONAL – RECOMMENDED)* The technician activates local diagnosis via the *Device* UI. (The Device MAY optionally require an additional authorization from the technician, e.g. “please reenter your password”).
4	The technician connects his laptop directly to the *Host* via the dedicated Ethernet interface for customer service diagnosis (point-to-point). The dedicated Ethernet interface is fixed configured to the IP address 192.168.10.90.
5	The laptop recognizes that no DHCP server is available and sets an IP address 192.168.10.XYZ (XYZ is not allowed to be in the range .90-.99).
6	The remaining steps are identical to case 1 steps 10-15.

Note: Due to service requirements, the MDU plug must be configured as 192.168.10.0/24 subnet. Depending on customer’s Network which is connected to the first Ethernet plug, conflicts may occur in that subnet for 192.168.10.1 up to 192.168.10.255.

Recommendation: It should be mentioned in the manual, that the subnet 192.168.10.0/24 should be excluded in the customer’s network.

8.1.4

Handling of Username and Password for Customer Service Login

8.1.4.1

In general, the service technician must login with a username and password according to the description given in the previous two sections.

Therefore, a corresponding user must be existing on the *Device*.

If no user is existing on the *Device*, and if the *Device* is a PCS *Device* the user for the MDU can be added manually via the local *Device* UI with the following steps:

1) The technician can perform a login via the local *Device* UI by using the so-called "Tagespasswort" (Daily-Password). Details regarding the Daily-Password can be requested by the editors of this document.
2) The technician can add a user for the MDU via the local *Device* UI.
3) The technician uses this user (added in 2) for the previous process in order to perform login with the MDU (Step 14)
4) After diagnosis has finished, the technician has to manually delete the user (added in 2) via the *Device* UI

The user (added in 2) must have a password with at least 8 characters, at least 1 numerical and at least 1 capital letter. If characters are not supported by the Device UI, a password with at least 10 digits must be used instead.

"Tagespasswort" (Daily-Password)
Although the Device and the MDU Tool are completely independent, both must know the "Tagespasswort". For this reason, **following algorithm has to be implemented** on both sides (inner brackets are processed first):

Tagespasswort := SubString(HEX_DEC(HMAC_SHA256(KEY_SECRET, CONTENT)), 10)

f(Tagespasswort) Part	Explanation
SubString	Extracts a fragment of the input string with the length 10. If the input string is less than 10, leading '0's have to be added. SubString must always result in a string with the length of 10 chars. Example: resultAsString := SubString(“3044298114683069345”, 10) resultAsString contains the string “3044298114”
HEX_DEC	*HEX:* the HMAC result is converted into a hex-string first Result Example: “e**3**b**0**c**44298**fc**1**c**14**b... ” *DEC*: then any (upper or lower case) hex-char is ignored Result Example: “3044298114...”
HMAC_SHA256	Standard IT algorithm (reference: rfc4634)

KEY_SECRET	A secret, e.g. a UUID as string UUID-Example: “5d935b43-4416-4054-b9ac-53d2523c150a” 1. It is known by very few persons which work on the Device or MDU. 2. It is strictly prohibited to email or publish it. 3. It must not be stored without further encrypting on persistent flash 4. it is only allowed to decrypt it in memory
CONTENT	The CONTENT is a concatenation of two values, separated by a ‘#’ with: **Timestamp** Today’s date without time, format: yyyymmdd as string (length is fixed to 8) *Example:* “20171214” or "20170101" (with leading '0') **FabNumber** according the chapter “Service: Device Ident” as string *Example:* “12345678” Example: CONTENT := “20170101” + ‘#’ + “12345678”

Example:
Tagespasswort *:=*
SubString(HEX_DEC(HMAC_SHA256("5d935b43-4416-4054-b9ac-53d2523c150a", "20170101#12345678")), 10)
results in
Tagespasswort *:= 7962592024*
(from HEX: *7f962d59c20bf2eb4216bf6c240db9e30090a54113cba70b9842b37c1cb11b31*)

8.2
WLAN: MDU Connection Establishment

8.2.1
Diagnosis Soft-AP

8.2.1.1
Local diagnosis is performed via a Soft-AP that is generated by the appliance. The SSID of the Soft-AP is constructed from the fabrication number and an additional random field with 8 characters constructed from 0-9, A-F.

It is constructed as

SSID := Miele-<Fabrication Number>-<Random Field>

If the Fabrication Number is empty/not existing, the field uses all 0s (resulting in "-000000000000-". For example:

SSID := Miele-000000000000-010203ABD10A

In addition it is also valid to use a single 0 (Miele-0-01020304...).

The Soft-AP uses WPA2 encryption. The passphrase is calculated according to the following rules

WPA_Passphrase := Base64(HMAC_SHA256(*KEY_DIAGNOSIS*, SSID) [max 32 characters]

The SSID is hashed with HMAC_SHA256 with the *KEY_DIAGNOSIS*. The result is encoded as Base64 and reduced to a maximum length of 32 characters.

Note: *KEY_DIAGNOSIS* can be requested from the editor of this document.

8.2.2
Activation Soft-AP and Connection Establishment (MDU)

8.2.2.1
The XKM opens the *Customer Service Soft-AP* after a user-interaction on the device UI. If the XKM is connected to the customer network, it saves the connection details in order to reconnect to it, after the *Customer Service Soft-AP* is closed. If the XKM is connected via LAN to the customer network, the LAN interface is disabled (only 1 interface Wifi or LAN can be active at the same time).

The XKM opens the *Customer Service Soft-AP* for 1 minute (*Note: Due to technical reasons the Soft-AP opens a time interval between 1 and 2 minutes, the minimum time is 1 minute*). The XKM offers a DHCP server on the *Customer Service Soft-AP*, and it can detect if a client has connected to the *Soft-AP*. The MDU is recognized as a Client as soon as it is connected.

As long as a connection to a client is established, the XKM leaves the *Customer Service Soft-AP* open (no timeout is used).

If the client disconnects, the XKM uses a timeout of 1..2 minutes (*Note: It might be larger than 1 minute due to technical reasons*). If no new client reconnects within 1..2 minutes, the XKM closes the *Customer Service Soft-AP*. It resumes with the previous connection (e.g. reconnect to customer network, etc.).

Since the XKM acts as DHCP server, the MDU can use the IP address of the DHCP server for the following connection steps. The port used is 443 (HTTPs) for IP Profile-Extended and 80 (HTTP) for IP Professional-Basic.

8.2.3
Additional Connection Establishment Steps for MDU

8.2.3.1
After the connection to the *Customer Service Soft-AP* has been established, the MDU can continue to read data and communicate with the Device via DOP2.

It is not necessary for the MDU to perform additional authorization. The authorization is performed via the connection to the Soft-AP.

8.2.4
Disconnection (MDU)

8.2.4.1
After the diagnosis is finished, the client PC must disconnect from the Soft-AP. In addition to that, the diagnosis software MUST delete all related credentials and profiles that might have been saved and used during the diagnosis on the PC that runs the diangnosis software (e.g. MDU).

Then, the Device will start a timer between 60s and 120s (randomly) and close the Soft-AP. If an additional diagnosis shall be performed, the Soft-AP must be activated again.

8.3
Write Ident

8.3.1
The *DeviceIdent* can be written by the customer service technician. Details are given in chapter 4.2.8.

8.4
Memory Dump

8.4.1
This function makes possible to analyse error states remotely. If a error state is active it can be reported to the cloud. The field service can get a memory dump which deliveres the same information like DUD-File. Memory dump contains all the readable DOP-Objects. Nevertheless it is not possible to write or change DOP-Objects. In summary it is a function which gives the field service the possibility to get all the DOP-Objects and start an analyse remotely to handle the error.

The memory dump has to be started over the cloud. If the dump is bigger than the buffer you will see on the cloud side "HasNext" which is a information that the dump is bigger and is has to be triggered again to complete the download.

9
QoS Requirements

9.1
A Miele Professional *Host* MUST be able to serve at least 3 parallel clients in the local network.

If the *Host* is connected to the Miele Cloud, it MUST be able to serve the *Miele Cloud* in addition to the three local clients.

If it is not possible to serve an additional client, the *Host* MUST signal this via **HTTP Status Code 503**. The *Host* MUST set to the **Retry-After** header to a reasonable value. The client shall try again after the time interval that is returned via *Retry-After*.

A *Host* MUST take measures in order to block *Denial of Service* attacks.

10
Miele Professional IP Profile Basic

10.1
This specification framework targets at professional appliances. In addition to that there are some types of appliances that can be located somewhere between professional and domestic appliances. These so-called semi-professional appliances are equipped with an onboard WiFi communication module. This module supports a subset of the IP Profile Professional in order to support simple remote status, monitoring and laundry use-cases.

The subset is called **IP Profile Professional Basic** and it is an extension of the IP Profile for domestic appliances.

Hosts have to comply with the mandatories from the domestic *Core profile* according to [Miele_IP_Profile_Core_Framework]. Therefore, the so-called *Groupmode* security is applied for the *IP Profile Professional Basic*. *Hosts* announce their mDNS service as *mielesemiprof._tcp.local.*

Additionally, *Hosts* support several optional (domestic-based) services according to [Miele_IP_Profile_Services], e.g. Remote Update, WiFi comissioning and customer service diagnosis via WiFi.

Regarding the professional services framework [Miele_Professional_IP_Profile_Services], it is described within the relevant chapters, if the corresponding service supports a Basic variant.

11
End-of-Line via Ethernet

11.1
This specification part applies, if the *Device* supports end-of-line (EOL) via Ethernet during production.

The *Device* Control sets the communication module into a certain state, in case EOL has not been successfully finished before. If the EOL has been finished, the *Device* Control MUST NOT set the communication module into this state. In this case it is only possible to activate the mode via the customer service diagnosis.

During EOL, the EOL terminal can read/modiy data and communicate with the *Device* via DOP2.

It is not necessary for the EOL terminal to perform additional authorization. Therefore, it must be ensured that this mode is ONLY used in a secure environment during the production! Besides that it MUST be ensured that this mode is only entered by the *Device* Control, if EOL has not been finished before.

Abbildungen:

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.