Lua User Manual

Rev. 2023-10-10

Contents

Getting started

Lua is a lightweight, high-level, multi-paradigm programming language designed primarily for embedded use in applications and extending their functionality.

The language has extensive documentation on its official site here.

It allows you to capture events in our system and perform specific actions or sequences using simple references to specific elements of the application, mainly in the form of automations and scenes popularly called **scripts**.

Managing (adding, removing, editing) scenes and automations is done via REST API or a web application served through the central unit server.

In scripts, you must not use blocking functions, e.g. `delay` or long-acting loops, the script should execute as soon as possible or it will block execution queue for other scripts.

Events

Event is an action or occurrence done by device, automation, scene or user in system that may be handled by user in automations.

If events occurs, system will trigger run-cycle through all lua automations which are not banned and enabled in order to perform user defined actions.

It will contain info only when executing in context of automation (its empty in scenes / deferred actions context). You can refer to it using `event` global scope object.

Properties

- `type` *(string)*

  Type of event

- `details` *(string)*

  More detailed info about event eg. if event refers to device state change then details will contain name of attribute that was recently changed.

Available types

- `application_initialized`

  Occurs once at application start, can be used as initializator of automations etc.

- `device_state_changed`

  Occurs when one of device attribute was changed by user, automation or scene.

- `minute_changed`

  Occurs cyclically once per minute.

- `scene_activated`

  Occurs on scene activation.

- `scene_state_changed`

  Occurs when one of scene attribute was changed by user, automation or scene.

- `scene_failed`

  Occurs when scene failed eg. due to syntax error.

- `automation_state_changed`

  Occurs when one of automation attribute was changed by user, automation or scene.

- `automation_failed`

  Occurs when automation failed eg. due to syntax error.

- `lua_timer_elapsed`

  Occurs when lua timer has elapsed after start for specific time.

- `sunrise`

  Occurs once a day at sunrise.

- `sunset`

  Occurs once a day at sunset.

- `mqtt_client_connected`

  Occurs when Mqtt client connects to broker (when CONACK received).

- `mqtt_client_disconnected`

  Occurs when Mqtt client disconnects from broker e.g. due to network error

- `mqtt_client_message_received`

  Occurs when Mqtt client receives message at subscribed topic.

- `http_client_request_failed`

  Occurs when request sent by http client failed, e.g. due to internet connection problems.

- `http_client_response`

  Occurs when received response on http client after sending request.

- `lua_http_server_request`

  Occurs when Lua Http Server receives request.

- `activate_scene_by_id`

  Occurs when external device activates scene.

- `custom_device_element_state_changed`

  Occurs when one of custom device element attribute was changed by user, automation or scene.

- `custom_device_element_stateless_event`

  Occurs when one of custom device element is touched without changing its state (eg. button is pressed).

Examples

More detailed event-based examples for specific object types can be found in chapters related to them.

Check if device parameter changed

```
if event.type == "device_state_changed" and event.details ==
   "target_temperature" then

  print("One of devices changed state!")

end
```

NOTE: You can't check to which device event refers to. In order to do it you need to use device object directly. Same rule applies to scenes, automations, timers etc, more detailed event-based examples for these object types can be found in chapters related to them. Example:

```
if wtp[4]:changedValue("target_temperature") then

  print(
     "Wireless WTP Temperature regulator with ID 4 changed target temperature!")

end
```

Catch scene activation or failure

```
if event.type == "scene_activated" then

  print("One of scenes was activated!")

end

if event.type == "scene_failed" then

  print("One of scenes failed!")

end
```

NOTE: You can't check to which scene event refers to. In order to do it you need to use scene object directly. Same rule applies to scenes, automations, devices, timers etc, more detailed event-based examples for these object types can be found in chapters related to them. Example:

```
if scene[4]:activated() then

  print("Your scene with ID 4 activated!")

end

if scene[4]:failed() then

  print("Your scene with ID 4 failed!")

end
```

Catch automation fail

```
if event.type == "automation_failed" then

  print("One of automations failed!")

end
```

NOTE: You can't check to which automation event refers to. In order to do it you need to use automation object directly. Same rule applies to scenes, automations, devices, timers etc, more detailed event-based examples for these object types can be found in chapters related to them. Example:

```
if automation[4]:failed() then
  print("Your automation with ID 4 failed!")
end
```

Check if a minute elapsed

```
if event.type == "minute_changed" then
  print("Another minute elepased!")
end
```

Check if a timer elapsed

```
if event.type == "lua_timer_elapsed" then
  print("One of lua timers elapsed!")
end
```

NOTE: You can't check to which timer event refers to. In order to do it you need to use timer object directly. Same rule applies to scenes, automations, devices etc, more detailed event-based examples for these object types can be found in chapters related to them. Example:

```
if timers[4]:isElapsed() then
  print("Lua timer with ID 4 elapsed!")
end
```

Catch sunrise event

```
if event.type == "sunrise" then
  print("Sunrise starts now!")
end
```

Catch sunset event

```
if event.type == "sunset" then
  print("Sunset starts now!")
end
```

Devices

Devices are exposed as key-based containers of objects with name matching their class. Currently available containers:

- `wtp` - wireless WTP devices
- `tech` - wired TECH RS devices
- `virtual` - virtual devices added via web-app
- `sbus` - wired SBUS devices
- `modbus` - wired MODBUS devices
- `system_module` - system modules (eg. transceivers or signal extenders)
- `alarm_system` - Integrated Alarm system devices
- `lora` - Lora devices (available only in Sinum Pro)

Containers store devices in the form of a key corresponding to the device ID. For example, when you want to refer to a **WTP** device with an **ID 4** you should use: `wtp[4]` object.

Same for the rest eg. `tech[66]` gives you access to **TECH RS** device with **ID 66**.

Devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` or `setValueAfter` methods.

For more details eg. available properties refer to specific device class/type documentation.

Attempting to reference a nonexistent device object, retrieve a nonexistent device property, or set the wrong value type will result in a script error.

Methods

- `changed()`

  Checks if one of device property has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of device has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property which should be checked

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string)* - name of property

- `setValue(property_name, property_value)`

  Sets value for object property.

  Returns:

  - *(userdata)* - reference to device object, for chained calls

  Arguments:

  - `property_name` *(string)* - name of property
  - `property_value` *(any)* - property type dependant value which should be set

- `setValueAfter(property_name, property_value, seconds_after)`

  Sets value for object property after certain time.

  Returns:

  - *(userdata)* - reference to device object, used for chained calls

  Arguments:

  - `property_name` *(string)* - name of property
  - `property_value` *(any)* - property type dependant value which should be set
  - `seconds_after` *(int)* - number of seconds after which the action will take place

- `call(command_name, arg)`

  Runs a device command.

  Returns:

  - *(userdata)* - reference to device object, for call chains

  Arguments:

  - `command_name` *(string)* - name of command available for device
  - `arg` *(any, optional)* - argument for command

- `hasTag(tag)`

  Returns true if device has tag specified in parameter.

  Returns:

  - *(boolean)*

  Arguments:

  - `tag` *(string)* - tag name

Commands

User can execute specific actions for devices by using commands (eg. fully open roller blinds) instead of changing attributes.

For more details eg. available commands refer to specific device class/type documentation.

Examples

Check if any property changed

```
if wtp[55]:changed() then
  print("Wireless WTP device with ID 55 changed!")
end
```

Check if specific property changed

```
if wtp[55]:changedValue("signal") then
  print("Wireless WTP device with ID 55 changed signal!")
end
```

Get value of a device property

```
if wtp[4]:getValue("open") then
  print("Window is open!")
else
  print("Window is closed!")
end
```

Set value of a device property

```
print("Lights ON!")
wtp[9]:setValue("state", true)
```

Set more than one property at once with chained calls

```
wtp[9]
  :setValue("state", true)
  :setValue("name","Lights ON")
  :setValueAfter("state", false, 300)
  :setValueAfter("name", "Lights OFF", 300)
```

Set value of device property after certain time

```
print("Lights will turn OFF after 30 seconds!")
wtp[9]:setValueAfter("state", false, 30)
```

Call device commands

```
tech[5]:call("toggle")
wtp[3]:call("open", 55)
```

Scenes

One-time execution of a sequence of actions programmed by the user, eg the scene **"I'm leaving the house"** may close the blinds and lower the target temperature in room.

Scene may be added, edited or deleted via REST API or a web application served through the central unit server.

Activation and property modification is possible via REST API, web app or directly from scripts using `scene` container eg. `scene[6]` gives you access to scene with **ID 6**.

Scenes have global scope and they are visible in all executions contexts.

NOTE: you must not use blocking functions, e.g. `delay` or long-acting loops, the scene should execute as soon as possible or it will block execution queue for other scenes.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent scene object, retrieve a nonexistent scene property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier

- `name` *(string)*

  User defined name of scene. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `enabled` *(boolean)*

  Defines if scene is enabled or not. In other words, it means if it's possible to execute that scene or not.

- `banned` *(boolean, read-only)*

  Smiliar to `enabled` proproperty but set by system. Defines if scene failed and is excluded (not able to execute) when condition `error_counter >= max_errors` is met.

- `error_counter` *(integer, read-only)*

  Error counter counts error on every fail of scene eg. syntax error or exceeding execution time.

- `max_errors` *(integer, read-only)*

  Maximum possible errors counted before scene gets banned. Defined by user via REST API.

- `max_execution_time` *(integer, read-only)*

  Maximum possible execution time in seconds before scene will get terminated with error. Defined by user via REST API.

- `labels` *(array, read-only)*

  Collection of scene specific labels.

  e.g. information if scene is added to room.

- `room_id` *(integer, read-only)*

  ID of room with which scene is associated or `nil` otherwise.

- `dir_id` *(integer, read-only)*

  ID of directory where the scene is.

- `tags` *(table, read-only)*

  Collection of tags assigned to scene.

Methods

- `changed()`

  Checks if one of scene property has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of scene has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property which should be checked

- `activated()`

  Checks if scene was activated (thus is source of event).

  Returns:

  - *(boolean)*

- `failed()`

  Checks if scene was failed (thus is source of event).

  Returns:

  - *(boolean)*

- `hasTag(tag)`

  Returns true if scene has tag specified in parameter.

Returns:

- *(boolean)*

Arguments:

- `tag` *(string)* - tag name

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string)* - name of property

- `setValue(property_name, property_value)`

  Sets value for object property.

  Returns:

  - *(userdata)* - reference to scene object, for call chains

  Arguments:

  - `property_name` *(string)* - name of property
  - `property_value` *(any)* - property type dependant value which should be set

- `activate()`

  Activates scene.

  Returns:

  - *(userdata)* - reference to scene object, for call chains

- `activateAfter(seconds_after)`

  Activates scene after certain time.

  Returns:

  - *(userdata)* - reference to scene object, for call chains

  Arguments:

  - `seconds_after` *(integer)* - number of seconds after which the action will take place

- `call(command_name, arg)`

  Calls scene to execute commmand.

  Returns:

  - *(reference to scene object)*

  Arguments:

  - `command_name` *(string)* - name of command available for scene
  - `arg` *(any, optional)* - argument for command

Commands

- `activate`

  Another way to activate a scene.

Examples

Activate a scene at sunrise

```
if event.type == "sunrise" then
  scene[3]:activate()
end
```

Change scene properties with chained calls

```
scene[3]
  :setValue("enabled", false)
  :setValue("name", "Temporary turned off")
```

Sample scenes: ”leaving home” and ”returning home”

”Leaving Home” scene saves current device presets to a variable before changing them, so the ”Returning Home” scene can restore them later.

NOTE: Global lua string variable is required, you can create one via application in configuration.

Leaving Home

```
-- store current configuration into local table
local dataToSave = {
  thermostat_temperature_1 = virtual[149]:getValue("target_temperature"),
  thermostat_temperature_2 = virtual[150]:getValue("target_temperature"),
  blind_opening_1 = wtp[290]:getValue("target_opening"),
  blind_opening_2 = wtp[291]:getValue("target_opening")
}

-- serialize data into string and save it to global variable
variable[4]:setValue(JSON:encode(dataToSave))

-- change device values to home away ones
virtual[149]:setValue("target_temperature", 150)
virtual[150]:setValue("target_temperature", 150)
wtp[290]:setValue("target_opening", 0)
wtp[291]:setValue("target_opening", 0)
```

Returning Home

```
-- Restore device parameters saved by the "leaving home" scene.

-- deserialize previously stored data into local table
local dataToLoad = JSON:decode(variable[4]:getValue())

-- restore previous configuration
virtual[149]:setValue("target_temperature", dataToLoad.thermostat_temperature_1)
```

```
virtual[150]:setValue("target_temperature", dataToLoad.thermostat_temperature_2)
wtp[290]:setValue("target_opening", dataToLoad.blind_opening_1)
wtp[291]:setValue("target_opening", dataToLoad.blind_opening_2)
```

Automations

Cyclical algorithms that are always ran on every event. The user is responsible for ”catching” the event and performing a specific action on its basis, eg based on the movement in the room, automatically turn on and off the light.

Automation may be added, edited or deleted via REST API or a web application served through the central unit server.

Access is possible via scripts using `automation` container eg. `automation[6]` gives you access to automation with **ID 6**.

Automation have global scope and they are visible in all executions contexts.

NOTE: you must not use blocking functions, e.g. `delay` or long-acting loops, the automation should execute as soon as possible or it will block execution queue for other automation.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent automation object, retrieve a nonexistent automation property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier

- `name` *(string)*

  User defined name of automation. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `enabled` *(boolean)*

  Defines if automation is enabled or not. In other words, it means if it's possible to execute that automation or not.

- `banned` *(boolean, read-only)*

  Smiliar to `enabled` proproperty but set by system. Defines if automation failed and is excluded (not able to execute) when condition `error_counter >= max_errors` is met.

- `ban_reason` *(string, read-only)*

  Reason why automation was banned.

- `error_counter` *(integer, read-only)*

  Error counter counts error on every fail of automation eg. syntax error or exceeding execution time.

- `max_errors` *(integer, read-only)*

  Maximum possible errors counted before automation gets banned. Defined by user via REST API.

- `max_execution_time` *(integer, read-only)*

  Maximum possible execution time in seconds before automation will get terminated with error. Defined by user via REST API.

- `dir_id` *(integer, read-only)*

  ID of directory where the automation is.

- `tags` *(table, read-only)*

  Collection of tags assigned to automation.

Methods

- `changed()`

  Checks if one of automation property has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of automation has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property which should be checked

- `failed()`

  Checks if automation has failed (thus is source of event).

  Returns:

  - *(boolean)*

- `hasTag(tag)`

  Returns true if automation has tag specified in parameter.

  Returns:

  - *(boolean)*

  Arguments:

  - `tag` *(string)* - tag name

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - property value

  Arguments:

  - `property_name` *(string)* - name of property

- `setValue(property_name, property_value)`

  Sets value for object property.

  Returns:

  - *(userdata)* - automation object reference, for chaining calls

  Arguments:

  - `property_name` *(string)* - name of property
  - `property_value` *(any)* - property type dependant value which should be set

Examples

Check if automation failed

```
if automation[2]:failed() then
  print("Automation failed!")
end
```

Change automation properties

```
automation[3]
  :setValue("enabled", false)
  :setValue("name", "Disabled")
```

Rooms

Rooms are exposed as a key-based container of objects `room`.

Container stores rooms in the form of a key corresponding to the room ID. For example, when you want to refer to a **Room** with **ID 4** you should use: `room[4]` object.

Attempting to reference a nonexistent room object, retrieve a nonexistent room property, or set the wrong value type will result in a script error.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier

- `name` *(string)*

  User defined name of room. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `icon` *(string)*

  User defined icon of room.

- `color` *(string, read-only)*

  User defined icon of room.

- `has_error` *(boolean, read-only)*

  Indicates if any associated device has error.

- `has_warning` *(boolean, read-only)*

  Indicates if any associated device has warning.

- `is_heating` *(boolean, read-only)*

  Indicates if any associated thermostat is currently in heating mode.

- `is_cooling` *(boolean, read-only)*

  Indicates if any associated thermostat is currently in cooling mode.

- `labels` *(array, read-only)*

  Collection of room specific labels. e.g. information if room is added to floor.

- `floor_id` *(integer, read-only)*

  ID of floor with which the room is associated or null otherwise.

- `is_window_open` *(boolean, read-only)*

  Informs whether there is window opened in any associated thermostat.

Methods

- `changed()`

  Checks if one of room property has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of room has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property which should be checked

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string)* - name of property

- `setValue(property_name, property_value)`

  Sets value for object property.

  Returns:

  - *(userdata)* - reference to room object, for call chains

  Arguments:

  - `property_name` *(string)* - name of property
  - `property_value` *(any)* - property type dependant value which should be set

- `foreach(function)`

  Executes function for each device added to room. Function should have following signature: `function (dev)` where `dev` is device in room.

  Arguments:

  - `function` *(function)* - function that will be executed for all devices

- `foreach(tag, function)`

  Executes function for each device added to room with specified tag. Function should have following signature: `function (dev)` where `dev` is device in room.

Arguments:

- `tag` *(string)* - tag of device which will be checked when choosing devices to execute function
- `function` *(function)* - function that will be executed for all devices with specified tag

- `getDevicesByTag(tag)`

  Returns all devices added to room with specified tag.

  Returns:

  - *(table)* - sequence with device objects

  Arguments:

  - `tag` *(string)* - tag of device which will be returned

Examples

Change room properties

```
room[3]:setValue("name", "Bedroom")
```

Turn on all devices in room

```
room[2]:foreach(function (dev)
  dev:setValue("state", true)
end)
```

Turn on all devices in a room which have tag 'light'

```
room[2]:foreach("light", function (dev)
  dev:call("turn_on")
end)
```

List all devices with tag 'regulator' in a room

```
utils.table:forEach(room[2]:getDevicesByTag('regulator'), function (reg)
  print(reg:getValue('name'))
end)
```

Floors

Floors are exposed as key-based container of objects `floor`.

Container stores floors in the form of key corresponding to the floor ID. For example, when you want to refer to a **Floor** with **ID 4** you should use: `floor[4]` object.

Attempting to reference a nonexistent floor object, retrieve a nonexistent floor property, or set the wrong value type will result in a script error.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of room. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `level` *(integer)*

  User defined value indicating at which level the floor is. This value has to be unique accross all floors.

Methods

- `changed()`

  Checks if one of the floor property has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of floor has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property which should be checked.

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

Arguments:

- `property_name` *(string)* - name of property.

- `setValue(property_name, property_value)`

  Sets value for object property.

  Returns:

  - *(userdata)* - reference to floor object, for call chains

  Arguments:

  - `property_name` *(string)* - name of property.
  - `property_value` *(any)* - property type dependant value which should be set

Examples

Change floor properties

```
floor[4]:setValue("name", "Ground floor");
floor[4]:setValue("level", 0);
```

DateTime

Global scope object containing date and time information.

Available in all contexts. You can access it using `dateTime` object.

Methods

- `changed()`

  Checks if minute changed.

  Returns:

  - *(boolean)*

- `getDay()`

  Day of month according to local time configured in system.

  Returns:

  - *(number)* - integer in 1–31 range

- `getMonth()`

  Month of year according to local time configured in system.

  Returns:

  - *(number)* - integer in 1–12 range

- `getYear()`

  Year according to local time configured in system.

  Returns:

  - *(number)* - integer, ≥2020

- `getSeconds()`

  Seconds according to local time configured in system.

  Returns:

  - *(number)* - integer in 0–61 range (leap seconds)

- `getMinutes()`

  Minutes of hour according to local time configured in system.

  Returns:

  - *(number)* - integer in 0–59 range

- `getHours()`

  Hour according to local time configured in system.

  Returns:

  - *(number)* - integer in 0–23 range

- `getWeekDay()`

  Day of week according to local time configured in system starting from sunday (index 0)

  Returns:

  - *(number)* - integer in 0–6 range

- `getWeekDayString()`

  Day of week according to local time configured in system represented as string

  Returns:

  - *(string)* - week day name, one of: `sunday`, `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, `saturday`

- `getTotalTime()`

  Total time elapsed since 1970-01-01 in seconds.

  Returns:

  - *(number)* - Unix timestamp integer

- `getTimeZoneOffset()`

  Returns current time zone offset in seconds including daylight saving time offset.

  Returns:

  - *(number)* - integer

- `isDaylightSavingTimeActive()`

  Return info whether DST is currently active or not.

  Returns:

  - *(boolean)*

- `getTimeOfDay()`

  Minutes of day in local time, since 00:00 eg. 750 is equal to 12:30

  Returns:

  - *(number)* - integer in 0–1439 range

Examples

Perform an action every minute

```
if dateTime:changed() then
  print("Another minute elepased!")
end
```

Get current time

```
message = string.format(
  "It is %s, %02d:%02d:%02d",
  dateTime:getWeekDayString(), dateTime:getHours(),
  dateTime:getMinutes(), dateTime:getSeconds())

print(message)
```

Perform actions on monday at 7:30

NOTE: When scheduling actions based on specific time-point, it is recommended to check if minute elapsed first. Otherwise action will be called on every run-cycle of lua (every event of system) at specific time-point.

```
if dateTime:changed() and dateTime:getWeekDayString() == "monday" then
  if dateTime:getHours() == 7 and dateTime:getMinutes() == 30 then
    print("Its monday, 7:30!")
  end
end
```

Perform an action every minute between 7:30 and 10:00 only at weekends

```
if dateTime:changed() then
  local day = dateTime:getWeekDayString()
  if day == "saturday" or day == "sunday" then
    if dateTime:getTimeOfDay() >= 450 and dateTime:getTimeOfDay() <= 600 then
      print("Its weekend morning!")
    end
  end
end
```

Variables

Variables defined in the script are not preserved between calls or subsequent cycles, you should use **Lua variables** to store the value between script calls.

Lua variables have global scope and they are visible in all executions contexts.

Variables are exposed in the key-based container of objects: `variable`. Container store variables in the form of a key corresponding to the variable ID. For example, when you want to refer to a **Lua variable** with **ID 4** you should use: `variable[4]` object.

Attempting to reference a nonexistent device object, retrieve a nonexistent device property, or set the wrong value type will result in a script error.

Types

There are now three types of variables that can be used in system.

- `boolean` - holds boolean values: true/false
- `integer` - holds integer numbers
- `string` - holds text

Methods

- `changed()`

  Checks if value which is holded by object changed.

  Returns:

  - *(boolean)*

- `getValue()`

  Returns value which is holded by object.

  Returns:

  - *(any)* - depends on variable type

- `setValue(value)`

  Sets value for object.

  Returns:

  - *reference to variable object*

  Arguments:

  - `value` *(any)* - variable type dependant value which should be set

- `save()`

  Copies current `value` to `default_value` and saves it to database. Next application start will use `default_value` to restore `value` property.

Examples

Set variable values

```
-- type: "string"
variable[1]:setValue("New text")

-- type: "integer"
variable[2]:setValue(42)

-- type: "boolean"
variable[3]:setValue(true)
```

Count failed scenes per day

```
if dateTime:changed() and dateTime:getHours() == 0 and dateTime:getMinutes() ==
   0 then
  variable[1]:setValue(0)
elseif event.type == "scene_failed" then
  variable[1]:setValue(variable[1]:getValue() + 1)
end
```

Use that counter in other automation

```
-- react to 10 scene fails
if variable[1]:changed() and variable[1]:getValue() >= 10 then
  print("Something is wrong!")
  wtp[3]:call("turn_on")
end
```

Timers

Timers are can be used to count-down time for performing actions based on intervals or periods of time (`milliseconds`, `seconds`, `minutes` or `hours`). They can also run in stopwatch mode to measure time.

In **timer mode**, it is as easy as setting the desired time using the `start` method. After the time has elapsed, the timer will trigger an event (`lua_timer_elapsed`) to inform you that the time has counted down.

In **stopwatch mode**, the time is counted continuously from the moment of starting with the `startFreeRun` method and it does not trigger an event because there is no time set. The total time elapsed can be retrieved using the `getElapsedTime` method.

Timer may be added, edited or deleted via REST API or a web application served through the central unit server.

They cannot be edited, but access to their methods is possible via scripts using `timer` container eg.

`timer[6]` gives you access to timer with **ID 6**.

Timers have global scope and they are visible in all executions contexts.

NOTE: it's not recommended to schedule multiple, parallel short intervals for timers, as this may degrade overall system performance due to large amount of events emitted.

Methods

- `startFreeRun()`

  Starts the stopwatch mode. Cancels previous modes and resets internal stopwatch counter if called again.

- `start(time)`

  Starts the count-down mode for certain amount of time in preconfigured units. Extends current interval if called again.

  NOTE: when using timer in count-down mode with `milliseconds` unit, minimum interval is 100 ms.

  Arguments:

  - `time` *(int)* - amount of time in preconfigured units

- `getElapsedTime()`

  Returns total elapsed time. If timer is running - time since last `startFreeRun` \ `start` call.

  If timer did stop - time counted until `stop` was called.

  If timer did elapse - time which was set as `start` argument.

  Returns:

- *(number)* - integer

- `isElapsed()`

  Returns information if timer is source of current execution context eg. you can catch moment of elapse.

  Returns:

  - *boolean*

- `getState()`

  Returns information of timer state. Can be one of following: `off`, `counting`, `elapsed`, `free_run`.

  Returns:

  - *string*

- `getUnit()`

  Returns timer unit. Can be one of following: `milliseconds`, `seconds`, `minutes`, `hours`.

  Returns:

  - *string*

- `stop()`

  Immediately stops (sets state to `off`) timer when running. In count-down mode, elapsed event won't fire afterwards. Does nothing if timer is already in `off` or `elapsed` state.

Examples

Start timer if it was not started before.

```
if timer[3]:getState() == "off" then
  timer[3]:start(1)
end
```

Start timer in stopwatch mode

```
timer[5]:startFreeRun()
```

Start timer in count-down timer mode for 5 seconds

```
timer[3]:start(5)
```

Start timer in count-down timer mode for 2 hours

```
timer[1]:start(2)
```

NOTE: Starting for 2 hours and 5 seconds may look similar interval but depends on `unit` property configured via REST API!

Count time between events

```
if wtp[5]:changedValue("state") then
  -- get elapsed time and start new round
  print("Last change took place %d seconds ago", timer[5]:getElapsedTime())
  timer[5]:startFreeRun()
end
```

Catch timer elapse

```
if timer[99]:isElapsed() then
  print("Timer has elapsed!")
end

-- Trigger conditions for timer

if dateTime:changed() then
  print("Count-down starts!")
  timer[99]:start(100)
end

if wtp[33]:changedValue("open") then
  print("Count-down starts!")
  timer[99]:start(500)
end
```

Statistics

User has possibility of adding custom statistic entries in lua scripts (scenes, automations and custom devices), which can be then displayed in statistics queries.

Points can be added using `statistics` object, which has global scope and is visible in all executions contexts.

Points are associated to exection context eg. when adding point from automation with id `5`, you should select this automation as source of statistics when configuring query in web/mobile application.

NOTE Throttling mechanism will prevent from adding too many points per time. User can add point only once every minute, attempting to do it more often will be ignored. See return status of `addPoint` function.

Methods

- `addPoint(name, value, unit)`

  Adds point with value for object property.

  Returns:

  - *(boolean)* - true if a point was added (can fail if called more often than once a minute)

  Arguments:

  - `name` *(string)* - user defined name of statistic serie
  - `value` *(number)* - value of stats point
  - `unit` *(unit)* - one of available units listed below

Units

List of available statistics units:

- `unit.celsius_x10`
- `unit.relative_humidity_x10`
- `unit.hectopascals_x10`
- `unit.indoor_air_quality`
- `unit.percent`
- `unit.bool_unit`
- `unit.ppm`
- `unit.lux`
- `unit.celsius`
- `unit.kwh`
- `unit.micro_grams_per_m3`
- `unit.liters_per_hour`
- `unit.watt`
- `unit.percent_x10`

- `unit.null`
- `unit.pascal`
- `unit.second`
- `unit.volt`
- `unit.millivolts`
- `unit.milliamp`
- `unit.milliwatt`
- `unit.wh`
- `unit.m3_H`
- `unit.milliseconds`
- `unit.percent_per_hertz`
- `unit.bar_x10`

Suffix `_x10` means that value is expected to be multiplied by 10. Eg. if you want to store 23.5°C using `unit.celsius_x10` you need to put 235 as value when adding point.

Examples

Log temperature once per minute

Lua variable may be fed with value eg. from http calls in another automation.

```
if dateTime:changed() then
  local value = variable[3]:getValue()
  statistics:addPoint("temperature", value, unit.celsius_x10)
end
```

Sunrise and Sunset

Global scope objects which will help user to do actions referring in time to sunrise and sunset

Available in all contexts. You can access to it using `sunrise` and `sunset` objects.

Methods

- `hour()`

  The hour when sunrise / sunset will occur.

  Returns:

  - *(number)* - integer in 0–23 range

- `minute()`

  The minute when sunrise / sunset will occur (minute in hour)

  Returns:

  - *(number)* - integer in 0–59 range

- `timeOfDay()`

  The minute when sunrise / sunset will occur (minute of day, since 00:00) eg. 750 is equal to 12:30

  Returns:

  - *(number)* - integer in 0–1439 range

Examples

Get time of sunrise

```
message = string.format(
  "Today sunrise will start at %02d:%02d",
  sunrise:hour(), sunrise:minute() )
print(message)
```

```
message = string.format(
  "Today sunrise will start at %02d:%02d (%d minutes of day)",
  sunrise:timeOfDay() / 60, sunrise:timeOfDay() % 60,
  sunrise:timeOfDay() )

print(message)
```

Catch sunrise event

```
if event.type == "sunrise" then
  print("Sunrise starts now!")
end
```

Catch sunset event

```
if event.type == "sunset" then
  print("Sunset starts now!")
end
```

Do an action 2 hours after sunrise

```
if dateTime:changed() then
  time = dateTime:getTimeOfDay()
  checkPoint = sunrise:timeOfDay() + (2 * 60)

  if checkPoint == time then
    print("This will be called once, 2 hours after sunrise!")
  end

  if checkPoint <= time then
    print(
       "This will be called once per minute, 2 hours after sunrise, until 24:00"
       )
  end
end
```

Using timer to delay the action

```
if event.type == "sunrise" then
  print("Sunrise starts now!")
  timer[1]:start(2)
end

if timer[1]:isElapsed() then
  print("This will be called once, 2 hours after sunrise!")
end
```

Defer action for device to 2 hours after sunrise

```
if event.type == "sunrise" then
  print("Light is enabled and will be disabled after 2 hours!")

  wtp[5]:setValue("state", true)
  wtp[5]:setValueAfter("state", false, 2 * 60 * 60)
end
```

System

Global scope object for accessing system data.

Available in all contexts. You can access to it using `system` object.

Methods

- `version()`

  Returns system version info object.

  Returns:

  - *(table)* - object with system version info with following properties:
    - `major*` *(number)*
    - `minor` *(number)*
    - `maintenance` *(number)*
    - `environment` *(string)*
    - `build` *(number)*
    - `semver` *(string)*

Examples

Print all

```lua
local version = system:version()
print("Major ", version.major)
print("Minor ", version.minor)
print("Maintenance ", version.maintenance)
print("Env ", version.environment)
print("Build ", version.build)
print("SemVer ", version.semver)
```

Weather

Global scope object which will help user to do actions referring on current and forecast weather conditions.

Available in all contexts. You can access to it using `weather` object.

Properties

- `enabled` *(string, read-only)*

  Indicates if weather feature is turned on. User must turn it on via web application in order to weather object get data from server and work properly.

Methods

- `current`

  Returns weather object containing information about current weather conditions.

  Returns:

  - *weather_info object*

- `hourly`

  Returns array of weather objects containing information about forecasted weather conditions for next 48 hours.

  Returns:

  - *array of 48 weather_info objects*

Weather Object

Object which is returned by `current` and `hourly` methods of global scope `weather` object. Contains information about weather conditions.

Methods

- `weather()`

  General weather description.

  Returns:

  - *string*, possible values: `Clear`, `Clouds`, `Rain`, `Snow`

- `temperature()`

  Measured or forecast temperature. Unit: °C with one decimal number, multiplied by 10.

  Returns:

  - *(number)*

- `feelsLikeTemperature()`

  Measured or forecast feels like temperature. Unit: °C with one decimal number, multiplied by 10.

  Returns:

  ○ *(number)*

- `humidity()`

  Measured or forecast humidity in percent.

  Returns:

  ○ *(number)*

- `pressure()`

  Measured or forecast pressure in hPa.

  Returns:

  ○ *(number)*

- `windSpeed()`

  Measured or forecast wind speed. Unit: m/s with one decimal number, multiplied by 10.

  Returns:

  ○ *(number)*

- `windDegrees()`

  Measured or forecast wind direction in meteorological degrees.

  Returns:

  ○ *(number)* - integer in 0-359 range

- `rain()`

  Rain volume that is predicted to fall. Unit: millimeters with one decimal number, multiplied by 10.

  Returns:

  ○ *(number)*

- `snow()`

  Snow volume that is predicted to fall. Unit: millimeters with one decimal number, multiplied by 10.

  Returns:

  ○ *(number)* - integer

- `cloudCoverage()`

  Cloud coverage in percentage.

  Returns:

  ○ *(number)* - integer

- `changed()`

  Check if weather data has changed.

  Returns:

  - *(boolean)*

Examples

Read weather data on update

```
if weather:changed() then
  print("Weather data updated")
  print(weather:current():weather())
  print(weather:current():temperature())
  print(weather:current():feelsLikeTemperature())
  print(weather:current():humidity())
  print(weather:current():pressure())
  print(weather:current():windSpeed())
  print(weather:current():windDegrees())
  print(weather:current():rain())
  print(weather:current():snow())
  print(weather:current():cloudCoverage())
end
```

Close the blind when there is strong wind and rain

```
if weather:current():windSpeed() > 400 and weather:current():rain() > 10 then
  wtp[3]:call("down")
  wtp[4]:call("down")
end
```

Signal alarm when there is strong wind expected in next 3–5 hours

```
function isStrongWind(data)
  return data:windSpeed() > 400
end

if weather:changed() then
  forecast = weather:hourly()
  if isStrongWind(forecast[3]) or isStrongWind(forecast[4]) or
     isStrongWind(forecast[5])
  then
    print("Strong wind expected!")
    -- signal alarm
    wtp[6]:call("turn_on")
  end
end
```

Notifications

Global scope object which allows user to send custom push or email notification from lua scripts to cloud users or local super admin (providing that its account is linked to cloud).

Available in all contexts. You can access it using `notify` object.

Methods

- `info(title, body, users)`

  Sends info notification.

  Arguments:

  - `title` *(string)* - notification title, parameter is required i.e. must be at least 1 character long, maximum 65 characters
  - `body` *(string)* - notification text, maximum 500 characters long
  - `users` *(int, array)* - optional parameter which allows to specify user/users (cloud user id or local super admin id) which will receive a notification. Can be single ID number or array of ID's. Will send to all users if not specified.

- `warning(title, body, users)`

  Sends warning notification.

  Arguments:

  - `title` *(string)* - notification title, parameter is required i.e. must be at least 1 character long, maximum 65 characters
  - `body` *(string)* - notification text, maximum 500 characters long
  - `users` *(int, array)* - optional parameter which allows to specify user/users (cloud user id or local super admin id) which will receive a notification. Can be single ID number or array of ID's. Will send to all users if not specified.

- `error(title, body, users)`

  Sends error notification.

  Arguments:

  - `title` *(string)* - notification title, parameter is required i.e. must be at least 1 character long, maximum 65 characters
  - `body` *(string)* - notification text, maximum 500 characters long
  - `users` *(int, array)* - optional parameter which allows to specify user/users (cloud user id or local super admin id) which will receive a notification. Can be single ID number or array of ID's. Will send to all users if not specified.

Examples

Notify user #1 of boiler state changes

```
boiler = tech[3]
if boiler:changedValue("state") then
  if boiler:getValue("state") then
    notify:info("Boiler", "Boiler turned on", 1)
  else
    notify:info("Boiler", "Boiler turned off", 1)
  end
end
```

Notify users #1 and #3 of poor air quality

```
if dateTime:changed() then
  if dateTime:getHours() == 8 and dateTime:getMinutes() == 0 then
    sensor = wtp[3]
    air_quality = sensor:getValue("air_quality")
    if utils.table:indexOf({'poor','unhealthy','very_unhealthy'}, air_quality)
    then
      notify:warning("Air quality", "There is bad air today", {1, 3})
    end
  end
end
```

Notify everyone when there is no connection with pellet boiler controller

```
boiler = tech[3]
if boiler:changedValue("status") and boiler:getValue("status") == "offline"
then
  notify:error("Pellet Boiler", "No connection with pellet boiler controller!")
end
```

Modbus Client

Global scope objects which allow user to send requests to devices via RS-485 using Modbus RTU protocol or via network using Modbus TCP protocol.

Both types (RS-485 and TCP) of clients are exposed in the key-based container of objects: `modbus_client`.

Container store clients in the form of a key corresponding to the client ID. For example, when you want to refer to a **Lua Modbus Client** with **ID 4** you should use: `modbus_client[4]` object.

Attempting to reference a nonexistent client or set the wrong value type will result in a script error.

When using `read` methods first call will send request to slave device and next calls will return the value from cache.

Cache values are refreshed periodically based on `cache_refresh` parameter from modbus settings.

Values are kept in cache for time specified in `keep_cached` parameter from modbus settings.

Modbus settings can be changed via web application.

When request fails in script due to error (e.g. `timeout` or `invalid write`) script will fail with error.

Methods

- `writeHoldingRegister(address, value)`

  Sends write request to slave modbus device with specified holding register address and value.

  Arguments:

  - `address` *(integer)* - address of a holding register.
  - `value` *(integer)* - value that should be written to holding register.

- `writeHoldingRegisters(startAddress, values)`

  Sends write request to slave modbus device with specified holding registers addresses and values.

  Arguments:

  - `startAddress` *(integer)* - start address of a holding register.
  - `values` *(array)* - integer values that should be written to consecutive holding registers starting with `startAddress`

- `writeHoldingRegisterAsync(address, value)`

  Asynchronously sends write request to slave modbus device with specified holding register address and value. Handle response state using `onRegisterAsyncWrite`, `onAsyncRequestFailure` methods.

  Arguments:

  - `address` *(integer)* - address of a holding register.
  - `value` *(integer)* - value that should be written to holding register.

- `writeHoldingRegistersAsync(startAddress, values)`

  Asynchronously sends write request to slave modbus device with specified holding registers addresses and values. Handle response state using `onRegisterAsyncWrite`, `onAsyncRequestFailure` methods.

  Arguments:

  - `startAddress` *(integer)* - start address of a holding register.
  - `values` *(array)* - integer values that should be written to consecutive holding registers starting with `startAddress`

- `readHoldingRegister(address)`

  Reads value from holding register.

  Returns:

  - *(number)* - unsigned 16-bit integer

  Arguments:

  - `address` *(integer)* - address of a holding register.

- `readHoldingRegisterAsync(address)`

  Asynchronously reads value from holding register. Handle response using `onRegisterAsyncRead`, `onAsyncRequestFailure` methods.

  Arguments:

  - `address` *(integer)* - address of a holding register.

- `readInputRegister(address)`

  Reads value from input register.

  Returns:

  - *(number)* - unsigned 16-bit integer

  Arguments:

  - `address` *(integer)* - address of a input register.

- `readInputRegisterAsync(address)`

  Asynchronously reads value from input register. Handle response using `onRegisterAsyncRead`, `onAsyncRequestFailure` methods.

  Arguments:

  - `address` *(integer)* - address of a input register.

- `writeCoil(address, bit_value)`

  Sends write request to slave modbus device with specified coil address and value.

  Arguments:

  - `address` *(integer)* - address of a coil.
  - `bit_value` *(boolean)* - value that should be written to coil.

- `writeCoils(startAddress, bit_values)`

  Sends write request to slave modbus device with specified coils addresses and values.

  Arguments:

  - `startAddress` *(integer)* - start address of a coil.
  - `bit_values` *(array)* - boolean values that should be written to consecutive coils starting with `startAddress`

- `writeCoilAsync(address, bit_value)`

  Asynchronously sends write request to slave modbus device with specified coil address and value. Handle response state using `onRegisterAsyncWrite`, `onAsyncRequestFailure` methods.

  Arguments:

  - `address` *(integer)* - address of a coil.
  - `bit_value` *(boolean)* - value that should be written to holding register.

- `writeCoilsAsync(startAddress, bit_values)`

  Asynchronously sends write request to slave modbus device with specified coils addresses and values. Handle response state using `onRegisterAsyncWrite`, `onAsyncRequestFailure` methods.

  Arguments:

  - `startAddress` *(integer)* - start address of a coil.
  - `bit_values` *(array)* - boolean values that should be written to consecutive coils starting with `startAddress`

- `readCoil(address)`

  Reads value from coil.

  Returns:

  - *(boolean)*

  Arguments:

  - `address` *(integer)* - address of a coil.

- `readCoilAsync(address)`

  Asynchronously reads value from coil. Handle response using `onRegisterAsyncRead`, `onAsyncRequestFailure` methods.

  Arguments:

  - `address` *(integer)* - address of a coil.

- `readDiscreteInput(address)`

  Reads value from discrete input.

  Returns:

  - *(boolean)*

  Arguments:

  - `address` *(integer)* - address of a discrete input.

- `readDiscreteInputAsync(address)`

  Asynchronously reads value from discrete input. Handle response using `onRegisterAsyncRead`, `onAsyncRequestFailure` methods.

  Arguments:

  - `address` *(integer)* - address of a discrete input.

- `isConnected()`

  Returns `true` if client's transceiver is connected to central unit, `false` otherwise.

  Returns:

  - *(boolean)*

- `onRegisterAsyncRead(callback)`

  Callback hook. Calls function passed in argument when asynchronous Modbus read request finishes successfully.

  Arguments:

  - `callback` *(function, required)* - callback function which should be called

    Arguments:

    - `registerType` *(string)* - type of read register (one of values: `COILS`, `DISCRETE_INPUTS`, `INPUT_REGISTERS`, `HOLDING_REGISTERS`)
    - `registerAddress` *(integer)* - address of read register
    - `value` *(integer/boolean)* - value of read register, type depends on register type

- `onRegisterAsyncWrite(callback)`

  Callback hook. Calls function passed in argument when asynchronous Modbus write request finishes successfully.

  Arguments:

  - `callback` *(function, required)* - callback function which should be called

    Arguments:

    - `registerType` *(string)* - type of written register (one of values: `COILS`, `DISCRETE_INPUTS`, `INPUT_REGISTERS`, `HOLDING_REGISTERS`)
    - `registerAddress` *(integer)* - address of written register

- `value` *(integer/boolean)* - value of written register, type depends on register type

- `onAsyncRequestFailure(callback)`

  Callback hook. Calls function passed in argument when asynchronous Modbus read or write request fails.

  Arguments:

  - `callback` *(function, required)* - callback function which should be called

    Arguments:

    - `requestType` *(string)* - type of request (one of values: `READ`, `WRITE`, `MULIPLE_WRITE`)
    - `error` *(string)* - error returned by device or `TIMEOUT` when there are connection problems
    - `registerType` *(string)* - type of register (one of values: `COILS`, `DISCRETE_INPUTS`, `INPUT_REGISTERS`, `HOLDING_REGISTERS`)
    - `registerAddress` *(integer)* - address of register
    - `value` *(integer/boolean)* - value of register to write (`0`/`false` for read requests), type depends on register type

Examples

Read data from a modbus device

```
print(modbus_client[1]:readHoldingRegister(104))
print(modbus_client[1]:readInputRegister(2))
print(modbus_client[1]:readCoil(1))
print(modbus_client[1]:readDiscreteInput(5))
```

Write data to a holding register and a coil

```
modbus_client[1]:writeHoldingRegister(104, 43)
modbus_client[1]:writeCoil(1, true)
```

Write multiple values to holding registers and coils

```
modbus_client[1]:writeHoldingRegisters(104, {42, 43, 44, 45})
modbus_client[1]:writeCoils(1, {true, false, false, true})
```

Turn on air conditioner using modbus protocol

```
local air_conditioner = modbus_client[1]

-- Set proper register and value according to
-- air conditioner modbus documentation.
local register = 40002
local turned_on_value = 1

air_conditioner:writeHoldingRegister(register, turned_on_value)
```

Turn off air conditioner using modbus protocol

```
local air_conditioner = modbus_client[1]

-- Set proper register and value according to
-- air conditioner modbus documentation.
local register = 40002
local turned_off_value = 1

air_conditioner:writeHoldingRegister(register, turned_off_value)
```

Handle asynchronous read request

```
modbus_client[1]:onRegisterAsyncRead(function(type, address, value)
  print ("Successfully read value")
  print (type, address, value)
end)
```

Handle asynchronous write request

```
modbus_client[1]:onRegisterAsyncWrite(function(type, address, value)
  print ("Successfully written value")
  print (type, address, value)
end)
```

Handle asynchronous request failure

```
modbus_client[1]:onAsyncRequestFailure(
  function(requestType, error, registerType, registerAddress, value)
    utils:printf(
      "%s register %s %x failed with error %s",
      requestType, registerType, registerAddress, error )

end)
```

Libraries - JSON

It is possible to encode and decode JSON data in Lua interpreter.

Methods

- `JSON:decode(text)`

  Decodes JSON to object representing it.

  Returns:

  - *(table)*

  Arguments:

  - `text` *(string)* - JSON data

- `JSON:encode(object)`

  Encodes passed Lua table as JSON.

  Returns:

  - *(string)*

  Arguments:

  - `object` *(any)* - variable to be encoded

- `JSON:encode_pretty(object)`

  Encodes passed Lua table as JSON with indentations.

  Returns:

  - *(string)*

  Arguments:

  - `object` *(any)* - variable to be encoded

Example

```lua
local json_text = "{\"name\":\"abc\"}"
local decoded = JSON:decode(json_text)

print(decoded.name)
-- abc

print(decoded["name"])
-- abc

local encoded_json = JSON:encode(decoded)
print(encoded_json)
-- {"name":"abc"}

local encoded_json_pretty = JSON:encode_pretty(decoded)
print(encoded_json_pretty)
--[[
{
    "name":"abc"
```

```
}
--]]

local data = { on = wtp[68]:getValue("state"), desc = "Test"}
print(JSON:encode(data))
-- {"on":false,"desc":"Test"}
```

Libraries - XML

It is possible to encode and decode XML data in Lua interpreter.

Methods

- `XML:decode(text)`

  Decodes XML to object representing it.

  Returns:

  - *(table)*

  Arguments:

  - `text` *(string)* - XML data

- `XML:encode(object)`

  Encodes passed Lua table as XML.

  Returns:

  - *(string)*

  Arguments:

  - `object` *(any)* - variable to be encoded

Example

XML input:

```
<devices>
  <device type="wtp">
    <id>1</id>
    <name>Relay</name>
    <state>true</state>
  </device>
  <device type="virtual">
    <id>1</id>
    <name>Thermostat</name>
    <temperature>
      <current>250</current>
      <target>300</target>
    </temperature>
  </device>
</devices>
```

Decoding:

```
local decoded = XML:decode(xml_input)

print (decoded.devices.device[1].name)
-- Relay

print (decoded.devices.device[1]._attr.type)
-- wtp

print (decoded.devices.device[2].temperature.target)
```

```
-- 300
```

Encoding:

```
local xml = {
  devices = {
    device = {
      {
        _attr = {type = "wtp"},
        id = 1,
        name = "Relay",
        state = "true"
      },
      {
        _attr = {type = "virtual"},
        id = 1,
        name = "Thermostat",
        temperature = {current = 250, target = 300}
      }
    }
  }
}

print (XML:encode(xml))
-- will print contents of the xml sample above
```

Libraries - hash

It is possible to calculate various hashes in Lua interpreter.

Methods

- `hash:sha256(input)`

  Calculates the sha256 hash for the given input string.

  Returns:

  - *(string)*

  Arguments:

  - `input` *(string)*

- `hash:md5(input)`

  Calculates the md5 hash for the given input string.

  Returns:

  - *(string)*

  Arguments:

  - `input` *(string)*

Example

```lua
local hash1 = hash:sha256("abc")
local hash2 = hash:md5("abc")

print(hash:sha256("abc"))
-- ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

print(hash:md5("abc"))
-- 900150983cd24fb0d6963f7d28e17f72
```

Utilities

Our interpreter is supplied with some lua utilities which should help you creating more complicated alghoritms.

Functions

`utils:printf(fmt, ...)`

Prints formatted string, refer to `printf(3)` man page for more information.

Arguments:

- `fmt` *(string)* - format string
- `...` *(any)* - values used to format the string

Example:

```
utils:printf("0x%04x", 32768)

-- stdout: [PRINT] 0x8000
```

`utils:ternary(condition, trueValue, falseValue)`

Returns proper value based on provided condition.

Returns:

- `trueValue` or `falseValue` directly

Arguments:

- `condition` *(bool)* - condition to check
- `trueValue` *(any)* - value to return if `condition` is true
- `falseValue` *(any)* - value to return if `condition` is false

Example:

```
function safeSqrt(value)
  return utils:ternary(value > 0, math.sqrt(value), 0)
end
```

`utils:integrateProperty(property, devices)`

Copy property change from one device to other ones.

Arguments:

- `property` *(string)* - property to copy
- `devices` *(table)* - sequence of devices to integrate

Example:

```
-- when one device is turned on, others are as well
utils:integrateProperty('state', { wtp[2], wtp[3], wtp[8] })
```

Deprecated methods

`utils:stair_light(devices)`

Binds all devices by *state* property allowing devices to change state at the same time.

For example when one relay changes *state* to *true* all other devices change *state* to *true* too.

Can be used to easily connect stair lighs together.

Arguments:

- `devices` *(array)* - devices array

Example:

```
utils:stair_light({wtp[4], wtp[6], sbus[1]})
```

`utils:integrate_property(property, devices)`

Similar to `utils:stair_light` but can connect provided devices by any other parameter.

For example can connect two or more blind controllers by property `target_opening` allowing them to open/close together, changing only blind controller value.

Arguments:

- `property` *(string)* - property name that devices will keep the same value for
- `devices` *(array)* - devices array

Examples:

```
utils:integrate_property("target_opening", {wtp[8], wtp[9], sbus[21]})
```

```
utils:integrate_property("target_temperature", {wtp[11], tech[7]})
```

Utilities - colorspace conversion

A set of routines for converting colors between color spaces, useful for various lighting devices.

Representation

`utils.color:philips_hue_normalize_hsb(hue, saturation, brightness)`

Hue API v1 uses `uint16_t` for hue and `uint8_t` for the rest, but the **HSV** color space represents hue as an angle on the color wheel and the rest as floats between 0 and 1. This routine can be used to convert Hue values into normalized **HSV**.

Returns:

- `hue` *(number)* - float, $\langle 0; 360)$
- `saturation` *(number)* - float, $\langle 0; 1\rangle$
- `value` *(number)* - float, $\langle 0; 1\rangle$

Arguments:

- `hue` *(number)* - fixed, $\langle 0; 65535\rangle$
- `saturation` *(number)* - fixed, $\langle 0; 255\rangle$
- `brightness` *(number)* - fixed, $\langle 0; 255\rangle$

Example:

```
-- darkblue color
local h, s, v = utils.color:philips_hue_normalize_hsb(43690, 127, 127)

-- h ≈ 240
-- s ≈ .5
-- v ≈ .5
```

`utils.color:normalize_rgb888(r, g, b)`

Most implementations store **RGB** values as **RGB888**, while the floating point representation is much more convenient for calculations.

This routine converts from `uint8_t` channel values to <0;1> float values.

Returns:

- `red` *(number)* - float, $\langle 0; 1\rangle$
- `green` *(number)* - float, $\langle 0; 1\rangle$
- `blue` *(number)* - float, $\langle 0; 1\rangle$

Arguments:

- `r` *(number)* - red channel, fixed, $\langle 0; 255\rangle$
- `g` *(number)* - green channel, fixed, $\langle 0; 255\rangle$
- `b` *(number)* - blue channel, fixed, $\langle 0; 255\rangle$

Example:

```
-- aqua color
local r, g, b = utils.color:normalize_rgb888(0, 0xff, 0xff)

-- r = 0
-- g = 1
-- b = 1
```

`utils.color:clamp_rgb(r, g, b)`

Most color spaces don't contain every color, so converting to **sRGB** from something like **CIEXYZ** can give channel values outside of range, like a negative channel value. **RGB** implementations cannot shine a negative amount of red for example, so a color like that cannot be represented in them. The color can be approximated by clamping values and it should be sufficient for most applications.

This routine clamps each channel value to $\langle 0; 1 \rangle$ range

Returns:

- `r` *(number)* - red channel, float, $\langle 0; 1 \rangle$
- `g` *(number)* - green channel, float, $\langle 0; 1 \rangle$
- `b` *(number)* - blue channel, float, $\langle 0; 1 \rangle$

Arguments:

- `r` *(number)* - red channel, float, $\langle 0; 1 \rangle$
- `g` *(number)* - green channel, float, $\langle 0; 1 \rangle$
- `b` *(number)* - blue channel, float, $\langle 0; 1 \rangle$

Example:

```
local r, g, b = utils.color:clamp_rgb(utils.color.ciexyz_to_rgb(X, Y, Z))
```

`utils.color:html(r, g, b)`

The **RGB** color value is often represented in `#rrggbb` form, as seen in HTML or CSS. This routine creates such string from three separate channel values.

Returns:

- `hex` *(string)* - string, formatted like `#%02x%02x%02x`

Arguments:

- `r` *(number)* - red channel, float, $\langle 0; 1 \rangle$
- `g` *(number)* - green channel, float, $\langle 0; 1 \rangle$
- `b` *(number)* - blue channel, float, $\langle 0; 1 \rangle$

Example:

```
virtual[1]:setValue('color', utils.color:html(r, g, b))
```

Gamma correction

`utils.color:gamma(gamma, channel)`

Perform gamma compression with `gamma` value on `channel` value. The formula is: $\sqrt[gamma]{channel}$

Returns:

- `channelp` *(number)* - compressed `channel`

Arguments:

- `gamma` *(number)* - gamma value, i.e. 2.2
- `channel` *(number)* - value to compress

Example:

```
local z = utils.color:gamma(1.8, .456)

-- z ≈ .64645
```

`utils.color:gamma3(gamma, channel1, channel2, channel3)`

Perform gamma compression with `gamma` value on three channel values. This is equivalent to three calls to `utils.color:gamma()`, but can be nicely inlined with other color conversion routines.

Returns:

- `channel1p` *(number)* - compressed `channel1`
- `channel2p` *(number)* - compressed `channel2`
- `channel3p` *(number)* - compressed `channel3`

Arguments:

- `gamma` *(number)* - gamma value, i.e. 2.2
- `channel1` *(number)* - value to compress
- `channel2` *(number)* - value to compress
- `channel3` *(number)* - value to compress

Example:

```
local rp, gp, bp = utils.color:gamma3(2.2, 0, .5, .5)

-- rp = 0
-- gp ≈ .72974
-- bp ≈ .72974
```

`utils.color:degamma(gamma, channelp)`

Perform gamma decompression (linearization) of `channelp` value with `gamma`. The formula is: $channelp^{gamma}$

Returns:

- `channel` *(number)* - decompressed `channelp`

Arguments:

- `gamma` *(number)* - gamma value, i.e. 2.2
- `channelp` *(number)* - value to decompress

Example:

```
local c = utils.color:degamma(2.2, .4958)

-- c ≈ .2136
```

`utils.color:degamma3(gamma, channel1p, channel2p, channel3p)`

Perform gamma decompression (linearization) of three channel values with `gamma`. This function is equivalent to three calls to `utils.color:degamma()`, but can be nicely inlined with other color conversion routines.

Returns:

- `channel1` *(number)* - decompressed `channel1p`
- `channel2` *(number)* - decompressed `channel2p`
- `channel3` *(number)* - decompressed `channel3p`

Arguments:

- `gamma` *(number)* - gamma value, i.e. 2.2
- `channel1p` *(number)* - value to decompress
- `channel2p` *(number)* - value to decompress
- `channel3p` *(number)* - value to decompress

Examples:

```
local r, g, b = utils.color:degamma3(2.2, 0, .5, .5)

-- r = 0
-- g ≈ .2176
-- b ≈ .2176
```

```
-- Philips Hue API v1 was able to interpret HSV. CLIP API v2 cannot do this,
-- but that behavior can be emulated

-- convert H, S, V to x, y for Hue
local x, y = utils.color:CIEXYZ_to_CIExyY(utils.color:lin_sRGB_to_CIEXYZ(
    utils.color:degamma3(2.2, utils.color:hsv_to_rgb(H, S, 1))
))

-- now use x and y to set the color and V to set the brightness
```

Color space conversion

`utils.color:hsv_to_rgb(hue, saturation, value)`

Converts color value from convenient for humans **HSV** model to convenient for computer displays **RGB** model without gamma correction.

Returns:

- `r` *(number)* - red channel, float, $\langle 0; 1 \rangle$
- `g` *(number)* - green channel, float, $\langle 0; 1 \rangle$
- `b` *(number)* - blue channel, float, $\langle 0; 1 \rangle$

Arguments:

- `hue` *(number)* - float, $\langle 0; 360)$
- `saturation` *(number)* - float, $\langle 0; 1 \rangle$
- `value` *(number)* - float, $\langle 0; 1 \rangle$

Example:

```
-- pure blue
local r, g, b = utils.color:hsv_to_rgb(240, 1, 1)

-- r = 0
-- g = 0
-- b = 1
```

`utils.color:rgb_to_hsv(r, g, b)`

Converts color value from **RGB** model to **HSV** model with no gamma correction.

Returns:

- `hue` *(number)* - float, $\langle 0; 360)$
- `saturation` *(number)* - float, $\langle 0; 1 \rangle$
- `value` *(number)* - float, $\langle 0; 1 \rangle$

Arguments:

- `r` *(number)* - red channel, float, $\langle 0; 1 \rangle$
- `g` *(number)* - green channel, float, $\langle 0; 1 \rangle$
- `b` *(number)* - blue channel, float, $\langle 0; 1 \rangle$

Example:

```
-- pure yellow
local h, s, v = utils.color:rgb_to_hsv(1, 1, 0)

-- h = 60
-- s = 1
-- v = 1
```

`utils.color:lin_sRGB_to_CIEXYZ(r, g, b)`

Transforms linear **sRGB** model color to a **CIEXYZ** model color.

Returns:

- `X` *(number)* - chromacity component
- `Y` *(number)* - luminosity
- `Z` *(number)* - chromacity component

Arguments:

- `r` *(number)* - red channel, linear; float, $\langle 0; 1 \rangle$
- `g` *(number)* - green channel, linear; float, $\langle 0; 1 \rangle$
- `b` *(number)* - blue channel, linear; float, $\langle 0; 1 \rangle$

Example:

```
local X, Y, Z = utils.color:lin_sRGB_to_CIEXYZ(.5, .5, .5)

-- X ≈ .47525
-- Y ≈ .5
-- Z ≈ .5445
```

`utils.color:CIEXYZ_to_lin_sRGB(X, Y, Z)`

Transforms color value from **CIEXYZ** to linear **sRGB**.

NOTE: sRGB colors are usually represented as gamma–compressed values, so values returned by this function should be passed to `utils.color:degamma3()` before passing them to i.e. `utils.color:html()`

NOTE: The result values can be out of range, as **sRGB** color space is ”smaller” than **CIEXYZ**. Out of range results should be fed to `utils.color:clamp_rgb` to approximate the color.

Returns:

- `r` *(number)* - red channel, linear; float, $\langle 0; 1 \rangle$
- `g` *(number)* - green channel, linear; float, $\langle 0; 1 \rangle$
- `b` *(number)* - blue channel, linear; float, $\langle 0; 1 \rangle$

Arguments:

- `X` *(number)* - chromacity component
- `Y` *(number)* - luminosity
- `Z` *(number)* - chromacity component

Example:

```
local r, g, b = utils.color:CIEXYZ_to_lin_sRGB(.5, .5, .5)

-- r = .6025
-- g = .4742
-- b = .4543
```

`utils.color:CIEXYZ_to_CIExyY(X, Y, Z)`

Converts color value from **CIEXYZ** to **CIExyY**, where x and y are coordinates on the chromacity diagram. The luminosity value stays the same. The **CIExyY** model is used by the Philips Hue API.

Returns:

- `x` *(number)* - chromacity coordinate, float
- `y` *(number)* - chromacity coordinate, float
- `Y` *(number)* - luminosity value, float

Arguments:

- `X` *(number)* - chromacity component, float
- `Y` *(number)* - luminosity, float
- `Z` *(number)* - chromacity component, float

Example:

```
-- white
local x, y, Y = utils.color:CIEXYZ_to_CIExyY(1, 1, 1)

-- x = 1 / 3
-- y = 1 / 3
-- Y = 1
```

`utils.color:CIExyY_to_CIEXYZ(x, y, Y)`

Converts **CIExyY** (chromacity diagram coordinates) to **CIEXYZ** model. The luminosity stays the same.

Returns:

- `X` *(number)* - chromacity component, float
- `Y` *(number)* - luminosity, float
- `Z` *(number)* - chromacity component, float

Arguments:

- `x` *(number)* - chromacity coordinate, float
- `y` *(number)* - chromacity coordinate, float
- `Y` *(number)* - luminosity value, float

Example:

```
-- orange
local X, Y, Z = utils.color:CIExyY_to_CIEXYZ(.6, .3, .2)

-- X = .4
-- Y = .2
-- Z ≈ .007
```

Utilities - ctype

A set of character type recognition routines based on `ctype.h`. Refer to `isalpha(3)` manpage for more information.

These routines work correctly only for ASCII characters (a lookup table for all UNICODE characters would be bigger than the utils module).

Each function takes a single character as an argument.

Functions

`utils.ctype:isalnum(c)`

Returns `true` for alphanumeric characters.

`utils.ctype:isalpha(c)`

Returns `true` for alphabetic characters.

`utils.ctype:isascii(c)`

Returns `true` for 7-bit characters.

`utils.ctype:isblank(c)`

Returns `true` for a space or a tab character.

`utils.ctype:iscntrl(c)`

Returns `true` for control characters.

`utils.ctype:isdigit(c)`

Returns `true` for decimal digit characters.

`utils.ctype:isgraph(c)`

Returns `true` for characters that have graphic representation.

`utils.ctype:islower(c)`

Returns `true` for lowercase alphabetic characters.

`utils.ctype:isprint(c)`

Returns `true` for characters with graphic representation (space included).

`utils.ctype:ispunct(c)`

Returns `true` for characters that have graphic representation and are not alphanumeric.

`utils.ctype:isspace(c)`

Returns `true` for one of:

- `" "` – *space*
- `"\f"` – *page feed*
- `"\n"` – *line feed*
- `"\r"` – *carriage return*
- `"\t"` – *horizontal tabulation*
- `"\v"` – *vetrical tabulation*

`utils.ctype:isupper(c)`

Returns `true` for uppercase alphabetic characters.

`utils.ctype:isxdigit(c)`

Returns `true` for characters used as hexadecimal digits, both upper and lower case.

Utilities - math

An addition to Lua's built-in math library

Functions

`utils.math:scale(oldMin, oldMax, newMin, newMax, value)`

Converts `value` between two linear scales.

Returns:

- *(number)* - scaled value

Arguments:

- `oldMin` *(number)* - bottom of the current scale
- `oldMax` *(number)* - top of the current scale
- `newMin` *(number)* - bottom of the target scale
- `newMax` *(number)* - top of the target scale
- `value` *(number)* - a value contained in current scale that will be converted to the target scale.

Example:

```
local adcOutput = 989
local voltage = utils.math:scale(0, 1023, 0, 5, adcOutput)
-- voltage ≈ 4.83
```

`utils.math:bounds(min, max, value)`

Throws an error if the `value` is not , range <`min`; `max`> .

Arguments:

- `min` *(number)* - bottom of the allowed `value` range
- `max` *(number)* - top of the allowed `value` range
- `value` *(number)* - a value to be checked against `min` and `max`

Example:

```
utils.math:bounds(0, 1, 12)
-- error: Argument 12 out of bounds
```

`utils.math:dot(vec1, vec2)`

Returns dot product of two vectors. If sequences representing those vectors are not equal in length, it is assumed that both have length of the shorter one.

Returns:

- *(number)* - dot product of the vectors

Arguments:

- `vec1` *(table)* - a sequence of numbers
- `vec2` *(table)* - a sequence of numbers

Example:

```lua
local area = utils.math:dot({2, 1}, {1, 2})
-- area == 4
```

Utilities - sequences

Set of routines that manipulate sequences (tables that only use positive integer indices and behave more like C arrays than hash maps)

Functions

`utils.seq:flat(sequence)`

Unpack embedded sequences into a copy of the parent one.

Returns:

- *(table)* - flattened sequence

Arguments:

- `seq` *(table)* - sequence to flatten

Example:

```
local flattened = utils.seq:flat({ 1, 2, {4, 8}, 16 })
-- flattened == { 1, 2, 4, 8, 16 }
```

`utils.seq:fromStr(str)`

Create a new character sequence from a string, so it can be used by table and sequence utilities.

Returns:

- *(table)* - `str` converted to sequence of characters

Arguments:

- `str` *(string)* - string to chop into sequence

Example:

```
local strtab = utils.seq:fromStr('abcd')
-- strtab == { 'a', 'b', 'c', 'd' }
```

`utils.table:join(sequence, separator)`

Build a string from `sequence` elements and join them with `separator`.

Returns:

- *(string)* - joined table elements

Arguments:

- `table` *(table)*
- `separator` *(string)*

Example:

```lua
local str = utils.seq:join({ 1, 2, 4, 8 }, '_')
-- str == '1_2_4_8'
```

`utils.seq:slice(sequence, from, to)`

Extract fragment of the given `sequence`.

Returns:

- *(table)* - extracted sequence

Arguments:

- `sequence` *(table)*
- `from` *(number)* - index of starting element, can be negative to count from the end
- `to` *(number)* - index of ending element, can be negative to count from the end

Example:

```lua
local fragment = utils.seq:slice({ 1, 2, 4, 8, 16 }, 3, -2)
-- fragment == { 4, 8 }
```

`utils.seq:toReversed(sequence)`

Creates new sequence with elements copied from source `sequence`, but reversed.

Returns:

- *(table)* - reversed sequence

Arguments:

- `sequence` *(table)* - sequence to reverse

Example:

```lua
local reverse = utils.seq:toReversed({ 1, 2, 4, 8 })
-- reverse == { 8, 4, 2, 1 }
```

Utilities - strings

These utilities supplement built-in `string` table.

Functions

`utils.str:ltrim(str)`

Create a copy of `str` with leading whitespace removed.

Returns:

- *(string)* - trimmed string

Arguments:

- `str` *(string)* - untrimmed string

Example:

```
local cleared = utils.str:ltrim("  aaaa  ")
-- cleared == "aaaa  "
```

`utils.str:rtrim(str)`

Create a copy of `str` with trailing whitespace removed.

Returns:

- *(string)* - trimmed string

Arguments:

- `str` *(string)* - untrimmed string

Example:

```
local cleared = utils.str:rtrim("  aaaa  ")
-- cleared == "  aaaa"
```

`utils.str:trim(str)`

Create a copy of `str` with leading and trailing whitespace removed.

Returns:

- *(string)* - trimmed string

Arguments:

- `str` *(string)* - untrimmed string

Example:

```
local cleared = utils.str:trim("  aaaa  ")
-- cleared == "aaaa"
```

`utils.str:lpad(str, length, char)`

Pad `str` to `length` with character `char`.

Returns:

- `string`

Arguments:

- `str` *(string)*
- `length` *(number)*
- `char` *(string, one character long)*

Example:

```
local fixedSize = utils.str:lpad("short", 8, '_')
-- fixedSize == "___short"
```

`utils.str:rpad(str, length, char)`

Pad `str` to `length` with character `char`.

Returns:

- `string`

Arguments:

- `str` *(string)*
- `length` *(number)*
- `char` *(string, one character long)*

Example:

```
local fixedSize = utils.str:rpad("short", 8, '_')
-- fixedSize == "short___"
```

`utils.str:contains(str, substr)`

Check whether `substr` is contained within `str`.

Returns:

- `boolean`

Arguments:

- `str` *(string)*
- `substr` *(string)

Example:

```lua
local options = "rw,_netdev,user,noauto"
if utils.str:contains(options, "user") then
  print("User access allowed")
end
```

`utils.str:split(str, delimiter)`

Splits `str` into a sequence of substrings. `delimiter` string supplies a set of characters to use as substring limits, in `strtok`-like fashion.

Returns:

- `table`

Arguments:

- `str` *(string)*
- `delimiter` *(string)*

Example:

```lua
local s = utils.str:split("a:b:c::d:;ef;", ":;")
-- s == { "a", "b", "c", "d", "ef" }
```

`utils.str:startsWith(str, prefix)`

Returns `true` if the `str` string starts with `prefix`.

Returns:

- *(boolean)* - test result

Arguments:

- `str` *(string)*
- `prefix` *(string)*

Example:

```lua
local prefixMatches = utils.str:startsWith("% 1444", "% ")
-- prefixMatches == true
```

`utils.str:endsWith(str, suffix)`

Returns `true` if the `str` ends with `suffix`.

Returns:

- *(boolean)* - test result

Arguments:

- `str` *(string)*
- `suffix` *(string)*

Example:

```
local suffixMatches = utils.str:endsWith("120 kWh", " kWh")
-- suffixMatches == true
```

`utils.str:randomUUID()`

Creates a random-number based UUID.

Returns:

- *(string)* - generated UUID

Example:

```
local device1 = utils.str:randomUUID()
-- device1 == "8816972a-be78-44b1-bcff-b64d550b9540"
```

`utils.str:random(length)`

Creates a string of size `length` containing random characters `[0-9A-Za-z]`.

Returns:

- *(string)* - generated value

Arguments:

- `length` *(number)*

Example:

```
local id = utils.str:random(12)
-- id == "aphoh4eiXoo1"
```

`utils.str:truncate(string, maxLength, suffix)`

Truncates a string to size. If `suffix` is provided, it will be placed at the end of the truncated string.

Returns:

- *(string)* - input cut to size

Arguments:

- `string` *(string)*
- `maxLength` *(number)*
- `suffix` *(string)*

Examples:

```
local label = utils.str:truncate("too long to fit", 11, "...")
-- label == "too long..."
```

```
function CustomDevice:onClick()
  local label = self:getState()

  -- text elements have a size limit!
  label = utils.str:truncate(label, 32, "...")
  self:getElement('status_label'):setValue('value', label)
end
```

Utilities - tables

JavaScript-like table manipulation routines.

NOTE: Lua tables which use indices other than positive integers are implemented internally as hash maps and are unsorted. The order of elements can be different every time program executes, so scripts have to cope with random element order.

Functions

`utils.table:copy(table)`

Returns a deep copy of given table.

Returns:

- *(table)* - a copy

Argument:

- `table` *(table)* - table to copy

Example:

```
local old = { 1, 2, { 'cc', 'dd' }, 12.8 }
local new = utils.table:copy(old)
new[2] = 3
-- old[2] == 2
```

`utils.table:hasKey(table, key)`

Checks if `table[key]` expression can be evaluated correctly and its value does not equal to `nil`. Works even if object throws error when using non-existent table subscript.

Returns:

- *(boolean)* - test result

Arguments:

- `table` *(any)* - any subscriptable object
- `key` *(any)* - possible subscript

Example:

```
if not utils.table:hasKey(virtual, 12) then
    print('Warning, virtual device #12 does not exist')
end
```

`utils.table:indexOf(table, value)`

Find first occurence of an element equal to `value` in the `table`.

Returns:

- *(number)* - requested index, nil if does not exist

Arguments:

- `table` *(table)*
- `value` *(any)* - value to look for

Examples:

```
local idx = utils.table:indexOf({ 1, 2, 4, 8, 16 }, 8)
-- idx == 4
```

```
local idx = utils.table:indexOf({ 1, 2, 4, 8, 16 }, 3)
-- idx == nil
```

`utils.table:reduce(table, callback, initialValue)`

"Reduces" array contents to a single value by calling user-provided function `callback` once for every table element.

Returns:

- *(any)* - final accumulator value

Arguments:

- `table` *(table)* - dataset to reduce
- `callback` *(function)* - performs the "reduction".

  Returns:

  - *(any)* - value to save to accumulator

  Arguments:

  - `accumulator` *(any)* - value returned by the previous call to the `callback`.
  - `value` *(any)* - current `table` element
  - `key` *(any)* - key of the `value`
  - `table` *(table)* - a reference to the reduced `table`
- `initialValue` *(any)* - An optional parameter that will initialize accumulator value. If not given, first table element is used instead and first iteration is skipped.

Iterative functions - intro

Shorthands for loops that iterate over all table elements.

All of these functions implement a specific protocol:

- prototype: `utils.table:FUNCTION(table, callback)`
- return values - function and callback dependent
- arguments:

  - `table` *(table)* - table to iterate over
  - `callback` *(function)* - function that will be called at most once for every `table` element.

The callback functions are user-defined and have to follow this protocol:

- Prototype: `function (value, key, table)`
- Return value - iterative function dependent
- Arguments passed to the callback:
  - `value` *(any)* - currently parsed table element
  - `key` *(any)* - key of the parsed table element
  - `table` *(table)* - reference to the `table` passed to the iterative function

Lua just pushes all arguments out, user callback can use any amount of them. Simple callback that only needs the value can look like this:

```
-- return cube of a number value
local function cube(value)
  return value^3
end

local cubes = utils.table:map(numbers, cube)
```

Table element's key can also be accessed by a callback by including it in the argument list.

```
-- convert elements to string only if their key is a string
local function makeTypesConsistent(value, key)
  if type(key) == type("")
  then
    return tostring(value)
  else
    return value
  end
end

-- format some input data
input = utils.table:map(input, makeTypesConsistent)
```

Third argument, `table`, is passed to allow callbacks to directly access the table, like this:

```
-- validate the response table
local function validate(value, key, table)
  if key == 'stats'
  then
    -- having 'stats' implies having 'statsMeta'
    if not utils.table:hasKey(table, 'statsMeta')
    then
      return false
    end
  end

  return true
end
```

```
-- perform the validation
local ok = utils.table:every(input, validate)
```

The idea comes from JavaScript, refer to its documentation for more examples.

Iterative functions

`utils.table:every(table, callback)`

Returns `true` only if the `callback` returns `true` for all elements in `table`. The function returns as soon as its return value is determined.

Can be used as a for-each loop: returning true continues loop and returning false breaks the loop.

Returns:

- *(bool)* - test result

Arguments:

- `table` *(table)*
- `callback` *(function)* - returns a *boolean*

Examples:

```
local allNumbersPositive = utils.table:every({1, 2, -4, 8}, function (n)
  return n > 0
end)

-- allNumbersPositive == false
```

```
local allDevicesOn = utils.table:every({ wtp[4], wtp[6] }, function (dev)
    return dev:getValue('state') == true
end)
```

```
-- breakable forEach loop:
utils.table:every(inputData, function (v)

  -- parse string values
  if type(v) == type("")
  then
    parseValue(v)

    -- continue loop
    return true
  else

    -- break loop when non-string value is detected
    return false
  end
end)
```

`utils.table:filter(table, callback)`

Returns new table, that contains elements for which `callback` returned `true`

Returns:

- *(table)* - filtered input `table`

Arguments:

- `table` *(table)*
- `callback` *(function)* - returns a *boolean*

Example:

```lua
local evenNumbers = utils.table:filter({ 1, 2, 3, 4 }, function (n)
  return n % 2 == 0
end)

-- evenNumbers == { 2, 4 }
```

`utils.table:find(table, callback)`

Returns first table element (and its key) for which `callback` returned `true`. The function returns as soon as its return value is determined.

Returns:

- `value` *(any)*
- `key` *(any)*

Arguments:

- `table` *(table)*
- `callback` *(function)* - returns a *boolean*

Example:

```lua
local smallestEven, key = utils.table:find({ 1, 3, 2, 4 }, function (n)
    return n % 2 == 0
end)

-- smallestEven == 2, key == 3
```

`utils.table:forEach(table, callback)`

Calls function `callback` once for every element in `table`. If loop breaking is required, use `utils.table:every` instead.

Arguments:

- `table` *(table)*
- `callback` *(function)*

Example:

```
-- list virtual devices
utils.table:forEach(virtual, function (dev, id)
    utils.printf("Virtual device '%s' has id #%d", dev, id)
end)
```

`utils.table:group(table, callback)`

Calls function `callback` for each table element to determine its group. A new table, containing elements from `table` grouped into tables, is returned.

Returns:

- *(table)* - contains selected groups in tables

Arguments:

- `table` *(table)* - values to group
- `callback` *(function)* - returns anything that can be used as a table key

Example:

```
local input = {
  { number = 1, name = "Jeden" },
  { number = 2, name = "Zwei"  },
  { number = 3, name = "Три"   },
  { number = 4, name = "Négy"  }
}

local grouped = utils.table:group(input, function (row)
  if row.number % 2 == 1 then
    return "odd"
  else
    return "even"
end)
```

variable `grouped` will contain:

```
{
  odd = {
    { number = 1, name = "Jeden" },
    { number = 3, name = "Три" }
  },
  even = {
    { number = 2, name = "Zwei" },
    { number = 4, name = "Négy" }
  }
}
```

`utils.table:map(table, callback)`

Calls function `callback` for each table element to determine its replacement. A new table, containing values returned by `callback`, is returned.

Returns:

- *(table)* - contains values returned by callback

Arguments:

- `table` *(table)*
- `callback` *(function)* - returns anything that can be stored in a table

Example:

```lua
local squares = utils.table:map({ 1, 2, 3, 4, 5 }, function (n) return n^2 end)

-- squares == { 1, 4, 9, 16, 25 }
```

`utils.table:some(table, callback)`

Returns `false` only if the `callback` returns `false` for every element of the table. The function returns as soon as its return value is determined.

Returns:

- *(boolean)* - test result

Arguments:

- `table` *(table)*
- `callback` *(function)* - returns `boolean`

Example:

```lua
local thereAreNegativeNumbers = utils.table:some({ 1, 2, -3, 4 }, function (v)
  return v < 0
end)

-- thereAreNegativeNumbers == true
```

Utilities - time

Methods

`utils.time:fromISO(iso)`

Converts ISO time string to Unix timestamp

Returns:

- *(number)* - Unix timestamp

Arguments:

- `iso` *(string)* - time string to convert

Example:

```
utils.time:fromISO('1986-04-26T01:23:00')
```

`utils.time:toISO(unix)`

Converts Unix timestamp to ISO time string

Returns:

- *(string)* - ISO time string

Arguments:

- `unix` *(number)* - Unix timestamp

Example:

```
utils.time:toISO(1688554570)
```

`utils.time:toTimeOfDay(timeString)`

Converts *hour:minute* string to a TOD value (minutes since day started)

Returns:

- *(number)* - minutes since midnight, in 0–1439 range

Arguments:

- `timeString` *(string)* - e.g. '14:27'

Example:

```
local tod = utils.time:toTimeOfDay('15:18')
-- tod == 918
```

Utilities - URL manipulation

Refer to RFC 3986 for more information.

Percent-encoding

This encoding method is widely used in URIs and HTTP forms. Refer to section 2.1 of the RFC 3986 for more information.

`utils.url:encode(str)`

Performs percent-encoding on `str`. This function encodes spaces as `%20`.

Returns:

- encoded *(string)*

Arguments:

- `str` *(string)* – any string

Example:

```
local q = utils.url:encode("it's over")

-- q = "it%27s%20over"
```

`utils.url:encodePlus(str)`

Performs percent-encoding on `str`. This function encodes spaces as `+`.

Returns:

- encoded *(string)*

Arguments:

- `str` *(string)* – any string

Example:

```
local q = utils.url:encode("it's so over")

-- q = "it%27s+so+over"
```

`utils.url:decode(str)`

Performs percent decoding on `str`. This function interprets only `%20` as a space.

Returns:

- decoded *(string)*

Arguments:

- `str` *(string)* – percent-encoded string

Example:

```
local q = utils.url:decode("we%20are%20back%21")

-- q = "we are back!"
```

`utils.url:decodePlus(str)`

Performs percent decoding on `str`. This function interprets both `%20` and `+` as a space.

Returns:

- decoded *(string)*

Arguments:

- `str` *(string)* – percent-encoded string

Example:

```
local q = utils.url:decodePlus("we%20are+soo+back%21")

-- q = "we are soo back!"
```

URL parsing

`utils.url:getScheme(url)`

Extracts the scheme component of the `url`.

Returns:

- scheme *(string)*

Arguments:

- `url` *(string)* – a valid URL

Example:

```
local scheme = utils.url:getScheme("http://www.project.d/")
-- scheme = "http"
```

`utils.url:getHost(url)`

Extracts the host subcomponent of the authority component.

Returns:

- host *(string)*

Arguments:

- `url` *(string)* - a valid URL

Example:

```
local host = utils.url:getHost("http://www.project.d/")
-- host = "www.project.d"
```

`utils.url:getPort(url, default)`

Extracts the port subcomponent of the authority component. If the port is present in the URL, it is returned as a *number*. Otherwise, the `default` is returned without any conversions.

Returns:

- port *(any)* – *number* or `type(default)`

Arguments:

- `url` *(string)* – a valid URL
- `default` *(any)* – a value to return if the port is not present

Example:

```
local port

port = utils.url:getPort("http://www.project.d:8080/")
-- port = 8080

port = utils.url:getPort("http://www.project.d/")
-- port = nil

port = utils.url:getPort("http://www.project.d/", 80)
-- port = 80
```

`utils.url:getPath(url)`

Extracts the path component.

Returns:

- path *(string)* – empty string if not present

Arguments:

- `url` *(string)* – a valid URL

Example:

```
local path

path = utils.url:getPath("http://www.project.d")
-- path = ""

path = utils.url:getPath("http://www.project.d/")
-- path = "/"

path = utils.url:getPath("http://www.project.d/index.html")
-- path = "/index.html"
```

`utils.url:getQueryParams(url)`

Extracts the query component, decodes it and puts it in a table. The `+` signs are not expanded to spaces.

Returns:

- query *(table)* – a map of decoded query parameters

Arguments:

- `url` *(string)* – a valid URL

Example:

```
local query =
   utils.url:getQueryParams("http://www.project.d?q=uphill%20results")
-- query = { q = "uphill results" }
```

`utils.url:stripQueryParams(url)`

Returns `url` with query component removed.

Returns:

- stripped url *(string)*

Arguments:

- `url` *(string)* - a valid URL

Example:

```
local clean =
   utils.url:stripQueryParams("http://www.project.d?q=downhill%20results")
-- clean = "http://www.project.d"
```

HTTP Client

Global scope objects which allow user to send http requests.

Clients are exposed in the key-based container of objects: `http_client`. Container store clients in the form of a key corresponding to the client ID. For example, when you want to refer to a **Lua HTTP Client** with **ID 4** you should use: `http_client[4]` object.

Attempting to reference a nonexistent client or set the wrong value type will result in a script error.

Properties

Http clients properties can't be changed via lua scripts but they will be used in some cases when sending requests.

Available Properties

- **default URL** *(string)*

  Default url for request that will be used if not specified in `GET`, `POST`, `DELETE`, `PATCH`, `PUT` methods.

- **default request body** *(string)*

  Default request body that will be used if `body()` method not called.

- **default headers** *(map string:string)*

  Set of default headers that will be used for each request. Header values can be overriden or extended via `header()` method.

  For example:

  ```
  local http = http_client[config.client_id]
  -- headers: "Content-Type: text/plain" (default)

  http:header("Content-Type", "application/json")
  -- headers: "Content-Type: application/json" (overridden)

  http:send()
  -- headers: "Content-Type: text/plain" (back to default)
  ```

  `Content-Type` HTTP header can also be changed using `contentType()` method, which has the highest priority.

  Header keys are case insesitive.

- **default query parameters** *(map string:string)*

  Set of default query parameters that will be appended to url for each request. Query parameter can be overriden or extended via `queryParam()` method.

  For example:

```lua
local http = http_client[config.client_id]
-- params: format=csv (default)

http:queryParam("format", "json")
-- params: format=json (overridden)

http:send()
-- params: format=csv (back to default)
```

Query parameters are case sensitive.

Methods

- `GET(url)`

  Sets the request method to GET, with optional url.

  - If URL not provided, default url will be used.
  - If only path (string that starts with `/`) provided it will be concatented with default url. eg. `GET("/test")` will add `/test` to default url for this single request.
  - If URL provided, it will replace default url for this single request.

  Returns:

  - *(userdata)* - http client reference for chained calls

  Arguments:

  - `url` *(string, optional)* - url on which request should be sent.

- `POST(url)`

  Sets the request method to POST, with optional url.

  - If URL not provided, default url will be used.
  - If only path (string that starts with `/`) provided it will be concatented with default url. eg. `POST("/test")` will add `/test` to default url for this single request.
  - If URL provided, it will replace default url for this single request.

  Returns:

  - *(userdata)* - http client reference, for chained calls

  Arguments:

  - `url` *(string, optional)* - url on which request should be sent.

- `DELETE(url)`

  Sets the request method to DELETE, with optional url.

  - If URL not provided, default url will be used.
  - If only path (string that starts with `/`) provided it will be concatented with default url. eg. `DELETE("/test")` will add `/test` to default url for this single request.
  - If URL provided, it will replace default url for this single request.

Returns:

- *(userdata)* - http client reference, for chained calls

Arguments:

- `url` *(string, optional)* - url on which request should be sent.

- `PATCH(url)`

  Sets the request method to PATCH, with optional url.

  - If URL not provided, default url will be used.
  - If only path (string that starts with `/`) provided it will be concatented with default url. eg. `PATCH("/test")` will add `/test` to default url for this single request.
  - If URL provided, it will replace default url for this single request.

  Returns:

  - *(userdata)* - http client reference

  Arguments:

  - `url` *(string, optional)* - url on which request should be sent.

- `PUT(url)`

  Sets the request method to PUT, with optional url.

  - If URL not provided, default url will be used.
  - If only path (string that starts with `/`) provided it will be concatented with default url. eg. `PUT("/test")` will add `/test` to default url for this single request.
  - If URL provided, it will replace default url for this single request.

  Returns:

  - *(userdata)* - http client reference, for chained calls

  Arguments:

  - `url` *(string, optional)* - url on which request should be sent.

- `header(key, value)`

  Adds HTTP header to next request.

  Returns:

  - *(userdata)* - http client reference for chained calls

  Arguments:

  - `key` *(string)* - header name
  - `value` *(string)* - header value

- `queryParam(key, value)`

  Adds a query parameter to next request.

  Returns:

- *(userdata)* - http client reference for chained calls

Arguments:

- `key` *(string)* - query parameter name
- `value` *(string)* - query parameter value

- `body(payload)`

  Sets request body for next request.

  Returns:

  - *(userdata)* - http client reference for chained calls

  Arguments:

  - `payload` *(string)* - request payload

- `contentType(type)`

  Sets content type for next request.

  Returns:

  - *(userdata)* - http client reference for chained calls

  Arguments:

  - `type` *(string)* - type of request content

- `timeout(sec)`

  Sets next request timeout.

  Returns:

  - *(userdata)* - http client reference for chained calls

  Arguments:

  - `sec` *(number)* - number of seconds for request timeout

- `send()`

  Sends prepared request.

- `hasResponse()`

  Checks whether client received response from server. If response is ready it caches the request result (response).

  Returns:

  - *(boolean)*

- `hasFailure()`

  Checks whether client request failed, e.g. due to invalid url. If request failed it caches the request result (failure).

  Returns:

  - *(boolean)*

- `response()`

  Returns last response body received from server. (from cache)

  Returns:

  - *(string)*

- `status()`

  Returns last response status received from server, e.g. 200 if request OK. (from cache)

  Returns:

  - *(number)*

- `failureCause()`

  Returns the cause of last request failure. (from cache)

  Returns:

  - *(string)*

- `onMessage(function(status, bodyOrFailureCause, requestUrl, responseHeaders) end)`

  Callback hook. Calls function passed in argument on http message received. It caches the request result.

  Returns:

  - *(userdata)* - http client reference for chained calls

  Arguments:

  - `function` *(function, required)* - callback function which should be called

    Arguments:

    - `status` *(number)* - response status received from server, e.g. 200 if request OK
    - `bodyOrFailureCause` *(string)* - response body received from server on success, failure cause on failure
    - `requestUrl` *(string)* - request url matching current response, can be used to select from multiple responses using single callback hook.
    - `responseHeaders` *(table, key-value container)* - response headers in form of lua table (key = header name, value = header value)

Examples

All methods in **HTTP Client** which return a *http client reference* can be chained.

Send GET request to custom.server.com at 19:00

```
-- with chained calls
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    http_client[1]
      :GET("https://custom.server.com")
      :send()
  end
end
```

```
-- without chained calls
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    http_client[1]:GET("https://custom.server.com")
    http_client[1]:send()
  end
end
```

POST data to custom.server.com at sunrise

```
-- without chained calls
if event.type == "sunrise"
then
  local http = http_client[1]
  http:POST("https://custom.server.com")
  http:header("Authorization", "Tk63TBJv5hhdnu5UN_F2dgj")
  http:header("Connection", "keep-alive")
  http:contentType("text/plain")
  http:body("request body")
  http:send()
end
```

```
-- with chained calls
if event.type == "sunrise"
then
  http_client[1]
    :POST("https://custom.server.com")
    :header("Authorization", "Tk63TBJv5hhdnu5UN_F2dgj")
    :header("Connection", "keep-alive")
    :queryParam("param", "value")
    :contentType("text/plain")
    :body("request body")
    :send()
end
```

POST data at sunrise with default values

Here we assume that at least default url is set for `http_client[1]`.

All default values will be used that are set for this client, i.e.: `url`, `headers`, `body`

```
if event.type == "sunrise" then
  http_client[1]:POST():send()
end
```

Handle received response

```
if http_client[1]:hasResponse()
then
  print("Response from client 1:")
  print(http_client[1]:response())

  print("Status from client 1:")
  print(http_client[1]:status())
end
```

Handle request failure

```
if http_client[1]:hasFailure()
then
  print("Request from client 1 failed!")
  print(http_client[1]:failureCause())
end
```

Handle response using a callback

```
http_client[1]:onMessage(
  function(status, bodyOrFailureCause, requestUrl, responseHeaders)
    if status == 200 then
      print ("Request succeeded with status")
      print (status)
      print ("Response from server:")
      print (bodyOrFailureCause)
    else
      print ("Request failed with status")
      print (status)
      print ("Failure cause:")
      print (bodyOrFailureCause)
    end

    print("Url requested:" .. requestUrl)
    print("Response headers:")
    utils.table:forEach(responseHeaders, function (value, name)
      utils:printf("%s: %s", name, value)
    end)
  end)
```

HTTP Server

Global scope object which allow user to receive custom http requests and generate custom responses. In order to trigger automation with http server calls inside, you need to send HTTP request to `/api/v1/lua/http-server/*` where `*` means you can put any suffix in url you want and have automation with attached request handler.

Features:

- supported http methods: **GET**, **POST**, **PUT**, **PATCH**, **DELETE**
- request path routing with dedicated handler per path/method
- url variable arguments
- json or raw string request/response content type

Authorization is needed while sending requests. There are two types of auth available:

- standard user login process with JWT (you will need to refresh it manually using token refresh endpoint)
- static API-TOKEN (via REST api) which doesnt need refreshing - is valid as long as exists in configuration. (Can be created using the API)

There two methods of token provision:

- use `Authorization` header with value of token (without `Bearer` prefix)
- use `access_token` url query parameter with value of token

Http server can generate automatic responses in some cases:

- `404 Not Found` if invalid url prefix was sent in request.
- `404 Results Not Found` if valid request was received but there wasnt handler declared for this url/method.
- `500 Server Error` if handler failed to execute. (check response body for error details.)
- `501 Not Implemented` if handler is declared but no response was generated by this handler.

Server is exposed as object: `http_server`.

Properties

Http server doesnt have properties.

Methods

- `on(method, path, handler)`

  Router hook. Attach handler to specific request method and path

  Returns:

  - `http server reference`

  Arguments:

- `method` *(string, required)* - case insensitive method name, one of `GET`, `POST`, `PUT`, `PATCH`, `DELETE`.
- `path` *(string, required)* - url template with or without variable arguments.

  The `/api/v1/lua/http-server` url prefix will be removed. eg. when you request `/lua/http-server/my-endpoint/5` it will get forwarded as `/my-endpoint/5` url.

  You can catch `5` as parameter (eg. named `id`) – put `/my-endpoint/:id` as path (note declaration of variable name `:id`) and obtain data via `request:argument("id")` method in handler.
- `handler` *(function, required)* - callback function which should be message received.

  Arguments:

  - `request` *(HttpServerRequest, required)* - received request, see HttpServerRequest description below for details.
  - `response` *(HttpServerResponse, required)* - used to generate response, see HttpServerResponse description below for details.

HttpServerRequest

This object (lua table) is passed to handler and can be used to read incoming http request.

Methods

- `url()`

  Returns requested url path.

  Returns:

  - *(string)*

- `method()`

  Returns request method name, one of **GET**, **POST**, **PUT**, **PATCH**, **DELETE**.

  Returns:

  - *(string)*

- `argument(name)`

  Returns variable argument, declared in request handler url template or nil if not found.

  Returns:

  - *(string)*

  Arguments:

  - `name` *(string, required)* - name of argument to get, declared in request handler url template

- `queryParam(name)`

  Returns url query parameter or nil if not found.

  Returns:

  - *(string)*

  Arguments:

  - `name` *(string, required)* - name of query parameter to get

- `body()`

  Returns request body.

  Returns:

  - *(string)*

HttpServerResponse

This object (lua table) is passed to handler and can be used to create outgoing http response. Methods which return reference can be used in chain-calls.

Methods

- `status(code)`

  Sets http response status. Using this is optional since calling `response:body(...)` method automatically sets code as `200` if none was set.

  Returns:

  - *(userdata)* http server response reference, for chained calls

  Arguments:

  - `code` *(number, required)* - Http response status, should be one of 2xx, 4xx or 5xx.

- `body(content)`

  Sets http response body (content). Automatically sets status code to `200` if none was set.

  Returns:

  - *(userdata)* http server response reference, for chained calls

  Arguments:

  - `content` *(string, required)* - String representation of body eg. raw text or serialized json.

Examples

All methods in **HTTP Server** which return `http server reference` can be called successively without calling `http_server` object every time.

Handle requests

Sending request to `/api/v1/lua/http-server/hello/world` will create response with message.

```
http_server:on("GET", "/hello/world", function(request, response)
  response:status(200):body("Hello world! You've reached GET handler.")
end)

http_server:on("POST", "/hello/world", function(request, response)
  response:status(200):body("Hello world! You've reached POST handler.")
end)

http_server:on("PUT", "/hello/world", function(request, response)
  response:status(200):body("Hello world! You've reached PUT handler.")
end)

http_server:on("PATCH", "/hello/world", function(request, response)
```

```
  response:status(200):body("Hello world! You've reached PATCH handler.")
end)

http_server:on("DELETE", "/hello/world", function(request, response)
  response:status(200):body("Hello world! You've reached DELETE handler.")
end)
```

Handle requests using local functions

Sending request to `/api/v1/lua/http-server/hello/world` will create response with message.

```
local function handleRequest(request, response)
  response
    :status(200)
    :body("Hello world! You've reached " .. request:method() .. " handler.")
end

http_server:on("GET", "/hello/world", handleRequest)
http_server:on("POST", "/hello/world", handleRequest)
http_server:on("PUT", "/hello/world", handleRequest)
http_server:on("PATCH", "/hello/world", handleRequest)
http_server:on("DELETE", "/hello/world", handleRequest)
```

Handle url template arguments

Sending request to `/api/v1/lua/http-server/hello/sinum/from/admin` will create response with message: `Hello sinum was sent by admin!`

```
http_server:on("GET", "/hello/:thing/from/:user", function(request, response)
  local thing = request:argument("thing")
  local user = request:argument("user")

  response:body("Hello " .. thing .. " was sent by " .. user .. "!")
end)
```

Handle url query parameters

Sending request to `/api/v1/lua/http-server/hello?user=admin&what=sinum` will create response with message: `Hello sinum was sent by admin!`

```
http_server:on("GET", "/hello", function(request, response)
  local what = request:queryParam("what")
  local user = request:queryParam("user")

  response:body("Hello " .. what .. " was sent by " .. user .. "!")
end)
```

Handle json body in request and response

Request to `/api/v1/lua/http-server/body-example` with body containing:

```
{
  "name": "External client",
  "data": [ 192, 168, 1, 1 ]
}
```

will cause printing `name` and `data` fields to automation log.

The response will contain:

```
{
  "name": "Sinum",
  "data": [ 66, 77, 88, 99],
  "success": true,
  "reason": null
}
```

Code:

```
http_server:on("POST", "/body-example", function(request, response)

  local body = JSON:decode(request:body())

  -- Note: Lua sequence indices start at 1!
  local dataString = string.format("%d.%d.%d.%d",
    body.data[1], body.data[2], body.data[3], body.data[4] )
  print("Received request from " .. body.name .. ", data " .. dataString)

  local responseBody = {
    name = "Sinum",
    data = {
      66,
      77,
      88,
      99
    },
    success = true
    reason = nil
  }

  response:body(JSON:encode(responseBody))
end)
```

ICMP Ping

Global scope utility which allow user to ping remote host, exposed as the `ping` object.

It may be used to check if internet connection is available, check if device is turned on or detect if certain local ip addresses are reachable (eg. when smartphone is reachable at local network, this may mean you are at home).

Methods

- `send(destination, timeout, dataSize)`

  Sends ICMP ping request to destination.

  Returns:

  - *(userdata)* - ping object reference for chained calls

  Arguments:

  - `destination` *(string, required)* - hostname or ip of destination.
  - `timeout` *(integer, optional, [0-30], default: 5)* - maximum waiting time for reply in seconds.
  - `dataSize` *(integer, optional, [0-256], default: 32)* - size of random data to send in request.

- `onReply(callback)`

  Callback hook. Calls function passed in argument on ping response or error received.

  Returns:

  - *(userdata)* - ping object reference for chained calls

  Arguments:

  - `callback` *(function, required)* - callback function used as response handler.

    Arguments:

    - `success` *(bool, required)* - status flag, on successful ping equals `true`, on fail equals `false`.
    - `errorMessage` *(string, required)* - error message, describes why ping failed. Empty on success.
    - `elapsed` *(integer, required)* - time spent while processing ping in milliseconds, either successful or not.
    - `destination` *(string, required)* - always equal to destination used in `send` function. May be used to distinguish between many responses at the same time.
    - `replyFrom` *(string, required)* - hostname or ip of remote which responsed to ping request.
    - `timeToLive` *(integer, required)* - time to live parameter, may be used measure how many router 'hops' was required to reach destination

Examples

All methods in **Ping** which return ping object reference can be called successively without calling `ping` every time.

Ping local IP address at 19:00

```
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    ping:send("192.168.1.200")
  end
end
```

Ping two remote hosts every minute

```
if dateTime:changed() then
  ping:send("192.168.1.1"):send("192.168.1.2")
end
```

Handle response with a callback

```
ping:onReply(
  function(success, errorMessage, elapsed, destination, replyFrom, timeToLive)
    if success
    then
      print("Success!")
    else
      print("Failed! Reason:")
      print(errorMessage)
    end

    -- Print diagnostic data
    print("Elapsed:", elapsed)
    print("Destination:", destination)
    print("Reply From:", replyFrom)
    print("TTL:", timeToLive)

    -- You may use destination to distinguish responses from different hosts

    local devices = {
      mike = '192.168.1.100',
      lucy = '192.168.1.200'
    }

    if destination == devices.mike
    then
      if success
      then
        print("Mike phone is reachable. Mike is at home.")
      else
        print("Mike phone is not reachable. Mike is out.")
      end

    elseif destination == devices.lucy
    then
      if success
      then
```

```
      print("Lucy phone is reachable. Lucy is at home.")
    else
      print("Lucy phone is not reachable. Lucy is out.")
    end
  end
end)
```

Mqtt Client

Global scope objects which allow user to exchange mqtt messages. (Currently mqtts and mqtt-over-ws is not supported)

Clients are exposed in the key-based container of objects: `mqtt_client`. Container store clients in the form of a key corresponding to the client ID. For example, when you want to refer to a **Lua Mqtt Client** with **ID 4** you should use: `mqtt_client[4]` object.

Attempting to reference a nonexistent client or set the wrong value type will result in a script error.

Subscriptions should be configured by REST.

Properties

Mqtt clients properties can't be changed directly by lua scripts. You should use methods and callback hooks to exchange messages and REST API to configure client (eg. subscriptions).

Methods

- `isConnected()`

  Checks whether client successfully connected to broker.

  Returns:

  - *(boolean)*

- `isSubscribed(topic)`

  Checks whether client successfully subscribed to desired topic.

  Returns:

  - *(boolean)*

  Arguments:

  - `topic` *(string, required)* - topic to check.

- `publish(topic, payload, qos, retain)`

  Publishes message on topic with desired payload, qos and retain.

  Returns:

  - *(userdata)* - mqtt client reference for chained calls

  Arguments:

  - `topic` *(string, required)* - topic on which message should be published.
  - `payload` *(string, required)* - message payload.
  - `qos` *(integer, required, [0-2])* - message qos.
  - `retain` *(string, required)* - message retain flag.

- `onConnected(function() end)`

  Callback hook. Calls function passed in argument on successful connection to broker (whenCONACK received).

  Returns:

  - *(userdata)* - mqtt client reference for chained calls

  Arguments:

  - `function` *(function, required)* - callback function which should be called on successful connection.

- `onDisconnected(function(error) end)`

  Callback hook. Calls function passed in argument on graceful disconnect or forced disconnect (eg. due to network error).

  Returns:

  - *(userdata)* - mqtt client reference for chained calls

  Arguments:

  - `function` *(function, required)* - callback function which should be called on disconnect or error.

    Arguments:

    - `error` *(bool, required)* - disconnection status - graceful (false) or error (true).

- `onSubscriptionEstablished(callback)`

  Callback hook. Calls function passed in argument on successful subcribe to topic.

  Returns:

  - *(userdata)* - mqtt client reference for chained calls

  Arguments:

  - `callback` *(function, required)* - callback function which should be called on subscription established.

    Arguments:

    - `topic` *(string, required)* - topic which was subscribed.

- `onMessage(function(topic, payload, qos, retain, dup) end)`

  Callback hook. Calls function passed in argument on message received at subscribed topics.

  Returns:

  - *(userdata)* - mqtt client reference for chained calls

  Arguments:

  - `function` *(function, required)* - callback function used as message handler.

    Arguments:

    - `topic` *(string, required)* - received message topic.

- `payload` *(string, required)* - received message payload.
- `qos` *(integer, required, [0-2])* - received message qos level.
- `retain` *(bool, required)* - received message retain flag.
- `dup` *(bool, required)* - received message duplicate flag.

Examples

All methods in **Mqtt Client** which return mqtt client reference can be called successively without calling `mqtt_client` container every time.

Receive message on subscribed topic.

```
mqtt_client[4]:onMessage(function(topic, payload, qos, retain, dup)
  if topic == "stat/tasmota_D9360D/POWER" then
    if payload == "ON" then
      wtp[68]:setValue("state", true)
    else
      wtp[68]:setValue("state", false)
    end
  elseif topic == "stat/tasmota_3C3AF1/POWER" then
    if payload == "ON" then
      wtp[69]:setValue("state", true)
    else
      wtp[69]:setValue("state", false)
    end
  elseif topic == "stat/tasmota_403B44/POWER" then
    if payload == "ON" then
        wtp[87]:setValue("state", true)
    else
        wtp[87]:setValue("state", false)
    end
  elseif topic == "zigbee2mqtt/Button" then
    data = JSON:decode(payload)

    if data["action"] ~= nil then
      if data["action"] == "1_single" then
        wtp[70]:call("toggle")
      elseif data["action"] == "2_single" then
        wtp[69]:call("toggle")
      elseif data["action"] == "3_single" then
        wtp[68]:call("toggle")
      end
    end
  end
end)
```

Publish message on topic ”greetings”

```
if dateTime:changed() then
  mqtt_client[4]:publish("greetings", "I am still alive mate!", 0, false)
end
```

Catch connect and disconnect

```
mqtt_client[4]:onConnected(function()
  print("Client with ID 4 connected to broker!")
end)

mqtt_client[4]:onDisconnected(function(error)
  if error then
    print("Client with ID 4 lost connection due to error.")
  else
    print("Client with ID 4 gracefully disconnected from broker.")
  end
end)
```

Catch subscription establish and publish data read request

```
mqtt_client[4]:onSubscriptionEstablished(function(topic)
  if topic == "/my-device/out" then
    mqtt_client[4]:publish("/my-device/in", "data-read-request", 0, false)
  end
end)
```

Wake On Lan

Global scope utility which allow user to send WakeOnLan magic packet to wake up device from standby, exposed as: `wakeOnLan` object.

NOTE: Remote device needs to support this function and you router should allow sending WOL packets.

NOTE: WOL Protocol does not support confirmations, so you can't check if device is turned on - but, you can use ICMP Ping utility in this case.

Properties

Wake On Lan utility doesn't have properties.

Methods

- `send(destination)`

  Sends WOL packet to destination.

  Returns:

  - *(userdata)* - wake on lan object reference for chained calls

  Arguments:

  - `destination` *(string, required)* - MAC Address of destination device.

Examples

All methods in **wake on lan** which return `wake on lan object reference` can be called successively without calling `wakeOnLan` every time.

Wake up devices at 19:00

```
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0
  then
    wakeOnLan
      :send("aa:bb:cc:dd:ee:ff")
      :send("A1:B2:C3:D4:E5:F6")
  end
end
```

EnergyCenter - FlowMonitor

The Goal of this module is to provide an easy to understand and visualize way of displaying the current Power Distribution coming from and to different sources, such as pv panels (inverter), grid, building and batteries etc.

Power distribution sources have to be associated using web application in order to get proper calculations available. Can be edited via REST API or a web application served through the central unit server.

Data access is possible via REST API, web app or directly from scripts using `flow_monitor` object eg. `flow_monitor:changed()`. Flow Monitor has global scope and is visible in all executions contexts.

Methods

- `changed()`

  Checks if any data has changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of object has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string)* - name of property

Properties

Properties direct access is not allowed. You can get values using `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `summary.building.available` *(boolean, read-only)*

Describes if building power distribution data is available. Becomes available if grid energy meter, pv or battery sources are configured (associated devices).

- `summary.building.value` *(number, read-only)*

  Current building power distribution. Only positive values are possible. The value represents the current consumption of the building.

  Unit: mW

- `summary.grid.available` *(boolean, read-only)*

  Describes if grid power distribution data is available. Becomes available if grid energy meter source is configured (associated device).

- `summary.grid.value` *(number, read-only)*

  Current grid power distribution. Positive value represents the power that is currently being imported from grid. Negative value represents the power that is currently being exported to the grid.

  Unit: mW

- `summary.pv.available` *(boolean, read-only)*

  Describes if pv power distribution data is available. Becomes available if pv panels (inverter) source is configured (associated device).

- `summary.pv.value` *(number, read-only)*

  Current pv power distribution. Only positive values are possible. The value represents the current production of the pv panels.

  Unit: mW

- `summary.battery.available` *(boolean, read-only)*

  Describes if battery power distribution data is available. Becomes available if battery source is configured (associated device).

- `summary.battery.value` *(number, read-only)*

  Current battery power distribution. Positive value represents the power that is currently used to charge battery. Negative value represents the power that is currently used to discharge battery.

  Unit: mW

- `summary.battery.state_of_charge.available` *(boolean, read-only)*

  Describes if battery state of charge data is available. Becomes available if battery device exposes such data.

- `summary.battery.state_of_charge.value` *(number, read-only)*

  Current battery state of charge.

  Unit: %

- `flow.pv_to_battery.value` *(number, read-only)*

  Represents value of current power flow from pv panels to battery. Only positive values are possible.

  Unit: mW

- `flow.pv_to_building.value` *(number, read-only)*

  Represents value of current power flow from pv panels to building. Only positive values are possible.

  Unit: mW

- `flow.pv_to_grid.value` *(number, read-only)*

  Represents value of current power flow from pv panels to grid. Only positive values are possible.

  Unit: mW

- `flow.grid_to_battery.value` *(number, read-only)*

  Represents value of current power flow from grid to battery (positive value) or from battery to grid (negative value).

  Unit: mW

- `flow.grid_to_building.value` *(number, read-only)*

  Represents value of current power flow from grid to building. Only positive values are possible.

  Unit: mW

- `flow.battery_to_building.value` *(number, read-only)*

  Represents value of current power flow from battery to building. Only positive values are possible.

  Unit: mW

- `building_consumption_details.rest` *(number, read-only)*

  Represents computed value of power consumption of devices that don't provide their individual power consumption data. Only positive values are possible.

  Unit: mW

- `building_consumption_details.by_devices` *(table, read-only)*

  Represents collection of devices that provide their power consumption.

  Unit: mW

Examples

Turn off relay when you start importing power from grid

```
if flow_monitor:changedValue("summary.grid.value") then
  local gridValue = flow_monitor:getValue("summary.grid.value")

  if gridValue > 0 and wtp[33]:getValue("state") then
    wtp[33]:call("turn_off")
  end
end
```

Turn on relay if there is pv production and it is being exported to grid

```
if flow_monitor:changedValue("flow.pv_to_grid.value") then
  local flowValue = flow_monitor:getValue("flow.pv_to_grid.value")

  if flowValue > 0 and not wtp[33]:getValue("state") then
    wtp[33]:call("turn_on")
  end
end
```

EnergyCenter - EnergyPrices

This module allows obtaining energy prices downloaded from various portals (configured via web application) or setting them manually via LUA.

The way prices are accessed can be edited via REST API or a web application served through the central unit server.

Data access is possible via REST API, web app or directly from scripts using `energy_prices` object eg. `energy_prices:changed()`. Energy Prices has global scope and is visible in all executions contexts.

Methods

- `changed()`

  Checks if any data has changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of object has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string)* - name of property

- `setValue(property_name, property_value)`

  Sets value for object property.

  Returns:

  - *(userdata)* - reference to Energy Prices object, for call chains

  Arguments:

  - `property_name` *(string)* - name of property
  - `property_value` *(any)* - property type dependant value which should be set

- `isHourPriceAvailable(hour)`

Check if energy price for `hour` is available. Returns false if `hour` is out of range, price for `hour` is not yet set or downloaded or Energy Prices feature is disabled.

Returns:

- *(boolean)*

Arguments:

- `hour` *(integer)* - hour in range [0, 23]

- `getHourPrice(hour)`

Returns energy price for `hour`. If `access_type` is set to `api` this method returns energy price downloaded from external API selected via web application. If `access_type` is set to `lua` it returns energy price set via LUA script.
NOTE: Returns `0` when price is not yet set or downloaded or Energy Prices feature is disabled.

Return:

- *(float)* or *(nil)* - energy price at `hour`. Return nil if `hour` is out range.

Arguments:

- `hour` *(integer)* - hour in range [0, 23]

- `setStaticPrice(price)`

Sets same energy price for every hour in a day.

Returns:

- *(userdata)* - reference to Energy Prices object, for call chains

Arguments:

- `price` *(float)* - energy price to set

- `setHourPrice(hour, price)`

Sets energy price for one `hour`.

Returns:

- *(userdata)* - reference to Energy Prices object, for call chains

Arguments:

- `hour` *(integer)* - selected hour in range [0, 23]
- `price` *(float)* - energy price to set

- `setHoursPrice(hours, price)`

Sets single energy price for multiple `hours`

Returns:

- *(userdata)* - reference to Energy Prices object, for call chains

Arguments:

- `hour` *(table)* - table of hours in range [0, 23]

- `price` *(float)* - energy price to set

- `setPrices(prices)`

  Sets prices.

  Returns:

  - *(userdata)* - reference to Energy Prices object, for call chains

  Arguments:

  - `prices` *(table)* - table of prices. If it's shorter than 24 elements only prices for first hours will be set.

Properties

Properties direct access is not allowed. You can get values using `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `enabled` *(boolean, read-only)*

  Informs if Energy Prices feature is enabled via web application.

- `access_type` *(string, read-only)*

  Current method to obtain prices. Possible values are:

  - `api` - prices are downloaded from selected portal
  - `lua` - prices are set via LUA script

- `country` *(string, read-only)*

  Country for which prices are downloaded via selected portal.
  NOTE: This parameter is not accessible when `access_type` is set to `lua`.

- `api_name` *(string, read-only)*

  Name of portal to download prices from.
  NOTE: This parameter is not accessible when `access_type` is set to `lua`.

- `currency` *(string)*

  Currency in which prices are represented.
  NOTE: This parameter is *read-only* when `access_type` is set to `api`.

EnergyCenter - EnergyStorage

The Goal of this module is to provide an easy to understand and visualize way of displaying the current Energy Storage (Battery) data.

Battery device have to be associated using web application in order to get proper calculations available. Can be edited via REST API or a web application served through the central unit server.

Data access is possible via REST API, web app or directly from scripts using `energy_storage` object eg. `energy_storage:changed()`. Energy Storage has global scope and is visible in all executions contexts.

Methods

- `changed()`

  Checks if any data has changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of object has recently changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string)* - name of property

Properties

Properties direct access is not allowed. You can get values using `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `available` *(boolean, read-only)*

  Describes if energy storage data is available. Becomes available if battery device association is configured (associated devices).

- `status` *(string, read-only)*

  Current status of energy storage. Possible values: *idle, charging, discharging*

- `power` *(number, read-only)*

  Current battery power distribution. Positive value represents the power that the battery is currently charged with. Negative value represents the power that the battery is currently discharged with.

  Unit: mW

- `energy_charged_today` *(number, read-only)*

  Daily sum of energy the battery was charged with.

  Unit: Wh

- `energy_discharged_today` *(boolean, read-only)*

  Daily sum of energy the battery was discharged with.

  Unit: Wh

- `state_of_charge.available` *(boolean, read-only)*

  Describes if battery state of charge data is available. Becomes available if battery device exposes such data.

- `state_of_charge.value` *(number, read-only)*

  Current battery state of charge.

  Unit: %

Examples

Turn off relay when battery is discharging and level drops to 20%

```
if energy_storage:changedValue("state_of_charge.value") then
  local level = energy_storage:getValue("state_of_charge.value")

  if level < 10 and wtp[33]:getValue("state") then
    wtp[33]:call("turn_off")
  end
end
```

EnergyCenter - EnergyConsumption

The Goal of this module is to provide the summary of energy consumption by registered power sockets and all other house appliances.

Power distribution sources have to be associated using web application in order to get proper calculations available. Can be edited via REST API or a web application served through the central unit server.

Accessing data is possible via REST API, web app or directly from scripts using `energy_consumption` object eg. `energy_consumption:changed()`. Energy Consumption has global scope and is visible in all executions contexts.

Methods

- `changed()`

  Checks if any data has changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of object has changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string)* - name of property

Properties

Properties direct access is not allowed. You can get values using `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `available` *(boolean, read-only)*

  Describes if energy consumption data is available. Becomes available if grid, pv or battery device association is configured (associated devices).

- `total.total_consumption` *(number, read-only)*

  Total summary of building energy consumption.

  Unit: Wh

- `total.house_consumption` *(number, read-only)*

  Total house energy consumption. Represents computed value of energy consumption of devices that don't provide their individual energy consumption data.

  Unit: Wh

- `total.electrical_outlets_consumption` *(number, read-only)*

  Total electrical outlets consumption. Represents computed value of energy consumption of devices that provide their power consumption.

  Unit: Wh

- `today.total_consumption` *(number, read-only)*

  Today summary of building energy consumption.

  Unit: Wh

- `today.house_consumption` *(number, read-only)*

  Today house energy consumption. Represents computed value of energy consumption of devices that don't provide their individual energy consumption data.

  Unit: Wh

- `today.electrical_outlets_consumption` *(number, read-only)*

  Today electrical outlets consumption. Represents computed value of energy consumption of devices that provide their power consumption.

  Unit: Wh

EnergyCenter - EnergyProduction

The Goal of this module is to provide the details of energy produced by PV inverter.

Inverter has to be associated using web application in order to get proper calculations available. Can be edited via REST API or a web application served through the central unit server.

Accessing data is possible via REST API, web app or directly from scripts using `energy_production` object eg. `energy_production:changed()`. Energy Production has global scope and is visible in all executions contexts.

Methods

- `changed()`

  Checks if any data has changed (thus is source of event).

  Returns:

  - *(boolean)*

- `changedValue(property_name)`

  Checks if specific property of object has changed (thus is source of event).

  Returns:

  - *(boolean)*

  Arguments:

  - `property_name` *(string)* - name of property

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string)* - name of property

Properties

Properties direct access is not allowed. You can get values using `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `available` *(boolean, read-only)*

  Describes if energy consumption data is available. Becomes available if FlowMonitor PV Summary is available and inverter exposed total energy produced parameter.

- `total.autoconsumption` *(number, read-only)*

  Total value of produced energy that was autoconsumed by building.

  Unit: Wh

- `total.energy_storage` *(number, read-only)*

  Total value of produced energy that was used to charge battery.

  Unit: Wh

- `total.grid_export` *(number, read-only)*

  Total value of produced energy that was exported to grid.

  Unit: Wh

- `today.autoconsumption` *(number, read-only)*

  Today value of produced energy that was autoconsumed by building.

  Unit: Wh

- `today.energy_storage` *(number, read-only)*

  Today value of produced energy that was used to charge battery.

  Unit: Wh

- `today.grid_export` *(number, read-only)*

  Today value of produced energy that was exported to grid.

WTP - AQSensor

Battery powered air quality sensor. Checks PM (particulate matter): 1.0, 2.5, 4.0, 10.0 concentration in the air.

Sensors measure values only every few minutes to save battery.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `pm1p0` *(number, read-only)*

  Sensed concentration of PM1.0 value (particulate matter).

  Unit: µg/m3.

- `pm2p5` *(number, read-only)*

  Sensed concentration of PM2.5 value (particulate matter).

  Unit: µg/m3.

- `pm4p0` *(number, read-only)*

  Sensed concentration of PM4.0 value (particulate matter).

  Unit: µg/m3.

- `pm10p0` *(number, read-only)*

  Sensed concentration of PM10.0 value (particulate matter).

Unit: µg/m3.

- `air_quality` *(string, read-only)*

  Descriptive name for air quality. Based on PM10.0 concetration.

raw	description
≤ 20 µg/m3	`very_good`
21 – 50 µg/m3	`good`
51 – 80 µg/m3	`moderate`
81 – 110 µg/m3	`poor`
111 – 150 µg/m3	`unhealthy`
> 150 µg/m3	`very_unhealthy`

WTP - BlindController

Controller based on time configurations opens and closes a roller shade or tilt blind.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `target_opening` *(number)*

  Desired setpoint opening, which device will try to achieve.

  NOTE: If device doesnt contain `percent_opening_control` label, target opening is limited to 0%, 50% or 100% (only these three).

  Unit: %.

- `current_opening` *(number, read-only)*

  Current opening value.

  Unit: %.

- `target_tilt` *(number, optional)*

  Desired tilt position.

  NOTE: Parameter is optional. Available when: `percent_tilt_control` label is provided.

  Unit: %.

- `current_tilt` *(number, optional, read-only)*

  Current tilt position

  NOTE: Parameter is optional. Available when: `percent_tilt_control` label is provided.

  Unit: %.

- `window_covering_type` *(string)*

  Determines wheter tilt should be possible or not.

  NOTE: Can be modified when: `percent_tilt_control` label is provided.

  Available values to set: *roller_shade, tilt_blind*

- `tilt_range` *(number, optional)*

  Determines tilt range.

  NOTE: Parameter is optional. Available when: `percent_tilt_control` label is provided. **NOTE:** Can be modified when: `window_covering_type` is equal to `tilt_blind`.

  Available values to set: *90, 180*

  Unit: angle (degrees).

- `full_cycle_duration` *(number, optional)*

  Time required by motor to do full cycle from 100% to 0% or 0% to 100% (select larger). Proper full open or full close action is based on this value.

  NOTE: Parameter is optional. Available when: `percent_opening_control` label is provided.

  Unit: seconds.

- `buttons_inverted` *(boolean, optional)*

  Replace up and down buttons directions.

  NOTE: Parameter is optional. Available when: `button_inversion_support` label is provided.

- `outputs_inverted` *(boolean, optional)*

  Replace up and down outputs directions.

  NOTE: Parameter is optional. Available when: `output_inversion_support` label is provided.

- `button_signal_type` *(string, optional)*

  Selected button specific behavior. eg. impulse = on/off impulse is required to start action.

  NOTE: Parameter is optional. Available when: `percent_opening_control` label is provided.

  Available values: *impulse, state_change*

- `output_signal_type` *(string, optional)*

  Selected output specific behavior. eg. impulse = on/off impulse is required to start motor.

  NOTE: Parameter is optional. Available when: `percent_opening_control` label is provided.

  Available values to set: *impulse, state_change*

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: *auto, fixed, off*

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle.

  Example: *#FF00FF*

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active eg. motor is working.

  Example: *#FFFF00*

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

Commands

- `open`

  Opens a blind to specific value in percent passed in argument.

  Argument:

  opening percentage *(number)*

- `up`

  Fully opens a blind.

- `down`

  Fully closes a blind.

- `stop`

  Immediately stops a blind motor.

- `calibration`

  Starts blind calibration cycle.

- `tilt`

  Calls tilt to the desired value.

  Argument:

  tilt percentage *(number)*

Examples

Open blind at sunrise and close at sunset

```
if event.type == "sunrise" then
  wtp[3]:call("up")
elseif event.type == "sunset" then
  wtp[3]:call("down")
end
```

Set blind to half-open at noon

```
if dateTime:changed() then
  if dateTime:getHours() == 12 and dateTime:getMinutes() == 0 then
    wtp[3]:call("open", 50)
  end
end
```

WTP - Button

Battery powered button, customizable in application. Every button action can be assigned different action. For example:

- Turn on first light when clicked once
- Turn on second light when clicked twice
- Turn off all lights when held down for 3 seconds

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `buttons_count` *(number, read-only)*

  Count of physical buttons.

- `action` *(string, read-only)*

  Last action performed by user. Example: *button_1_clicked_10_times , button_3_hold_started , button_2_held_3_seconds*

- `buzzer` *(string)*

  Embedded buzzer (speaker) settings. One of following: *on, off, unsupported*

Examples

Turn on lights when button clicked once

```
local button = wtp[9]
local lights = {wtp[2], wtp[3], wtp[4]}

if button:changedValue("action") and button:getValue("action") ==
   "button_1_clicked_1_times"
then
  utils.table:forEach(lights, function (light) light:call("turn_on") end)
end
```

Close blinds when button held for 3 seconds

```
local button = wtp[9]
local blinds = {wtp[5], wtp[6], wtp[7]}

if button:changedValue("action") and button:getValue("action") ==
   "button_1_held_3_seconds"
then
  utils.table:forEach(blinds, function (blind) blind:call("down") end)
end
```

WTP - CO2Sensor

Battery powered CO_2 sensor. Measures CO_2 concentration in the air and sends measurement to central unit. Sensors measure value only every few minutes to save battery.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `co2` *(number, read-only)*

  Sensed CO_2 value.

  Unit: PPM.

WTP - Dimmer

Device that controlls light intensity of output LED.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online*, *offline*, *unknown*, *service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date*, *update_available*, *recovery*, *pending*, *downloading*, *updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state` *(boolean)*

  State of the output. On/Off.

- `target_level` *(number)*

  Desired light intensity level on which device is set or level on which device will be set when turned on. (depending on `state`) Unit: %.

Commands

- `turn_on`

  Turns on output.

- `turn_off`

  Turns off output.

- `toggle`

  Changes state to opposite.

- `set_level`

  Set light intensity level to desired level smoothly during given time.

  Argument:

  packed arguments *(table)*:

  - light intensity in % *(number)*

    - minumum: 0
    - maximum: 100
  - transition time in 0.1s *(number)*
    - minimum: 1 *(100 ms)*
    - maximum: 6000 *(10 minutes)*
    - parameter is optional *(500 ms default)*

- `stop`

  Calls Dimmer to stop current level moving action. Does nothing if no action is in progress.

Examples

Turn on light at 19:00 and turn off at 21:00

```
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    wtp[4]:call("turn_on")
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    wtp[4]:call("turn_off")
  end
end
```

Set the light intensity to 75% during 2 minutes

```
wtp[4]:call("set_level", {75, 1200})
```

Begin dimming on button hold start and finish immediately after release (simple version)

Solution Drawback: will always take constant time to move from 0%->100%, 50%->100%, 10%->0% etc

```
local dimmerID = 4
local buttonID = 98

if wtp[buttonID]:changedValue("action") then

  local action = wtp[buttonID]:getValue("action")
  local fadeTime = 50 -- 5s / 5000ms

  if action == "button_1_hold_started"
  then
    -- start moving to 100% from current target level
     wtp[dimmerID]:call("set_level", {100, fadeTime})
  elseif action == "button_2_hold_started"
  then
    -- start moving to 0% from current target level
    wtp[dimmerID]:call("set_level", {0, fadeTime})
  elseif action:find("button_1_held_") ~= nil or action:find("button_2_held_")
  then
    -- stop current moving action
    wtp[dimmerID]:call("stop")
```

```
  end

end
```

Begin dimming on button hold start and finish immediately after release (advanced version)

Note: will adjust move time regarding current to target level difference

```
-- this function will compute required dimming time (adjusted to current level)
-- you shouldnt need to modify this function
local function computeMoveParameters(currentValue, desiredValue, fullFadeTime)

  -- calculate diff between current and desired level
  local diff = math.abs(desiredValue - currentValue)

  -- calculate how long move will take for this diff
  local reqTime = utils.math:scale(0, 100, 0, fullFadeTime, diff)

  -- clamping / rounding
  return {desiredValue, math.floor(utils.math:clamp(0, fullFadeTime, reqTime))}
end

-- the actual dimming action
local dimmer = wtp[4]
local button = wtp[9]

if button:changedValue("action") then

  local action = button:getValue("action")
  local fadeTime = 50 -- 5s / 5000ms

  if action == "button_1_hold_started" then
    -- start moving to 100% from current target
    dimmer:call(
      "set_level",
      computeMoveParameters(wtp[dimmerID]:getValue("target_level"), 100,
         fadeTime) )
  elseif action == "button_2_hold_started" then
    -- start moving to 0% from current target level
    dimmer:call(
      "set_level",
      computeMoveParameters(wtp[dimmerID]:getValue("target_level"), 0, fadeTime)
         )
  elseif action:find("button_1_held_") ~= nil or action:find("button_2_held_")
     then
    -- stop current moving action
    dimmer:call("stop")
  end

end
```

WTP - EnergyMeter

Energy meter is a device which can track and count consumed energy (total so far and daily) and sense voltage/current and active power.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `current` *(number, read-only)*

  Recent AC current measument.

  Unit: mA.

  Note: Parameter is optional. Available when sensor is supported - check if *has_current_sensor* label is provided.

- `voltage` *(number, read-only)*

  Recent AC voltage measurement.

  Unit: mV.

  Note: Parameter is optional. Available when sensor is supported - check if *has_voltage_sensor* label is provided.

- `active_power` *(number, read-only)*

  Recent AC active power measurement.

  Unit: mW.

- `energy_consumed_today` *(number, read-only)*

  Sum of energy used today.

  Unit: Wh.

- `energy_consumed_yesterday` *(number, read-only)*

  Sum of energy used yesterday.

  Unit: Wh.

- `energy_consumed_total` *(number, read-only)*

  Total sum of energy used Unit: Wh.

Commands

- `reset_energy_consumed`

  Calls Energy meter to reset energy consumed data.

- `calibration`

  Calls Energy meter to calibrate sensor, adjusting measurements to expected values. Calibration should be done using a resistive load (or as close as possible to the perfect power factor ($\cos\varphi = 1$)) !

  Arguments:

  packed arguments *(table)*:

  - expected voltage in mV
  - expected current in mA
  - expected active power in mW

Examples

Send notification when active power usage rises above 2.5kW

```
-- 2.5 kW = 2500 W = 2500000 mW
local threshold = 2500000
if wtp[10]:changedValue("active_power")
then
  if wtp[10]:getValue("active_power") > treshold
  then
    notify:error("Power usage too high!", "Check your device!")
  end
end
```

WTP - FloodSensor

Battery powered, flood sensor. Detects water leak on flat surfaces.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `flood_detected` *(boolean, read-only)*

  A flag representing the detection of flood / water leak by the sensor.

Examples

Catching alarms

```
if wtp[5]:changedValue("flood_detected") and wtp[5]:getValue("flood_detected")
then
  print("Sensor detected water leak!!!")
  notify:warning("Water leak!", "Water leak detected in toilet!", {1, 3})
end
```

Close the valve and turn on siren on water leak

```
local valve, siren, floodSensor = wtp[1], wtp[2], wtp[3]

if floodSensor:changedValue("flood_detected") and
   floodSensor:getValue("flood_detected")
then
  valve:call("turn_off")
  siren:call("turn_on")
end
```

WTP - HumiditySensor

Battery powered humidity sensor. Measures humidity and sends measurement to central unit.

Sensors measure humidity only every few minutes to save battery. Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `humidity` *(number, read-only)*

  Sensed humidity value.

  Unit: rH% with one decimal number, multiplied by 10.

WTP - IAQSensor

Battery powered Index of Air Quality sensor. Calculates Air Quality Index based on various measures like CO2 or particles level and relative humidity.

Sensors measure values only every few minutes to save battery.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(integer, read-only)*

  Unique network address.

- `signal` *(integer, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(integer, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `iaq` *(integer, read-only)*

  Calculated Index of Air Quality.

- `iaq_accuracy` *(string, read-only)*

  Index of Air Quality calculation accuracy. One of: `unreliable`, `low`, `medium`, `high`.

value	meaning
`unreliable`	The sensor is not yet stabilized or in a run-in status
`low`	Calibration required and will be soon started
`medium`	Calibration on-going
`high`	Calibration is done, now IAQ estimate achieves best performance

- `air_quality` *(string, read-only)*

  Descriptive name for air quality.

raw	description
≤ 20	`very_good`
21 – 50	`good`
51 – 100	`moderate`
101 – 150	`poor`
151 – 200	`unhealthy`
201 – 300	`very_unhealthy`
301 – 500	`hazardous`
> 500	`extreme`

WTP - LightSensor

Battery powered light sensor. Measures light illuminance in lux and sends measurement to central unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(integer, read-only)*

  Unique network address.

- `signal` *(integer, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(integer, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `illuminance` *(integer, read-only)*

  Sensed light illuminance value.

  Unit: lx.

WTP - MotionSensor

Battery powered motion sensor. Based on custom configuration checks whether motion was detected.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `enabled` *(boolean)*

  Enable or disable sensor. eg. If you want sense only at night time, you can setup automation to enable/disable sensor.

- `blind_duration` *(number)*

  Duration of sensor being off after detecting motion.

  Unit: seconds.

- `pulses_threshold` *(number)*

  Sensitivity factor. How many pulses from sensor are needed to treat action as motion. The higher the value, the sensitivity decreases.

- `pulses_window` *(number)*

  Sensitivity factor. Maximum time window in which `pulses_threshold` must occur to treat action as motion. The higher the value, the sensitivity increases.

  Unit: seconds.

- `motion_detected` *(boolean, read-only)*

  Holds latest motion detection state. Remains *true* on motion detection and *false* when `blind_duration` time elapses.

  This parameter doesn't emit event when switch from *true* to *false* happens. If you need to observe such action, you need to use `time_since_motion` parameter.

- `time_since_motion` *(number, read-only)*

  Time since last motion detected. Value of -1 means there wasn't any motion since last system startup.

  Unit: seconds.

Commands

- `enable`

  Enables motion detector.

- `disable`

  Disables motion detection.

- `add_time_since_motion_event`

  Adds additional emitting `time_since_motion` event in seconds passed in argument.

  Arguments:

  event reemission delay in seconds *(number)*

Examples

Catching motion events

```
if wtp[4]:changedValue("motion_detected") then
    print("someone is moving aroung!")
end
```

```
if wtp[4]:changedValue("time_since_motion")
  then
  if wtp[4]:getValue("time_since_motion") == 0
  then
    print("someone is moving aroung!")
  end
end
```

Delayed action

```
if dateTime:changed() then
  -- add 30 second delay
  wtp[4]:call("add_time_since_motion_event", 30)
end
```

```
if wtp[4]:changedValue("time_since_motion")
  then
  if wtp[4]:getValue("time_since_motion") == 30
  then
    print("someone was here 30 seconds ago")
  end
end
```

Enable motion detection at sunset and disable it at sunrise

```
if event.type == "sunrise" then
  wtp[3]:call("disable")
elseif event.type == "sunset" then
  wtp[3]:call("enable")
end
```

Enable a light for 5 minutes on motion detection

```
if wtp[4]:changedValue("motion_detected") then
  wtp[60]:setValue("state", true)
  wtp[60]:setValueAfter("state", false, 5 * 60)
end
```

Reconfigure thermostat when motion detected

```
if wtp[4]:changedValue("motion_detected") then
  -- time limited to 2 hours, temperature 23.5°C
  virtual[1]:call("enable_time_limited_mode", {120, 235})
end
```

WTP - OpeningSensor

Battery powered opening sensor. Checks whether window or door is open. Based on that information system can do some action, for example, turn off heating in that room. Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online*, *offline*, *unknown*, *service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date*, *update_available*, *recovery*, *pending*, *downloading*, *updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `open` *(boolean, read-only)*

  Opening sensor state. Open/Closed.

- `acknowledgment` *(string)*

  Newer sensors support communication protocol with acknowledgment. When enabled, sensor will try deliver state change message three times or until ack is received. May increase battery usage if communication is noisy, but data transfer is more reliable.

  Available values: *on*, *off*, *unsupported*

Examples

Catch open and close events

```
if wtp[4]:changedValue("open") then
  if wtp[4]:getValue("open") then
    print("The window is now open!")
  else
```

```
    print("The window is now closed!")
  end
end
```

WTP - PressureSensor

Battery powered pressure sensor. Measures pressure and sends measurement to central unit.

Sensors measure pressure only every few minutes to save battery.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(integer, read-only)*

  Unique network address.

- `signal` *(integer, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(integer, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `pressure` *(integer, read-only)*

  Sensed pressure value.

  Unit: hPa with one decimal number, multiplied by 10.

WTP - RadiatorActuator

Battery powered radiator actuator. Controls valve opening e.g. based temperature regulator or thermostat state.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `opening` *(number)*

  Current valve opening level.

  Unit: %.

- `opening_minimum` *(number)*

  Lower valve opening level. Could not be greater than maximum. Setting minimum value above current opening value, will also change current opening value to minimum.

  Unit: %.

- `opening_maximum` *(number)*

  Upper valve opening level. Could not be less than minimum. Setting maximum value below current opening value, will also change current opening value to maximum.

  Unit: %.

Commands

- `open`

  Opens radiator actuator to desired value in percent passed in argument.

  Argument:

  actuator opening in 1% *(number)*

- `calibration`

  Calls Radiator Actuator to calibrate on next communication cycle.

  NOTE: Cannot be executed when radiator actuator does not have `calibration_support` label!

Examples

Regulate valve based on room temperature

```
sensor = wtp[1]
valve = wtp[2]

if sensor:changedValue("temperature") then
  current_temperature = sensor:getValue("temperature")

  if current_temperature > 220 then
    valve:call("open", 0)
  elseif current_temperature > 200 then
    valve:call("open", 50)
  else
    valve:call("open", 100)
  end
end
```

WTP - Relay

Execution module that changes state depending on the control signal. Relay can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state` *(boolean)*

  State of the output. On/Off.

- `timeout` *(number)*

  Protection functionality, that will set device state to off if there are communication problems.

  Unit: minutes.

- `timeout_enabled` *(boolean)*

  Parameter that indicates if timeout functionality is enabled.

- `current` *(number, read-only)*

  Recent AC current measument in mA.

  Note: Parameter is optional. Available when power meter is supported - check if *has_power_meter* label is provided.

- `voltage` *(number, read-only)*

  Recent AC voltage measurement in V.

  Note: Parameter is optional. Available when power meter is supported - check if *has_power_meter* label is provided.

- `active_power` *(number, read-only)*

  Recent AC active power measurement in W.

Note: Parameter is optional. Available when power meter is supported - check if *has_power_meter* label is provided.

- `energy_consumption` *(double/real, read-only)*

  Sum of energy consumed by output in last 5 minutes.

  Note: Parameter is optional. Available when power meter is supported - check if *has_power_meter* label is provided.

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: *auto, fixed, off*

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle.

  Example: *#FF00FF*

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active eg. motor is working.

  Example: *#FFFF00*

  Note: Available when backlight is supported
  - check if *has_backlight* label is provided.

- `inverted` *(boolean)*

  Indicates if should invert physical state of relay compared to represented state in application.

Commands

- `turn_on`

  Turns on relay output.

- `turn_off`

  Turns off relay output.

- `toggle`

  Changes relay output to opposite.

Examples

Turn relay on between 19:00 and 21:00

```
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    wtp[4]:call("turn_on")
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    wtp[4]:call("turn_off")
  end
end
```

Turn on the light for 5 minutes when motion detected

```
if wtp[4]:changedValue("motion_detected") then
  wtp[60]:setValue("state", true)
  wtp[60]:setValueAfter("state", false, 5 * 60)
end
```

WTP - RGB Controller

Device that controlls color and light intensity of output LED.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to device with **ID 6**.

WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state` *(boolean)*

  State of the output. On/Off.

- `brightness` *(number, read-only)*

  Desired light intensity level on which device is set or level on which device will be set when turned on. (depending on `state`) Unit: %.

- `led_color` *(string, read-only)*

  HTML/Hex RGB color that device will set on its output led strip.

- `white_temperature` *(number, read-only)*

  White temperature that device will set on its output led strip.

  Unit: Kelvins

- `color_mode` *(string, read-only)*

  Color mode that device is set on. One of: `rgb`, `temperature`, `animation`.

- `led_strip_type` *(string)*

  Led strip type that is connected with device. One of: `rgb`, `rgbw`, `rgbww`.

- `white_temperature_correction` *(number)*

  White color temperature correction. Applies when led_strip_type set to rgbw.

- `cool_white_temperature_correction` *(number)*

  Cool white color temperature correction. Applies when led_strip_type set to rgbww.

- `warm_white_temperature_correction` *(number)*

  Warm white color temperature correction. Applies when led_strip_type set to rgbww.

- `active_animation` *(number, read-only)*

  Active animation id if animation was activated. Null value when no animation active.

Commands

- `turn_on`

  Turns on output.

- `turn_off`

  Turns off output.

- `toggle`

  Changes state to opposite.

- `set_brightness`

  Sets light intensity level to desired level smoothly during given time.

  Argument:

  packed arguments *(table)*:

  - light intensity in % *(number)*:
    - minumum: 1
    - maximum: 100
  - transition time in 0.1s *(number)*:
    - minimum: 1 *(100 ms)*
    - maximum: 6000 *(10 minutes)*
    - parameter is optional *(500 ms default)*

- `set_color`

  Sets device output to requested color in RGB mode during requested period of time. Set `color_mode` to `rgb`.

  Argument:

  packed arguments *(table)*:

  - HTML/Hex RGB color representation *(string)*
    - example: `#88fb1c`
  - transition time in 0.1s *(number)*
    - minimum: 1 *(100 ms)*
    - maximum: 6000 *(10 minutes)*
    - parameter is optional *(500 ms default)*

- `set_temperature`

  Sets device output to requested white temperature during requested period of time. Set `color_mode` to `temperature`.

  Argument:

  packed arguments *(table)*:

  - color temperature in Kelvins *(number)*
    - minumum: 1000
    - maximum: 40000
  - transition time in 0.1s *(number)*
    - minimum: 1 *(100 ms)*
    - maximum: 6000 *(10 minutes)*
    - parameter is optional *(500 ms default)*

- `activate_animation`

  Activate animation with specified id.

  Argument:

  packed arguments *(table)*:

  - `id` - ID of animation that will be activated *(number)*

- `stop_animation`

  Stops active animation and call device to return to previous `color_mode`.

Examples

Turn on light to specific color at 19:00 and turn off at 21:00

```
local rgb = wtp[4]

if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    rgb:call("set_color", {"#eedd11", 10})
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    rgb:call("turn_off")
  end
end
```

Tune color temperature based on the time of day

```
local rgb = wtp[79]

if dateTime:changed() then
  if dateTime:getHours() == 16 and dateTime:getMinutes() == 0 then
    -- afternoon, neutral white at 75%
    rgb:call("set_temperature", {5000})
    rgb:call("set_brightness", {75})
  elseif dateTime:getHours() == 18 and dateTime:getMinutes() == 30 then
    -- evening, warm white at 45%
    rgb:call("set_temperature", {3000, 600})
```

```
    rgb:call("set_brightness", {45, 600})
  end
end
```

Activate an animation by id

```
local rgb = wtp[79]
local animation_id = 2

rgb:call("activate_animation", {id=animation_id})
```

Stop active animation

```
local rgb = wtp[79]
rgb:call("stop_animation")
```

Activate an animation by id when device state changes

```
local rgb = wtp[79]
local animation_id = 3

if wtp[3]:changedValue("state") then
  rgb:call("activate_animation", {id=animation_id})
end
```

WTP - SmokeSensor

Battery powered, optical Smoke sensor. Detects smoke presence, high temperature (eg. fire) and tamper.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `locked` *(boolean)*

  Sensing/detection lock status. If `true` it means sensor won't report high temperature and smoke detection alarms.

- `dirt_level` *(number, read-only)*

  The current dirt (contamination) level of the optical sensor.

  Unit: %.

- `smoke_detected` *(boolean, read-only)*

  A flag representing the detection of smoke by the sensor.

- `high_temperature_detected` *(boolean, read-only)*

  A flag representing the detection of high temperature (eg. fire) by the sensor.

- `tamper_detected` *(boolean, read-only)*

  A flag representing the detection of tamper (eg. the sensor is not in the correct position or someone is trying to take it off).

- `uptime` *(number, read-only)*

  Time since sensor start.

  Unit: seconds.

Commands

- `lock`

  Locks the sensor. Smoke detection and high temperature alarms will not be reported.

- `unlock`

  Unlocks the sensor. Smoke detection and high temperature alarms will be reported if detected.

- `test`

  Starts device self-test.

- `reset`

  Resets current device alarms.

Examples

Catching different alarms

```
if wtp[5]:changedValue("smoke_detected") and wtp[5]:getValue("smoke_detected")
then
  print("Sensor detected smoke!!!")
end

if ( wtp[5]:changedValue("high_temperature_detected")
  and wtp[5]:getValue("high_temperature_detected") )
then
  print("Sensor detected high temperature!!!")
end

if wtp[5]:changedValue("tamper_detected") and wtp[5]:getValue("tamper_detected")
then
  print("Someone is trying to steal your sensor!")
end
```

Locking and unlocking

```
-- lock using parameter
wtp[5]:setValue("locked", true)

--unlock using parameter
wtp[5]:setValue("locked", false)

--lock using command
wtp[5]:call("lock")

--unlock using command
wtp[5]:call("unlock")
```

Reacting to smoke

```
local fan, siren, smokeSensor = wtp[2], wtp[4], wtp[8]

if smokeSensor:changedValue("smoke_detected") and
   smokeSensor:getValue("smoke_detected")
then
  fan:call("turn_on")
  siren:call("turn_on")
end
```

WTP - TemperatureRegulator

Temperature regulator notifies when desired temperature is reached in room. Can by battery or AC 230V powered. Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_mode.current` *(string, read-only)*

  Regulator target temperature mode. Specifies if regulator works in `constant` mode with one target temperature, `time_limited` mode with one temporary target temperature or according to schedule in `schedule` mode with many target temperatures in time, configured by user.

  Parameter is read only, use commands to change target temperature mode! Parameter cannot be `schedule` if thermostat doesnt have `has_schedule` label!

When not associated with Virtual Thermostat it will always work in `constant` mode.

Available values: *constant, schedule, time_limited*. Default: *constant*

- `target_temperature_mode.remaining_time` *(number, read-only)*

  Remaining time until `time_limited` mode ends. Cannot be modified directly - use commands.

  Unit: minutes.

- `target_temperature_miniumum` *(number)*

  Lower limit of the target temperature. Could not be greater than maximum. Setting minimum value above target value, will also change target value to minimum.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_maximum` *(number)*

  Upper limit of the target temperature. Could not be less than minimum. Setting maximum below target, will also change target value to minimum. Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_reached` *(boolean)*

  Controls device‘s algorithm state indicator (available on some regulators). eg LED Diode. May be controlled by external algorithms or devices such as Thermostat (when thermostat is active, indicator will blink)

- `system_mode` *(string)*

  Indicates external system work mode. Used to display proper icon on the regulator.

  Available only if device has label *has_system_mode*.

  May only be changed if device is not assigned to thermostat (has not label *managed_by_thermostat*).

  Available values: *off, heating, cooling*. Default: *heating*

- `keylock` *(string)*

  Device keylock state. Available values: *on, off, unsupported*

- `confirm_time_mode` *(boolean, read-only)*

  Mainly for Mobile/Web App purposes. Indicates if time mode modal should be displayed when changing thermostat temperature. Controlled by Virtual Thermostat.

Commands

- `set_target_temperature`

  Calls Temperature Regulator to change `constant` or `time_limited` mode target temperature to the desired value.

If regulator works in `time_limited` mode it will change target temperature only, not affecting `remaining_time`.

If regulator works in `schedule` mode it will change target temperature mode to `constant`.

Argument:

target temperature in 0.1°C *(number)*

- `enable_constant_mode`

  Calls Temperature Regulator to change target temperature mode to `constant`. When regulator is already in `constant` mode, it will change mode `target_temperature` only.

  NOTE: Cannot be executed when regulator is not associated with Thermostat.

  Argument:

  target temperature in 0.1°C *(number)*

- `enable_time_limited_mode`

  Calls Temperature Regulator to change mode and target temperature mode to `time_limited` for desired time.

  When regulator is already in `time_limited` mode, it will change `remaining_time` or/and `target_temperature` depending on payload.

  NOTE: Cannot be executed when regulator is not associated with Thermostat.

  Argument:

  packed arguments *(table)*:

  - remaining time in minutes *(number)*
  - target temperature in 0.1°C *(number)*

- `disable_time_limited_mode`

  Calls Temperature Regulator to disable `time_limited` and go back to previous target temperature mode. When regulator is not in `time_limited` mode, it will do nothing.

  NOTE: Cannot be executed when regulator is not associated with Thermostat.

Examples

Raise target temperature between 15:00 and 20:00

```
if dateTime:changed() then
  if dateTime:getHours() == 15 and dateTime:getMinutes() == 0 then
    wtp[5]:call("set_target_temperature", 220)
  elseif dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
    wtp[5]:call("set_target_temperature", 190)
  end
end
```

WTP - TemperatureSensor

Battery powered temperature sensor. Measures temperature and sends measurement to central unit. Temperature sensors measure temperature only every few minutes to save battery. Can be assigned to virtual thermostat in web application as room or floor sensor.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(integer, read-only)*

  Unique network address.

- `signal` *(integer, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `battery` *(integer, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `temperature` *(integer, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

- `calibration` *(integer)*

  Static point temperature calibration, used to adjust measurments.

  Unit: °C with one decimal number, multiplied by 10.

WTP - Throttle

Standalone radio controlled throttle.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `opening` *(number)*

  Current opening level.

  Unit: %.

- `impulses` *(number, read-only)*

  Current fan speed-o-meter impulses reading.

  Unit: %.

- `flow` *(double/real, read-only)*

  Calculated throttle flow based on opening and impulses.

  Flow is calculated using formula in the `formula` parameter.

- `formula` *(string)*

  Formula used to calculate flow. Refering to `object` you can get data you need to calculate, for example get `opening` from object: `object.opening`. Should contain only calculations returning number. Should not contain any condition statements, loops and more complicated code.

  Example:

  ```
  object.opening * 2 + math.sqrt(object.impulses)
  ```

  Default:

```
8 + (object.impulses * ((1.32 - object.opening / 100)^2 * -0.35 + 1.9)) *
   0.055
```

Commands

- `calibration`

  Requests immediate calibration.

- `factory_reset`

  Requests device factory reset.

Examples

Synchronize throttle with radiator actuator

```
actuator = wtp[1]
throttle = wtp[2]

if actuator:changedValue("opening") then
  throttle:setValue("opening", actuator:getValue("opening"))
end
```

WTP - TwoStateInputSensor

Boolean input sensor checks input state and send it to central unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to device with **ID 6**.

WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(integer, read-only)*

  Unique network address.

- `signal` *(integer, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state` *(boolean, read-only)*

  State of the input.

- `inverted` *(boolean)*

  Indicates if physical state of input compared to represented state in application should be inverted.

WTP - FanControl

Fan Control is a device which is used to control ventilation fan.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `wtp` container eg. `wtp[6]` gives you access to wireless device with **ID 6**. WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online*, *offline*, *unknown*, *service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date*, *update_available*, *recovery*, *pending*, *downloading*, *updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state.current` *(string, read-only)*

  Current state the fan is working in. Available states are: *off*, *automatic*, *holiday*, *hurricane*, *party*, *hearth*, *flaccid*.

- `state.previous` *(string, read-only)*

  Previous state the fan was working in. Available states are: *off*, *automatic*, *holiday*, *hurricane*, *party*, *hearth*, *flaccid*.

- `state.remaining_time` *(integer, read-only)*

  Remaining time of the temporal state. When passes current state is set to previous state. Temporal states are: *hurricane*, *party*, *hearth*, *flaccid*.

- `state_configuration.auto.co2_thresholds` *(table of size 3)*

  Three steps of CO2 thresholds specifying wroking in *automatic* state.

- `state_configuration.holiday.air_out_interval` *(number)*

  Interval for airing in *holiday* state. Unit: days

- `state_configuration.hurricane.default_duration` *(number)*

  Default duration of *hurricane* state. Unit: seconds

- `state_configuration.party.default_duration` *(number)*

  Default duration of *party* state. Unit: seconds

- `state_configuration.hearth.default_duration` *(number)*

  Default duration of *hearth* state. Unit: seconds

- `state_configuration.flaccid.default_duration` *(number)*

  Default duration of *flaccid* state. Unit: seconds

- `computed_flow` *(number)*

  Value of computed flow passed from other devices.

Commands

- `set_state`

  Calls Fan Control to change its current state.

  Arguments:

  packed arguments *(table)*:

  - state to set *(string)*
  - duration the state should be active in seconds *(number)* **Note**: This parameter is forbidden for permanent states and is optional for temporal states. If it is not passed default duration is used.

TECH - CommonHeatBuffer

Device plugged into RS input in central unit. Heat buffer representation. Allows user to read and modify buffer parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired target (setpoint) temperature, which device will try to achieve.

  Unit: °C

  NOTE: Can be changed only if device is in `fixed` target temperature mode.

- `target_temperature_mode` *(string)*

  Defines whether target temperature is fixed or dynamic eg. computed by heat curve.

  NOTE: Can be changed only if device has associated temperature curve.

  Available values: *fixed*, *heat_curve*. Default: *fixed*

- `temperature_down` *(number, read-only)*

  Sensed temperature in lower part of buffer.

  Unit: °C with one decimal number, multiplied by 10.

  Note: Parameter is optional. Available when: check if *temperature_down_available* label is provided.

- `temperature_up` *(number, read-only)*

  Sensed temperature in upper part of buffer.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_reached` *(boolean, read-only)*

  Indicates if target temperature is reached.

- `name_text` *(number, read-only)*

  Buffer name ID. ID text from TECH translations.

TECH - CH PumpAdditional

Device plugged into RS input in central unit. Additional CH Pump representation. Allows user to read and modify CH Pump parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `pump_work` *(boolean, read-only)*

  Current pump working state on/off.

- `algorithm_type` *(number, read-only)*

  Device name ID. ID text from TECH translations.

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `temperature_central_heating` *(number, read-only)*

  Current central heating temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_hysteresis` *(number, read-only)*

  Current hysteresis temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_threshold` *(number, read-only)*

  Current threshold temperature.

  Unit: °C with one decimal number, multiplied by 10.

TECH - CommonDHW

Device plugged into RS input in central unit. Common DHW representation. Allows user to read and modify DHW parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired target (setpoint) temperature, which device will try to achieve. Unit: °C

  NOTE: Can be changed only if device is in `fixed` target temperature mode.

- `target_temperature_mode` *(string)*

  Defines whether target temperature is fixed or dynamic eg. computed by heat curve.

  NOTE: Can be changed only if device has associated temperature curve.

  Available values: *fixed, heat_curve*. Default: *fixed*

- `target_temperature_minimum` *(number, read-only)*

  Lower limit of the target temperature. Could not be greater than maximum. Unit: °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature. Could not be less than minimum.

  Unit: °C

- `correction` *(number, read-only)*

  Target temperature correction.

  Unit: °C

- `temperature_central_heating` *(number, read-only)*

  Current central heating temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_domestic_hot_water` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `pump_work` *(boolean, read-only)*

  Current pump working state on/off.

Examples

Set target temperature to 45 in summer mode and 55 in other modes

```
pellet_ch_main = tech[7]
dhw = tech[8]

if dateTime:changed() then
  if pellet_ch_main:getValue("operations_mode") == "summer_mode" then
    dhw:setValue("target_temperature", 45)
  else
    dhw:setValue("target_temperature", 55)
  end
end
```

TECH - DHW PumpAdditional

Device plugged into RS input in central unit. Additional DHW Pump representation. Allows user to read and modify DHW Pump parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `pump_work` *(boolean, read-only)*

  Current pump working state on/off.

- `algorithm_type` *(number, read-only)*

  Device name ID. ID text from TECH translations.

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C

- `temperature_domestic_hot_water` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature. Setting maximum below target, will also change target value to maximum.

  Unit: °C

- `temperature_threshold` *(number, read-only)*

  Current threshold temperature.

  Unit: °C with one decimal number, multiplied by 10.

TECH - FloorPumpAdditional

Device plugged into RS input in central unit. Additional Floor Pump representation. Allows user to read pump parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `pump_work` *(boolean, read-only)*

  Current pump working state on/off.

- `algorithm_type` *(number, read-only)*

  Device name ID. ID text from TECH translations.

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `temperature_floor` *(number, read-only)*

  Current floor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `minimum_temperature` *(number, read-only)*

  Lower limit of the floor temperature. .

  Unit: °C

- `maximum_temperature` *(number, read-only)*

  Upper limit of the floor temperature.

  Unit: °C

TECH - HeatPump

Device plugged into RS input in central unit. Heat pump representation. Allows user to read and modify heat pump parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `blockade` *(boolean)*

  Valve work blockade. If set to true valve will stop working.

- `work_mode` *(number)*

  The heat pump current operating mode (text id)

- `work_mode_list` *(number, read-only)*

  Available work mode list (text id)

- `fan` *(number, read-only)*

  Current fan state (0 - 100%)

- `compressor_state` *(boolean, read-only*

  Current compressor state

- `cop` *(number, read-only)*

  Coefficient of performance

- `cop_text` *(number, read-only)*

  Text id for cop (cooling/heating)

- `temperature_outdoor` *(number, read-only)*

  Current outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `actual_power` *(number, read-only)*

  Current heating power Unit: WAT

- `actual_power_text` *(number, read-only)*

  Text id for cop (cooling/heating)

- `upper_source_in_temp` *(number, read-only)*

  Current upper source temperatrure.

  Unit: °C with one decimal number, multiplied by 10.

- `electrical_power` *(number, read-only)*

  Current consumed electrical power Unit: WAT

- `valve_buffer_state_text` *(number, read-only)*

  Current valve-buffer state text id

- `ehome_work_mode` *(string)*

  Current heat pump work mode (auto,heating,cooling)

- `compressor_oil_temperature` *(number, read-only)*

  Current compressor oil temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `current_flow` *(number, read-only)*

  Current flow Unit: l/h (-1 error)

- `current_power_consumption` *(number, read-only)*

  Current power consumption Unit: WAT

- `evd_valve_opening` *(number, read-only)*

  Current evd valve opening Unit: percent with one decimal number, multiplied by 10.

- `upper_source_pump_state` *(number, read-only)*

  Current upper source pump state Unit: percent with one decimal number, multiplied by 10.

- `evd_condensing_pressure` *(number, read-only)*

  Current evd condensing pressure Unit: paskal.

- `compressor_last_work_time` *(number, read-only)*

  Current compressor last work time Unit: second.

Commands

- `set_ehome_work_mode`

  Calls HeatPump to change ehome work mode (device climate mode)

  Argument:

  ehome work mode (climate mode), one of: (auto,heating,cooling) *(string)*

- `set_work_mode`

  Calls HeatPump to change work mode

  Argument:

  work mode id, one of available in property `work_mode_list` *(number)*

TECH - HumiditySensor

Device plugged into RS input in central unit. Humidity sensor. Measures humidity and sends measurement to central unit. Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `humidity` *(number, read-only)*

  Sensed humidity value.

  Unit: rH% with one decimal number, multiplied by 10.

TECH - PelletBoiler

Device plugged into RS input in central unit. Pellet boiler representation. Allows user to read and modify boiler parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `feeder` *(boolean, read-only)*

  Feeder working state. On/Off.

- `stocker` *(boolean, read-only)*

  Secondary feeder working state. On/Off.

- `fan` *(number, read-only)*

  Current fan speed (0-100).

  Unit: %

- `grid` *(boolean, read-only)*

  Current grid working state. On/Off.

- `heater` *(boolean, read-only)*

  Current heater working state. On/Off.

- `state` *(boolean, read-only)*

  Current pellet boiler working state. On/Off.

- `state_text` *(number, read-only)*

  Pellet boiler working state ID name. ID text from TECH translations.

- `temperature_central_heating` *(number, read-only)*

  Current central heating temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_exhaust` *(number, read-only)*

  Current exhaust temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_return` *(number, read-only)*

  Current return temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_feeder` *(number, read-only)*

  Current feeder temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `fire` *(boolean, read-only)*

  Current fire state.

- `target_temperature` *(number)*

  Desired target (setpoint) temperature, which device will try to achieve. Unit: °C

  NOTE: Can be changed only if device is in `fixed` target temperature mode.

- `target_temperature_mode` *(string)*

  Defines whether target temperature is fixed or dynamic eg. computed by heat curve.

  NOTE: Can be changed only if device has associated temperature curve.

  Available values: *fixed*, *heat_curve*. Default: *fixed*

- `target_temperature_miniumum` *(number, read-only)*

  Lower limit of the target temperature.

  Unit: °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature.

  Unit: °C

- `correction` *(number, read-only)*

  Target temperature correction resulting from some algorithms in pellet controller.

  Unit: °C

- `blockade` *(boolean)*

  Pellet boiler work blockade. If set to true pellet will stop working.

- `tray_calibrate` *(boolean, read-only)*

  Parameter which indicates if tray is calibrated.

- `tray_percent` *(number, read-only)*

  Percentage of tray filling. Will show proper value only if tray is calibrated.

- `cause_of_damping` *(table of numbers, read-only)*

  Array of text IDs which show a cause of damping. ID text from TECH translations.

Examples

Stop pellet boiler when all thermostats reach their target temperatures

```
thermostats = {virt[3], virt[4], virt[5], virt[6]}
pellet = tech[2]

if dateTime:changed()
then
  local temperature_reached = utils.table:every(thermostats, function (th)
    return not th:getValue('state')
  end)

  pellet:setValue("blockade", temperature_reached)
end
```

TECH - PelletCHMain

Device plugged into RS input in central unit. Pellet CH representation. Allows user to read and modify CH parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `pump_work` *(boolean, read-only)*

  Current pump working state on/off.

- `operations_mode` *(string)*

  Current pump mode. One of following: `house_heating`, `boiler_priority`, `parallel_pumps`, `summer_mode`

Examples

Change modes based on current season

```
if dateTime:changed()
then
  if dateTime:getHours() == 0 and dateTime:getMinutes() == 0
  then
    if dateTime:getMonth() >= 4 and dateTime:getMonth() <= 9
    then
      -- april – september, change to summer mode
      tech[7]:setValue("operations_mode", "summer_mode")
    else
      -- rest of the year, change to boiler priority mode
      tech[7]:setValue("operations_mode", "boiler_priority")
    end
  end
end
```

TECH - ProtectPumpAdditional

Device plugged into RS input in central unit. Additional Protect Pump representation. Allows user to read additional pump parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `pump_work` *(boolean, read-only)*

  Current pump working state on/off.

- `algorithm_type` *(number, read-only)*

  Device name ID. ID text from TECH translations.

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `temperature_central_heating` *(number, read-only)*

  Current central heating temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_return` *(number, read-only)*

  Current return temperature.

  Unit: °C with one decimal number, multiplied by 10.

TECH - RelayAdditional

Device plugged into RS input in central unit. Additional Relay representation. Allows user to read additional relay parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `pump_work` *(boolean, read-only)*

  Current relay state on/off.

- `algorithm_type` *(number, read-only)*

  Device name ID. ID text from TECH translations.

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

TECH - Relay

Device plugged into RS input in central unit. Execution module that changes state depending on the control signal. Relay can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(boolean)*

  State of the output. On/Off.

- `timeout` *(number)*

  Protection functionality, that will set device state to off if there are communication problems.

  Unit: minutes.

- `timeout_enabled` *(boolean)*

  Parameter that indicates if timeout functionality is enabled.

- `inverted` *(boolean)*

  Indicates if should invert physical state of relay compared to represented state in application.

Commands

- `turn_on`

  Turns on relay output.

- `turn_off`

  Turns off relay output.

- `toggle`

  Changes relay output to opposite.

Examples

Turn on relay between 19:00 and 21:00

```
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    tech[4]:call("turn_on")
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    tech[4]:call("turn_off")
  end
end
```

TECH - TemperatureRegulator

Device plugged into RS input in central unit. Temperature regulator notifies when desired temperature is reached in room. Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_mode.current` *(string, read-only)*

  Regulator target temperature mode. Specifies if regulator works in `constant` mode with one target temperature, `time_limited` mode with one temporary target temperature or according to schedule in `schedule` mode with many target temperatures in time, configured by user.

  Parameter is read only, use commands to change target temperature mode! Parameter cannot be `schedule` if thermostat doesnt have `has_schedule` label!

  When not associated with Virtual Thermostat it will always work in `constant` mode.

  Available values: *constant, schedule, time_limited*. Default: *constant*

- `target_temperature_mode.remaining_time` *(number, read-only)*

  Remaining time until `time_limited` mode ends. Cannot be modified directly - use commands.

  Unit: minutes.

- `target_temperature_miniumum` *(number)*

  Lower limit of the target temperature. Could not be greater than maximum. Setting minimum value above target value, will also change target value to minimum.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_maximum` *(number)*

  Upper limit of the target temperature. Could not be less than minimum. Setting maximum below target, will also change target value to minimum. Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_reached` *(boolean)*

  Controls device‘s algorithm state indicator (available on some regulators). eg LED Diode. May be controlled by external algorithms or devices such as Thermostat (when thermostat is active, indicator will blink)

- `confirm_time_mode` *(boolean, read-only)*

  Mainly for Mobile/Web App purposes. Indicates if time mode modal should be displayed when changing thermostat temperature. Controlled by Virtual Thermostat.

Commands

- `set_target_temperature`

  Calls Temperature Regulator to change `constant` or `time_limited` mode target temperature to the desired value.

  If regulator works in `time_limited` mode it will change target temperature only, not affecting `remaining_time`.

  If regulator works in `schedule` mode it will change target temperature mode to `constant`.

  Argument: target temperature in 0.1°C *(number)*

- `enable_constant_mode`

  Calls Temperature Regulator to change target temperature mode to `constant`. When regulator is already in `constant` mode, it will change mode `target_temperature` only.

  NOTE: Cannot be executed when regulator is not associated with Thermostat.

  Argument: target temperature in 0.1°C *(number)*

- `enable_time_limited_mode`

  Calls Temperature Regulator to change mode and target temperature mode to `time_limited` for desired time.

  When regulator is already in `time_limited` mode, it will change `remaining_time` or/and `target_temperature` depending on payload.

First parameter is `remaining_time`, second is `target_temperature`.

NOTE: Cannot be executed when regulator is not associated with Thermostat.

Argument:

packed arguments *(table)*:

- remaining time in minutes *(number)*
- target temperature in 0.1°C *(number)*

- `disable_time_limited_mode`

  Calls Temperature Regulator to disable `time_limited` and go back to previous target temperature mode. When regulator is not in `time_limited` mode, it will do nothing.

  NOTE: Cannot be executed when regulator is not associated with Thermostat.

Examples

Raise target temperature between 15:00 and 20:00

```
if dateTime:changed() then
  if dateTime:getHours() == 15 and dateTime:getMinutes() == 0 then
    tech[5]:call("set_target_temperature", 220)
  elseif dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
    tech[5]:call("set_target_temperature", 190)
  end
end
```

TECH - TemperatureSensor

Device plugged into RS input in central unit. Temperature sensor. Measures temperature and sends measurement to central unit. Can be assigned to virtual thermostat in web application as room or floor sensor.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `temperature` *(number, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

- `calibration` *(number)*

  Static point temperature calibration, used to adjust measurments.

  Unit: °C with one decimal number, multiplied by 10.

TECH - TwoStateInputSensor

Device plugged into RS input in central unit. Boolean input sensor checks input state and send it to central unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(boolean, read-only)*

  State of the input. On/Off.

- `inverted` *(boolean)*

  Indicates if physical state of input compared to represented state in application should be inverted.

TECH - Valve

Device plugged into RS input in central unit. Valve representation. Allows user to read and modify valve parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server. Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**. TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired target (setpoint) temperature, which device will try to achieve. Unit: °C

  NOTE: Can be changed only if device is in `fixed` target temperature mode.

- `target_temperature_mode` *(string)*

  Defines whether target temperature is fixed or dynamic eg. computed by heat curve.

  NOTE: Can be changed only if device has associated temperature curve.

  Available values: *fixed, heat_curve*. Default: *fixed*

- `target_temperature_miniumum` *(number, read-only)*

  Lower limit of the target temperature. Unit: °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature. Unit: °C

- `correction` *(number, read-only)*

  Target temperature correction resulting from some algorithms in valve controller. Unit: °C

- `temperature_valve` *(number, read-only)*

  Current valve temperature. Unit: °C with one decimal number, multiplied by 10.

- `open_percent` *(number, read-only)*

  Current open percentage. Unit: %

- `state` *(number, read-only)*

  Valve working state. With following meanings. 1 - Off 2 - Calibration 4 - Return protection 5 - Boiler protection 6 - Working 7 - Blockade 8 - Alarm 9

  - Manual work

- `state_text` *(number, read-only)*

  Valve working state ID name. ID text from TECH translations.

- `temperature_return` *(number, read-only)*

  Current return temperature. Unit: °C with one decimal number, multiplied by 10.

- `temperature_central_heating` *(number, read-only)*

  Current central heating temperature. Unit: °C with one decimal number, multiplied by 10.

- `room_regulator` *(boolean, read-only)*

  Current room regulator state (target temperature reached).

- `pump_work` *(boolean, read-only)*

  Current pump working state. On/Off.

- `blockade` *(boolean)*

  Valve work blockade. If set to true valve will stop working.

- `weather_control` *(boolean, read-only)*

  Parameter which indicates if weather control is enabled.

- `temperature_outdoor` *(number, read-only)*

  Current outdoor temperature. Unit: °C with one decimal number, multiplied by 10.

- `work_mode` *(string)*

  Current valve work mode (heating,cooling)

Examples

Close valve if thermostat reached target temperature

```
if dateTime:changed() then
  thermostat = virt[3]
  valve = tech[3]

  temperature_reached = not thermostat:getValue("state")
  valve:setValue("blockade", temperature_reached)
end
```

TECH - Ventilation

Device plugged into RS input in central unit. Valve representation. Allows user to read and modify valve parameters.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container eg. `tech[6]` gives you access to device with **ID 6**.

TECH devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C

- `target_temperature_miniumum` *(number, read-only)*

  Lower limit of the target temperature.

  Unit: °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature.

  Unit: °C

- `cooling` *(boolean, read-only)*

  Current cooling state,

- `pre_heating` *(boolean, read-only)*

  Current pre_heating state,

- `post_heating` *(boolean, read-only)*

  Current post_heating state,

- `gwc` *(boolean, read-only)*

  Current gwc state,

- `humidifier` *(boolean, read-only)*

  Current humidifier state,

- `bypass` *(boolean, read-only)*

  Current bypass state,

- `intake_temperature` *(number, read-only)*

  current temperature entering the house Unit: °C with one decimal number, multiplied by 10.

- `exhaust_temperature` *(number, read-only)*

  current exhaust temperature Unit: °C with one decimal number, multiplied by 10.

- `extract_temperature` *(number, read-only)*

  current extract temperature Unit: °C with one decimal number, multiplied by 10.

- `supply_temperature` *(number, read-only)*

  current supply temperature Unit: °C with one decimal number, multiplied by 10.

- `additional_temperature_supply` *(number, read-only)*

  current additional_temperature_supply Unit: °C with one decimal number, multiplied by 10.

  Note: Parameter is optional. Available when: check if *additional_temperature_supply_available* label is provided.

- `additional_temperature_outside` *(number, read-only)*

  current additional_temperature_outside Unit: °C with one decimal number, multiplied by 10.

  Note: Parameter is optional. Available when: check if *additional_temperature_outside_available* label is provided.

- `additional_temperature_outside` *(number, read-only)*

  current additional_temperature_outside Unit: °C with one decimal number, multiplied by 10.

- `humidity` *(number, read-only)*

  current humidity Unit: %

- `co2ppm` *(number, read-only)*

  current co2 level Unit: ppm

- `supply_fan_gear` *(number, read-only)*

  current supply fan gear Example: 0-4

- `extract_fan_gear` *(number, read-only)*

  current extract fan gear Example: 0-4

- `supply_fan_flow` *(number, read-only)*

  current supply fan flow Unit: m3/h

- `extract_fan_flow` *(number, read-only)*

  current extract fan flow Unit: m3/h

- `is_flow` *(boolean, read-only)*

  Ventilation flow mode. If set to true ventilation working with flow settings

- `target_flow_supply` *(number)*

  Desired setpoint flow supply, which device will try to achieve.

  Unit: m3/h

- `target_flow_extract` *(number)*

  Desired setpoint flow extract, which device will try to achieve.

  Unit: m3/h

- `min_flow` *(number, read-only)*

  Lower limit of the target target_flow.

  Unit: m3/h

- `max_flow` *(number, read-only)*

  Upper limit of the target target_flow.

  Unit: m3/h

- `work_mode` *(number)*

  Ventilation work mode. If set to true ventilation working with sinum parameters else working standalone

- `state` *(number, read-only)*

  Ventilation working state. With following meanings.

- `state_text` *(number, read-only)*

  Ventilation working state ID name. ID text from TECH translations.

- `target_gear_supply` *(number)*

  Desired setpoint gear supply, which device will try to achieve.

  Unit: %

- `target_gear_extract` *(number)*

  Desired setpoint gear extract, which device will try to achieve.

  Unit: %

- `bypass_work_mode` *(number)*

  The bypass current operating mode (text id)

  Note: Parameter is optional. Available when: check if *bypass_available* label is provided.

- `bypass_work_mode_list` *(number, read-only)*

  Available bypass work mode list (text id)

Commands

- `cooling_on_request`

  Calls Ventilation to send cooling on request.

- `heating_on_request`

  Calls Ventilation to send heating on request.

- `humidifier_on_request`

  Calls Ventilation to send humidifier on request.

- `cooling_off_request`

  Calls Ventilation to send cooling off request.

- `heating_off_request`

  Calls Ventilation to send heating off request.

- `humidifier_off_request`

  Calls Ventilation to send humidifier off request.

- `set_work_mode`

  Calls Ventilation to change work mode

  Argument:

  work mode, one of (auto, sinum) *(string)*

- `set_target_temperature`

  Calls device to change target temperature.

  Argument:

  target temperature in °C without decimals *(number)*

- `set_bypass_work_mode`

  Calls device to change bypass work mode

  Argument:

  bypass work mode text id, one of available in property `bypass_work_mode_list` *(number)*

Modbus - Alpha-Innotec - Heat Pump

Representation of Heat Pump related parameters of Alpha-Innotec device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `temperature_indoor` *(number, read-only)*

  Indoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_indoor` *(number, read-only)*

  Set indoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `fixed_heating_target_temperature` *(number)*

  Set temperature for heating in fixed temperature mode.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `hot_gas_temperature` *(number, read-only)*

  Hot gas temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `overheating` *(number, read-only)*

  Overheating.

  Unit: K with one decimal number, multiplied by 10.

- `lower_source_out_temperature` *(number, read-only)*

  Lower source out temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `lower_source_in_temperature` *(number, read-only)*

  Lower source in temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heat_quantity_hot_water` *(number, read-only)*

  Heat quantity domestic hot water.

  Unit: kW/h with one decimal number, multiplied by 10.

- `heat_quantity_heating` *(number, read-only)*

  Heat quantity heating.

  Unit: kW/h with one decimal number, multiplied by 10.

- `heat_quantity_total` *(number, read-only)*

  Heat quantity total.

  Unit: kW/h with one decimal number, multiplied by 10.

- `electric_heater_active` *(boolean)*

  Indicates electric heater active state.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `operating_hours_heating` *(number, read-only)*

  Operating hours for central heating.

- `operating_hours_hot_water` *(number, read-only)*

  Operating hours for domestic hot water.

- `heat_curve_end_point` *(number, read-only)*

  Heat curve end point.

  Unit: °C with one decimal number, multiplied by 10.

- `heat_curve_parallel_shift` *(number, read-only)*

  Heat curve parallel shift.

  Unit: °C with one decimal number, multiplied by 10.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Modbus - Alpha-Innotec - Main DHW

Representation of DHW related parameters of Alpha-Innotec device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(integer)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(integer, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `emergency_electric_element_dhw_active` *(boolean)*

  Indicates electric heater active state.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Modbus - Alpha Innotec - Temperature Sensor

Representation of Temperature sensor related parameters of Alpha Innotec device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `temperature` *(number, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

Modbus - Eastron SDM630 - Energy Meter

Representation of Energy Meter related parameters of Eastron SDM630 device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `phase_1.active_power` *(number, read-only)*

  First phase active power.

  Unit: mW

- `phase_1.voltage` *(number, read-only)*

  First phase voltage.

  Unit: mV

- `phase_1.current` *(number, read-only)*

  First phase current.

  Unit: mA

- `phase_1.apparent_power` *(number, read-only)*

  First phase apparent power.

  Unit: mVA

- `phase_1.reactive_power` *(number, read-only)*

  First phase reactive power.

  Unit: mVAr

- `phase_1.energy_consumed_total` *(number, read-only)*

  Energy consumed lifetime on first phase.

  Unit: Wh

- `phase_1.energy_consumed_today` *(number, read-only)*

  Energy consumed today on first phase.

  Unit: Wh

- `phase_1.energy_fed_total` *(number, read-only)*

  Energy fed lifetime on first phase.

  Unit: Wh

- `phase_1.energy_fed_today` *(number, read-only)*

  Energy fed today on first phase.

  Unit: Wh
- `phase_1.energy_sum_total` *(number, read-only)*

  Energy sum (consumed + fed) lifetime on first phase.

  Unit: Wh
- `phase_1.energy_sum_today` *(number, read-only)*

  Energy sum (consumed + fed) today on first phase.

  Unit: Wh
- `phase_1.reactive_energy_consumed_total` *(number, read-only)*

  Reactive energy consumed lifetime on first phase.

  Unit: VArh
- `phase_1.reactive_energy_consumed_today` *(number, read-only)*

  Reactive energy consumed today on first phase.

  Unit: VArh
- `phase_1.reactive_energy_fed_total` *(number, read-only)*

  Reactive energy fed lifetime on first phase.

  Unit: VArh
- `phase_1.reactive_energy_fed_today` *(number, read-only)*

  Reactive energy fed today on first phase.

  Unit: VArh
- `phase_1.reactive_energy_sum_total` *(number, read-only)*

  Reactive energy sum (consumed + fed) lifetime on first phase.

  Unit: VArh
- `phase_1.reactive_energy_sum_today` *(number, read-only)*

  Reactive energy sum (consumed + fed) today on first phase.

  Unit: VArh
- `phase_2.active_power` *(number, read-only)*

  Second phase active power.

  Unit: mW
- `phase_2.voltage` *(number, read-only)*

  Second phase voltage.

  Unit: mV
- `phase_2.current` *(number, read-only)*

  Second phase current.

Unit: mA

- `phase_2.apparent_power` *(number, read-only)*

  Second phase apparent power.

  Unit: mVA

- `phase_2.reactive_power` *(number, read-only)*

  Second phase reactive power.

  Unit: mVAr

- `phase_2.energy_consumed_total` *(number, read-only)*

  Energy consumed lifetime on second phase.

  Unit: Wh

- `phase_2.energy_consumed_today` *(number, read-only)*

  Energy consumed today on second phase.

  Unit: Wh

- `phase_2.energy_fed_total` *(number, read-only)*

  Energy fed lifetime on second phase.

  Unit: Wh

- `phase_2.energy_fed_today` *(number, read-only)*

  Energy fed today on second phase.

  Unit: Wh

- `phase_2.energy_sum_total` *(number, read-only)*

  Energy sum (consumed + fed) lifetime on second phase.

  Unit: Wh

- `phase_2.energy_sum_today` *(number, read-only)*

  Energy sum (consumed + fed) today on second phase.

  Unit: Wh

- `phase_2.reactive_energy_consumed_total` *(number, read-only)*

  Reactive energy consumed lifetime on second phase.

  Unit: VArh

- `phase_2.reactive_energy_consumed_today` *(number, read-only)*

  Reactive energy consumed today on second phase.

  Unit: VArh

- `phase_2.reactive_energy_fed_total` *(number, read-only)*

  Reactive energy fed lifetime on second phase.

  Unit: VArh

- `phase_2.reactive_energy_fed_today` *(number, read-only)*

  Reactive energy fed today on second phase.

  Unit: VArh
- `phase_2.reactive_energy_sum_total` *(number, read-only)*

  Reactive energy sum (consumed + fed) lifetime on second phase.

  Unit: VArh
- `phase_2.reactive_energy_sum_today` *(number, read-only)*

  Reactive energy sum (consumed + fed) today on second phase.

  Unit: VArh
- `phase_3.active_power` *(number, read-only)*

  Third phase active power.

  Unit: mW
- `phase_3.voltage` *(number, read-only)*

  Third phase voltage.

  Unit: mV
- `phase_3.current` *(number, read-only)*

  Third phase current.

  Unit: mA
- `phase_3.apparent_power` *(number, read-only)*

  Third phase apparent power.

  Unit: mVA
- `phase_3.reactive_power` *(number, read-only)*

  Third phase reactive power.

  Unit: mVAr
- `phase_3.energy_consumed_total` *(number, read-only)*

  Energy consumed lifetime on third phase.

  Unit: Wh
- `phase_3.energy_consumed_today` *(number, read-only)*

  Energy consumed today on third phase.

  Unit: Wh
- `phase_3.energy_fed_total` *(number, read-only)*

  Energy fed lifetime on third phase.

  Unit: Wh
- `phase_3.energy_fed_today` *(number, read-only)*

  Energy fed today on third phase.

Unit: Wh

- `phase_3.energy_sum_total` *(number, read-only)*

  Energy sum (consumed + fed) lifetime on third phase.

  Unit: Wh

- `phase_3.energy_sum_today` *(number, read-only)*

  Energy sum (consumed + fed) today on third phase.

  Unit: Wh

- `phase_3.reactive_energy_consumed_total` *(number, read-only)*

  Reactive energy consumed lifetime on third phase.

  Unit: VArh

- `phase_3.reactive_energy_consumed_today` *(number, read-only)*

  Reactive energy consumed today on third phase.

  Unit: VArh

- `phase_3.reactive_energy_fed_total` *(number, read-only)*

  Reactive energy fed lifetime on third phase.

  Unit: VArh

- `phase_3.reactive_energy_fed_today` *(number, read-only)*

  Reactive energy fed today on third phase.

  Unit: VArh

- `phase_3.reactive_energy_sum_total` *(number, read-only)*

  Reactive energy sum (consumed + fed) lifetime on third phase.

  Unit: VArh

- `phase_3.reactive_energy_sum_today` *(number, read-only)*

  Reactive energy sum (consumed + fed) today on third phase.

  Unit: VArh

- `total_active_power` *(number, read-only)*

  Total active power on all phases.

  Unit: mW

- `total_apparent_power` *(number, read-only)*

  Total apparent power on all phases.

  Unit: mVA

- `total_reactive_power` *(number, read-only)*

  Total reactive power on all phases.

  Unit: mVAr

- `energy_sum_total` *(number, read-only)*

  Energy sum (consumed + fed) lifetime on all phases.

  Unit: Wh

- `energy_sum_today` *(number, read-only)*

  Energy sum (consumed + fed) today on all phases.

  Unit: Wh

- `reactive_energy_sum_total` *(number, read-only)*

  Reactive energy sum (consumed + fed) lifetime on all phases.

  Unit: VArh

- `reactive_energy_sum_today` *(number, read-only)*

  Reactive energy sum (consumed + fed) today on all phases.

  Unit: VArh

Modbus - EcoAir - Heat Pump

Representation of Heat Pump related parameters of EcoAir device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(string)*

  State of the heat pump: *on, off, emergency*

- `work_mode` *(string)*

  Current work mode of the heat pump: *automatic, cooling, heating*.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: Bar with one decimal number, multiplied by 10.

- `hot_gas_temperature` *(number, read-only)*

  Hot Gas temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `number_of_starts` *(number, read-only)*

  Number of heat pump starts.

- `electric_heater_emergency` *(boolean)*

  Indicates electric heater emergency state.

- `electric_heater_active` *(boolean)*

  Indicates electric heater activation state.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Commands

- `reset_alarms`

  Sends request to heat pump device to reset alarms.

Modbus - EcoAir - Main DHW

Representation of DHW related parameters of EcoAir device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```
if dateTime:changed() then
  local heat_pump = modbus[7]
  local dhw = modbus[8]
  if heat_pump:getValue("work_mode") == "cooling" then
    dhw:setValue("target_temperature", 450)
  else
    dhw:setValue("target_temperature", 550)
  end
end
```

Modbus - EcoGeo - Heat Pump

Representation of Heat Pump related parameters of EcoGeo device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(string)*

  State of the heat pump: *on, off, emergency*

- `work_mode` *(string)*

  Current work mode of the heat pump: *automatic, cooling, heating* **NOTE:** Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `brine_out_temperature` *(number, read-only)*

  Brine out temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `brine_in_temperature` *(number, read-only)*

  Brine in temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `brine_pressure` *(number, read-only)*

  Brine pressure.

  Unit: Bar with one decimal number, multiplied by 10.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

Unit: Bar with one decimal number, multiplied by 10.

- `hot_gas_temperature` *(number, read-only)*

  Hot Gas temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `number_of_starts` *(number, read-only)*

  Number of heat pump starts.

- `electric_heater_emergency` *(boolean)*

  Indicates electric heater emergency state.

- `electric_heater_active` *(boolean)*

  Indicates electric heater activation state.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Commands

- `reset_alarms`

  Sends request to heat pump device to reset alarms.

Modbus - EcoGeo - Main DHW

Representation of DHW related parameters of EcoGeo device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```
if dateTime:changed() then
  local heat_pump = modbus[7]
  local dhw = modbus[8]
  if heat_pump:getValue("work_mode") == "cooling" then
    dhw:setValue("target_temperature", 450)
  else
    dhw:setValue("target_temperature", 550)
  end
end
```

Modbus - Galmet Prima - Heat Pump

Representation of Heat Pump related parameters of Galmet Prima device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `work_mode` *(string)*

  Current work mode of the heat pump: *automatic, cooling, heating*.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `fixed_target_temperature` *(number)*

  Set temperature for heating or cooling in fixed temperature mode.

  Unit: °C.

- `fixed_target_temperature_minimum` *(number, read-only)*

  Minimum value of fixed_target_temperature parameter.

  Unit: °C.

- `fixed_target_temperature_maximum` *(number, read-only)*

  Maximum value of fixed_target_temperature parameter.

  Unit: °C.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: Bar with one decimal number, multiplied by 10.

- `hot_gas_temperature` *(number, read-only)*

  Hot gas temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `water_inlet_temperature` *(number, read-only)*

  Water inlet temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `water_outlet_temperature` *(number, read-only)*

  Water outlet temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `electric_heater_active` *(boolean)*

  Indicates electric heater active state.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Modbus - Galmet Prima - Main DHW

Representation of DHW related parameters of Galmet Prima device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `electric_heater_active` *(boolean)*

  Indicates electric heater active state.

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```
if dateTime:changed() then
  local heat_pump = modbus[7]
  local dhw = modbus[8]
  if heat_pump:getValue("work_mode") == "cooling" then
    dhw:setValue("target_temperature", 45)
  else
    dhw:setValue("target_temperature", 55)
  end
end
```

Modbus - Galmet Prima - Temperature Sensor

Representation of Temperature sensor related parameters of Galmet Prima device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `temperature` *(number, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

Modbus - GoodWe MT/SMT - Inverter

Representation of Inverter related parameters of GoodWe MT/SMT device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `run_mode` *(string, read-only)*

  Inverter current run mode. Available values are: *waiting, normal, fault*

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: mW

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: mW
- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: mV
- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: mA
- `pv_3.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: mW
- `pv_3.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: mV
- `pv_3.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: mA
- `pv_4.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: mW
- `pv_4.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: mV
- `pv_4.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: mA
- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid.

  Unit: mV
- `phase_1.current` *(number, read-only)*

  Current current on first phase of power grid.

  Unit: mA
- `phase_2.voltage` *(number, read-only)*

  Current voltage on second phase of power grid.

Unit: mV

- `phase_2.current` *(number, read-only)*

  Current current on second phase of power grid.

  Unit: mA

- `phase_3.voltage` *(number, read-only)*

  Current voltage on third phase of power grid.

  Unit: mV

- `phase_3.current` *(number, read-only)*

  Current current on third phase of power grid.

  Unit: mA

Commands

- `turn_on`

  Turns on inverter.

- `turn_off`

  Turns off inverter.

Modbus - GoodWe SDT/MS/DNS/XS - Inverter

Representation of Inverter related parameters of GoodWe SDT/MS/DNS/XS device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `run_mode` *(string, read-only)*

  Inverter current run mode. Available values are: *waiting, normal, fault*

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

Unit: mV

- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: mA

- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid.

  Unit: mV

- `phase_1.current` *(number, read-only)*

  Current current on first phase of power grid.

  Unit: mA

- `phase_2.voltage` *(number, read-only)*

  Current voltage on second phase of power grid.

  Unit: mV

- `phase_2.current` *(number, read-only)*

  Current current on second phase of power grid.

  Unit: mA

- `phase_3.voltage` *(number, read-only)*

  Current voltage on third phase of power grid.

  Unit: mV

- `phase_3.current` *(number, read-only)*

  Current current on third phase of power grid.

  Unit: mA

Commands

- `turn_on`

  Turns on inverter.

- `turn_off`

  Turns off inverter.

Modbus - Heatcomp - Heat Pump

Representation of Heat Pump related parameters of Heatcomp device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(string)*

  State of the heat pump: *on, off*

- `work_mode` *(string)*

  Current work mode of the heat pump: *cooling, heating*

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `hot_gas_temperature` *(number, read-only)*

  Hot Gas temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `compressor_percentage` *(number, read-only)*

  Compresor percentage.

  Unit: %/Hz.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `fixed_heating_target_temperature` *(number)*

  Fixed heating target temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `fixed_cooling_target_temperature` *(number)*

  Fixed cooling target temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_mode` *(string)*

  Target temperature mode: *fixed, heat_curve*

- `heat_curve_slope` *(number)*

  Heat curve slope.

  Unit: °C with one decimal number, multiplied by 10.

- `heat_curve_offset` *(number)*

  Heat curve offset.

  Unit: °C with one decimal number, multiplied by 10.

Modbus - Heatcomp - Main DHW

Representation of DHW related parameters of Heatcomp device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```
if dateTime:changed() then
  local heat_pump = modbus[7]
  local dhw = modbus[8]
  if heat_pump:getValue("work_mode") == "cooling" then
    dhw:setValue("target_temperature", 450)
  else
    dhw:setValue("target_temperature", 550)
  end
end
```

Modbus - HeatEco - Heat Pump

Representation of Heat Pump related parameters of HeatEco device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `work_mode` *(string, read-only)*

  Current work mode of the heat pump. Possible values: *cooling_only, heating_only, dhw_only, cooling_with_dhw, heating_with_dhw*

- `fixed_heating_target_temperature` *(integer)*

  Target heating temperature. Unit: °C with one decimal number, multiplied by 10.

- `fixed_cooling_target_temperature` *(integer)*

  Target cooling temperature. Unit: °C with one decimal number, multiplied by 10.

- `bottom_hysteresis` *(integer)*

  Damper factor, which will protect from continous on/off switching when current temperature is below target value. Unit: °C with one decimal number, multiplied by 10.

- `top_hysteresis` *(integer)*

  Damper factor, which will protect from continous on/off switching when current temperature is above target value. Unit: °C with one decimal number, multiplied by 10.

- `pid.proportional_gain` *(integer)*

  (Kp) Proportional gain factor of PID controller.

- `pid.integral_time` *(integer)*

  (Ti) Integral time factor of PID controller.

- `pid.differential_time` *(integer)*

  (Td) Differential time factor of PID controller.

- `water_inlet_temperature` *(integer, read-only)*

  Water inlet temperature. Unit: °C with one decimal number, multiplied by 10.

- `water_outlet_temperature` *(integer, read-only)*

  Water outlet temperature. Unit: °C with one decimal number, multiplied by 10.

- `temperature_outdoor` *(integer, read-only)*

  Outdoor temperature. Unit: °C with one decimal number, multiplied by 10.
- `discharge_gas_temperature` *(integer, read-only)*

  Discharge Gas temperature. Unit: °C with one decimal number, multiplied by 10.
- `suction_gas_temperature` *(integer, read-only)*

  Suction Gas temperature. Unit: °C with one decimal number, multiplied by 10.
- `discharge_pressure` *(integer, read-only)*

  Discharge pressure. Unit: Pascals.
- `suction_pressure` *(integer, read-only)*

  Suction pressure. Unit: Pascals.
- `coil_temperature` *(integer, read-only)*

  Coil temperature. Unit: °C with one decimal number, multiplied by 10.
- `evaporation_temperature` *(integer, read-only)*

  Evaporation temperature. Unit: °C with one decimal number, multiplied by 10.
- `flow_switch_active` *(boolean, read-only)*

  Indicates flow switch state.
- `emergency_switch_active` *(boolean, read-only)*

  Indicates emergency switch state.
- `terminal_signal_switch_active` *(boolean, read-only)*

  Indicates terminal signal switch state.
- `sequential_protection_switch_active` *(boolean, read-only)*

  Indicates sequential protection switch state.
- `fan_high_speed_active` *(boolean, read-only)*

  Indicates whether fan high speed is active.
- `fan_low_speed_active` *(boolean, read-only)*

  Indicates whether fan low speed is active.
- `four_way_valve_active` *(boolean, read-only)*

  Indicates whether four way valve is active.
- `pump_active` *(boolean, read-only)*

  Indicates whether pump is active.
- `three_way_valve_active` *(boolean, read-only)*

  Indicates whether three way valve is active.
- `crankshaft_heater_active` *(boolean, read-only)*

  Indicates electric heater of crankshaft is active.

- `chassis_heater_active` *(boolean, read-only)*

  Indicates electric heater of chassis is active.

- `electric_heater_active` *(boolean, read-only)*

  Indicates electric heater activation state.

- `fan_output` *(integer, read-only)*

  Current fan output value. Unit: % with one decimal number, multiplied by 10.

- `pump_output` *(integer, read-only)*

  Current pump output value. Unit: % with one decimal number, multiplied by 10.

- `fan_mode` *(string, read-only)*

  Current fan mode. Possible values: *day, night, eco, pressure*

- `pump_mode` *(string, read-only)*

  Current pump mode. Possible values: *normal, demand, interval*

- `eev_opening` *(integer, read-only)*

  Current opening of electrinc expansion valve. Unit: %.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not. Indirectly controls the heat pump work mode. **NOTE:** Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not. Indirectly controls the heat pump work mode. **NOTE:** Cannot be modified when device is associated with Heat Pump Manager.

Modbus - HeatEco - Main DHW

Representation of DHW related parameters of HeatEco device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(integer)*

  Desired setpoint temperature, which device will try to achieve. Unit: °C. **NOTE:** Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(integer, read-only)*

  Current domestic hot water temperature. Unit: °C with one decimal number, multiplied by 10.

- `bottom_hysteresis` *(integer)*

  Damper factor, which will protect from continous on/off switching when current temperature is below target value. Unit: °C with one decimal number, multiplied by 10.

- `top_hysteresis` *(integer)*

  Damper factor, which will protect from continous on/off switching when current temperature is above target value. Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand. **NOTE:** Cannot be modified when device is associated with Heat Pump Manager.

Modbus - Huawei SUN2000 - Battery

Representation of Battery related parameters of Huawei SUN2000 device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(string, read-only)*

  Battery current state. Available values are: *offline, standby, running, fault, sleep_mode*

- `energy_charged_total` *(integer, read-only)*

  Amount of energy charged to the battery over a lifetime. Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_charged_today` *(integer, read-only)*

  Amount of energy charged to the battery today. Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_discharged_total` *(integer, read-only)*

  Amount of energy consumed from the battery over a lifetime. Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_discharged_today` *(integer, read-only)*

  Amount of energy consumed from the battery today. Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `charge_power` *(integer, read-only)*

  Current charing (positive number) or discharging (negative number) power. Unit: mW

- `maximum_charging_power` *(integer)*

  Maximum charging power. Unit: W with three decimal number, multiplied by 1000 (mW).

- `maximum_discharging_power` *(integer)*

  Maximum charging power. Unit: W with three decimal number, multiplied by 1000 (mW).

- `charging_cutoff_capacity` *(integer)*

  Charging cutoff capacity. Unit: % with one decimal number, multiplied by 10.

- `discharge_cutoff_capacity` *(integer)*

  Discharge cutoff capacity. Unit: % with one decimal number, multiplied by 10.

Modbus - Huawei SUN2000 - Energy Meter

Representation of Energy Meter related parameters of Huawei SUN2000 device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `pv_total_active_power` *(integer, read-only)*

  Current total power produced by all photovoltaic panels. Unit: mW

- `energy_produced_total` *(integer, read-only)*

  Total amount of energy produced by PV over a lifetime. Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `energy_produced_today` *(integer, read-only)*

  Amount of energy produced by PV today. Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `power_to_grid` *(integer, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid. Unit: mW

- `phase_1.voltage` *(integer, read-only)*

  First phase voltage. Unit: mV

- `phase_1.current` *(integer, read-only)*

  First phase current. Unit: mA

- `phase_2.voltage` *(integer, read-only)*

  Second phase voltage. Unit: mV

- `phase_2.current` *(integer, read-only)*

  Second phase current. Unit: mA

- `phase_3.voltage` *(integer, read-only)*

  Third phase voltage. Unit: mV

- `phase_3.current` *(integer, read-only)*

  Third phase current. Unit: mA

Modbus - Huawei SUN2000 - Inverter

Representation of Inverter related parameters of Huawei SUN2000 device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online*, *offline*, *unknown*, *service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date*, *update_available*, *recovery*, *pending*, *downloading*, *updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `run_mode` *(string, read-only)*

  Inverter current run mode. Available values are: *standby*, *starting*, *on_grid*, *grid_power_limited*, *grid_self_derating*, *shutdown_fault*, *shutdown_command*, *grid_scheduling*, *spot_check_ready*, *spot_checking*, *inspecting*, *afci_self_check*, *iv_scanning*, *dc_input_detection*, *running*

- `pv_1.active_power` *(integer, read-only)*

  Current power produced by first group of photovoltaic panels. Unit: mW

- `pv_1.voltage` *(integer, read-only)*

  Current voltage on first group of photovoltaic panels. Unit: mV

- `pv_1.current` *(integer, read-only)*

  Current current on first group of photovoltaic panels. Unit: mA

- `pv_2.active_power` *(integer, read-only)*

  Current power produced by second group of photovoltaic panels. Unit: mW

- `pv_2.voltage` *(integer, read-only)*

  Current voltage on second group of photovoltaic panels. Unit: mV

- `pv_2.current` *(integer, read-only)*

  Current current on second group of photovoltaic panels. Unit: mA

- `pv_3.active_power` *(integer, read-only)*

  Current power produced by third group of photovoltaic panels. Unit: mW

- `pv_3.voltage` *(integer, read-only)*

  Current voltage on third group of photovoltaic panels. Unit: mV

- `pv_3.current` *(integer, read-only)*

  Current current on third group of photovoltaic panels. Unit: mA

- `pv_4.active_power` *(integer, read-only)*

  Current power produced by fourth group of photovoltaic panels. Unit: mW

- `pv_4.voltage` *(integer, read-only)*

  Current voltage on fourth group of photovoltaic panels. Unit: mV

- `pv_4.current` *(integer, read-only)*

  Current current on fourth group of photovoltaic panels. Unit: mA

Commands

- `turn_on`

  Turns on inverter.

- `turn_off`

  Turns off inverter.

Modbus - Itho - Heat Pump

Representation of Heat Pump related parameters of Itho device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(string)*

  State of the heat pump: *on, off*

- `fixed_heating_target_temperature` *(number)*

  Set temperature for heating in fixed temperature mode.

  Unit: °C.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_indoor` *(number, read-only)*

  Indoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_indoor` *(number, read-only)*

  Set indoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: Bar with one decimal number, multiplied by 10.

- `hot_gas_temperature` *(number, read-only)*

  Hot gas temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `brine_out_temperature` *(number, read-only)*

  Brine out temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `brine_in_temperature` *(number, read-only)*

  Brine in temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `energy_used_for_hot_water` *(number, read-only)*

  Energy used for hot water.

  Unit: kW/h.

- `energy_used_for_heating` *(number, read-only)*

  Energy used for heating.

  Unit: kW/h.

- `energy_used_for_cooling` *(number, read-only)*

  Energy used for cooling.

  Unit: kW/h.

- `energy_used_in_stand_by` *(number, read-only)*

  Energy used in stand-by.

  Unit: kW/h.

- `energy_used_total` *(number, read-only)*

  Energy used total.

  Unit: kW/h.

- `source_supply_energy` *(number, read-only)*

  Energy in source supply.

  Unit: MWh with two decimal numbers, multiplied by 100.

- `source_return_energy` *(number, read-only)*

  Energy in source return.

  Unit: MWh with two decimal numbers, multiplied by 100.

- `electric_heater_active` *(boolean)*

  Indicates electric heater active state.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `operating_hours_heating` *(number, read-only)*

  Operating hours for central heating.

- `operating_hours_hot_water` *(number, read-only)*

  Operating hours for domestic hot water.

- `number_of_starts` *(number, read-only)*

  Number of compressor starts.

- `heat_curve_end_point` *(number, read-only)*

  Heat curve end point.

  Unit: °C with one decimal number, multiplied by 10.

- `heat_curve_base_point` *(number, read-only)*

  Heat curve base point.

  Unit: °C with one decimal number, multiplied by 10.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Modbus - Itho - Main DHW

Representation of DHW related parameters of Itho device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `hysteresis` *(number)*

  Damper factor, which will protect from continous on/off switching when current temperature is near target value.

  Unit: °C with one decimal number, multiplied by 10.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water_lower_tank` *(number, read-only)*

  Current water temperature of lower tank sensor.

  Unit: °C with one decimal number, multiplied by 10.

Modbus - Itho - Temperature Sensor

Representation of Temperature sensor related parameters of Itho device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `temperature` *(number, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

Modbus - Kaisai KHC - Heat Pump

Representation of Heat Pump related parameters of Kaisai KHC device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `work_mode` *(string)*

  Current work mode of the heat pump: *automatic, cooling, heating*.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `fixed_target_temperature` *(number)*

  Set temperature for heating or cooling in fixed temperature mode.

  Unit: °C.

- `fixed_target_temperature_minimum` *(number, read-only)*

  Minimum value of fixed_target_temperature parameter.

  Unit: °C.

- `fixed_target_temperature_maximum` *(number, read-only)*

  Maximum value of fixed_target_temperature parameter.

  Unit: °C.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: Bar with one decimal number, multiplied by 10.

- `hot_gas_temperature` *(number, read-only)*

  Hot gas temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `water_inlet_temperature` *(number, read-only)*

  Water inlet temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `water_outlet_temperature` *(number, read-only)*

  Water outlet temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `electric_heater_active` *(boolean)*

  Indicates electric heater active state.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Modbus - Kaisai KHC - Main DHW

Representation of DHW related parameters of Kaisai KHC device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(integer)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C. **NOTE:** Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(integer, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `electric_heater_active` *(boolean)*

  Indicates electric heater active state.

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```
if dateTime:changed() then
  local heat_pump = modbus[7]
  local dhw = modbus[8]
  if heat_pump:getValue("work_mode") == "cooling" then
    dhw:setValue("target_temperature", 45)
  else
    dhw:setValue("target_temperature", 55)
  end
end
```

Modbus - Kaisai KHC - Temperature Sensor

Representation of Temperature sensor related parameters of Kaisai KHC device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `temperature` *(number, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

Modbus - Mitsubishi Ecodan - Heat Pump

Representation of Heat Pump related parameters of Mitsubishi Ecodan device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(string)*

  State of the heat pump: *on, off*

- `temperature_outdoor` *(integer, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_supply` *(integer, read-only)*

  Heating supply temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_return` *(integer, read-only)*

  Heating return temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `running_hours` *(integer, read-only)*

  Hours heat pump is working.

- `zone1.target_temperature` *(integer)*

  Target temperature at first zone.

  Unit: °C with one decimal number, multiplied by 10.

- `zone1.current_temperature` *(integer, read-only)*

  Current temperature at first zone.

  Unit: °C with one decimal number, multiplied by 10.

- `zone1.work_mode` *(string)*

  Work mode at first zone: *heating_room_temp, heating_flow_temp, heating_heat_curve, cooling_flow_temp*

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `zone2.target_temperature` *(integer)*

  Target temperature at second zone.

Unit: °C with one decimal number, multiplied by 10.

- `zone2.current_temperature` *(integer, read-only)*

  Current temperature at second zone.

  Unit: °C with one decimal number, multiplied by 10.

- `zone2.work_mode` *(string)*

  Work mode at second zone: *heating_room_temp, heating_flow_temp, heating_heat_curve, cooling_flow_temp*

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `heat_demand` *(boolean, read-only)*

  Informs that heating is demanded.

- `cool_demand` *(boolean, read-only)*

  Informs that cooling is demanded.

Modbus - Mitsubishi Ecodan - Main DHW

Representation of DHW related parameters of Mitsubishi Ecodan device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_domestic_hot_water` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `dhw_demand` *(boolean, read-only)*

  Domestic Hot Water demand.

Modbus - Remeha Elga ACE - Heat Pump

Representation of Heat Pump related parameters of Remeha Elga ACE device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `work_mode` *(string)*

  Current work mode of the heat pump: *cooling, heating*

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `temperature_indoor` *(number, read-only)*

  Indoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_indoor` *(number, read-only)*

  Set indoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `central_heating_target_temperature` *(number, read-only)*

  Central heating set temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `fixed_heating_target_temperature` *(number)*

  Set temperature for heating in fixed temperature mode.

  Unit: °C with one decimal number, multiplied by 10.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `heating_system_pressure` *(number, read-only)*

  Heating system pressure.

  Unit: Bar with one decimal number, multiplied by 10.

- `energy_used_for_heating` *(number, read-only)*

  Energy used for heating.

  Unit: kW/h with one decimal number, multiplied by 10.

- `current_power` *(number, read-only)*

  Current relative power produced.

  Unit: kW with one decimal number, multiplied by 10.

- `alarm_code` *(number, read-only)*

  Device flow alarm code.

- `alarm_description` *(number, read-only)*

  Alarm code description id.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `operating_hours_heating` *(number, read-only)*

  Operating hours for central heating.

- `heat_demand` *(boolean)*

  Informs device that heat is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

- `cool_demand` *(boolean)*

  Informs device that cool is demanded or not.

  NOTE: Cannot be modified when device is associated with Heat Pump Manager.

Commands

- `reset_alarms`

  Sends request to heat pump device to reset alarms.

Modbus - Remeha Elga ACE - Temperature Sensor

Representation of Temperature sensor related parameters of Remeha Elga ACE device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `temperature` *(number, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

Modbus - SolarEdge with MTTP Extension Model - Inverter

Representation of Inverter related parameters of SolarEdge device with MTTP Extension Model.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `run_mode` *(string, read-only)*

  Inverter current run mode. Available values are: *off, sleeping, starting, working, throttled, shutting_down, fault, standby*

- `energy_produced_total` *(number, read-only)*

  Total amount of energy produced by PV over a lifetime.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: mW

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: mW

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: mA
- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: mW
- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: mV
- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: mA
- `pv_3.active_power` *(number, read-only)*

  Current power produced by third group of photovoltaic panels.

  Unit: mW
- `pv_3.voltage` *(number, read-only)*

  Current voltage on third group of photovoltaic panels.

  Unit: mV
- `pv_3.current` *(number, read-only)*

  Current current on third group of photovoltaic panels.

  Unit: mA
- `advanced_power_control_enabled` *(boolean)*

  Allows to set advanced power control settings.
- `reactive_power_config` *(string)*

  Reactive power config. Available values are: *fixed_cosphi, fixed_q, cosphi, q, rrcr*.
- `active_power_limit` *(number)*

  Percent of max power at which inverter is going to work. Unit: percent Requires `advanced_power_control_enabled` to be set to `true` and `reactive_power_config` to `rrcr`.

Modbus - SolarEdge - Inverter

Representation of Inverter related parameters of SolarEdge device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server. Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: `up_to_date`, `update_available`, `recovery`, `pending`, `downloading`, `updating`

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `run_mode` *(string, read-only)*

  Inverter current run mode. Available values are: *off, sleeping, starting, working, throttled, shutting_down, fault, standby*

- `energy_produced_total` *(number, read-only)*

  Total amount of energy produced by PV over a lifetime. Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today. Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid. Unit: mW

- `pv.active_power` *(number, read-only)*

  Current power produced by photovoltaic panels. Unit: mW

- `pv.voltage` *(number, read-only)*

  Current voltage on photovoltaic panels. Unit: mV

- `pv.current` *(number, read-only)*

  Current current on photovoltaic panels. Unit: mA

- `advanced_power_control_enabled` *(boolean)*

  Allows to set advanced power control settings.

- `reactive_power_config` *(string)*

  Reactive power config. Available values are: *fixed_cosphi, fixed_q, cosphi, q, rrcr*.

- `active_power_limit` *(number)*

  Percent of max power at which inverter is going to work. Unit: percent Requires `advanced_power_control_enabled` to be set to `true` and `reactive_power_config` to `rrcr`.

Modbus - Solax X1 - Battery

Representation of Battery related parameters of Solax X1 device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `soc` *(number, read-only)*

  Current state of charge.

  Unit: %

- `energy_charged_total` *(number, read-only)*

  Amount of energy charged to the battery over a lifetime.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the battery today.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_discharged_total` *(number, read-only)*

  Amount of energy consumed from the battery over a lifetime.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_discharged_today` *(number, read-only)*

  Amount of energy consumed from the battery today.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `charge_power` *(number, read-only)*

  Current charing (positive number) or discharging (negative number) power. Unit: mW

Commands

- `charge`

  Calls battery to charge during given period of time.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*

    - minumum: 0
    - maximum: 2147483647
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 65535

- `discharge`

  Calls battery to discharge during given period of time.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 0
    - maximum: 2147483647
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 65535

Examples

Turn on battery charging with 1kW for 1 hour at 1:00PM

```
if dateTime:changed() then
  if dateTime:getHours() == 13 and dateTime:getMinutes() == 0 then
    modbus[2]:call("charge", { 1000000, 3600 })
  end
end
```

Modbus - Solax X1 - Inverter

Representation of Inverter related parameters of Solax X1 device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `run_mode` *(string, read-only)*

  Inverter current run mode. Available values are: *waiting, checking, normal, fault, permanent_fault, update, off_grid_waiting, off_grid, self_testing, idle, standby*

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: mW

- `energy_produced_total` *(number, read-only)*

  Total amount of energy produced by PV over a lifetime.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `energy_consumed_total` *(number, read-only)*

  Amount of energy consumed from the power grid over a lifetime.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `energy_consumed_today` *(number, read-only)*

  Amount of energy consumed from the power grid over today.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: mV

- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: mA

- `grid.active_power` *(number, read-only)*

  Current power on the power grid.

  Unit: mW

- `grid.voltage` *(number, read-only)*

  Current voltage on the power grid.

  Unit: mV

- `grid.current` *(number, read-only)*

  Current current on the power grid.

  Unit: mA

Modbus - Solax X3 - Battery

Representation of Battery related parameters of Solax X3 device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `soc` *(number, read-only)*

  Current state of charge.

  Unit: %

- `energy_charged_total` *(number, read-only)*

  Amount of energy charged to the battery over a lifetime.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the battery today.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_discharged_total` *(number, read-only)*

  Amount of energy consumed from the battery over a lifetime.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `energy_discharged_today` *(number, read-only)*

  Amount of energy consumed from the battery today.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh)

- `charge_power` *(number, read-only)*

  Current charing (positive number) or discharging (negative number) power. Unit: mW

Commands

- `charge`

  Calls battery to charge during given period of time.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*

    - minumum: 0
    - maximum: 2147483647
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 65535

- `discharge`

  Calls battery to discharge during given period of time.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 0
    - maximum: 2147483647
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 65535

Examples

Turn on battery charging with 1kW for 1 hour at 1:00PM

```
if dateTime:changed() then
  if dateTime:getHours() == 13 and dateTime:getMinutes() == 0 then
    modbus[2]:call("charge", { 1000000, 3600 })
  end
end
```

Modbus - Solax X3 - Inverter

Representation of Inverter related parameters of Solax X3 device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `run_mode` *(string, read-only)*

  Inverter current run mode. Available values are: *waiting, checking, normal, fault, permanent_fault, update, off_grid_waiting, off_grid, self_testing, idle, standby*

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: mW

- `energy_produced_total` *(number, read-only)*

  Total amount of energy produced by PV over a lifetime.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today.

  Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `energy_consumed_total` *(number, read-only)*

  Amount of energy consumed from the power grid over a lifetime.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `energy_consumed_today` *(number, read-only)*

  Amount of energy consumed from the power grid over today.

  Unit: kWh with two decimal numbers, multiplied by 1000 (Wh)

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: mV

- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: mA

- `grid_total_active_power` *(number, read-only)*

  Current total power on the grid.

  Unit: mW

- `phase_1.active_power` *(number, read-only)*

  Current power on first phase of power grid.

  Unit: mW

- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid.

  Unit: mV

- `phase_1.current` *(number, read-only)*

  Current current on first phase of power grid.

  Unit: mA

- `phase_2.active_power` *(number, read-only)*

  Current power on second phase of power grid.

Unit: mW

- `phase_2.voltage` *(number, read-only)*

  Current voltage on second phase of power grid.

  Unit: mV

- `phase_2.current` *(number, read-only)*

  Current current on second phase of power grid.

  Unit: mA

- `phase_3.active_power` *(number, read-only)*

  Current power on third phase of power grid.

  Unit: mW

- `phase_3.voltage` *(number, read-only)*

  Current voltage on third phase of power grid.

  Unit: mV

- `phase_3.current` *(number, read-only)*

  Current current on third phase of power grid.

  Unit: mA

Modbus - Solis - Inverter

Representation of Inverter related parameters of Solis device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online*, *offline*, *unknown*, *service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date*, *update_available*, *recovery*, *pending*, *downloading*, *updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `run_mode` *(string, read-only)*

  Inverter current run mode. Available values are: *unknown*, *normal*, *initializing*, *grid_off*, *fault_to_skip*, *standby*, *derating*, *limitating*, *backup_ov_load*, *grid_surge*, *fan_fault*

- `pv_total_active_power` *(integer, read-only)*

  Current total power produced by all photovoltaic panels. Unit: mW

- `grid_total_active_power` *(integer, read-only)*

  Current total power on the grid. Unit: mW

- `energy_produced_total` *(integer, read-only)*

  Total amount of energy produced by PV over a lifetime. Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `energy_produced_today` *(integer, read-only)*

  Amount of energy produced by PV today. Unit: kWh with one decimal number, multiplied by 1000 (Wh).

- `power_to_grid` *(integer, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid. Unit: mW

- `pv_1.active_power` *(integer, read-only)*

  Current power produced by first group of photovoltaic panels. Unit: mW

- `pv_1.voltage` *(integer, read-only)*

  Current voltage on first group of photovoltaic panels. Unit: mV

- `pv_1.current` *(integer, read-only)*

  Current current on first group of photovoltaic panels. Unit: mA

- `pv_2.active_power` *(integer, read-only)*

  Current power produced by second group of photovoltaic panels. Unit: mW

- `pv_2.voltage` *(integer, read-only)*

  Current voltage on second group of photovoltaic panels. Unit: mV
- `pv_2.current` *(integer, read-only)*

  Current current on second group of photovoltaic panels. Unit: mA
- `pv_3.active_power` *(integer, read-only)*

  Current power produced by third group of photovoltaic panels. Unit: mW
- `pv_3.voltage` *(integer, read-only)*

  Current voltage on third group of photovoltaic panels. Unit: mV
- `pv_3.current` *(integer, read-only)*

  Current current on third group of photovoltaic panels. Unit: mA
- `pv_4.active_power` *(integer, read-only)*

  Current power produced by fourth group of photovoltaic panels. Unit: mW
- `pv_4.voltage` *(integer, read-only)*

  Current voltage on fourth group of photovoltaic panels. Unit: mV
- `pv_4.current` *(integer, read-only)*

  Current current on fourth group of photovoltaic panels. Unit: mA
- `phase_1.voltage` *(integer, read-only)*

  Current voltage on first phase of power grid. Unit: mV
- `phase_1.current` *(integer, read-only)*

  Current current on first phase of power grid. Unit: mA
- `phase_2.voltage` *(integer, read-only)*

  Current voltage on second phase of power grid. Unit: mV
- `phase_2.current` *(integer, read-only)*

  Current current on second phase of power grid. Unit: mA
- `phase_3.voltage` *(integer, read-only)*

  Current voltage on third phase of power grid. Unit: mV
- `phase_3.current` *(integer, read-only)*

  Current current on third phase of power grid. Unit: mA

Modbus - P1 Energy Meter

Representation of P1 Energy Meter device.

Device may be added using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `modbus` container eg. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `tariff_1.energy_consumed_total` *(integer, read-only)*

  Total energy delivered to client in tariff 1. Unit: kWh with three decimal numbers, multiplied by 1000 (Wh).

- `tariff_2.energy_consumed_total` *(integer, read-only)*

  Total energy delivered to client in tariff 2. Unit: kWh with three decimal numbers, multiplied by 1000 (Wh).

- `tariff_1.energy_fed_total` *(integer, read-only)*

  Total energy delivered by client in tariff 1. Unit: kWh with three decimal numbers, multiplied by 1000 (Wh).

- `tariff_2.energy_fed_total` *(integer, read-only)*

  Total energy delivered by client in tariff 2. Unit: kWh with three decimal numbers, multiplied by 1000 (Wh).

- `tariff_indicator` *(integer, read-only)*

  Electricity tariff indicator.

- `power_to_grid` *(integer, read-only)*

  Current power fed to the power grid. Unit: mW

- `power_from_grid` *(integer, read-only)*

  Current power consumed from the power grid. Unit: mW

- `number_of_power_failures` *(integer, read-only)*

  Number of power failures.

- `number_of_long_power_failures` *(integer, read-only)*

  Number of long power failures.

- `phase_1.voltage` *(integer, read-only)*

  First phase voltage. Unit: mV

- `phase_1.current` *(integer, read-only)*

  First phase current. Unit: mA

- `phase_1.active_power` *(integer, read-only)*

  First phase active power. Unit: mW
- `phase_1.number_of_voltage_sags` *(integer, read-only)*

  First phase total number of voltage sags.
- `phase_1.number_of_voltage_swells` *(integer, read-only)*

  First phase total number of voltage swells.
- `phase_2.voltage` *(integer, read-only)*

  Second phase voltage. Unit: mV
- `phase_2.current` *(integer, read-only)*

  Second phase current. Unit: mA
- `phase_2.active_power` *(integer, read-only)*

  Second phase active power. Unit: mW
- `phase_2.number_of_voltage_sags` *(integer, read-only)*

  Second phase total number of voltage sags.
- `phase_2.number_of_voltage_swells` *(integer, read-only)*

  Second phase total number of voltage swells.
- `phase_3.voltage` *(integer, read-only)*

  Third phase voltage. Unit: mV
- `phase_3.current` *(integer, read-only)*

  Third phase current. Unit: mA
- `phase_3.active_power` *(integer, read-only)*

  Third phase active power. Unit: mW
- `phase_3.number_of_voltage_sags` *(integer, read-only)*

  Third phase total number of voltage sags.
- `phase_3.number_of_voltage_swells` *(integer, read-only)*

  Third phase total number of voltage swells.
- `p1_version_id` *(integer, read-only)*

  P1 version information.
- `software_version` *(string, read-only)*

  P1 converter software version.

Virtual - Thermostat

The virtual thermostat controls the output devices based on the readings from the sensors.

Thermostat has three working modes: `schedule`, `time_limited` and `constant`. By default thermostat works in `constant` mode.

- In `constant` mode thermostat has one target temperature which is used for algorithm.
- In `time_limited` mode thermostat has one target temperature which is used for algorithm until `target_temperature_mode.remaining_time` reaches 0. It will switch back to previous target temperature mode afterwards.
- In `schedule` mode there are many target temperatures in time. User can set several time ranges during the day in which target temperature applies. `Fallback` temperature applies otherwise.

  User can set different working schedule for every week day.

  For example: If user setted schedule to be 6:00 - 20:00, temperature applies between 6:00 - 20:00. Fallback temperature applies between 0:00 - 5:59 and 20:01 - 23:59.

The thermostat can be associated with: Room sensor (temperature sensor), floor sensor (temperature sensor), humidity sensor, temperature regulator, radiator actuator, relay, window/door opening sensor, two state input sensor.

The room sensor is a required device.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(boolean, read-only)*

  Current working state (*active (true) / idle (false)*).

- `temperature` *(integer, read-only)*

  Temperature value forwarded from associated sensor or 0 if not associated. Unit: °C with one decimal number, multiplied by 10.

- `floor_temperature` *(integer, read-only)*

  Floor Temperature value forwarded from associated sensor or 0 if not associated.

  Unit: °C with one decimal number, multiplied by 10.

- `humidity` *(integer, read-only)*

  Humidity value forwarded from associated sensor or 0 if not associated. Unit: rH% with one decimal number, multiplied by 10.

- `target_temperature` *(integer)*

  Current target temperature. Modification will result in change of `constant` or `time_limited` target temperature to the desired value. If thermostat works in `time_limited` mode it will change target temperature only, not affecting `remaining_time`. If thermostat works in `schedule` mode it will change target temperature mode to `constant`.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_mode.current` *(string, read-only)*

  Thermostat target temperature mode. Specifies if thermostat is working in `constant` mode with one target temperature, `time_limited` mode with one temporary target temperature or according to schedule in `schedule` mode with many target temperatures in time, configured by user. Parameter is read only, use commands to change target temperature mode! Parameter cannot be `schedule` if thermostat doesnt have `has_schedule` label! Available values: *constant, schedule, time_limited*. Default: *constant*

- `target_temperature_mode.previous` *(string, read-only)*

  Thermostat previous target temperature mode.

  Available values: *constant, schedule, time_limited*. Default: *constant*

- `target_temperature_mode.remaining_time` *(integer, read-only)*

  Remaining time until `time_limited` mode ends. Cannot be modified directly - use commands.

  Unit: minutes.

- `target_temperature_miniumum` *(integer)*

  Lower limit of the target temperature. Could not be greater than maximum. Setting minimum value above target value, will also change all target values to minimum.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_maximum` *(integer)*

  Upper limit of the target temperature. Could not be less than minimum. Setting maximum below target, will also change target values to maximum. Unit: °C with one decimal number, multiplied by 10.

- `hysteresis` *(integer)*

  Damper factor, which will protect from continous on/off switching when current temperature is near target value.

  Unit: °C with one decimal number, multiplied by 10.

- `mode` *(string)*

  Current working mode. Available values: *heating, cooling, off*

- `overheat_protection.active` *(boolean, read-only)*

  State of algorithm that disables heating if temperature is higher than target.

- `overheat_protection.enabled` *(boolean)*

  Enables or disables overheat protection algorithm.

- `overheat_protection.range` *(integer)*

  A value above target temperature that triggers overheat protection.

  Unit: °C with one decimal number, multiplied by 10.

- `sigma_control.enabled` *(boolean)*

  Sigma smooth control state. If disabled, opening value of actuator will jump between min/max instead of smooth control.

- `sigma_control.range` *(integer)*

  Temperature range below target temperature that is used to scale opening from 100 (or maximum opening) - 0 (or minimum opening) percent when current room temperature is equal to target temperature. Sigma causes actuators to open/close in small, smooth steps instead of full open/full close.

  Unit: °C with one decimal number, multiplied by 10.

- `sigma_control.opening_factor` *(integer, read-only)*

  Current calculated valve opening factor in percent used to scale desired opening between min and max.

- `floor_control.enabled` *(boolean)*

  State of algorithm that controls floor heating processes.

- `floor_control.lower_target_temperature` *(integer)*

  Lower limit of floor temperature fluctuation (due to material inertia). Could not be greater than upper value.

  Unit: °C with one decimal number, multiplied by 10.

- `floor_control.upper_target_temperature` *(integer)*

  Upper limit of floor temperature fluctuation (due to material inertia). Could not be less than lower value.

  Unit: °C with one decimal number, multiplied by 10.

- `floor_control.hysteresis` *(integer)*

  Damper factor, which will protect from continous on/off switching when current temperature is near target value.

  Unit: °C with one decimal number, multiplied by 10.

- `floor_control.condition` *(string, read-only)*

  Floor control condition. Informs whether floor temperature is in min-max range or not. Available values: *optimal, overheated, underheated*

- `antifrost_protection` *(boolean)*

  State of algorithm that turns on heating if temperature drops under 6°C.

- `opening_sensors_delay` *(integer)*

  Delay after which thermsotat will react when opening sensor detects window openned.

  Unit: seconds

- `is_window_open` *(boolean, read-only))*

  Informs whether there is window opened.

- `confirm_time_mode` *(boolean)*

  Mainly for Mobile/Web App purposes. Indicates if time mode modal should be displayed when changing thermostat temperature.

Commands

- `set_target_temperature`

  Calls Thermostat to change `constant` or `time_limited` mode target temperature to the desired value. If thermostat works in `time_limited` mode it will change target temperature only, not affecting `remaining_time`. If thermostat works in `schedule` mode it will change target temperature mode to `constant`.

  Argument:

  target temperature in 0.1°C *(number)*

- `enable_schedule_mode`

  Changes thermostat target temperature mode to `schedule`. In this mode target temperature is set based on schedule set by user. Command cannot be called to if thermostat doesnt have `has_schedule` label!

- `enable_constant_mode`

  Calls Thermostat to change mode and target temperature mode to `constant`. While thermostat is already in `constant` mode, it will change mode `target_temperature` only.

  Unit: °C with one decimal number, multiplied by 10.

  Argument:

  target temperature in 0.1°C *(number)*

- `enable_time_limited_mode`

  Calls Thermostat to change mode and target temperature mode to `time_limited` for desired time. While thermostat is already in `time_limited` mode, it will change `remaining_time` or/and `target_temperature` depending on payload. First parameter is `remaining_time`, second is `target_temperature`.

  Unit: minutes and °C with one decimal number, multiplied by 10.

  Argument:

  packed arguments *(table)*:

  - remaining time in minutes *(number)*
  - target temperature in 0.1°C *(number)*

- `disable_time_limited_mode`

  Calls Thermostat to disable `time_limited` and go back to previous target temperature mode. When thermostat is not in `time_limited` mode, it will do

nothing.

- `set_mode`

  Calls Thermostat to change mode to one of `heating`, `cooling`, `off`.

  Argument:

  mode name *(string)*

Examples

Raise target temperature between 15:00 and 20:00

```
if dateTime:changed() then
  if dateTime:getHours() == 15 and dateTime:getMinutes() == 0 then
    virtual[1]:call("set_target_temperature", 220)
  elseif dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
    virtual[1]:call("set_target_temperature", 190)
  end
end
```

Raise target temperature on saturday and lower on monday

```
if dateTime:getTimeOfDay() == 00 then
  if dateTime:getWeekDayString() == "saturday" then
    virtual[1]:call("set_target_temperature", 230)
  elseif dateTime:getWeekDayString() == "monday" then
    virtual[1]:call("set_target_temperature", 210)
  end
end
```

Enable schedule work monday to friday and disable during weekends

```
if dateTime:getTimeOfDay() == 00 then
  if dateTime:getWeekDayString() == "monday" then
    virtual[1]:call("enable_schedule_mode")
  elseif dateTime:getWeekDayString() == "saturday" then
    virtual[1]:call("enable_constant_mode", 200)
  end
end
```

Reconfigure thermostat when motion sensor triggers

```
if wtp[4]:changedValue("motion_detected") then
  -- time limited to 2 hours, 23.5°C
  virtual[1]:call("enable_time_limited_mode", {120, 235})
end
```

Change thermostat modes based on temperature

```
local sensor_temperature = wtp[3]:getValue("temperature")

if sensor_temperature > 250 then
  -- above 25°C, enable cooling
  virtual[1]:call("set_mode", "cooling")
elseif sensor_temperature < 200 then
  -- below 20°C, enable heating
  virtual[1]:call("set_mode", "heating")
end
```

Virtual - Thermostat Output Group

The virtual thermostat output group controls the output devices based on the readings from the virtual thermostats eg. turning on and off gas boiler, pumps or valves via associated relay or allowing heating by pellet boiler.

Currently only heating is supported.

The thermostat output group can be associated with: Thermostats (input devices), relays (output for gas boiler, pump or valve), pellet boiler, heat pumps or two state input sensor (switches Thermostats heating/cooling mode).

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `enabled` *(boolean)*

  Current device state (*enabled (true) / disabled (false)*).

- `state` *(boolean, read-only)*

  Current working state (*active (true) / idle (false)*).

- `mode` *(string, read-only)*

  Current calculated working mode. Available values: *heating, cooling*

- `propagation_delay` *(number)*

  Delays active state propagation for output devices eg. delays switching from off to on for relays/pellet boiler when heat is requested. Useful for setup with gas boiler (quick achieve of heating setpoint) and radiator actuators (long response, up to several minutes) where boiler can go into alarm status when there is no heat extraction. Unit: seconds.

Commands

- `enable`

  Enables device.

- `disable`

  Disables device.

- `set_propagation_delay` *(number)*

  Sets desired propagation delay for outputs.

  Argument:

  delay in seconds *(number)*

Examples

Check when heat/cooling is requested

```
if virtual[55]:changedValue("state") then
  if virtual[55]:getValue("mode") == "heating" then
    if virtual[55]:getValue("state") then
      print("HEAT IS REQUESTED")
    else
      print("COOLING IS REQUESTED")
    end
  else
    print("ACTION NO LONGER NEEDED")
  end
end
```

Disable between 15:00 and 20:00

```
if dateTime:changed() then
  if dateTime:getHours() == 15 and dateTime:getMinutes() == 0 then
    virtual[1]:call("enable")
  elseif dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
    virtual[1]:call("disable")
  end
end
```

Virtual - Relay Integrator

The virtual relay intergrator keeps all associated relays in the same state. If one of assigned relays changes state, integrator changes state of all associated relays to new state.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(boolean)*

  Current relays output state.

Examples

Turn on all assigned relays when motion sensor triggers

```
if wtp[3]:changedValue("motion_detected") then
  virtual[5]:setValue("state", true)
end
```

Virtual - Blind Controller Integrator

The virtual blind controller intergrator allows setting the same target opening to all associated blind controllers. Control logic is one-way - If one of assigned blind controllers changes target opening, integrator will not affect target opening of the rest associated blind controllers.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `last_set_target_opening` *(number, read-only)*

  Contains last set target opening via integrator.

- `action_in_progress` *(boolean, read-only)*

  Indicates if control action requested via integrator is in progress.

Examples

Open all assigned blinds at sunrise and close at sunset

```
if event.type == "sunrise" then
  virtual[3]:call("up")
elseif event.type == "sunset" then
  virtual[3]:call("down")
end
```

Set all assigned blinds to half-open at noon

```
if dateTime:changed() then
  if dateTime:getHours() == 12 and dateTime:getMinutes() == 0 then
    virtual[3]:call("open", 50)
  end
end
```

Catch actions starting and ending

```
if virtual[3]:changedValue("action_in_progress") then
  if virtual[3]:getValue("action_in_progress") then
    print("Somebody started action via integrator")
  else
    print("Integration action has eneded.")
  end
end
```

Virtual - CustomDevice

Custom Device is a special type of device in which the user can design the layout of the controls on the widget and in the options window, and then program their behavior in Lua language. This functionality allows you to easily expand the system with further integrations and functionalities. This requires knowledge of the Sinum Lua development environment.

Custom Device lua code is a private extension (not available outside of the device execution context) of the standard devices functionality. This means that in the context of device lua code / execution context, you can use standard methods like `getValue`, `setValue`, `setValueAfter` etc. referring to the `self` object e.g. `self:getValue("name")`, `self:setValue("name", "new_name")`. See examples for more info.

From the automation/scene context, the device is visible like the rest. The difference is an additional method, `getElement` which allows you to refer to a specific control by its name and get or set properties.

The names control and element are used interchangeably and represent the predefined parts from which the user builds his device. Read `Virtual - CustomDevice - Controls` chapter for more information.

Device and controls may be added, edited or removed via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts (excluding adding/removing controls and changing positions) using `virtual` container eg.

`virtual[6]` gives you access to virtual device with **ID 6**. VIRTUAL devices have global scope and they are visible in all executions contexts.

Methods

This is an extension of the methods available in standard devices.

- `getElement(element_name)`

  Returns reference to control or nil if it doesnt exist.

  Returns:

  - *(any)* - depends on element type

  Arguments:

  - `element_name` *(string)* - name of element configured by user

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `enabled` *(boolean)*

  Current device state (*enabled (true) / disabled (false)*). When disabled, lua code wont be executed.

Commands

- `enable`

  Enables device.

- `disable`

  Disables device.

User specific commands

There is possibility to add user commands to custom device, which can be called from REST api or other lua scripts. All you need is to define callback handler for commands and write its logic in Custom Device Code eg.

```
function CustomDevice:onCommand(command, arg)
  if command == "my_command_1" then
    print("You called first command without arg.")
  else if command == "my_command_2_with_arg" then
    print("You called second command with arg: " .. tostring(arg))
  end
end
```

and use it in other scripts eg:

```
virtual[5]:call("my_command_1")

virtual[5]:call("my_command_2_with_arg", 77)
```

Virtual - CustomDevice - Lua code

Custom device `lua` property (available only via REST api) is a place when you write entire code for your custom device including all controls callbacks. When some element refers to callback name it must exists in `lua` code.

Available callbacks

- `onEvent`

  Executed when any event in system occurs. Allows user to react to other parts of the system. Defining this callback in code is not neccessary if user doesn't want to catch events not related to custom device. (see examples section for more details)

  Arguments:

  - `event_object` *(object)* - Current system event. See Events section for more details about events

- `onCommand`

  Executed when command call requested via REST API (see device commands endpoints) or via other Lua scripts (see device commands `call` function description in Devices section.). Allows user to handle custom defined commands eg. to control multiple elements at once from automations or scenes. Defining this callback in code is not neccessary if user doesn't want to have commands. (see examples section for more details)

  Arguments:

  - `command` *(string)* - Name of command to handle
  - `arg` *(any)* - Argument passed by user to command. Type of arg can be number, string, boolean or nil (null)

Virtual - CustomDevice - Controls

Controls form the appearance and logical part of a custom device. They allow you to change parameters, display their values and react to actions such as clicking a button.

Each type of control has its own properties and the ability to attach a lua function that will be executed when an specific event occurs.

Currently available controls:

- `button` - Button with text and/or icon that may react to a click
- `progress_bar` - A bar with a percentage value from 0% to 100% that can be changed by the user (only from lua side)
- `slider` - A bar with a numerical value of any range and step, which can be changed by the user from the Lua context and the widget/option context
- `switcher` - Element for switching values between true / false
- `text` - Text field that displays the value entered from the Lua context on the widget / options
- `combo_box` - A user defined drop-down list, which can be change by user from the Lua context and the widget/option context.

Methods

The methods are common to all types of controls.

- `getValue(property_name)`

  Returns value of object property.

  Returns:

  - *(any)* - depends on property type

  Arguments:

  - `property_name` *(string, required)* - name of property

- `setValue(property_name, property_value, stop_propagation)`

  Sets value for object property.

  Returns:

  - *(userdata)* - reference to element object for chained calls

  Arguments:

  - `property_name` *(string, required)* - name of property
  - `property_value` *(any, required)* - property type dependant value which should be set
  - `stop_propagation` *(boolean, optional)* - defines whether futher callback propagation should be stopped (= `true`) or not (= `false` / empty). In other words, if = true, then associated callback (eg. `onChange`) won't be executed

after value change. This may help reducing callback propagation infinite loops - see explanation below.

- `call(command_name, arg)`

  Calls element to execute commmand.

  Returns:

  - *(userdata)* - reference to element object for chained calls

  Arguments:

  - `command_name` *(string, required)* - name of command available for element
  - `arg` *(any, optional)* - argument for command

Virtual - CustomDevice - Controls - Text

This element represents text field that displays the value entered from the Lua context on the widget / options. It is possible to attach a Lua callback which will be executed when text changes.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `type` *(string, read-only)*

  Element type description, based on role and functionality.

- `name` *(string, read-only)*

  User defined name of element. Cannot contain special characters except `_`

- `uuid` *(string, read-only)*

  Universal Unique Identifier of element used by frontend app to properly render and position element.

- `enabled` *(boolean)*

  Indicates if element is enabled/disabled. If element is disabled, user cannot change its properties or call commands (will result in validation error)

- `device_id` *(number, read-only)*

  ID of device from which the control comes from.

- `value` *(string)*

  User defined text value. Max 32 characters. Value change will trigger `on_change` callback.

- `font_weight` *(string)*

  Font weight of displayed text. Available values: `normal`, `bold`.

- `font_size` *(string)*

  Font size of displayed text. Available values: `small`, `normal`, `large`.

- `on_change` *(string, read-only)*

  Name of method (function) from CustomDevice lua code which will be executed on text value change. Should not contain `CustomDevice:` prefix, only name.

Commands

Commands cannot be executed if control is not `enabled` (will result in validation error).

- `set_value`

  Sets new text value. Value change will trigger `on_change` callback.

  Arguments:

  - `string`

- `set_font_weight`

  Sets new font weight. Available values: `normal`, `bold`.

  Arguments:

  - `string`

- `set_font_size`

  Sets new font weight. Available values: `small`, `normal`, `large`.

  Arguments:

  - `string`

Lua Callback signature

- `on_change`

  Executed on text value change. Takes new value and reference to element as arguments.

  Arguments:

  - `string`
  - `element_reference`

Virtual - CustomDevice - Controls - Button

This element represents button with text and/or icon that may react to a click. It is possible to attach a Lua callback which will be executed when press event happens.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `type` *(string, read-only)*

  Element type description, based on role and functionality.

- `name` *(string, read-only)*

  User defined name of element. Cannot contain special characters except `_`

- `uuid` *(string, read-only)*

  Universal Unique Identifier of element used by frontend app to properly render and position element.

- `enabled` *(boolean)*

  Indicates if element is enabled/disabled. If element is disabled, user cannot change its properties or call commands (will result in validation error)

- `device_id` *(number, read-only)*

  ID of device from which the control comes from.

- `text` *(string)*

  User defined text value. Max 32 characters.

- `icon` *(string)*

  User defined icon value. Max 64 characters.

- `on_press` *(string, read-only)*

  Name of method (function) from CustomDevice lua code which will be executed on press event. Should not contain `CustomDevice:` prefix, only name.

Commands

Commands cannot be executed if control is not `enabled` (will result in validation error).

- `press`

  Emits press event and executes callback if attached.

- `set_text`

  Sets new text value.

  Arguments:

  - `string`

- `set_icon`

  Sets new icon value.

  Arguments:

  - `string`

Lua Callback signature

- `on_press`

  Executed on press event. Takes reference to element as argument.

  Arguments:

  - `element_reference`

Virtual - CustomDevice - Controls - Switcher

This element represents switchable value between true / false. It is possible to attach a Lua callback which will be executed when value changes.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `type` *(string, read-only)*

  Element type description, based on role and functionality.

- `name` *(string, read-only)*

  User defined name of element. Cannot contain special characters except `_`

- `uuid` *(string, read-only)*

  Universal Unique Identifier of element used by frontend app to properly render and position element.

- `enabled` *(boolean)*

  Indicates if element is enabled/disabled. If element is disabled, user cannot change its properties or call commands (will result in validation error)

- `device_id` *(number, read-only)*

  ID of device from which the control comes from.

- `value` *(boolean)*

  User defined value. Value change will trigger `on_change` callback.

- `on_change` *(string, read-only)*

  Name of method (function) from CustomDevice lua code which will be executed on value change. Should not contain `CustomDevice:` prefix, only name.

Commands

Commands cannot be executed if control is not `enabled` (will result in validation error).

- `set_value`

  Sets new value. Value change will trigger `on_change` callback.

  Arguments:

  - `boolean`

- `toggle`

  Sets value to opposite. Value change will trigger `on_change` callback.

Lua Callback signature

- `on_change`

  Executed on value change. Takes new value and reference to element as arguments.

  Arguments:

  - changed *(boolean)*
  - element reference *(userdata)*

Virtual - CustomDevice - Controls - Progress Bar

This element represents bar with a percentage value from 0% to 100% that can be changed by the user (only from lua side). It is possible to attach a Lua callback which will be executed when value changes.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `type` *(string, read-only)*

  Element type description, based on role and functionality.

- `name` *(string, read-only)*

  User defined name of element. Cannot contain special characters except `_`

- `uuid` *(string, read-only)*

  Universal Unique Identifier of element used by frontend app to properly render and position element.

- `enabled` *(boolean)*

  Indicates if element is enabled/disabled. If element is disabled, user cannot change its properties or call commands (will result in validation error)

- `device_id` *(number, read-only)*

  ID of device from which the control comes from.

- `value` *(number)*

  User defined value between 0 - 100. Value change will trigger `on_change` callback.

- `on_change` *(string, read-only)*

  Name of method (function) from CustomDevice lua code which will be executed on value change. Should not contain `CustomDevice:` prefix, only name.

Commands

Commands cannot be executed if control is not `enabled` (will result in validation error).

- `set_value`

  Sets new value. Value change will trigger `on_change` callback.

  Arguments:

  - `number`

- `increment`

  Adds 1% to value. Value change will trigger `on_change` callback.

- `decrement`

  Substracts 1% from value. Value change will trigger `on_change` callback.

Lua Callback signature

- `on_change`

  Executed on value change. Takes new value and reference to element as arguments.

  Arguments:

  - new value *(number)*
  - element reference *(userdata)*

Virtual - CustomDevice - Controls - Slider

This element represents bar with a numerical value of any range and step, which can be changed by the user from the Lua context and the widget/option context. It is possible to attach a Lua callback which will be executed when value changes.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `type` *(string, read-only)*

  Element type description, based on role and functionality.

- `name` *(string, read-only)*

  User defined name of element. Cannot contain special characters except `_`

- `uuid` *(string, read-only)*

  Universal Unique Identifier of element used by frontend app to properly render and position element.

- `enabled` *(boolean)*

  Indicates if element is enabled/disabled. If element is disabled, user cannot change its properties or call commands (will result in validation error)

- `device_id` *(number, read-only)*

  ID of device from which the control comes from.

- `minimum` *(number, read-only)*

  User defined minimum value.

- `maximum` *(number, read-only)*

  User defined maximum value.

- `value` *(number)*

  User defined value between `minimum` and `maximum`. Value change will trigger `on_change` callback.

- `step` *(number)*

  User defined step for value when using GUI slider or `increment` / `decrement` commands.

- `on_change` *(string, read-only)*

Name of method (function) from CustomDevice lua code which will be executed on value change. Should not contain `CustomDevice:` prefix, only name.

Commands

Commands cannot be executed if control is not `enabled` (will result in validation error).

- `set_value`

  Sets new value. Value change will trigger `on_change` callback.

  Arguments:

  - `number`

- `increment`

  Adds `step` to value. Value change will trigger `on_change` callback.

- `decrement`

  Substracts `step` from value. Value change will trigger `on_change` callback.

Lua Callback signature

- `on_change`

  Executed on value change. Takes new value and reference to element as arguments.

  Arguments:

  - new value *(number)*
  - element reference *(userdata)*

Virtual - CustomDevice - Controls - ComboBox

This element represents a drop-down list that displays the value selected from the Lua context or the widget / options. It is possible to attach a Lua callback which will be executed when selected value changes. Available options can be changed from the Lua.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `type` *(string, read-only)*

  Element type description, based on role and functionality.

- `name` *(string, read-only)*

  User defined name of element. Cannot contain special characters except `_`

- `uuid` *(string, read-only)*

  Universal Unique Identifier of element used by frontend app to properly render and position element.

- `enabled` *(boolean)*

  Indicates if element is enabled/disabled. If element is disabled, user cannot change its properties or call commands (will result in validation error)

- `device_id` *(number, read-only)*

  ID of device from which the control comes from.

- `value` *(string)*

  Selected value. Has to be one of the available values stored in `options`. Value change will trigger `on_change` callback.

- `options` *(string, read-only)*

  JSON formatted string with list of objects with fields `label` and `value` representing all available options in combo box.

- `on_change` *(string, read-only)*

  Name of method (function) from CustomDevice lua code which will be executed on selected value change. Should not contain `CustomDevice:` prefix, only name.

Commands

Commands cannot be executed if control is not `enabled` (will result in validation error).

- `set_value`

  Sets new selected value. Value change will trigger `on_change` callback. Has to be one of `value` strings stored in `options`.

  Arguments:

  - `string`

- `add_option`

  Adds new available value to `options`. Available values has to be unique.

  Arguments:

  - `array` of size 2 where first field is `label` and second is `value`.

- `remove_option_by_label`

  Removes one of the available option from `options` where `label` is equal to passed argument. If currently selected value is removed `value` is changed to empty string and `on_change` callback is triggered.

  Arguments:

  - `string`

- `remove_option_by_value`

  Removes one of the available option from `options` where `value` is equal to passed argument. If currently selected value is removed `value` is changed to empty string and `on_change` callback is triggered.

  Arguments:

  - `string`

- `clear_options`

  Removes all available values from `options`. If any value is selected `value` is changed to empty string and `on_change` callback is triggered.

Lua Callback signature

- `on_change`

  Executed on selected value change. Takes new value and reference to element as arguments.

  Arguments:

  - `string`

- `element_reference`

Examples

Using callbacks

Assuming the user created a device with one of each type of control e.g.:

(Example device structure)

```
{
  "elements": [
    {
      "type": "text",
      "name": "my_text_field",
      [...]
      "on_change": "onMyTextFieldChange"
    },
    {
      "type": "button",
      "name": "my_button",
      [...]
      "on_press": "onMyButtonPress"
    },
    {
      "type": "slider",
      "name": "my_slider",
      [...]
      "on_change": "onMySliderValueChange"
    },
    {
      "type": "switcher",
      "name": "my_switcher",
      [...]
      "on_change": "onMySwitcherValueChange"
    },
    {
      "type": "progress_bar",
      "name": "my_progress_bar",
      [...]
      "on_change": "onMyProgressValueChange"
    },
    {
      "type": "combo_box",
      "name": "my_combo_box",
      [...]
      "on_change": "onMyComboBoxValueChange"
    },
  ]
}
```

(Custom Device logic)

```
-- on_change callback handler for text
function CustomDevice:onMyTextFieldChange(newValue, element)
  -- print new value
  print("Text changed in element " .. element:getValue("name") .. " to " ..
     newValue)

  -- set device name to this value
  self:setValue("name", newValue)
```

```
  -- set button text to this value (other control from this device)
  self:getElement("my_button"):setValue("text", newValue)
end

-- on_press callback handler for button
function CustomDevice:onMyButtonPress(element)
  -- print info
  print("Somebody pressed on " .. element:getValue("name"))

  -- toggle switch (other control from this device)
  self:getElement("my_switcher"):call("toggle")

  -- activate scene after 5 seconds
  scene[5]:activateAfter(5)
end

-- on_change callback handler for slider
function CustomDevice:onMySliderValueChange(newValue, element)
  -- print new value
  print("Value changed in element " .. element:getValue("name") .. " to " ..
     newValue)

  -- send this slider value as json to http server
  local body = {
    value = newValue,
    device_name = self:getValue("name"),
    device_id = element:getValue("device_id")
  }

  http_client[1]:POST("https://custom.server.com")
                :header("Authorization", "Tk63TBJv5hhdnu5UN_F2dgj")
                :contentType("application/json")
                :body(JSON:encode(body))
                :send()
end

-- on_change callback handler for switcher
function CustomDevice:onMySwitcherValueChange(newValue, element)
  -- print new value
  print("Value changed in element " .. element:getValue("name") .. " to " ..
     newValue)

  -- control relays based on new value
  wtp[5]:setValue("state", newValue)

  if newValue then
    sbus[9]:call("turn_on")
  else
    sbus[9]:call("turn_off")
  end
end

-- on_change callback handler for progress bar
function CustomDevice:onMyProgressValueChange(newValue, element)
  -- print new value
  print("Value changed in element " .. element:getValue("name") .. " to " ..
     newValue)

  -- publish to mqtt
  mqtt_client[4]:publish("progress_bar/value", tostring(newValue), 0, false)
end

-- onEvent callback, can catch events that occur in system
function CustomDevice:onEvent(event)
  -- change switcher value when device state changes
```

```
  if wtp[3]:changedValue("state") then
    self:getElement("my_switcher"):call("set_value", wtp[3]:getValue("state"))
  end

  -- Set all switchers to off when sunrise
  if event.type == "sunrise" then
    self:getElement("my_switcher"):call("set_value", false)
    self:getElement("my_switcher_2"):call("set_value", false)
  end

  -- set the text in text field with http client response
  if event.type == "http_client_response" then
    -- check if it is client we are interested in
    if http_client[4]:hasResponse() then
      local decoded = JSON:decode(http_client[4]:response())
      self:getElement("my_text_field"):call("set_value", decoded.data)
    end
  end
end

-- onCommand callback, can catch custom commands executed
function CustomDevice:onCommand(command, arg)

  if command == "modify_elements" then
    print(string.format("Got command %s with argument of type %s.", command,
       type(arg)))
    self:getElement("my_text_field"):call("set_value", command)
    self:getElement("my_switcher"):call("set_value", false)
    self:getElement("my_switcher_2"):call("set_value", false)
    scene[5]:activateAfter(5)
  else
    print(string.format("Command %s not implemented!", command))
  end

end
```

Change element values/call commands from scene or automation

```
virtual[7]:getElement("my_button"):call("press")
virtual[7]:getElement("my_slider"):setValue("value", 55)

if virtual[7]:getElement("my_progress"):getValue("value") > 95 then
  print("Its almost ready!")
end

-- this is custom command defined, see onCommand function example above
virtual[7]:call("modify_elements")
virtual[7]:call("modify_elements", "my-string-val")
virtual[7]:call("modifiy_elements", false)
```

Call custom commands from scene or automation

```
-- this is custom command defined, see onCommand function example above
virtual[7]:call("modify_elements")
virtual[7]:call("modify_elements", "my-string-val")
virtual[7]:call("modifiy_elements", false)

-- this will print "Command non_existing_weird_command not implemented!
virtual[7]:call("non_existing_weird_command", 123.77)
```

Infinite event loops / callback propagation stop

In general, this feature allows to stop custom device element callback propagation and prevent from infinite callback loops.

Consider following case:

Changing state of switcher sends mqtt message to remote device. Changing state of remote device sends mqtt message to custom device switcher.

```
local tasmotaName = "tasmota_D9360D"

function CustomDevice:onChange(newValue, element)
  -- send message to remote device
  self:getElement("text"):setValue("value", utils:ternary(newValue, "On",
     "Off"))
  mqtt_client[4]:publish(
    "cmnd/" .. tasmotaName .. "/POWER",
    utils:ternary(newValue, "ON", "OFF"), 0, false )
end

function CustomDevice:onEvent(event)
 -- message from remote device received
 mqtt_client[4]:onMessage(function(topic, payload, qos, retain, dup)
  -- this is the status when some one toggled it remotely
    -- or response for toggle from publish above (cannot distinguish)
    if topic == "stat/" .. tasmotaName .. "/POWER" then
      self:getElement("switch"):setValue("value", payload == "ON")
    end
  end)
end
```

By default, every change of Custom Device element state will emit event and if there is callback associated it will be executed.

When mqtt latency happens and you toggle switcher 2/3 times from REST API / Web or Mobile app in a row, you may end up with infline loops. When you send ON command (#1), from mqtt you get previous OFF response, this sets switcher to OFF and publishes message, again you get ON payload as response (from message #1) and have inflinite toggling loop. To prevent this situation, you may stop element event propagation in mqtt response when third argument of `setValue` (or `call`) for this element is set to `true`:

This is fixed case, note the third argument in mqtt `onMessage` for element `setValue` function.

```
local tasmotaName = "tasmota_D9360D"

function CustomDevice:onChange(newValue, element)
  self:getElement("text"):setValue("value", utils:ternary(newValue, "On",
     "Off"))
  mqtt_client[4]:publish(
    "cmnd/" .. tasmotaName .. "/POWER",
    utils:ternary(newValue, "ON", "OFF"),
    0,
    false )
end

function CustomDevice:onEvent(event)
  mqtt_client[4]:onMessage(function(topic, payload, qos, retain, dup)
```

```
    -- this is the status when some one toggled it remotely
    -- or response for toggle from publish above (cannot distinguish)
    if topic == "stat/" .. tasmotaName .. "/POWER"
    then
      self:getElement("switch"):setValue("value", payload == "ON", true)
    end
  end)
end
```

Now the received response will still set element value but won't execute `onChange` callback.

Virtual - Heat Pump Manager

The virtual heat pump manager controls the associated devices based on the readings from the sensors and configured target temperatures.

Manager has four work modes: `heating`, `cooling`, `automatic` and `fireplace`. By default works in `heating` mode.

- In `heating` mode computes state of heat demand and controls remote heat pump.
- In `cooling` mode computes state of cool demand and controls remote heat pump.
- In `automatic` mode computes state of heat and cool demand and controls remote heat pump (switches between work modes).
- In `fireplace` mode forces remote heat pump to be in never ending heat demand.

At least one temperature sensor is required.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `temperature` *(number, read-only)*

  Temperature value forwarded from associated sensor or 0 if no sensor associated or computed average temperature if more than one sensor associated Unit: °C with one decimal number, multiplied by 10.

- `enabled` *(boolean)*

  Enable or disable device.

- `work_mode` *(string)*

  Current work mode algorithm. Available values: *heating, cooling, automatic, fireplace*. Default: *heating*

- `state` *(boolean, read-only)*

  Current working state (*active (true) / idle (false)*).

- `target_temperature.current` *(number, read-only)*

  Current target temperature. This is read-only value. Unit: °C with one decimal number, multiplied by 10.

- `target_temperature.heating` *(number)*

  Target temperature for heating work mode Unit: °C with one decimal number, multiplied by 10.

- `target_temperature.cooling` *(number)*

  Target temperature for cooling work mode Unit: °C with one decimal number, multiplied by 10.

- `target_temperature.automatic` *(number)*

  Target temperature for automatic work mode Unit: °C with one decimal number, multiplied by 10.

- `hysteresis.heating` *(number)*

  Damper factor, which will protect from continous on/off switching when current temperature is near target value in heating work mode. Unit: °C with one decimal number, multiplied by 10.

- `hysteresis.cooling` *(number)*

Damper factor, which will protect from continous on/off switching when current temperature is near target value in cooling work mode. Unit: °C with one decimal number, multiplied by 10.

- `hysteresis.automatic` *(number)*

  Damper factor, which will protect from continous on/off and heat pump work mode switching when current temperature is near target value in automatic work mode. Unit: °C with one decimal number, multiplied by 10.

- `dhw_control.enabled` *(boolean)*

  Enable or disable domestic hot water control.

- `dhw_control.temperature` *(number, read-only)*

  Temperature value forwarded from DHW device built-in sensor Unit: °C with one decimal number, multiplied by 10.

- `dhw_control.target_temperature` *(number)*

  Target temperature for DHW control Unit: °C with one decimal number, multiplied by 10.

- `dhw_control.hysteresis` *(number)*

  Damper factor, which will protect from continous on/off switching when current temperature is near target value. Unit: °C with one decimal number, multiplied by 10.

- `dhw_control.state` *(boolean, read-only)*

  Current working state of DHW control (Working/Idle).

- `electric_heater_active` *(boolean, read-only)*

  Indicates electric heater active state in associated heat pump.

Commands

- `enable`

  Calls Heat Pump Manager to enable control.

- `disable`

  Calls Heat Pump Manager to disable control.

- `toggle`

  Calls Heat Pump Manager to toggle control.

- `set_heating_target_temperature`

  Calls Heat Pump Manager to change `heating` work mode target temperature to the desired value.

  Argument:

  target temperature in 0.1°C *(number)*

- `set_cooling_target_temperature`

  Calls Heat Pump Manager to change `cooling` work mode target temperature to the desired value.

  Argument:

  cooling target in 0.1°C *(number)*

- `set_automatic_target_temperature`

  Calls Heat Pump Manager to change `automatic` work mode target temperature to the desired value.

  Argument:

  auto mode target in 0.1°C *(number)*

- `set_dhw_target_temperature`

  Calls Heat Pump Manager to change DHW control target temperature to the desired value.

  Argument:

  DHW target temperature in 0.1°C *(number)*

Examples

Raise heating target temperature between 15:00 and 20:00

```
if dateTime:changed() then
  if dateTime:getHours() == 15 and dateTime:getMinutes() == 0 then
    virtual[1]:call("set_heating_target_temperature", 220)
  elseif dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
    virtual[1]:call("set_heating_target_temperature", 190)
  end
end
```

Raise cooling target temperature on saturday and lower it on monday

```
if dateTime:getTimeOfDay() == 00 then
  if dateTime:getWeekDayString() == "saturday" then
    virtual[1]:call("set_cooling_target_temperature", 230)
  elseif dateTime:getWeekDayString() == "monday" then
    virtual[1]:call("set_cooling_target_temperature", 210)
  end
end
```

Disable manager between June and August

```
if dateTime:changed() then
  if dateTime:getMonth() >= 6 and dateTime:getMonth() <= 8 then
    virtual[1]:setValue("enabled", false)
    virtual[1]:setValue("dhw_control.enabled", false)
  else
    virtual[1]:setValue("enabled", true)
```

```
    virtual[1]:setValue("dhw_control.enabled", true)
  end
end
```

Virtual - Gate

The virtual gate controls the sliding gate, swing gate or garage gate depending on configured variant, using associated devices:

- `full_open_close_output`, commands the gate controller device to fully open or close gate depe1nding on current `state` by sending on/off impulse for 500ms.
- `partial_open_close_output`, commands the gate controller device to partially open or close gate depending on current `state` by sending on/off impulse for 500ms.
- `close_status_sensor`, this is feedback device, detects physical gate `state` between open (open circuit = false) / closed (closed circuit = true)
- `trigger_sensor`, this device can be used to catch external signal (eg. wall switch impulse, or RC remote output impulse) and trigger `full_move` action

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device. Can be either `swing_gate`, `sliding_gate` or `garage_gate`. Can be set only once, when creating device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(string, read-only)*

  Current state of gate: *no_move, moving, closed, closing, open, opening*. No move and moving states are present only when there is no close status sensor associated, as device cannot determine physical state of the gate.

- `full_move_duration` *(number)*

  Maximum time in seconds required to fully close or fully open (select greater) the gate. Valid range: [1 - 120] seconds.

- `partial_move_duration` *(number)*

  Maximum time in seconds required to partial close or partial open (select greater) the gate. Valid range: [1 - full_move_duration] seconds.

Commands

- `full_move`

  Commands the gate controller to do the full move action. Should be used when no `close_status_sensor` is associated or you want to just do the counter direction move according to current `state`.

- `partial_move`

  Commands the gate controller to do the partial move action. Should be used when no `close_status_sensor` is associated or you want to just do the counter direction move according to current `state`. This command requires the `partial_open_close_output` to be associated!

- `full_open`

  Commands the gate controller to do the full open action. This command requires the `close_status_sensor` to be associated!

- `partial_open`

  Commands the gate controller to do the partial open action. This command requires the `close_status_sensor` and `partial_open_close_output` to be associated!

- `close`

  Commands the gate controller to do the close action. This command requires the `close_status_sensor` to be associated!

Examples

Open gates when smoke sensor triggers

```
if wtp[5]:changedValue("smoke_detected") and wtp[5]:getValue("smoke_detected")
then
  print("Sensor detected smoke!!! Opening gates!")

  virtual[5]:call("full_open")
  virtual[6]:call("partial_open")
end
```

Close a gate 10 minutes after opening it

NOTE: This requires adding a timer via API or WebApp with minute unit.

```
if virtual[10]:changedValue("state") then
  if virtual[10]:getValue("state") == "open" then
    timer[3]:start(10)
  else
    timer[3]:stop()
  end
end

if timer[3]:isElapsed() then
  virtual[10]:call("close")
end
```

Virtual - Wicket

The virtual wicket controls the electric strike of wicket or gate, using associated devices:

- `electric_strike_output`, controls the electric strike locking and unlocking
- `close_status_sensor`, this is feedback device, detects physical wicket `state` between open (open circuit = false) / closed (closed circuit = true)
- `trigger_sensor`, this device can be used to catch external signal (eg. wall switch impulse, or RC remote output impulse) and trigger `unlock` action

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device. Can be either `swing_gate`, `sliding_gate` or `garage_gate`. Can be set only once, when creating device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `state` *(string, read-only)*

  Current state of wicket: *locked, unlocked, closed, open*. Open states is present only when there is close status sensor associated, as device can determine physical state of the wicket.

- `unlock_duration` *(number)*

  Time in seconds of electric strike output being active (buzzing). Valid range: [1 - 45] seconds.

Commands

- `lock`

  Locks (turns off) electric strike if its already unlocked.

- `unlock`

  Unlocks (turns on) electric strike if its already locked.

Examples

Unlock wicket when smoke sensor detects smoke

```
if wtp[5]:changedValue("smoke_detected") and wtp[5]:getValue("smoke_detected")
then
  print("Sensor detected smoke!!! Opening gates!")

  virtual[5]:call("unlock")
end
```

SBUS - AnalogInput

Analog input sensor representation. Measures value from analog input and sends it to central unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(integer, read-only)*

  Unique network address.

- `endpoint` *(integer, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `raw_value` *(integer, read-only)*

  Raw value read from analog input.

- `value` *(double/real, read-only)*

  Value from analog input after formula calculation or raw value when no formula specified.

- `formula` *(string)*

  Formula used to calculate value. Refering to `object` you can get data you need to calculate, for example get `raw_value` from object: `object.raw_value`.

  Should contain only calculations returning number. Should not contain any condition statements, loops and more complicated code.

  Example: `object.raw_value * 2 + math.sqrt(object.raw_value)`

- `unit` *(string)*

  Value unit used for statistics.

  Example: `mV`

SBUS - Button

Button customizable in application. Every button action can be assigned different action. For example:

- Turn on first light when clicked once
- Turn on second light when clicked twice
- Turn off all lights when hold 3 seconds

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `action` *(string, read-only)*

  Last action performed by user.

  Example: `button_1_clicked_10_times`, `button_1_hold_started`, `button_1_held_3_seconds`

Examples

Turn on lights when button clicked once

```
local button = sbus[9]
local lights = { wtp[2], wtp[3], wtp[4] }

if ( button:changedValue("action")
and button:getValue("action") == "button_1_clicked_1_times" )
then
  utils.table:forEach(lights, function (light) light:call("turn_on") end)
end
```

Close blinds when button held for 3 seconds

```
local button = sbus[9]
local blinds = {wtp[5], wtp[6], wtp[7]}

if ( button:changedValue("action")
  and button:getValue("action") == "button_1_held_3_seconds" )
then
  utils.table:forEach(blinds, function (blind) blind:call("down") end)
end
```

SBUS - CO2Sensor

Battery powered CO_2 sensor. Measures CO_2 concentration in the air and sends measurement to central unit.

Sensors measure value only every few minutes to save battery.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to wireless device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `co2` *(number, read-only)*

  Sensed CO_2 value. Unit: PPM.

SBUS - Dimmer

Device that controlls light intensity of output LED.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state` *(boolean)*

  State of the output. On/Off.

- `target_level` *(number)*

  Desired light intensity level on which device is set or level on which device will be set when turned on. (depending on `state`) Unit: %.

Commands

- `turn_on`

  Turns on output.

- `turn_off`

  Turns off output.

- `toggle`

  Changes state to opposite.

- `set_level`

  Set light intensity level to desired level smoothly during given time.

  Argument:

  packed arguments *(table)*:

  - light intensity in 1% *(number)*:

    - minumum: 0
    - maximum: 100
  - transition time in 0.1s *(number)*:
    - minimum: 1 *(100 ms)*
    - maximum: 6000 *(10 minutes)*
    - parameter is optional *(500 ms default)*

- `stop`

  Calls Dimmer to stop current level moving action. Does nothing if no action is in progress.

Examples

Turn on light at 19:00 and turn off at 21:00

```
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    sbus[4]:call("turn_on")
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    sbus[4]:call("turn_off")
  end
end
```

Set the light intensity to 75% during 2 minutes

```
sbus[4]:call("set_level", {75, 1200})
```

Dim or brighten lights while button is pressed (simple version)

Solution Drawback: will always take constant time to move from 0%->100%, 50%->100%, 10%->0% etc

```
local dimmer = sbus[4]
local button = sbus[98]

if button:changedValue("action") then

  local action = button:getValue("action")
  local fadeTime = 50 -- 5s / 5000ms

  if action == "button_1_hold_started"
  then
    -- start moving to 100% from current target level
    dimmer:call("set_level", {100, fadeTime})

  elseif action == "button_2_hold_started"
  then
    -- start moving to 0% from current target level
    dimmer:call("set_level", {0, fadeTime})

  elseif action:find("button_1_held_") ~= nil or action:find("button_2_held_")
  then
    -- stop current moving action
    dimmer:call("stop")
```

```
  end

end
```

Dim or brighten lights while button is pressed (advanced version)

Note: will adjust move time regarding current to target level difference

```
-- this function will compute required dimming time (adjusted to current level)
local function computeMoveParameters(currentValue, desiredValue, fullFadeTime)

  -- calculate diff between current and desired level
  local diff = math.abs(desiredValue - currentValue)

  -- calculate how long move will take for this diff
  local reqTime = math.utils:scale(0, 100, 0, fullFadeTime, diff)

  -- clamping / rounding
  return {desiredValue, math.floor(math.utils:clamp(0, fullFadeTime, reqTime))}
end

-- the actual dimming action
local dimmer = sbus[4]
local button = sbus[98]

if button:changedValue("action") then

  local action = button:getValue("action")
  local fadeTime = 50 -- 5s / 5000ms

  if action == "button_1_hold_started"
  then
    -- start moving to 100% from current target
    dimmer:call(
      "set_level",
      computeMoveParameters(dimmer:getValue("target_level"), 100, fadeTime) )

  elseif action == "button_2_hold_started"
  then
    -- start moving to 0% from current target level
    dimmer:call(
      "set_level",
      computeMoveParameters(dimmer:getValue("target_level"), 0, fadeTime) )

  elseif action:find("button_1_held_") ~= nil or action:find("button_2_held_")
  then
    -- stop current moving action
    dimmer:call("stop")

  end
end
```

SBUS - HumiditySensor

Humidity sensor. Measures humidity and sends measurement to central unit. Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `humidity` *(number, read-only)*

  Sensed humidity value.

  Unit: rH% with one decimal number, multiplied by 10.

SBUS - IAQSensor

Battery powered Index of Air Quality sensor. Calculates Air Quality Index based on various measures like CO2 or particles level and relative humidity. Sensors measure values only every few minutes to save battery.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to wireless device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(integer, read-only)*

  Unique network address.

- `endpoint` *(integer, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `iaq` *(integer, read-only)*

  Calculated Index of Air Quality.

- `iaq_accuracy` *(string, read-only)*

  Index of Air Quality calculation accuracy. One of: `unreliable`, `low`, `medium`, `high`.

value	meaning
unreliable	The sensor is not yet stabilized or in a run-in status
low	Calibration required and will be soon started
medium	Calibration on-going
high	Calibration is done, now IAQ estimate achieves best performance

- `air_quality` *(string, read-only)*

  Descriptive name for air quality.

raw	description
≤ 20	`very_good`
21 - 50	`good`
51 - 100	`moderate`
101 - 150	`poor`
151 - 200	`unhealthy`
201 - 300	`very_unhealthy`
301 - 500	`hazardous`
> 500	`extreme`

SBUS - LightSensor

Light sensor measures light illuminance in lux and sends measurement to central unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to wireless device with **ID 6**.

WTP devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `illuminance` *(number, read-only)*

  Sensed light illuminance value.

  Unit: lx.

SBUS - MotionSensor

Motion sensor based on custom configuration checks whether motion was detected.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `enabled` *(boolean)*

  Enable or disable sensor. eg. If you want sense only at night time, you can setup automation to enable/disable sensor.

- `blind_duration` *(number)*

  Duration of sensor being off after detecting motion.

  Unit: seconds.

- `pulses_threshold` *(number)*

  Sensitivity factor. How many pulses from sensor are needed to treat action as motion. The higher the value, the sensitivity decreases.

- `pulses_window` *(number)*

  Sensitivity factor. Maximum time window in which `pulses_threshold` must occur to treat action as motion. The higher the value, the sensitivity increases.

  Unit: seconds.

- `motion_detected` *(boolean, read-only)*

  Holds latest motion detection state. Remains *true* on motion detection and *false* when `blind_duration` time elapses.

  This parameter doesn't emit event when switch from *true* to *false* happens. If you need to observe such action, you need to use `time_since_motion` parameter.

- `time_since_motion` *(number, read-only)*

  Time since last motion detected. Value of -1 means there wasn't any motion since last system startup.

  Unit: seconds.

Commands

- `enable`

  Enables motion detector.

- `disable`

  Disables motion detection.

- `add_time_since_motion_event`

  Adds additional emitting `time_since_motion` event in seconds passed in argument.

  Argument:

  event reemission delay in seconds *(number)*

Examples

Catching motion events

```
if sbus[4]:changedValue("motion_detected") then
  print("someone is moving aroung!")
end
```

```
if sbus[4]:changedValue("time_since_motion") and
   sbus[4]:getValue("time_since_motion") == 0
then
  print("someone is moving aroung!")
end
```

Delayed action

```
if dateTime:changed() then
  sbus[4]:call("add_time_since_motion_event", 30)
end

if sbus[4]:changedValue("time_since_motion") and
   sbus[4]:getValue("time_since_motion") == 30
then
  print("someone was here 30 seconds ago")
end
```

Enable motion detection at sunset and disable it at sunrise

```
if event.type == "sunrise" then
  sbus[3]:call("disable")
elseif event.type == "sunset" then
  sbus[3]:call("enable")
end
```

Enable a light for 5 minutes on motion detection

```
if sbus[4]:changedValue("motion_detected") then
  sbus[60]:setValue("state", true)
  sbus[60]:setValueAfter("state", false, 5 * 60)
end
```

Reconfigure thermostat when motion detected

```
if sbus[4]:changedValue("motion_detected") then
  -- time limited to 2 hours, temperature 23.5°C
  virtual[1]:call("enable_time_limited_mode", {120, 235})
end
```

SBUS - PressureSensor

Pressure senosr measures pressure and sends measurement to central unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to wireless device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `pressure` *(number, read-only)*

  Sensed pressure value.

  Unit: hPa with one decimal number, multiplied by 10.

SBUS - Relay

Execution module that changes state depending on the control signal. Relay can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state` *(boolean)*

  State of the output. On/Off.

- `timeout` *(number)*

  Protection functionality, that will set device state to off if there are communication problems.

  Unit: minutes.

- `timeout_enabled` *(boolean)*

  Parameter that indicates if timeout functionality is enabled.

- `inverted` *(boolean)*

  Indicates if should invert physical state of relay compared to represented state in application.

Commands

- `turn_on`

  Turns on relay output.

- `turn_off`

  Turns off relay output.

- `toggle`

  Changes relay output to opposite.

Examples

Turn on relay between 19:00 and 21:00

```
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    sbus[4]:call("turn_on")
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    sbus[4]:call("turn_off")
  end
end
```

SBUS - RGB Controller

Device that controlls color and light intensity of output LED.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state` *(boolean)*

  State of the output. On/Off.

- `brightness` *(number, read-only)*

  Desired light intensity level on which device is set or level on which device will be set when turned on. (depending on `state`) Unit: %.

- `led_color` *(string, read-only)*

  HTML/Hex RGB color that device will set on its output led strip.

- `white_temperature` *(number, read-only)*

  White temperature that device will set on its output led strip.

  Unit: Kelvins

- `color_mode` *(string, read-only)*

  Color mode that device is set on. One of: `rgb`, `temperature`, `animation`.

- `led_strip_type` *(string)*

  Led strip type that is connected with device. One of: `rgb`, `rgbw`, `rgbww`.

- `white_temperature_correction` *(number)*

  White color temperature correction. Applies when led_strip_type set to rgbw.

- `cool_white_temperature_correction` *(number)*

  Cool white color temperature correction. Applies when led_strip_type set to rgbww.

- `warm_white_temperature_correction` *(number)*

  Warm white color temperature correction. Applies when led_strip_type set to rgbww.

- `active_animation` *(number, read-only)*

  Active animation id if animation was activated. Null value when no animation active.

Commands

- `turn_on`

  Turns on output.

- `turn_off`

  Turns off output.

- `toggle`

  Changes state to opposite.

- `set_brightness`

  Sets light intensity level to desired level smoothly during given time.

  Argument:

  packed arguments *(table)*:

  - light intensity in % *(number)*:
    - minumum: 1
    - maximum: 100
  - transition time in 0.1s *(number)*:
    - minimum: 1 *(100 ms)*
    - maximum: 6000 *(10 minutes)*
    - parameter is optional *(500 ms default)*

- `set_color`

  Sets device output to requested color in RGB mode during requested period of time. Set `color_mode` to `rgb`.

  Argument:

  packed arguments *(table)*

  - HTML/Hex RGB color representation *(string)* * example: `#88fb1c`
  - transition time in 0.1s *(number)*
    - minimum: 1 *(100 ms)*
    - maximum: 6000 *(10 minutes)*
    - parameter is optional *(500 ms default)*

- `set_temperature`

  Sets device output to requested white temperature during requested period of time. Set `color_mode` to `temperature`.

  Argument:

  packed arguments *(table)*:

  - color temperature *(number)*:
    - minumum: 1000
    - maximum: 40000
    - unit: Kelvin
  - transition time *(number)*:
    - minimum: 1 *(100 ms)*
    - maximum: 6000 *(10 minutes)*
    - unit: 100ms
    - parameter is optional *(500 ms default)*

- `activate_animation`

  Activate animation with specified id.

  Arguments:

  packed arguments *(table)*:

  - `id` - ID of animation that will be activated *(number)*

- `stop_animation`

  Stops active animation and call device to return to previous `color_mode`.

Examples

Turn on light to specific color at 7:00 and turn off at 8:00

```
local rgb = sbus[4]

if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    rgb:call("set_color", {"#eedd11", 10})
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    rgb:call("turn_off")
  end
end
```

Tune color temperature based on the time of day

```
local rgb = sbus[79]

if dateTime:changed() then
  if dateTime:getHours() == 16 and dateTime:getMinutes() == 0 then
    -- afternoon, neutral white at 75%
    rgb:call("set_temperature", {5000})
    rgb:call("set_brightness", {75})
```

```
  elseif dateTime:getHours() == 18 and dateTime:getMinutes() == 30 then
    -- evening, warm white at 45%
    rgb:call("set_temperature", {3000, 600})
    rgb:call("set_brightness", {45, 600})
  end
end
```

Activate an animation by id

```
local rgb = sbus[79]
local animation_id = 2

rgb:call("activate_animation", {id=animation_id})
```

Stop active animation

```
local rgb = sbus[79]
rgb:call("stop_animation")
```

Activate an animation by id when device state changes

```
local rgb = sbus[79]
local animation_id = 3

if wtp[3]:changedValue("state") then
  rgb:call("activate_animation", {id=animation_id})
end
```

SBUS - TemperatureRegulator

Temperature regulator notifies when desired temperature is reached in room. Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_mode.current` *(string, read-only)*

  Regulator target temperature mode. Specifies if regulator works in `constant` mode with one target temperature, `time_limited` mode with one temporary target temperature or according to schedule in `schedule` mode with many target temperatures in time, configured by user.

  Parameter is read only, use commands to change target temperature mode! Parameter cannot be `schedule` if thermostat doesnt have `has_schedule` label! When not associated with Virtual Thermostat it will always work in `constant` mode.

  Available values: *constant, schedule, time_limited*. Default: *constant*

- `target_temperature_mode.remaining_time` *(number, read-only)*

Remaining time until `time_limited` mode ends. Cannot be modified directly - use commands.

Unit: minutes.

- `target_temperature_miniumum` *(number)*

  Lower limit of the target temperature. Could not be greater than maximum. Setting minimum value above target value, will also change target value to minimum.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_maximum` *(number)*

  Upper limit of the target temperature. Could not be less than minimum. Setting maximum below target, will also change target value to minimum. Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_reached` *(boolean)*

  Controls device‘s algorithm state indicator (available on some regulators). eg LED Diode. May be controlled by external algorithms or devices such as Thermostat (when thermostat is active, indicator will blink)

- `system_mode` *(string)*

  Indicates external system work mode. Used to display proper icon on the regulator.

  May only be changed if device is not assigned to thermostat (has not label *managed_by_thermostat*).

  Available values: *off, heating, cooling*. Default: *heating*

- `keylock` *(string)*

  Device keylock state. Available values: *on, off, unsupported*

- `confirm_time_mode` *(boolean, read-only)*

  Mainly for Mobile/Web App purposes. Indicates if time mode modal should be displayed when changing thermostat temperature. Controlled by Virtual Thermostat.

Commands

- `set_target_temperature`

  Calls Temperature Regulator to change `constant` or `time_limited` mode target temperature to the desired value.

  If regulator works in `time_limited` mode it will change target temperature only, not affecting `remaining_time`.

  If regulator works in `schedule` mode it will change target temperature mode to `constant`.

  Argument:

  target temperature in 0.1°C *(number)*

- `enable_constant_mode`

  Calls Temperature Regulator to change target temperature mode to `constant`. When regulator is already in `constant` mode, it will change mode `target_temperature` only.

  NOTE: Cannot be executed when regulator is not associated with Thermostat.

  Argument:

  target temperature in 0.1°C *(number)*

- `enable_time_limited_mode`

  Calls Temperature Regulator to change mode and target temperature mode to `time_limited` for desired time.

  When regulator is already in `time_limited` mode, it will change `remaining_time` or/and `target_temperature` depending on payload.

  NOTE: Cannot be executed when regulator is not associated with Thermostat.

  Arguments:

  packed arguments *(table)*:

  - remaining time in minutes *(number)*
  - target temperature in 0.1°C *(number)*

- `disable_time_limited_mode`

  Calls Temperature Regulator to disable `time_limited` and go back to previous target temperature mode. When regulator is not in `time_limited` mode, it will do nothing.

  NOTE: Cannot be executed when regulator is not associated with Thermostat.

Examples

Raise target temperature between 15:00 and 20:00

```
if dateTime:changed() then
  if dateTime:getHours() == 15 and dateTime:getMinutes() == 0 then
    sbus[5]:call("set_target_temperature", 220)
  elseif dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
    sbus[5]:call("set_target_temperature", 190)
  end
end
```

SBUS - TemperatureSensor

Temperature sensor. Measures temperature and sends measurement to central unit. Can be assigned to virtual thermostat in web application as room or floor sensor.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `temperature` *(number, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

- `calibration` *(number)*

  Static point temperature calibration, used to adjust measurments.

  Unit: °C with one decimal number, multiplied by 10.

SBUS - TwoStateInputSensor

Boolean input sensor checks input state and send it to central unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `state` *(boolean, read-only)*

  State of the input.

- `inverted` *(boolean)*

  Indicates if physical state of input compared to represented state in application should be inverted.

SBUS - Blind Controller

Controller opens and closes a roller shade or tilt blind.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container eg. `sbus[6]` gives you access to device with **ID 6**.

SBUS devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `endpoint` *(number, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `target_opening` *(number)*

  Desired setpoint opening, which device will try to achieve.

  NOTE: If device doesnt contain `percent_opening_control` label, target opening is limited to 0%, 50% or 100% (only these three).

  Unit: %.

- `current_opening` *(number, read-only)*

  Current opening value.

  Unit: %.

- `target_tilt` *(number, optional)*

  Desired tilt position.

  NOTE: Parameter is optional. Available when: `percent_tilt_control` label is provided.

  Unit: %.

- `current_tilt` *(number, optional, read-only)*

  Current tilt position

  NOTE: Parameter is optional. Available when: `percent_tilt_control` label is provided.

  Unit: %.

- `window_covering_type` *(string)*

  Determines wheter tilt should be possible or not.

  NOTE: Can be modified when: `percent_tilt_control` label is provided.

  Available values to set: *roller_shade, tilt_blind*

- `tilt_range` *(number, optional)*

  Determines tilt range.

  NOTE: Parameter is optional. Available when: `percent_tilt_control` label is provided. **NOTE:** Can be modified when: `window_covering_type` is equal to `tilt_blind`.

  Available values to set: *90, 180*

  Unit: angle (degrees).

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: *auto, fixed, off*

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle.

  Example: *#FF00FF*

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active eg. motor is working.

  Example: *#FFFF00*

  Note: Available when backlight is supported - check if *has_backlight* label is provided.

Commands

- `open`

  Opens a blind to specific value in percent passed in argument.

  Argument:

  opening percentage *(number)*

- `up`

  Fully opens a blind.

- `down`

  Fully closes a blind.

- `stop`

  Immediately stops a blind motor.

- `calibration`

  Starts blind calibration cycle.

- `tilt`

  Calls tilt to the desired value.

  Argument:

  tilt percentage *(number)*

Examples

Open blind at sunrise and close at sunset

```
if event.type == "sunrise" then
  sbus[3]:call("up")
elseif event.type == "sunset" then
  sbus[3]:call("down")
end
```

Set blind to half-open at noon

```
if dateTime:changed() then
  if dateTime:getHours() == 12 and dateTime:getMinutes() == 0 then
    sbus[3]:call("open", 50)
  end
end
```

AlarmSystem - Satel - AlarmZone

Object from Satel central unit. Zone from Satel central unit representation.

Zone may be added by configuration read using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `alarm_system` container eg. `alarm_system[6]` gives you access to device with **ID 6**.

Alarm system devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application. Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(integer, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_version` *(string, read-only)*

  Satel central unit version.

- `parent_id` *(integer, read-only)*

  ID of parent device which device belongs to.

- `armed` *(boolean, read-only)*

  Indicates if zone is armed.

- `violated` *(boolean, read-only)*

  Zone violation state.

Commands

- `arm`

  Arm zone. *(This command will send request to Satel central unit and armed state will change only if central unit approves the command. So subsequent call to getValue("armed") in script will most likely return previous armed state. Used should check if "armed" parameter changes by using changedValue("armed") method in scripts)*

  Argument:

  pin code used to arm zone *(string)*

- `arm_in_mode`

  Arm zone in requested mode. *(This command will send request to Satel central unit and armed state will change only if central unit approves the command. So subsequent call to getValue("armed") in script will most likely return previous armed state. Used should check if "armed" parameter changes by using changedValue("armed") method in scripts)*

Argument:

packed arguments *(table)*:

- `pin_code` - pin code used to arm zone *(string)*
- `mode` - Mode which will be used to arm zone (0-3). *(integer)*. See Satel documentation for differences between modes.

- `disarm`

  Disarm zone. *(This command will send request to Satel central unit and armed state will change only if central unit approves the command. So subsequent call to getValue("armed") in script will most likely return previous armed state. Used should check if "armed" parameter changes by using changedValue("armed") method in scripts)*

  Argument:

  pin code used to disarm zone *(string)*

Examples

Arm zone at 8:00PM and disarm at 7:00AM

```
local zone = alarm_system[3]

if dateTime:changed() then
    if dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
        zone:call("arm", "1234")
    elseif dateTime:getHours() == 7 and dateTime:getMinutes() == 0 then
        zone:call("disarm", "1234")
    end
end
```

Arm zone in mode 1

```
alarm_system[3]:call("arm_in_mode", {pin_code="1234", mode=1})
```

AlarmSystem - Satel - TwoStateInputSensor

Device from Satel central unit.

Boolean input sensor state (violation) is read from Satel central unit.

Device may be added by configuration read using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `alarm_system` container eg. `alarm_system[6]` gives you access to device with **ID 6**.

Alarm system devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `sub_id` *(integer, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_version` *(string, read-only)*

  Satel central unit version.

- `parent_id` *(integer, read-only)*

  ID of parent device which device belongs to.

- `state` *(boolean, read-only)*

  State of the input. On/Off.

- `inverted` *(boolean)*

  Indicates if physical state of input compared to represented state in application should be inverted.

AlarmSystem - Satel - TwoStateOutput

Device from Satel central unit.

Execution module that changes state depending on the control signal.

NOTE: Only some types of outputs can be controlled remotely (via Sinum). Check Satel documentation for more information.

Device may be added by configuration read using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `alarm_system` container eg. `alarm_system[6]` gives you access to device with **ID 6**.

Alarm system devices have global scope and they are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application (like `#a800b0`)

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `software_status` *(string, read-only)*

  Current device software update status: `up_to_date`, `update_available`, `recovery`, `pending`, `downloading`, `updating`

- `sub_id` *(integer, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_version` *(string, read-only)*

  Satel central unit version.

- `parent_id` *(integer, read-only)*

  ID of parent device which device belongs to.

- `state` *(boolean, read-only)*

  State of the output. On/Off.

Commands

- `turn_on`

  Turns on output.

  Argument:

  pin code used to turn on output *(string)*

- `turn_off`

  Turns off output.

  Argument:

  pin code used to turn off output *(string)*

- `toggle`

  Changes output to opposite state.

Argument:

pin code used to toggle output state *(string)*

Examples

Set output state based on alarm input sensor violated state

```
local input = alarm_system[1]
local output = alarm_system[2]

if input:changedValue("state") then
    if input:getValue("state") then
        output:call("turn_on", "1234")
    else
        output:call("turn_off", "1234")
    end
end
```

Lora - FloodSensor

Battery powered, flood sensor. Detects water leak on flat surfaces.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `lora` container eg. `lora[6]` gives you access to device with **ID 6**.

Lora devices have global scope and they are visible in all executions contexts.

Note: Lora devices are available only in Sinum Pro.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online*, *offline*, *unknown*, *service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date*, *update_available*, *recovery*, *pending*, *downloading*, *updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `eui` *(string, read-only)*

  Lora 64-bit device EUI (Extended Unique Identifier).

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `flood_detected` *(boolean, read-only)*

  A flag representing the detection of flood / water leak by the sensor.

Examples

Catching alarms

```
if lora[1]:changedValue("flood_detected") and lora[1]:getValue("flood_detected")
then
  print("Sensor detected water leak!!!")
  notify:warning("Water leak!", "Water leak detected in toilet!", {1, 3})
end
```

Close the valve and turn on siren on water leak

```
valve = wtp[1]
siren = wtp[2]
floodSensor = lora[1]

if floodSensor:changedValue("flood_detected") and
   floodSensor:getValue("flood_detected")
then
  valve:call("turn_off")
  siren:call("turn_on")
end
```

Lora - HumiditySensor

Battery powered humidity sensor. Measures humidity and sends measurement to central unit.

Sensors measure humidity only every few minutes to save battery. Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `lora` container eg. `lora[6]` gives you access to device with **ID 6**.

Lora devices have global scope and they are visible in all executions contexts.

Note: Lora devices are available only in Sinum Pro.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `eui` *(string, read-only)*

  Lora 64-bit device EUI (Extended Unique Identifier).

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `humidity` *(number, read-only)*

  Sensed humidity value.

  Unit: rH% with one decimal number, multiplied by 10.

Lora - OpeningSensor

Battery powered opening sensor. Checks whether window or door is open. Based on that information system can do some action, for example, turn off heating in that room.

Can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `lora` container eg. `lora[6]` gives you access to device with **ID 6**.

Lora devices have global scope and they are visible in all executions contexts.

Note: Lora devices are available only in Sinum Pro.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `eui` *(string, read-only)*

  Lora 64-bit device EUI (Extended Unique Identifier).

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `open` *(boolean, read-only)*

  Opening sensor state. Open/Closed.

Examples

Catch open and close events

```
if lora[12]:changedValue("open") then
  if lora[12]:getValue("open") then
    print("The window is now open!")
  else
    print("The window is now closed!")
  end
```

```
end
```

Lora - Relay

Execution module that changes state depending on the control signal.

Relay can be assigned to virtual thermostat in web application.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `lora` container eg. `lora[6]` gives you access to wireless device with **ID 6**.

Lora devices have global scope and they are visible in all executions contexts.

Note: Lora devices are available only in Sinum Pro.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(integer, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(array, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array, read-only)*

  Collection of tags assigned to device.

- `room_id` *(integer, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(integer, read-only)*

  Unique network address.

- `signal` *(integer, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `eui` *(string, read-only)*

  Lora 64-bit device EUI (Extended Unique Identifier).

- `state` *(boolean)*

  State of the output. On/Off.

- `inverted` *(boolean)*

  Indicates if should invert physical state of relay compared to represented state in application.

Commands

- `turn_on`

  Turns on relay output.

- `turn_off`

  Turns off relay output.

- `toggle`

  Changes relay output to opposite.

Examples

Turn on relay between 19:00 and 21:00

```
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    lora[4]:call("turn_on")
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    lora[4]:call("turn_off")
  end
end
```

Turn on the light for 5 minutes when motion detected

```
if lora[7]:changedValue("motion_detected") then
  lora[11]:setValue("state", true)
  lora[11]:setValueAfter("state", false, 5 * 60)
end
```

Lora - TemperatureSensor

Measures temperature and sends measurement to central unit.

Temperature sensors measure temperature only every few minutes to save battery.

Can be assigned to virtual thermostat in web application as room or floor sensor.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `lora` container eg. `lora[6]` gives you access to device with **ID 6**.

Lora devices have global scope and they are visible in all executions contexts.

Note: Lora devices are available only in Sinum Pro.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `eui` *(string, read-only)*

  Lora 64-bit device EUI (Extended Unique Identifier).

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  Note: Parameter is optional. Available when device is battery powered - check if *battery_powered* label is provided.

- `temperature` *(number, read-only)*

  Sensed temperature value.

  Unit: °C with one decimal number, multiplied by 10.

- `calibration` *(number)*

  Static point temperature calibration, used to adjust measurments.

  Unit: °C with one decimal number, multiplied by 10.

Lora - TwoStateInputSensor

Boolean input sensor checks input state and send it to central unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `lora` container eg. `lora[6]` gives you access to device with **ID 6**.

Lora devices have global scope and they are visible in all executions contexts.

Note: Lora devices are available only in Sinum Pro.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `address` *(number, read-only)*

  Unique network address.

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `software_version` *(string, read-only)*

  Software name and version description.

- `eui` *(string, read-only)*

  Lora 64-bit device EUI (Extended Unique Identifier).

- `state` *(boolean, read-only)*

  State of the input.

- `inverted` *(boolean)*

  Indicates if physical state of input compared to represented state in application should be inverted.

System Module - WTP, SBus or Modbus Extenders

Device extends signal range of wireless WTP devices, extends SBus communication line or Modbus RTU communication line. It passes all communication with WTP/SBus/Modbus devices to Sinum Central Unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `system_module` container eg. `system_module[2]` gives you access to module with **ID 2**.

System modules have global scope and the are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of device. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `icon` *(string)*

  Name of the icon associated with device.

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `status` *(string, read-only)*

  Current device connection status: *online, offline, unknown, service*

- `messages` *(table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to device.

- `software_status` *(string, read-only)*

  Current device software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which device is associated or nil otherwise.

- `visible` *(boolean, read-only)*

  Indicates if device is enabled/viable to use.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color in application.

  Example: *#FFFF00*

- `uuid` *(string, read-only)*

  Unique identifier of system module, used for communication.

- `enabled` *(boolean)*

  Indicates if device is enabled. Disabled extender disables communication with all WTP devices connected to it.

- `software_version` *(string, read-only)*

  Current software version.

- `transceiver_uuid` *(number, read-only)*

  Unique indentifier of transceiver.

- `link_latency` *(number, read-only)*

  Average communication latency in last 10 minutes.

- `latest_link_latency` *(number, read-only)*

  Latest reported communication latency.

- `network_name` *(string, read-only)*

  Name of WiFi network extender is connected to.

- `network_signal` *(int, read-only)*

  Value from 0 to 100 indicating how strong WiFi signal is.

- `network_channel` *(int, read-only)*

  WiFi network channel.

- `ethernet_connected` *(boolean, read-only)*

  Indicates if extender has ethernet cable connected.

  Note: Parameter is optional. Available when device has ethernet - check if *has_ethernet* label is provided.

System Module - Lora Gateway

Responsible for communication with Lora devices connected to Sinum Central Unit.

Device may be added by registration using web application. Can be edited or deleted via REST API or a web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `system_module` container eg. `system_module[2]` gives you access to module with **ID 2**.

System modules have global scope and the are visible in all executions contexts.

Note: Lora devices are available only in Sinum Pro.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of module. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `icon` *(string)*

  Name of the icon associated with module.

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `status` *(string, read-only)*

  Current module connection status: *online*, *offline*, *unknown*, *service*

- `messages` *(table, read-only)*

  Collection of module specific messages. Contains module error/warning details.

- `labels` *(table, read-only)*

Collection of module specific labels. Contains module specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to module.

- `software_status` *(string, read-only)*

  Current module software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which module is associated or nil otherwise.

- `visible` *(boolean, read-only)*

  Indicates if module is enabled/viable to use.

- `color` *(string)*

  HTML/Hex RGB representation of module widget color in application.

  Example: *#FFFF00*

- `uuid` *(string, read-only)*

  Unique identifier of system module, used for communication.

- `enabled` *(boolean)*

  Indicates if module is enabled. Disabled transceiver disables communication with all devices of certain type connected to Sinum Central Unit.

- `software_version` *(string, read-only)*

  Current software version.

System Module - WTP, SBus or Modbus Transceiver

Representation of built-in module which is responsible for communication with certain type (WTP, SBus or Modbus) of devices connected to Sinum Central Unit.

This system module cannot be added or removed by user.

Property modification is possible via REST API, web app or directly from scripts using `system_module` container eg. `system_module[2]` gives you access to module with **ID 2**.

System modules have global scope and the are visible in all executions contexts.

Properties

Properties direct access is not allowed. You can get or set values using `setValue`, `getValue` methods.

Attempting to reference a nonexistent object, retrieve a nonexistent object property, or set the wrong value type will result in a script error.

Available Properties

- `id` *(number, read-only)*

  Unique object identifier.

- `name` *(string)*

  User defined name of module. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

- `icon` *(string)*

  Name of the icon associated with module.

- `type` *(string, read-only)*

  Device type description, based on role and functionality.

- `variant` *(string, read-only)*

  Defines the more detailed functionality of the device.

- `class` *(string, read-only)*

  Device class description, based on communication type, manufacturer etc.

- `status` *(string, read-only)*

  Current module connection status: *online*, *offline*, *unknown*, *service*

- `messages` *(table, read-only)*

  Collection of module specific messages. Contains module error/warning details.

- `labels` *(table, read-only)*

  Collection of module specific labels. Contains module specification and additional flags.

- `tags` *(table, read-only)*

  Collection of tags assigned to module.

- `software_status` *(string, read-only)*

  Current module software update status: *up_to_date, update_available, recovery, pending, downloading, updating*

- `room_id` *(number, read-only, optional, nilable)*

  ID of room with which module is associated or nil otherwise.

- `visible` *(boolean, read-only)*

  Indicates if module is enabled/viable to use.

- `color` *(string)*

  HTML/Hex RGB representation of module widget color in application. Example: *#FFFF00*

- `uuid` *(string, read-only)*

  Unique identifier of system module, used for communication.

- `enabled` *(boolean)*

  Indicates if module is enabled. Disabled transceiver disables communication with all devices of certain type connected to Sinum Central Unit.

- `software_version` *(string, read-only)*

  Current software version.

- `transceiver_uuid` *(number, read-only)*

  Unique indentifier of transceiver connected to module.

- `link_latency` *(number, read-only)*

  Average communication latency in last 10 minutes.

- `latest_link_latency` *(number, read-only)*

  Latest reported communication latency.