Lua API Manual

Rev. 2025-07-03

Contents

Energy center   182

Schedules   201

WTP devices   213

TECH RS devices   **287**

Modbus devices   **329**

General

Lua is a lightweight, high-level, multi-paradigm programming language designed primarily for embedding in applications and extending their functionality. The language has extensive documentation on its [official website](#).

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

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

Lua interpreter

Sinum ships with Lua version *5.4*. Features that normally interface with operating system have been removed or modified to work within the Sinum software.

Global environment differences

Following objects are not available:

- `_VERSION`
- `collectgarbage`
- `coroutine`
- `debug`
- `dofile`
- `io`
- `load`
- `loadfile`
- `package`
- `rawequal`
- `rawget`
- `rawlen`
- `rawset`
- `require`
- `select`
- `warn`

Standard output and standard error streams are attached to an appropriate script log, therefore functions `assert`, `error` and `print` create log entries[1]

`os` library differences

Following functions are not available:

- `execute`
- `exit`
- `getenv`
- `remove`
- `rename`
- `setlocale`
- `tmpname`

---

[1]Logs can be accessed by `>_` button in the bottom left corner of the web interface.

`string` library differences

Following functions are not available:

- `dump`

Events

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

If events occurs, system will trigger run-cycle through all **enabled** / **not banned** Lua automations and CustomDevice event handlers in order to perform user defined actions.

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

Properties

- `type` *(string)*

  Type of the event

- `details` *(string)*

  More detailed info about the event e.g. if the event refers to device state change, this property will contain the name of the attribute that was recently changed.

- `source` *(table)*

  The source object of event, in other words, the object which e.g. just changed. Properties:

  - `type` *(string)* — possible values: `wtp`, `sbus`, `slink`, `tech`, `virtual`, `custom_device_module`, `system_module`, `alarm_system`, `modbus`, `lora`, `variable`, `automation`, `scene`
  - `id` *(number)*

    > **Note**
    >
    > You may use type and ID to refer to specific object in global containers, see examples.
    > Currently available only for devices, variables, scenes and automations, for other sources it will return empty type and ID equal to 0.

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 e.g. 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 e.g. due to syntax error.

- `lua_timer_elapsed`

  Occurs when a Lua timer elapses after start for specific time.

- `lua_ping_reply`

  Occurs when received `ping` response.

- `lua_port_knock_finished`

  Occurs when received `port_knock` response.

- `sunrise`

  Occurs once a day at sunrise.

- `sunset`

  Occurs once a day at sunset.

- `modbus_client_async_read_response`

  Occurs when a Modbus client finishes asynchronous read request.

- `modbus_client_async_write_response`

  Occurs when a Modbus client finishes asynchronous write request.

- `modbus_client_async_request_failure`

  Occurs when a Modbus client asnychronous request fails.

- `modbus_client_state_changed`

  Occurs when a Modbus client attribute was changed by user, automation or scene.

- `mqtt_client_connected`

  Occurs when an MQTT client connects to broker (when CONACK received).

- `mqtt_client_disconnected`

  Occurs when an MQTT client disconnects from broker, e.g. due to network error

- `mqtt_client_message_received`

  Occurs when an MQTT client receives message at subscribed topic.

- `http_client_request_failed`

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

- `mqtt_client_subscription_established`

  Occurs when an MQTT client establishes new subscription.

- `mqtt_client_state_changed`

  Occurs when an MQTT client attribute was changed by user, automation or scene.

- `http_client_request_completed`

  Occurs when received response on an HTTP client after sending a request.

- `ws_client_connected`

  Occurs when an Websocket client connects to server.

- `ws_client_disconnected`

  Occurs when an Websocket client disconnects from server, e.g. due to network error

- `ws_client_message_received`

  Occurs when an Websocket client receives message.

- `ws_client_state_changed`

  Occurs when an Websocket client attribute was changed by user, automation or scene.

- `lua_http_server_request`

  Occurs when the Lua HTTP server receives a request.

- `lua_variable_state_changed`

  Occurs when a Lua variable attribute was changed by user, automation or scene.

- `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 (e.g. button is pressed).

- `custom_device_command_call`

  Occurs when command on custom device was called.

- `custom_device_component_state_changed`

  Occurs when custom device on of component attribute was changed by user, automation or scene.

- `weather_state_changed`

  Occurs when weather full data was changed.

- `weather_partial_update`

  Occurs when some weather data (e.g. temperature) was changed.

- `flow_monitor_state_changed`

  Occurs when Energy center Flow monitor attribute was changed by user, automation or scene.

- `energy_prices_state_changed`

  Occurs when Energy center prices was changed by user, automation, scene or new data was received.

- `energy_storage_state_changed`

  Occurs when Energy center Storage attribute was changed by user, automation or scene.

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 == "temperature"
then
  print("A device has updated its temperature readout!")
end
```

> **Note**
>
> You can check which device event refers to using `event.source` property, see this example.
> If you know your object in advance and you just want to check if event refers to it:
>
> ```
> if wtp[4]:changedValue("target_temperature") then
>   print("WTP temperature regulator #4 has changed its setpoint!")
> 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 check which device event refers to using `event.source` property, see this example.
> If you know your object in advance and you just want to check if event referts to it:
>
> ```lua
> 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

```lua
if event.type == "automation_failed" then
  print("One of automations failed!")
end
```

> **Note**
>
> You can check which device event refers to using `event.source` property, see this example.
> If you know your object in advance and you just want to check if event referts to it:
>
> ```lua
> if automation[4]:failed() then
>   print("Your automation with ID 4 failed!")
> end
> ```

Grab object that is source of event

```lua
local source = event.source

if source.id ~= 0 then
  -- access
  local object = _G[source.type][source.id]
  print(object)
else
  print("Event source unavailable!")
end
```

Perform an action every minute

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

Check if a timer elapsed

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

> **Note**
>
> You can check which device event refers to using `event.source` property, see **Grab object that is source of event** example.
> If you know your object in advance and you just want to check if event referts to it:
>
> ```lua
> if timers[4]:isElapsed() then
>   print("Lua timer with ID 4 elapsed!")
> end
> ```

Catch sunrise event

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

Catch sunset event

```lua
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 the web-app
- `custom_device_module` — [Custom Device modules](#) added in custom devices
- `sbus` — wired [SBUS devices](#)
- `slink` — wireless [SLINK devices](#)
- `modbus` — wired [Modbus devices](#)
- `system_module` — [system modules](#) (e.g. transceivers or signal extenders)
- `alarm_system` — integrated [alarm system devices](#)
- `lora` — wireless [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 e.g. `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.

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

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

Common device properties

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

  Device class description, based on communication type, manufacturer etc.

- `color` *(string)*

  HTML/Hex RGB representation of device widget color.

- `icon` *(string)*

  One of font awesome font set, presented in frontend app.

  Maximum length: 64

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

Unique (per system) instance identifier.

Value range: ⟨1; +∞)

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

Collection of device specific labels. Contains device specification and additional flags.

- `messages` *(sequence, read-only)*

Collection of device specific messages. Contains device error/warning details.

Properties:

- `code` *(string, read-only)* — error specific code
- `message` *(string, read-only)* — english translation of error message
- `name` *(integer, read-only)* — error message translation ID
- `typeText` *(string, read-only)* — message type: `"error"` or `"warning"`

- `name` *(string)*

Custom device name for user purposes. Cannot contain special characters except `:` `,` `;` `.` `-` `_`

Maximum length: 64

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

ID of room with which device is associated or null otherwise.

Value range: ⟨1; +∞)

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

Current device connection status.

Has to be one of: `online` `offline` `unknown` `service`

- `tags` *(sequence)*

Collection of device tags. Tags can be set by user to differentiate devices in room.

- `type` *(read-only)*

Device type description, based on role and functionality.

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

Defines the more detailed functionality of the device.

One of: `generic` `eco_geo` `eco_air` `satel` `heatcomp` `remeha_elga_ace` `alpha_innotec` `solax_x1` `solax_x3` `itho` `eastron_sdm630` `solar_edge_single` `solar_edge_multiple` `mitsubishi_ecodan` `galmet_prima` `kaisai_khc` `sliding_gate` `swing_gate` `garage_gate` `goodwe_mt_smt` `goodwe_sdt_ms_dns_xs` `heat_eco` `solis` `huawei_sun_2000` `p1` `ampowr_ampi_home_1_phase` `ampowr_ampi_home_3_phase` `wallbox_ev` `daikin_altherma` `eco_geo_high_power` `tech_le3x230mb` `heatcomp_inverter` `temperature_sensor` `heat_pump` `inverter` `energy_meter` `battery` `common_dhw_main` `relay` `temperature_regulator`

`two_state_input_sensor` `floor_temperature_sensor` `simple_fan_coil` `pergola` `alarm_siren` , `car_charger`

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

  Indicates if device is enabled/viable to use.

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

  Device type defined for voice assistants (Goolge Home/Alexa etc.) integrations.

  One of: `not_set` `light` `thermostat` `blind_controller` `blind_controller_percent_control` `two_state_switch` `dimmer` `rgb_controller` `door` `lock` `fireplace` `garage_door` `gate` `smoke_detector` `sprinkler` `carbon_dioxide_level` `water_leak`

- `purpose` *(string)*

  Some devices can have a specific, changeable purpose in the system, eg. `relay` or `two_state_output` can be used as a light switch, sprinkler controller, etc. This parameter may be read-only for some devices that have a fixed purpose.

  One of: `general` , `light` , `sprinkler` , `door` , `lock` , `fireplace` , `heater` , `cooler` , `humidifier` , `fan` , `pump` , `valve`

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

  List of properties that are supervised by other device. Any property on the list cannot be modified.

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` *(number)* — number of seconds after which the action will take place. Should be not less than 0.1 seconds.

- `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

- `hasLabel(label)`

  Returns true if device has label specified in parameter.

  Returns:

  - *boolean*

Arguments:

∘ `label` *(string)* — label name

Commands

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

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

Examples

Check if any property changed

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

Check if specific property changed

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

Get value of a device property

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

Set value of a device property

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

Modify and read tags

```lua
wtp[9]:setValue("tags", { "tag1", "tag2", "tag3" })

print(wtp[9]:hasTag("tag1"))
-- true

print(wtp[9]:hasTag("tag99"))
-- false
```

Check if device contains label

```
print(wtp[9]:hasLabel("battery_powered"))
-- true

print(virtual[9]:hasLabel("has_schedule"))
-- false
```

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, e.g. 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 e.g. `scene[6]` gives you access to scene with **ID 6**.

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

> **Note**
>
> The scene should execute as soon as possible, or it will block execution queue for other scenes. Using long-acting code will slow down the system and may result in script termination.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

- `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 e.g. 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` *(string[], 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` *(string[])*

  Collection of tags (array-like table of strings) 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` *(number)* — number of seconds after which the action will take place. Should be not less than 0.1 seconds.

- `call(command_name, arg)`

  Calls scene to execute commmand.

  Returns:

  - *userdata* — reference to scene object, for call chains

  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

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

Change scene properties with chained calls

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

Change tags

```lua
scene[9]:setValue("tags", {"tag1", "tag2", "tag3"})
```

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 the web application in configuration.

Leaving Home

```lua
-- 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

```lua
-- 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 which are executed on every event. The user is responsible for "catching" the event and performing a specific action on its basis, e.g. 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 the web application served through the central unit server.

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

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

> **Note**
>
> An automation should execute as soon as possible, or it will block execution queue for other automations. Using long-acting code will slow down the system and may result in script termination.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

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

- `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 e.g. 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` *(array-like table)*

Collection of tags (array-like table of strings) 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

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

Change automation properties

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

Change tags

```lua
automation[9]:setValue("tags", {"tag1", "tag2", "tag3"})
```

React to automation being turned on

```lua
if automation[context().id]:changedValue('enabled') then
  print 'automation was enabled'
end
```

Function `context()` is described here.

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

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 color 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` *(string[], 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.

- `heating_configuration_finished` *(boolean)*

  Informs whether heating configuration is finished for a room.

- `climate_control_status.issues` *(string[], read-only)*

  Issues related to heating configuration. Any of: `climate_control_unavailable`, `temperature_sensor_without_thermostat`, `temperature_regulator_without_thermostat`, `thermostat_without_temperature_sensor`, `thermostat_without_temperature_regulator`.

- `climate_control_status.skipped` *(string[])*

  Issues related to heating ignored by the user. Any of: `climate_control_unavailable`, `temperature_sensor_without_thermostat`, `temperature_regulator_without_thermostat`, `thermostat_without_temperature_sensor`, `thermostat_without_temperature_regulator`.

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

  Informs whether there is any light on in the room.

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 the 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 the 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'

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

List all devices with tag 'regulator' in a room

```lua
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

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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);
```

Schedules

Schedules are exposed as a key-based container of objects `schedule`.

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

Schedules 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` or `getValue` methods.

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

For more details e.g. available properties refer to specific schedule type documentation.

Common schedule properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

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

  Unique object identifier

- `name` *(string)*

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

- `icon` *(string)*

  User defined icon of schedule.

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

  Schedule type. One of: `thermal`, `temperature_curve`, `relay_control`

Methods

- `changed()`

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

  Returns:

  ◦ *boolean*

- `changedValue(property_name)`

  Checks if specific property of schedule 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 schedule object, for call chains

  Arguments:

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

Examples

Change schedule properties

```
schedule[3]:setValue("name", "My Schedule")
```

Get current target temperature computed by thermal schedule

```
local target = schedule[2]:getValue("current_target_temperature")

print(target) -- [PRINT] 230
```

Date and time

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:

    ◦ *integer* — 1–31

- `getMonth()`

  Month of year according to local time configured in system.

  Returns:

    ◦ *integer* — 1–12

- `getYear()`

  Year according to local time configured in system.

  Returns:

    ◦ *integer*

- `getSeconds()`

  Seconds according to local time configured in system.

  Returns:

    ◦ *integer* — 1–61 (> 59 in case of a [leap second](.))

- `getMinutes()`

  Minutes of hour according to local time configured in system.

  Returns:

    ◦ *integer* — integer, 0–59

- `getHours()`

  Hour according to local time configured in system.

  Returns:

- *integer* — 0–23

- `getWeekDay()`

  Day of week according to local time configured in system, where Sunday = 0, Monday = 1, ...

  Returns:

  - *integer* — 0–6

- `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 (Unix timestamp).

  Returns:

  - *integer*

- `getTimeZoneOffset()`

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

  Returns:

  - *integer*

- `isDaylightSavingTimeActive()`

  Return info whether DST is currently active or not.

  Returns:

  - *boolean*

- `getTimeOfDay()`

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

  Returns:

  - *integer* — 0--1439 minutes

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

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

> **Note**
>
> Without the `dateTime:changed()` subcondition, the `print` function would run on every event while the clock shows 7:30 on monday.

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
- `table` - holds table (object or array)

Methods

- `getName()`

  Return variable name.

  Returns:

   ◦ *(string)*

- `getDescription()`

  Return variable description.

  Returns:

   ◦ *string*

- `getType()`

  Return variable type. One of: `"boolean"`, `"integer"`, `"string"`, `"table"`.

  Returns:

   ◦ *string*

- `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, stop_propagation)`

  Sets value for object.

  Returns:

  - *reference to variable object*

  Arguments:

  - `value` *(any)* — variable type dependant value which should be set
  - `stop_propagation` *(boolean, optional)* — defines whether futher event propagation should be stopped (= `true`) or not (= `false` / empty). In other words, if = true, then `changed()` method will not return `true` (`lua_variable_state_changed` event will not be emitted) on modification and automation cycle won't be invoked. This may greatly improve performance if variable is frequently modified and doesn't require notifying another automation / custom devices instances about such changes.

- `setValueByPath(path, value, stop_propagation)`

  > **Note**
  >
  > This method is available for variables of type `table` only!

  Sets partial value using provided path.

  Returns:

  - *reference to variable object*

  Arguments:

  - `path` *(string)* — path to the value which should be set. Path is represented as a string with dot separated keys. For example: `key1.key2` or `key1[1].key2[2]` (for array). If the path doesn't exist, it will be created. If the path is invalid, error will be thrown.
  - `value` *(any)* — value which should be set in place pointed by path
  - `stop_propagation` *(boolean, optional)* — defines whether futher event propagation should be stopped (= `true`) or not (= `false` / empty). In other words, if = true, then `changed()` method will not return `true` (`lua_variable_state_changed` event will not be emitted) on modification and automation cycle won't be invoked. This may greatly improve performance if variable is frequently modified and doesn't require notifying another automation / custom devices instances about such changes.

- `save(stop_propagation)`

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

  Returns:

○ *boolean* — `true`, when the content was successfully saved, `false` otherwise

Arguments:

○ `stop_propagation` *(boolean, optional)* — defines whether futher event propagation should be stopped (= `true`) or not (= `false` / empty). In other words, if = true, then `changed()` method will not return `true` (`lua_variable_state_changed` event will not be emitted) on modification and automation cycle won't be invoked. This may greatly improve performance if variable is frequently modified and doesn't require notifying another automation / custom devices instances about such changes.

- `revert(stop_propagation)`

Copies current `default_value` to `value`.

Returns:

○ *boolean* — `true`, when the content was successfully reverted to default, `false` otherwise

Arguments:

○ `stop_propagation` *(boolean, optional)* — defines whether futher event propagation should be stopped (= `true`) or not (= `false` / empty). In other words, if = true, then `changed()` method will not return `true` (`lua_variable_state_changed` event will not be emitted) on modification and automation cycle won't be invoked. This may greatly improve performance if variable is frequently modified and doesn't require notifying another automation / custom devices instances about such changes.

Examples

Set variable values

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

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

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

--type: "table" with string keys (object)
variable[4]:setValue({ key = "value", other_key = "other_value" })

--type "table" with integer keys (array)
variable[4]:setValue({ 10, 20, 30, 40, 50 })
-- NOTE: modification of single element in table is not supported
```

Get variable values

```lua
-- type: "string"
print(variable[1]:getValue()) -- prints "New text"

-- type: "integer"
```

```lua
print(variable[2]:getValue()) -- prints 42

-- type: "boolean"
print(variable[3]:getValue()) -- prints true

-- type: "table" with string keys (object)
print(variable[4]:getValue())
    -- prints {"key":"value", "other_key":"other_value"} (json representation of table)
-- NOTE: access to single table element is possible
print(variable[4]:getValue().key) -- prints only "value"

-- type "table" with integer keys (array)
print(variable[4]:getValue())
    -- prints [10, 20, 30, 40, 50] (json representation of table)
-- NOTE: access to single table element is possible
print(variable[4]:getValue()[1]) -- prints only 10
```

Set table variable partial value by path

```lua
-- lets consider that our variable is already set to:
-- { int = 55 }

-- modify using number
variable[5]:setValueByPath("int", 100)
print(variable[5]:getValue().int) -- prints 100, table is now { int = 100 }

-- modify using bool
variable[5]:setValueByPath("int", true)
print(variable[5]:getValue().int) -- prints true, table is now { int = true }

-- modify using string
variable[5]:setValueByPath("int", "works")
print(variable[5]:getValue().int)
    -- prints works, table is now { int = "works" }

-- modify using table
variable[5]:setValueByPath("int", {
    key = "value",
    arr = { 11,22,33 },
    bool = false,
    obj = {
        arr = { 100, 200, 300 }
    }
})
print(variable[5]:getValue().int.key) -- prints "value"
print(variable[5]:getValue().int.arr[1]) -- prints 11
print(variable[5]:getValue().int.arr[2]) -- prints 22
print(variable[5]:getValue().int.arr[3]) -- prints 33
print(variable[5]:getValue().int.bool) -- prints false
print(variable[5]:getValue().int.obj.arr[1]) -- prints 100
print(variable[5]:getValue().int.obj.arr[2]) -- prints 200
print(variable[5]:getValue().int.obj.arr[3]) -- prints 300

--modify single nested prop by longer path
variable[5]:setValueByPath("int.obj.arr[1]", 999)
print(variable[5]:getValue().int.obj.arr[1])
    -- prints 999, rest of values remain not modified

--create nested prop if doesnt exist (path points to object key)
print(variable[5]:getValue().int.obj.new_prop) --prints nil (doesnt exist)

variable[5]:setValueByPath("int.obj.new_prop", "nested_value")
```

```
print(variable[5]:getValue().int.obj.new_prop) --prints "nested_value"

--create nested prop if doesnt exist (path points to array index)
print(variable[5]:getValue().int.obj.arr[4]) --prints nil (doesnt exist)

variable[5]:setValueByPath("int.obj.arr[4]", 888)
print(variable[5]:getValue().int.obj.arr[4]) --prints 888
```

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
```

Variable which saves itself

```
local function getNvm()
    local var = variable[108] --[[@as TableVariable]]
    return setmetatable(var, { __close = function (data)
        var:setValue(data, true):save(true)
    end })
end

local function toggleWithCooldown()
    local nvm <close> = getNvm()
    local now = os.time()

    -- toggle relay only if at least 10 minutes have passed since the last time
    if now > nvm.lastToggle - (10 * 60) then
        sbus[10]:call("toggle")
        nvm.lastToggle = now
    end
    -- nvm will save itself after leaving function toggleWithCooldown
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 the web application served through the central unit server.

They cannot be edited, but access to their methods is possible via scripts using `timer` container e.g. `timer[6]` gives you access to timer with **ID 6**.

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

> **Note**
>
> It is 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` *(integer)* — amount of time in preconfigured units

- `getElapsedTime()`

  Returns total elapsed time, depending on the timer state:

  Running
  > time since last `startFreeRun` / `start` call.

Stopped

    time counted until `stop` was called.

Elapsed

    time which was set as `start` argument.

Returns:

- *integer*

- `isElapsed()`

  Returns information if timer is source of current execution context e.g. 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

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

Start timer in stopwatch mode

```lua
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

```lua
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

```lua
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 e.g. 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 use up to 60 statistic points at a time. Every minute a point is added to available pool (up to 60 maximum). When the pool is exhausted, method calls will be ignored. See return status of `addPoint` function.
>
> Limit is set per execution context and statistic series name (e.g. custom device with ID 3 and parameter `temperature` has separate pool from custom device with ID 3 and parameter `humidity`).

Methods

- `addPoint(name, value, unit)`

  Adds point with value for object property.

  Returns:

  - *boolean* — true if a point was added (can fail if called when all points from pool used)

  Arguments:

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

Units

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

Similarly, point with value 5001 and unit with suffix `_x100` will be shown as 50.01 in the statistics page.

Unit	Object name
1 A	`unit.ampere`
1 mA	`unit.milliamp`

Unit	Object name
1 µA	`unit.microamp`
1 atm	`unit.atm`
0.1 bar	`unit.bar_x10`
0 / 1	`unit.bool_unit`
1 °C	`unit.celsius`
0.1 °C	`unit.celsius_x10`
1°	`unit.degree`
1 µg/m^3	`unit.micro_grams_per_m3`
1 Hz	`unit.hertz`
0.01 Hz	`unit.hertz_x100`
IAQ	`unit.indoor_air_quality`
1 J	`unit.joule`
0.1 J	`unit.joule_x10`
0.1 K	`unit.kelvin_x10`
1 L	`unit.litres`
1 L/h	`unit.liters_per_hour`
0.01 L/min	`unit.litres_per_minute_x100`
1 lx	`unit.lux`
1 m^3	`unit.m3`
1 m^3/h	`unit.m3_H`
1 m/s	`unit.metre_per_second`
1 mm	`unit.millimetre`
1 km/h	`unit.kilometre_per_hour`
1	`unit.null`
1 Pa	`unit.pascal`
1 kPa	`unit.kPa`
0.1 hPa	`unit.hectopascals_x10`
1 %/Hz	`unit.percent_per_hertz`
1 %	`unit.percent`
0.1 %	`unit.percent_x10`
1 PPM	`unit.ppm`
0.1 % RH	`unit.relative_humidity_x10`
1 RPM	`unit.rpm`
1 s	`unit.second`
1 ms	`unit.milliseconds`
1 var	`unit.var`
1 mvar	`unit.mVAr`
1 varh	`unit.VArh`
1 V	`unit.volt`
1 mV	`unit.millivolts`

Unit	Object name
1 VA	`unit.VA`
1 mVA	`unit.mVA`
1 W	`unit.watt`
1 W/m^2	`unit.watt_per_m2`
0.1 kW	`unit.kw_x10`
1 mW	`unit.milliwatt`
1 Wh	`unit.wh`
1 kWh	`unit.kwh`
1 PLN	`unit.pln`

Examples

Log temperature once per minute

Lua variable may be fed with value e.g. from HTTP calls in another automation.

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

Statistics Queries

User has possibility to get the statistics points from lua code.

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

> **Note**
>
> User can execute up to 5 queries simultaneously. Limit is set globally, so for all execution contexts. Queries slots are freed after query callback execution.

Methods

- `query(queryData)`

  Adds a statistics query to execution queue. Does not perform the query immediately. Use callback `onQueryResult` to get the results.

  Arguments:

  - `queryData` *(object)* — query parameters with given properties:

    - `property` *(string) required* - property name

      Must consists of alphanumerical characters or `_` or `.` only
      Minimum length: 1
      Maximum length: 128

    - `time_start` *(string or integer) optional* - Start time for requested data series. Can be Unix timestamp or ISO 8601-like time string. Will request for data from the beginning of the device/object data if not provided.
      Should be UTC time!

    - `time_end` *(string or integer) optional* - End time for requested data series. Can be Unix timestamp or ISO 8601-like time string. Current time used as `time_end` when not provided.
      Should be UTC time!

    - `sources` *(array of objects) optional* - Sources list for requested property statistics.

      - `id` *(integer) required* - source object ID
        Minimum: 1
        Maximum: 4294967295

      - `class` *(string) required* - source class (device class or global object name)
        One of: `scene`, `automation`, `energy_consumption`, `energy_production`, `energy_storage`, `wtp`, `tech`, `virtual`, `system_module`, `sbus`, `slink`, `lora`, `modbus`, `alarm_system`, `custom_device_module`

- `onQueryResult(func)`

Calls function passed in argument on HTTP message received. It caches the request result.

Arguments:

- `func` *(function)* — callback function which will be called when statistics data is retrieved from the database.

  Arguments:

  - `queryData` *(object)* — query data used to get the statistics. Object used in `query` method.

  - `status` *(boolean)* — indicates if the query succeeded. Query can fail if there is problem with connection to database, there is no statistics storage or there is some other unexpected error.

  - `result` *(array of objects)* — array of statistics data points retrieved from database. Objects with properties:

    - `id` *(integer)* — source ID

    - `class` *(string)* — source class

    - `name` *(string)* — property name

    - `unit` *(string)* — value unit

    - `value` *(number)* — value

Examples

Query for the temperature statistics from devices for the specific timestamps

```lua
local dev1 = wtp[3]
local dev2 = sbus[1]
local dev3 = modbus[2]

local queryData = {
    property = "temperature",
    time_start = 1745418251,
    time_end = 1745926579,
    sources = {
        {
            id = dev1:getValue("id"),
            class = dev1:getValue("class")
        },
        {
            id = dev2:getValue("id"),
            class = dev2:getValue("class")
        },
        {
            id = dev3:getValue("id"),
            class = dev3:getValue("class")
        }
    }
}

statistics:query(queryData)
```

Query for the humidity statistics from device every hour for the last hour and find the lowest and the highest value in callback

```lua
local device = wtp[5]

if dateTime:changed() then
    -- every hour
    if dateTime:getMinutes() == 0 then
        local queryData = {
            property = "humidity",
            -- from the last hour using ISO time (UTC is required)
            time_start = utils.time:toISO(dateTime:getTotalTime() - (1 * 60 *
                60), utils.time.utc),
            sources = {
                {
                    id = device:getValue("id"),
                    class = device:getValue("class")
                }
            }
        }
        statistics:query(queryData)
    end
end

-- callback for the statistics requests
statistics:onQueryResult(function (queryData, status, result)

    print("Statistics query for " .. queryData["property"] .. " and " ..
        (queryData["sources"] and #queryData["sources"] or 0) ..
        " sources status: " .. (status and "SUCCESS" or "FAILED"))

    -- do not process if failed
    if status == false then
        return
    end

    -- check if the request is the one we need
    -- check if property is humidity
    if queryData["property"] ~= "humidity" then
        return
    end

    -- check if this is one source, as we requested
    local sources = queryData["sources"]
    if sources == nil or #sources ~= 1 then
        return
    end

    -- check if source is requested device
    if (sources[1]["id"] ~= device:getValue("id")) or (sources[1]["class"] ~=
        device:getValue("class")) then
        return
    end

    -- sources match, check the results

    -- do not check if result empty
    if #result == 0 then
        print("No data from last hour!")
        return
    end

    local lowest = nil
    local highest = nil
```

```lua
    for _, point in pairs(result) do
        local value = point["value"]

        if lowest == nil or value < lowest then
            lowest = value
        end

        if highest == nil or value > highest then
            highest = value
        end
    end

    print("The lowest value of humidity from last hour: " .. tostring(lowest))
    print("The highest value of humidity from last hour: " .. tostring(highest))

end)
```

Sun Position

Global scope objects which will help user to do actions referring time for sunrise, sunset, dusk and dawk events. It is based on the location set in the system settings and current local time.

Available in all contexts.

You can access it using following objects:

- `dawn` - The moment when the first light appears in the sky before sunrise. During this time, there is enough light for basic outdoor activities without artificial illumination.
- `sunrise` - The moment when the sun first appears above the horizon in the morning.
- `sunset` - The moment when the sun disappears below the horizon in the evening.
- `dusk` - The time after sunset when natural light is fading but not yet completely dark. This event characterizes end of time that it is enough light for most outdoor activities.

Following methods are available for each object.

Methods

- `hour()`

  The hour when certain sun position event will occur for current day.

  Returns:

  ◦ *number* — integer in 0–23 range

- `minute()`

  The minute when certain sun position event will occur (minute in hour) for current day.

  Returns:

  ◦ *number* — integer in 0–59 range

- `timeOfDay()`

  The time in minutes of day (minute of day, since 00:00, e.g. 12:30 is equal to 750 == 12 * 60 + 30) when certain sun position event will occur for current day.

  Returns:

  ◦ *number* — integer in 0–1439 range

Examples

Get time of sunrise

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

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

```lua
-- same methods for the rest of objects

print("sunrise", sunrise:hour(), sunrise:minute(), sunrise:timeOfDay())
print("dawn", dawn:hour(), dawn:minute(), dawn:timeOfDay())
print("sunset", sunset:hour(), sunset:minute(), sunset:timeOfDay())
print("dusk", dusk:hour(), dusk:minute(), dusk:timeOfDay())
```

Catch sunrise event

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

Catch dawn event

```lua
if event.type == "dawn" then
    print("Dawn starts now!")
end
```

Catch sunset event

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

Catch dusk event

```lua
if event.type == "dusk" then
    print("Dusk starts now!")
end
```

Do an action 2 hours after sunrise

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

    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)
        -- 2 hours == 2 * 60 * 60 seconds
end
```

System

Global scope object for accessing system data.

Available in all contexts. You can access 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)*

- `network()`

  Returns network info object.

  > **Note**
  >
  > This is blocking call, do not use it frequently, as it may reduce your Lua (or overall system) responsiveness.

  Returns:

  - *table?* — object with network info with following properties or nil if not available:

    - `ethernet` *(table)*

      - `connected` *(boolean)* — Status of the Ethernet connection
      - `dhcp` *(boolean)* — Status of the DHCP client (on / off)
      - `dns` *(string)*
      - `gateway` *(string)*
      - `ip` *(string)*
      - `mask` *(string)*
      - `mac` *(string)*

    - `wifi` *(table)*

      - `connected` *(boolean)* — Status of the Wi-Fi connection
      - `dhcp` *(boolean)* — Status of the DHCP client (on / off)

- dns *(string)*
- gateway *(string)*
- ip *(string)*
- mask *(string)*
- mac *(string)*
- secure *(boolean)* — Tells whether current Wi-Fi network is password protected
- signal *(number)* — Current Wi-Fi network signal strength
- ssid *(string)* — Current Wi-Fi network name

- hostname()

  Returns actual hostname of central.

  > **Note**
  >
  > This is blocking call, do not use it frequently, as it may reduce your Lua (or overall system) responsiveness.

  Returns:

  - *string?* — hostname or nil if not available

- uid()

  Returns central UID.

  Returns:

  - *string* — Unique central identifier.

  Example:

  ```lua
  local uid = system:uid()
  local versionMap = {
      ['1'] = "Sinum",
      ['2'] = "Sinum Pro",
      ['3'] = "Sinum Lite",
  }

  print(versionMap[uid:sub(4, 4)], uid)
  ```

- reboot()

  Requests system reboot.

  Returns:

  - *boolean* — status of request, true when command accepted and will be executed soon

- shutdown()

  Requests system shutdown.

  Returns:

  - *boolean* — status of request, true when command accepted and will be executed soon

Examples

Print version info

```
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)

-- [PRINT] Major, 1
-- [PRINT] Minor, 10
-- [PRINT] Maintenance, 0
-- [PRINT] Env,        -- empty means production env
-- [PRINT] Build, 1
-- [PRINT] SemVer, 1.10.0
```

Turn on device if networks are not available

```
-- check once per minute
if dateTime:changed() then
  local network = system:network()

  if not network.ethernet.connected and not network.wifi.connected then
    sbus[44]:call("turn_on")
  end
end
```

Print hostname

```
print("Hostname ",  system:hostname())

-- [PRINT] Hostname , sinum
```

Users

Global scope object for accessing users e.g. when you want to send notification and don't know user ID but name.

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

Methods

- `all()`

  Returns collection of all available users.

  Returns:

  ○ *(table/nil)* — collection (table) of objects or nil if not available:
    - *(table)* — object with user data with following properties:
      ○ `id` *(number)*
      ○ `type` *(string)*
      ○ `role` *(string)*
      ○ `username` *(string)*
      ○ `email` *(string)*
      ○ `language` *(string)*

- `get(nameOrId)`

  Returns certain user data.

  > **Note**
  >
  > This is a blocking call, do not use it frequently as it may reduce your Lua (or overall system) responsiveness.

  Arguments:

  ○ `nameOrId` *(string/number)* — username/email of user or numeric ID to lookup

  Returns:

  ○ *(table/nil)* — object with user data with following properties or nil if not available:
    - `id` *(number)*
    - `type` *(string)*
    - `role` *(string)*
    - `username` *(string)*
    - `email` *(string)*
    - `language` *(string)*

Examples

Print all users data

```lua
local all = users:all()

if all == nil then
    print("Users not available")
else
    for _, user in pairs(all) do
        print(user.id, user.type, user.role, user.username, user.email,
            user.language)
    end
end

-- [PRINT] 1, local_user, SUPER_ADMIN, admin, admin@sinum.tech, pl
-- [PRINT] 7, cloud_user, USER, user, user@sinum.tech, en
```

Query user by ID

```lua
local user = users:get(7)

if user == nil then
    print("User not found")
else
    print(user.id, user.type, user.role, user.username, user.email,
        user.language)
end

-- [PRINT] 7, cloud_user, USER, user, user@sinum.tech, en
```

Query user by username

```lua
local user = users:get("admin")

if user == nil then
    print("User not found")
else
    print(user.id, user.type, user.role, user.username, user.email,
        user.language)
end

-- [PRINT] 1, local_user, SUPER_ADMIN, admin, admin@sinum.tech, pl
```

Query user by email

```lua
local user = users:get("user@sinum.tech")

if user == nil then
    print("User not found")
else
    print(user.id, user.type, user.role, user.username, user.email,
        user.language)
end
```

```
-- [PRINT] 7, cloud_user, USER, user, user@sinum.tech, en
```

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

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `enabled` *(boolean)*

  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.

- `associations.outdoor_sensor` *(device)*

  Reference to outdoor temperature sensor. Has to be reference to **temperature_sensor** device type.

- `available` *(boolean)*

  Indicates if weather feature is turned on and weather data is available (received from weather service).

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

  Current method to obtain weather data. Possible values are:

  - `"api"` — data is retrieved from weather service
  - `"lua"` — data is provided by user using lua scripts

- `city_name` *(string)*

  City name displayed by application. Can be changed only when parameter `access_type` is set to `lua`.

- `sunrise` *(integer)*

  Timestamp of sunrise for current day. Can be used in determining day or night in devices. Can be changed only when parameter `access_type` is set to `lua`.

- `sunset` *(integer)*

  Timestamp of sunset for current day. Can be used in determining day or night in devices. Can be changed only when parameter `access_type` is set to `lua`.

Methods

- `current()`

Returns weather object containing information about current weather conditions.

Returns:

- *userdata* — weather data object

- `hourly()`

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

Returns:

- *userdata[]* — table of weather data objects

- `changed()`

Check if weather data has changed.

Returns:

- *boolean*

- `setCurrentWeatherData(data)`

Set the current weather conditions. Parameter `access_type` needs to be set to `lua` to take effect.

Arguments:

- `data` *(table)* - object with weather conditions data:
  - `timestamp` *(integer)* - current conditions timestamp. Parameter is optional - will be set to current timestamp when not provided.
  - `temperature` *(integer)* - current temperature in 0.1 °C
  - `feels_like` *(integer)* - current feels like temperature in 0.1 °C
  - `humidity` *(integer)* - current humidity in percent
  - `pressure` *(integer)* - current pressure in hPa
  - `wind` *(table)* - object with wind conditions data
    - `speed` *(integer)* - current wind speed in 0.1 km/h
    - `gust` *(integer)* - current wind gust speed in 0.1 km/h
    - `degrees` *(integer)* - current wind direction in meteorological degrees, 0° – 359°
  - `rain` *(integer)* - current rain volume that is predicted to fall in 0.1 mm/h
  - `snow` *(integer)* - current snow volume that is predicted to fall in 0.1 mm/h
  - `cloud_coverage` *(integer)* - current cloud coverage in percentage
  - `precipitation_probability` *(integer)* - probability of precipitation in percentage
  - `weather` *(table)* - object with weather information
    - `main` *(string)* - general weather description
    - `description` *(string)* - detailed weather description
    - `id` *(integer)* - weather condition identifier, one of values defined in enum `weather_id` (check below)

- `setHourlyWeatherData(hoursData)`

Set the hourly forecast weather conditions. Parameter `access_type` needs to be set to `lua` to take effect.

Arguments:

- `hoursData` *(table)* - array of weather conditions objects with data:
  - `timestamp` *(integer)* - hour conditions timestamp. Parameter is optional - will be set to current timestamp plus hour number hours when not provided.
  - `temperature` *(integer)* - hour temperature in 0.1 °C
  - `feels_like` *(integer)* - hour feels like temperature in 0.1 °C
  - `humidity` *(integer)* - hour humidity in percent
  - `pressure` *(integer)* - hour pressure in hPa
  - `wind` *(table)* - object with wind conditions data
    - `speed` *(integer)* - hour wind speed in 0.1 km/h
    - `gust` *(integer)* - hour wind gust speed in 0.1 km/h
    - `degrees` *(integer)* - hour wind direction in meteorological degrees, 0° – 359°
  - `rain` *(integer)* - hour rain volume that is predicted to fall in 0.1 mm/h
  - `snow` *(integer)* - hour snow volume that is predicted to fall in 0.1 mm/h
  - `cloud_coverage` *(integer)* - hour cloud coverage in percentage
  - `precipitation_probability` *(integer)* - probability of precipitation in percentage
  - `weather` *(table)* - object with weather information
    - `main` *(string)* - general weather description
    - `description` *(string)* - detailed weather description
    - `id` *(integer)* - weather condition identifier, one of values defined in enum `weather_id` (check below)

- `setDailyWeatherData(daysData)`

  Set the daily forecast weather conditions. Parameter `access_type` needs to be set to `lua` to take effect.

  Arguments:

  - `daysData` *(table)* - array of weather conditions objects with data:
    - `timestamp` *(integer)* - day conditions timestamp. Parameter is optional - will be set to current timestamp plus day number days when not provided.
    - `temperature` *(object)* - temperature data object for given day with parameters:
      - `min` *(integer)* - minimum temperature predicted for a day in 0.1 °C
      - `max` *(integer)* - maximum temperature predicted for a day in 0.1 °C
      - `day` *(integer)* - average temperature predicted for a day in 0.1 °C
    - `feels_like` *(integer)* - feels like temperature data object for given day with parameters:
      - `day` *(integer)* - average feels like temperature predicted for a day in 0.1 °C
    - `humidity` *(integer)* - day humidity in percent
    - `pressure` *(integer)* - day pressure in hPa

- `wind` *(table)* - object with wind conditions data
  - `speed` *(integer)* - day wind speed in 0.1 km/d
  - `gust` *(integer)* - day wind gust speed in 0.1 km/d
  - `degrees` *(integer)* - day wind direction in meteorological degrees, 0° – 359°
- `rain` *(integer)* - day rain volume that is predicted to fall in 0.1 mm/d
- `snow` *(integer)* - day snow volume that is predicted to fall in 0.1 mm/d
- `cloud_coverage` *(integer)* - day cloud coverage in percentage
- `precipitation_probability` *(integer)* - probability of precipitation in percentage
- `weather` *(table)* - object with weather information
  - `main` *(string)* - general weather description
  - `description` *(string)* - detailed weather description
  - `id` *(integer)* - weather condition identifier, one of values defined in enum `weather_id` (check below)

- `setHourWeatherData(hourNumber, data)`

  Update the hour forecast data for a given hour. Hour must already exists in data (set by `setHourlyWeatherData` method). Parameter `access_type` needs to be set to `lua` to take effect.

  Arguments:

  - `hourNumber` *(integer)* - hour number that will be updated where 0 is first hour, 1 is second hour etc.
  - `data` *(table)* - object with weather conditions data:
    - `timestamp` *(integer)* - hour conditions timestamp. Parameter is optional - will be set to current timestamp plus hour number hours when not provided.
    - `temperature` *(integer)* - hour temperature in 0.1 °C
    - `feels_like` *(integer)* - hour feels like temperature in 0.1 °C
    - `humidity` *(integer)* - hour humidity in percent
    - `pressure` *(integer)* - hour pressure in hPa
    - `wind` *(table)* - object with wind conditions data
      - `speed` *(integer)* - hour wind speed in 0.1 km/h
      - `gust` *(integer)* - hour wind gust speed in 0.1 km/h
      - `degrees` *(integer)* - hour wind direction in meteorological degrees, 0° – 359°
    - `rain` *(integer)* - hour rain volume that is predicted to fall in 0.1 mm/h
    - `snow` *(integer)* - hour snow volume that is predicted to fall in 0.1 mm/h
    - `cloud_coverage` *(integer)* - hour cloud coverage in percentage
    - `precipitation_probability` *(integer)* - probability of precipitation in percentage
    - `weather` *(table)* - object with weather information
      - `main` *(string)* - general weather description
      - `description` *(string)* - detailed weather description
      - `id` *(integer)* - weather condition identifier, one of values defined in enum `weather_id` (check below)

- `setDayWeatherData(dayNumber, data)`

  Update the day forecast data for a given day. Day must already exists in data (set by `setDailyWeatherData` method). Parameter `access_type` needs to be set to `lua` to take effect.

  Arguments:

  - `dayNumber` *(integer)* - day number that will be updated where 0 is first day, 1 is second day etc.
  - `data` *(table)* - object with weather conditions data:
    - `timestamp` *(integer)* - day conditions timestamp. Parameter is optional - will be set to current timestamp plus day number days when not provided.
    - `temperature` *(object)* - temperature data object for given day with parameters:
      - `min` *(integer)* - minimum temperature predicted for a day in 0.1 °C
      - `max` *(integer)* - maximum temperature predicted for a day in 0.1 °C
      - `day` *(integer)* - average temperature predicted for a day in 0.1 °C
    - `feels_like` *(integer)* - feels like temperature data object for given day with parameters:
      - `day` *(integer)* - average feels like temperature predicted for a day in 0.1 °C
    - `humidity` *(integer)* - day humidity in percent
    - `pressure` *(integer)* - day pressure in hPa
    - `wind` *(table)* - object with wind conditions data
      - `speed` *(integer)* - day wind speed in 0.1 km/d
      - `gust` *(integer)* - day wind gust speed in 0.1 km/d
      - `degrees` *(integer)* - day wind direction in meteorological degrees, 0° – 359°
    - `rain` *(integer)* - day rain volume that is predicted to fall in 0.1 mm/d
    - `snow` *(integer)* - day snow volume that is predicted to fall in 0.1 mm/d
    - `cloud_coverage` *(integer)* - day cloud coverage in percentage
    - `precipitation_probability` *(integer)* - probability of precipitation in percentage
    - `weather` *(table)* - object with weather information
      - `main` *(string)* - general weather description
      - `description` *(string)* - detailed weather description
      - `id` *(integer)* - weather condition identifier, one of values defined in enum `weather_id` (check below)

- `setWeatherData(data)`

  Update all weather data components, current and hourly, daily forecasts. Parameter `access_type` needs to be set to `lua` to take effect.

  Arguments:

  - `data` *(object)* - object with weather data components:
    - `current` *(object)* - object with current weather conditions data as defined in `setCurrentWeatherData` method
    - `hourly` *(table)* - array of objects with hourly forecast weather conditions data as

defined in `setHourlyWeatherData`
- `daily` *(table)* - array of objects with daily forecast weather conditions data as defined in `setDailyWeatherData`

Weather object

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

Methods

- `timestamp()`

  Unix timestamp in local time of current or forecast weather conditions.

  Returns:

  - *integer* — timestamp

- `weather()`

  General weather description (empty string when data is not available yet).

  Returns:

  - *string* — one of: `""` `"Clear"` `"Clouds"` `"Rain"` `"Snow"`

- `weatherId()`

  Weather condition identificator.

  Returns:

  - *integer* — one of values defined in enum `weather_id` (check below).

- `temperature()`

  Measured or forecast temperature.

  Returns:

  - *integer* — Temperature in 0.1 °C

- `feelsLikeTemperature()`

  Measured or forecast feels like temperature.

  Returns:

  - *integer* — Feels-like temperature in 0.1 °C

- `humidity()`

  Measured or forecast humidity in percent.

  Returns:

  - *integer* — Relative humidity in 1 %

- `pressure()`

  Measured or forecast pressure.

  Returns:

  - *integer* — Atmospheric pressure in 1 hPa

- `windSpeed()`

  Measured or forecast wind speed.

  Returns:

  - *integer* — Speed in 0.1 km/h

- `windDegrees()`

  Measured or forecast wind direction in meteorological degrees.

  Returns:

  - *integer* — Azimuth, 0° – 359°

- `rain()`

  Rain volume that is predicted to fall.

  Returns:

  - *integer* — Rainfall in 0.1 mm/h

- `snow()`

  Snow volume that is predicted to fall.

  Returns:

  - *integer* — Snowfall in 0.1 mm/h

- `cloudCoverage()`

  Cloud coverage in percentage.

  Returns:

  - *integer* — Coverage in 1 %

> **Note**
>
> When weather data is not yet available, most methods return 0. To check for data availability, use `:weather()` method. It will return an empty string, which is distinguishable from correct values.
> Data can be unavailable if the weather service was enabled just now and the Sinum central has yet to download the weather data.

Weather conditions

Weather conditions identifiers are stored in enum with global scope `weather_id`. To use or check given condition use for example: `weather_id.clearSky`. Available conditions are:

- `thunderstormWithLightRain`

- `thunderstormWithRain`
- `thunderstormWithHeavyRain`
- `lightThunderstorm`
- `thunderstorm`
- `heavyThunderstorm`
- `raggedThunderstorm`
- `thunderstormWithLightDrizzle`
- `thunderstormWithDrizzle`
- `thunderstormWithHeavyDrizzle`
- `lightIntensityDrizzle`
- `drizzle`
- `heavyIntensityDrizzle`
- `lightIntensityDrizzleRain`
- `drizzleRain`
- `heavyIntensityDrizzleRain`
- `showerRainAndDrizzle`
- `heavyShowerRainAndDrizzle`
- `showerDrizzle`
- `lightRain`
- `moderateRain`
- `heavyIntensityRain`
- `veryHeavyRain`
- `extremeRain`
- `freezingRain`
- `lightIntensityShowerRain`
- `showerRain`
- `heavyIntensityShowerRain`
- `raggedShowerRain`
- `lightSnow`
- `snow`
- `heavySnow`
- `sleet`
- `lightShowerSleet`
- `showerSleet`
- `lightRainAndSnow`
- `rainAndSnow`
- `lightShowerSnow`
- `showerSnow`
- `heavyShowerSnow`

- `mist`
- `smoke`
- `haze`
- `sandDustWhirls`
- `fog`
- `sand`
- `dust`
- `volcanicAsh`
- `squalls`
- `tornado`
- `clearSky`
- `fewClouds11to25`
- `scatteredClouds25to50`
- `brokenClouds51to84`
- `overcastClouds85to100`

Examples

Read weather data on update

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

Close the blind when there is strong wind and rain

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

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

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

if weather:changed() then
    local 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
```

Enable weather and assign temperature sensor

```lua
local sensor = sbus[12]

weather:setValue("enabled", true)
weather:setValue("associations.outdoor_sensor", sensor)
```

Check if weather is available

```lua
local available = weather:getValue("available")
```

Change weather access to lua and set the weather conditions

```lua
weather:setValue("access_type", "lua")

-- set current weather conditions
weather:setCurrentWeatherData({
    temperature = 123,
    feels_like = 134,
    humidity = 65,
    pressure = 1013,
    wind = {
        speed = 5,
        gust = 15,
        degrees = 180
    },
    rain = 0,
    snow = 0,
    precipitation_probability = 0,
    cloud_coverage = 0,
    weather = {
        id = weather_id.clearSky
        main = "Clear",
        description = "Clear sky"
    }
})
```

```lua
-- set hourly forecast conditions
weather:setHourlyWeatherData({
    {
        temperature = 136,
        feels_like = 154,
        humidity = 65,
        pressure = 1013,
        wind = {
            speed = 5,
            gust = 15,
            degrees = 180
        },
        rain = 0,
        snow = 0,
        precipitation_probability = 0,
        cloud_coverage = 0,
        weather = {
            id = weather_id.clearSky
            main = "Clear",
            description = "Clear sky"
        }
    },
    {
        temperature = 147,
        feels_like = 166,
        humidity = 65,
        pressure = 1013,
        wind = {
            speed = 3,
            gust = 12,
            degrees = 180
        },
        rain = 2,
        snow = 0,
        precipitation_probability = 20,
        cloud_coverage = 77,
        weather = {
            id = weather_id.lightRain
            main = "Rain",
            description = "Light rain"
        }
    },
    {
        temperature = 133,
        feels_like = 156,
        humidity = 77,
        pressure = 1010,
        wind = {
            speed = 12,
            gust = 22,
            degrees = 166
        },
        rain = 33,
        snow = 0,
        precipitation_probability = 100,
        cloud_coverage = 100,
        weather = {
            id = weather_id.moderateRain
            main = "Rain",
            description = "Rain"
        }
    }
})

weather:setDailyWeatherData({
```

```
    {
        temperature = {
            day = 153,
            min = 105,
            max = 188
        },
        feels_like = { day = 130 },
        humidity = 80,
        pressure = 1005,
        wind = {
            speed = 12,
            gust = 20,
            degrees = 200
        },
        rain = 55,
        snow = 0,
        cloud_coverage = 100,
        weather = {
            main = "Rain",
            description = "Rain",
            id = weather_id.moderateRain
        }
    },
    {
        temperature = {
            day = 132,
            min = 92,
            max = 142
        },
        feels_like = { day = 110 },
        humidity = 92,
        pressure = 1012,
        wind = {
            speed = 7,
            gust = 15,
            degrees = 150
        },
        rain = 122,
        snow = 0,
        cloud_coverage = 100,
        weather = {
            main = "Rain",
            description = "Heavy rain",
            id = weather_id.heavyIntensityRain
        }
    }
})
```

Update single parameter in current weather conditions

```
-- only temperature changed, the rest stays as before
weather:setCurrentWeatherData({
    temperature = 116,
})
```

Update single hour weather conditions

```lua
-- update only first hour
local hourNumber = 0

weather:setHourWeatherData(hourNumber,
    {
        temperature = 136,
        feels_like = 154,
        humidity = 65,
        pressure = 1013,
        wind = {
            speed = 5,
            gust = 15,
            degrees = 180
        },
        rain = 0,
        snow = 0,
        precipitation_probability = 0,
        cloud_coverage = 0,
        weather = {
            id = weather_id.clearSky
            main = "Clear",
            description = "Clear sky"
        }
    })
```

Update single day weather conditions

```lua
-- update only second day
local dayNumber = 1
weather:setDailyWeatherData(1,
    {
        temperature = {
            day = 153,
            min = 105,
            max = 188
        },
        feels_like = { day = 130 },
        humidity = 80,
        pressure = 1005,
        wind = {
            speed = 12,
            gust = 20,
            degrees = 200
        },
        rain = 55,
        snow = 0,
        cloud_coverage = 100,
        weather = {
            main = "Rain",
            description = "Rain",
            id = weather_id.moderateRain
        }
    })
```

Update all weather conditions using setWeatherData method

```lua
weather:setValue("access_type", "lua")
weather:setWeatherData({
    current = {
        temperature = 123,
        feels_like = 134,
        humidity = 65,
        pressure = 1013,
        wind = {
            speed = 5,
            gust = 15,
            degrees = 180
        },
        rain = 0,
        snow = 0,
        precipitation_probability = 0,
        cloud_coverage = 0,
        weather = {
            id = weather_id.clearSky
            main = "Clear",
            description = "Clear sky"
        }
    },

    hourly = {
        {
            temperature = 136,
            feels_like = 154,
            humidity = 65,
            pressure = 1013,
            wind = {
                speed = 5,
                gust = 15,
                degrees = 180
            },
            rain = 0,
            snow = 0,
            precipitation_probability = 0,
            cloud_coverage = 0,
            weather = {
                id = weather_id.clearSky
                main = "Clear",
                description = "Clear sky"
            }
        },
        {
            temperature = 147,
            feels_like = 166,
            humidity = 65,
            pressure = 1013,
            wind = {
                speed = 3,
                gust = 12,
                degrees = 180
            },
            rain = 2,
            snow = 0,
            precipitation_probability = 20,
            cloud_coverage = 77,
            weather = {
                id = weather_id.lightRain
                main = "Rain",
                description = "Light rain"
            }
```

```
        },
        {
            temperature = 133,
            feels_like = 156,
            humidity = 77,
            pressure = 1010,
            wind = {
                speed = 12,
                gust = 22,
                degrees = 166
            },
            rain = 33,
            snow = 0,
            precipitation_probability = 100,
            cloud_coverage = 100,
            weather = {
                id = weather_id.moderateRain
                main = "Rain",
                description = "Rain"
            }
        }
    },

    daily = {
        {
            temperature = {
                day = 153,
                min = 105,
                max = 188
            },
            feels_like = { day = 130 },
            humidity = 80,
            pressure = 1005,
            wind = {
                speed = 12,
                gust = 20,
                degrees = 200
            },
            rain = 55,
            snow = 0,
            cloud_coverage = 100,
            weather = {
                main = "Rain",
                description = "Rain",
                id = weather_id.moderateRain
            }
        },
        {
            temperature = {
                day = 132,
                min = 92,
                max = 142
            },
            feels_like = { day = 110 },
            humidity = 92,
            pressure = 1012,
            wind = {
                speed = 7,
                gust = 15,
                degrees = 150
            },
            rain = 122,
            snow = 0,
            cloud_coverage = 100,
            weather = {
```

```
                main = "Rain",
                description = "Heavy rain",
                id = weather_id.heavyIntensityRain
            }
        }
    }
})
```

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` *(integer, sequence)* — optional parameter which allows specifying user/users (cloud user ID or local super admin ID) which will receive a notification. Can be single ID number or table of IDs. 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` *(integer, sequence)* — optional parameter which allows specifying user/users (cloud user ID or local super admin ID) which will receive a notification. Can be single ID number or table of IDs. 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` *(integer, sequence)* — optional parameter which allows specifying user/users (cloud user ID or local super admin ID) which will receive a notification. Can be single ID number or table of IDs. 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
    local sensor = wtp[3]
    local airQuality = sensor:getValue("air_quality")
    if utils.table:indexOf({
      "poor",
      "unhealthy",
      "very_unhealthy"
      }, airQuality)
    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
```

Toasts

Global scope object which allows user to show custom toasts / snackbars from Lua scripts in Sinum application.

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

Methods

- `success()`

  Creates a toast object which can be used to report completion of an action.

  Returns: ToastInfo

  Success toast.

- `failure()`

  Creates a toast object which can be used to report that an action could not be completed.

  Returns: ToastInfo

  Failure toast.

- `info()`

  Creates a toast object which can be used to report additional information.

  Returns: ToastInfo

  Information toast

- `warning()`

  Creates a toast object which can be used to report non-critical errors.

  Returns: ToastInfo

  Warning toast

ToastInfo Methods

These methods allow customizing text displayed in the toast and also specify target users. Calling `show` before setting any text will **fail** with an error.

- `text(plainText)`

  Sets text passed as argument to be displayed in the toast.

  Arguments:

  - `plainText` *(string)* — text to be displated in the toast

  Returns: ToastInfo

- `textID(id, params)`

  Sets text from translation database to be displayed in the toast. Text will be displayed in the language selected by user. Optionally you can also pass parameters to text.

  Arguments:

  - `idOrIdt` *(integer/string)* — numeric ID or string IDT of the text in database

  - `params` *(array-like table)* — list of string parameters to be injected into text (optional)

  Returns: ToastInfo

- `users(users)`

  Allows to specify users the toast will be displated for. Will show to all users if not specified.

  Arguments:

  - `users` *(integer, array-like table)* — user/users (cloud user ID or local super admin ID) which will receive a toast. Can be single ID number or table of IDs.

  Returns: ToastInfo

- `show()`

  Triggers the toast to show. Can be called when a text is set.

Examples

Show success toast with plain text to all users

```
-- will show green toast with text "Device started successfully"
toast:success()
      :text('Device started successfully')
      :show()
```

Show failure toast with translated text to users #1 and #3

```
-- will show red toast with text "No Modbus communication"
toast:failure()
     :textID(3679)
     :users({1,3})
     :show()
```

Show warning toast with translated text and params to user #1

```
-- will show orange toast with text
-- "Parameter name should be equal to Modbus"
toast:warning()
     :textID('IDT_PARAMETR_POWINIEN_BYC_ROWNY', {'name', 'Modbus'})
     :users(1)
     :show()
```

Deferred actions

Sometimes there is a need to perform an action after some time, e.g. when we want to turn on the light and then turn it off several seconds later after motion is detected by the motion sensor.

This can be done in many ways, e.g. using global timers or setValueAfter methods or using *deferred actions*.

This chapter will describe *deferred actions*, a universal tool for scheduling function execution in the future.

Utility is available as two global function. It has a similar API to a utility used in JavaScript, called `setTimeout`.

Methods

- `setTimeout(function, time, arg)`

  Schedules `function` to be executed after `time` with `arg` as argument.

  > **Note**
  >
  > Local scope variables are not preserved, if your action needs context variables, use `arg`! (See examples)

  Arguments:

  - `function` *(function)* — function to be executed

  - `time` *(number)* — the time, in seconds that the system should wait before the specified function is executed. Should be not less than 0.1 seconds.

  - `arg` *(any?)* — optional parameter which allows passing context as an argument to the function

  Returns:

  - *(string)* — six characters long ID, which can be used to cancel scheduled action before it executes

- `clearTimeout(id)`

  Cancels `function` before it is executed using ID.

  Arguments:

  - `id` *(string)* — six characters long ID, generated by `setTimeout`

  Returns:

  - *(boolean)* — information whether function was cancelled (`true`) or not (`false`)

Examples

Turn on the light and then turn it off after 5 secs

```
wtp[15]:call("turn_on")
setTimeout(function()
  wtp[15]:call("turn_off")
end, 5)
```

Turn on the light and then turn it off after 500 milliseconds

```
wtp[15]:call("turn_on")
setTimeout(function()
  wtp[15]:call("turn_off")
end, 0.5)
```

Blink the light 2 times in a row (nested `setTimeout` calls)

```
-- call this immediately
wtp[15]:call("turn_on")
setTimeout(function()
  -- call this after 1 sec
  wtp[15]:call("turn_off")
  setTimeout(function()
    -- call this after another 1 sec to prev action
    wtp[15]:call("turn_on")
    setTimeout(function()
      -- call this after another 1 sec to prev action
      tp[15]:call("turn_off")
    end, 1)
  end, 1)
end, 1)
```

> **Note**
>
> Here, first `setTimeout` will "wait" for 1 sec, then change relay state and schedule next `setTimeout` to "wait" again.

Pass custom context variable to future function call

```
local thisGonnaVanish = 12345;

setTimeout(function(arg)
  print(arg, thisGonnaVanish)
end, 5, thisGonnaVanish)

-- prints 12345, nil
```

> **Note**
>
> Local scope variable `thisGonnaVanish` disappeared, but its value was passed as
> `arg`, thus its preserved this way.

Schedule and then cancel action

```lua
local id = setTimeout(function(arg)
  print("Hello!")
end, 5)

if clearTimeout(id) then
  print("Bye!")
else
  print("Too slow!")
end


-- prints Bye!
```

> **Note**
>
> Schedule action was cancelled before it executed, so such script will never print
> `Hello!`

Translations

Global scope object for fetching translations' data.

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

Methods

- `get(idOrIdt, language)`

  Fetches translation for given `id` or `idt` in specified `language`.

  Arguments:

  - `idOrIdt` *(integer/string)* — numeric id or string idt of the translation

  - `language` *(string)* — language code of the translation. One of: `en`, `pl`, `cs`, `sk`, `de`, `ru`, `ro`, `hu`, `nl`, `lt`.

  Returns:

  - *(string)* — translation for given `id` or `idt` in specified `language` or empty string if not found.

Examples

Get the translation for the given id

```
print(translations:get(280, "en")) -- [PRINT] "Flow function"
print(translations:get(11870, "cs")) -- [PRINT] "Správa pater"
```

Get the translation for given idt

```
print(translations:get("IDT_FUNKCJA_PRZEPLYWU", "en"))
    -- [PRINT] "Flow function"
print(translations:get("IDT_ZARZADZANIE_PIETRAMI", "cs"))
    -- [PRINT] "Správa pater"
```

Basic library extensions

Various extensions to the basic library are present in the interpreter.

Value casting

`asInt16(value)`

Interprets value as a 16-bit signed integer.

Returns:

- *number* - conversion result

Arguments:

- `value` *(number)* — 16-bit value to convert

`asFloat(msw, lsw)`

Converts two 16-bit values into a IEEE754 binary32 (float) number.

Returns:

- *number* - conversion result

Arguments:

- `msw` *(number)* — most significant word
- `lsw` *(number)* — least significant word

`asInt32(msw, lsw)`

Converts two 16-bit values into a 32-bit signed integer.

Returns:

- *number* - conversion result

Arguments:

- `msw` *(number)* — most significant word
- `lsw` *(number)* — least significant word

`asUint32(msw, lsw)`

Converts two 16-bit values into a 32-bit unsigned integer.

Returns:

- *number* - conversion result

Arguments:

- `msw` *(number)* — most significant word

- `lsw` *(number)* — least significant word

Execution context

`context()`

Returns information about current execution context.

Returns:

- *userdata* - context object with following properties:
  - `type` *(string)* — script source: one of `automation`, `scene`, `custom_device`
  - `id` *(number)* — ID of the script source
  - `name` *(string)* — user provided name of the script source

```lua
local cx = context()
print("This is " .. cx.type .. "[" .. tostring(cx.id) .. "]")
print("also known as", '"' .. cx.name .. '"')

-- sample output:
-- [PRINT] This is automation[6]
-- [PRINT] also known as "My automation"
```

Localization

Global scope object which will help user to do actions referring on localization.

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

Methods

- `longitude()`

  Returns longitude setup for central unit.

  Returns:
  - *number* — longitude

- `latitude()`

  Returns latitude setup for central unit.

  Returns:
  - *number* — latitude

Examples

Read current localization

```
print("Current longitude:")
print(localization:longitude())
print("Current latitude:")
print(localization:latitude())
```

Libraries

Additional Lua 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:

  ◦ *any*

  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"}
```

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:

```xml
<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:

```lua
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
```

hash

It is possible to calculate various hashes in Lua interpreter.

Methods

- `hash:sha256(input)`

  Calculates the SHA-256 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
```

base64

It is possible to encode and decode Base64 data in Lua interpreter.

Methods

- `base64:encode(input)`

  Encodes input string to base64.

  Returns:

  - *(string)*

  Arguments:

  - `input_string` *(string)*

- `base64:decode(input)`

  Decodes input string from base64.

  Returns:

  - *(string)*

  Arguments:

  - `input` *(string)*

Example

```lua
local encoded = base64:encode("Hello Base64!")
local decoded = base64:decode(encoded)

print(encoded)
-- SGVsbG8gQmFzZTY0IQ==

print(decoded)
-- Hello Base64!
```

Utilities

The Lua interpreter is supplied with a utility package which implements frequently used algorithms efficiently.

Utility package is available as `utils` global.

Basic

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:

```
-- synchronize multiple thermostats
utils:integrateProperty('target_temperature', { wtp[2], wtp[3], wtp[8] })
```

`utils:stairLight(devices)`

Copies change of `state` property from one device to other ones. Equivalent to
`utils:integrateProperty('state', devices)`.

Arguments:

- `devices` *(table)* — sequence of devices to integrate

Example:

```
-- synchronize multiple light switches
utils:stairLight { wtp[2], wtp[3], wtp[8] }
```

Color operations

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

Representation

`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 ⟨0; 255⟩ fixed point channel values to ⟨0; 1⟩ float values.

Returns:

- `red` *(number)* — float, ⟨0; 1⟩
- `green` *(number)* — float, ⟨0; 1⟩
- `blue` *(number)* — float, ⟨0; 1⟩

Arguments:

- `r` *(number)* — red channel, fixed, ⟨0; 255⟩
- `g` *(number)* — green channel, fixed, ⟨0; 255⟩
- `b` *(number)* — blue channel, fixed, ⟨0; 255⟩

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 ⟨0; 1⟩ range

Returns:

- `r` *(number)* — red channel, float, ⟨0; 1⟩
- `g` *(number)* — green channel, float, ⟨0; 1⟩
- `b` *(number)* — blue channel, float, ⟨0; 1⟩

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_lin_sRGB(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)* — color in hexadecimal RGB notation

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 (γ) compression with `gamma` value on `channel` value. The formula is: $\sqrt[\gamma]{channel}$
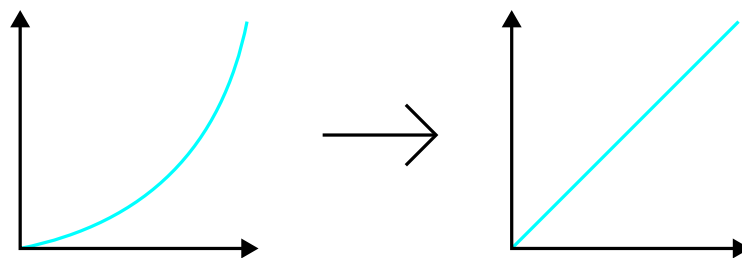
Gamma compression

Returns:

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

Arguments:

- `gamma` *(number)* — gamma value
- `channel` *(number)* — value to compress

Example:

```lua
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
- `channel1` *(number)* — value to compress
- `channel2` *(number)* — value to compress
- `channel3` *(number)* — value to compress

Example:

```lua
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 (γ) expansion of `channelp` value with `gamma`. The formula is:
$(channelp)^{\gamma}$

Gamma expansion
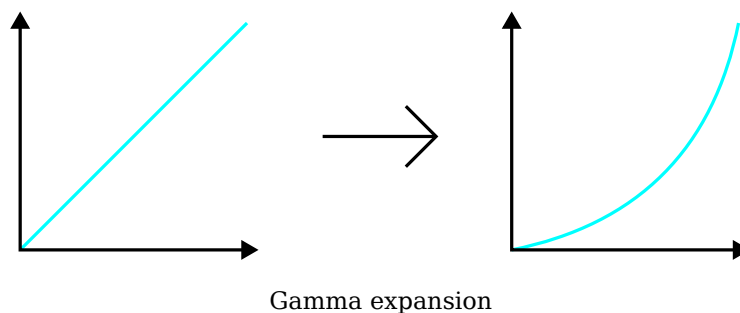
Returns:

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

Arguments:

- `gamma` *(number)* — gamma value
- `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
- `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
```

```
-- convert HSV to CIExyY
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, ⟨0; 1⟩
- `g` *(number)* — green channel, float, ⟨0; 1⟩
- `b` *(number)* — blue channel, float, ⟨0; 1⟩

Arguments:

- `hue` *(number)* — float, ⟨0; 360)
- `saturation` *(number)* — float, ⟨0; 1⟩
- `value` *(number)* — float, ⟨0; 1⟩

Example:

```lua
-- 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, ⟨0; 360)
- `saturation` *(number)* — float, ⟨0; 1⟩
- `value` *(number)* — float, ⟨0; 1⟩

Arguments:

- `r` *(number)* — red channel, float, ⟨0; 1⟩
- `g` *(number)* — green channel, float, ⟨0; 1⟩
- `b` *(number)* — blue channel, float, ⟨0; 1⟩

Example:

```lua
-- 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, ⟨0; 1⟩
- `g` *(number)* — green channel, linear; float, ⟨0; 1⟩
- `b` *(number)* — blue channel, linear; float, ⟨0; 1⟩

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()`
> The result values may 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, ⟨0; 1⟩
- `g` *(number)* — green channel, linear; float, ⟨0; 1⟩
- `b` *(number)* — blue channel, linear; float, ⟨0; 1⟩

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.

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:

```lua
-- 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:

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

-- X = .4
-- Y = .2
-- Z ≈ .007
```

Color temperature

utils.color:kelvins_to_mireds(t)

Converts color temperature in kelvins to mireds.

Returns:

- *number* — color temperature in mireds

Arguments:

- `t` *(number)* — color temperature in kelvins

`utils.color:mireds_to_kelvins(t)`

Converts color temperature in mireds to kelvins.

Returns:

- *number* — color temperature in kelvins

Arguments:

- `t` *(number)* — color temperature in mireds

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.

Date

Methods

`utils.date:currentWeekdayIn(table)`

One-line helper method to determine if the current day of the week is in a given table of weekdays.

Returns:

- *boolean* — `true` if the current day of the week is in the table, `false` otherwise

Arguments:

- `table` *(table)* — table of weekdays, e.g. `{1, 3, 5}`, where Sunday = 0, Monday = 1, …

Example:

```
utils.date:currentWeekdayIn({1, 3, 5})
    -- true if today is Monday, Wednesday or Friday
```

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:

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

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

This function returns the number between range of numbers or it's minimum or maximum.

Returns:

- *number* — clamped value

Arguments:

- `min` *(number)* — bottom of the allowed `value` range
- `max` *(number)* — top of the allowed `value` range
- `value` *(number)* — a value to be clamped into range of `min` to `max`

Example:

```lua
local value = 0

value = utils.math:clamp(0, 4, 12)
-- value = 4

value = utils.math:clamp(0, 4, -8)
-- value = 0
```

```
value = utils.math:clamp(0, 4, 2)
-- value = 2
```

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

Throws an error if the `value` does not fit inside ⟨*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:

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

Profiler

Performance tuning utility. Profiler allows to measure total execution time and execution time between checkpoints in code. User can instantiate multiple profilers to measure different parts of the code at the same time.

The instance of object can be created by calling `utils:profiler()` and saving it to a variable eg.

```
local profiler = utils:profiler()
```

Following function calls should be chained to instance of profiler object.

Functions

start()

Starts the profiler. It should be called at the beginning of the code to measure. Profiler cannot be started twice.

checkpoint(name)

Creates a named checkpoint. Will be used to measure time between previous checkpoint or start of the profiler and this checkpoint. Checkpoint cannot be created when profiler is not started or is stopped.

Arguments:

- `name` *(string)* — checkpoint name

stop()

Stops the profiler - in other words marks the final checkpoint. Profiler cannot be stopped twice.

print()

Prints the results of the profiler to the console. It will print the total time of execution and time between checkpoints. Can be called only on profiler that was started and then stopped.

result()

Returns table with total time of execution and time between checkpoints that can be used in other functions. Can be called only on profiler that was started and then stopped.

Returns:

- *table* - result of the profiler

  Schema of result table:

- ◦ `total_time` *(number)*

  Total execution time in milliseconds.

- ◦ `checkpoints` *(table)*

  Table with checkpoints. Each checkpoint is a table with following fields:

  - • `name` *(string)*

  - • `time` *(number)*

    Time between previous checkpoint (or start if there wasn't any previous checkpoint) and this checkpoint in milliseconds.

- ◦ `stop_time` *(boolean)*

  Time between last checkpoint (or start if there was not checkpoints) and stop in milliseconds.

Example:

```lua
local profiler = utils:profiler()

profiler:start()

for i = 1, 100 do
  if i % 10 == 0 then
    profiler:checkpoint("Checkpoint " .. i)
  end
end

profiler:stop()
profiler:print()
```

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. The unpacking is not recursive.

Returns:

- *table* — flattened sequence

Arguments:

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

Example:

```lua
local flattened = utils.seq:flat { 1, 2, { 4, 8, { 12, 13 } }, 16 }
-- flattened == { 1, 2, 4, 8, { 12, 13 }, 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:

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

`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:

```
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:

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

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:

```lua
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:

```lua
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:

```lua
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:

```lua
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:

```lua
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:

```
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* - sequence which contains all separated tokens

Arguments:

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

Example:

```
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:

```
for line in input:gmatch("(.*)\n") do
    -- ignore comments
    if not utils.str:startsWith(line, "//") then
        parse(line)
    end
end
```

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 (uppercase and lowercase consonants and digits).

Returns:

- *string* - generated value

Arguments:

- `length` *(number)*

Example:

```
local id = utils.str:random(12)
-- id == "5FjjNd3zVpT6"
```

`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:

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

```lua
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
```

Tables

JavaScript-like table manipulation routines.

> **Note**
>
> Lua tables which use indices other than positive integers are implemented internally as hash maps and are not sorted. 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:

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

`utils.table:equal(t1, t2)`

Returns `true` if objects `t1` and `t2` are equal or if they are tables with equal elements. Tables are checked recursively.

Returns:

- *boolean*

Arguments:

- `t1` *(any)*
- `t2` *(any)*

`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-like table 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` *(fun(accumulator: any, value: any, key: any, table: table): any)* — 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.

Example:

```lua
local input = { 1, 2, 4, 8 }
local sum = utils.table:reduce(input, function (A, value)
    return A + value
end)

--> sum = 15
```

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:

```lua
-- 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.

```lua
-- convert elements to string only if their key is a string
local function makeTypesConsistent(value, key)
  if type(key) == 'string' 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:

```lua
-- 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
  else
    return true
  end
end

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

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:

- *boolean* - test result

Arguments:

- `table` *(table)*
- `callback` *(fun(item: any, key: any, table: table): boolean)*

Examples:

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

--> allNumbersPositive = false
```

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

```lua
-- breakable forEach loop:
utils.table:every(inputData, function (v)
    -- parse only string values
    if type(v) == 'string' then
        parseValue(v)

        -- continue loop
        return true
    else
        -- break loop
        return false
    end
end)
```

> **Note**
>
> `utils.table:some()` method can also be used, but with return value meanings reversed, therefore callback would not have to return a value at all to continue the loop.

utils.table:filter(table, callback)

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

Returns:

- *table* — filtered input `table`

Arguments:

- `table` *(table)*
- `callback` *(fun(item: any, key: any, table: table): boolean)*

Example:

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

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

```lua
-- make a list of relays which are turned on right now
local relays = { wtp[4], wtp[5], ... }
local activeRelays = utils.table:filter(relays, function (relay)
    return relay:getValue("state") == true
end)
```

`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:

- *any* — found table element
- *any* — element's key

Arguments:

- `table` *(table)*
- `callback` *(fun(item: any, key: any, table: table): boolean)*

Example:

```lua
local input = { "a", "b", "", "d", "" }
local empty, key = utils.table:find(input, function (str)
    return #str == 0
end)

-- empty = "", key = 3
```

```lua
-- find a relay which was toggled right now
local relays = { wtp[4], wtp[5], ... }
local relay = utils.table:find(relays, function (relay)
    return relay:changedValue("state")
end)
```

`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` *(fun(item: any, key: any, table: table))*

Example:

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

> **Note**
>
> If ability to break the loop is required, consider using `utils.table:some()` or `utils.table:every()`.

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. If the `callback` returns `nil`, the element is ignored.

Returns:

- *table* — contains selected groups in tables

Arguments:

- `table` *(table)* — values to group
- `callback` *(fun(item: any, key: any, table: table): any)* — returns anything that can be used as a table key

Example:

```lua
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)
    return row.number % 2 == 1 and "odd" or "even"
end)
```

variable `grouped` will contain:

```lua
{
    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` *(fun(item: any, key: any, table: table): any)* — 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 }
```

```lua
local names = utils.table:map({ 1, 4, 16 }, function (id)
    return wtp[id]:getValue("name")
end)

--[[
> names = {
    "Bedroom temp sensor",
    "Kitchen regulator",
    "Corridor lights"
}
]]
```

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` *(fun(item: any, key: any, table: table): boolean)*

Examples:

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

--> thereAreNegativeNumbers = true
```

```lua
local relays = { wtp[4], wtp[6], sbus[8] }
local lightIsOn = utils.table:some(relays, function (dev)
    return dev:getValue('state') == true
end)
```

```lua
-- breakable forEach loop:
utils.table:some(inputData, function (v)
    -- parse only string values
    if type(v) == 'string' then
        parseValue(v)
        -- (function returns nothing, continue)
    else
        return true -- break loop
```

```
        end
    end)
```

utils.table:xform(table, callback)

Creates a new table, using `callback` function to determine keys and values.

Returns:

- *table* — transformation result

Arguments:

- `table` *(table)*
- `callback` *(fun(item: any, key: any, table: table): any, any)* — returns key and value

Example:

```lua
local array = {
    {
        id = 1,
        name = "Power",
        value = 4
    },
    {
        id = 5,
        name = "Energy",
        value = 14
    },
}

-- convert array to a map
local map = utils.table:xform(array, function (value)
    return value.id, value
end)

--[[
> map = {
    [1] = {
        id = 1,
        name = "Power",
        value = 4
    },
    [5] = {
        id = 5,
        name = "Energy",
        value = 14
    },
}
]]
```

```lua
local names = utils.table:xform({ 1, 4, 16 }, function (id)
    return id, wtp[id]:getValue("name")
end)

--[[
> names = {
    [ 1] = "Bedroom temp sensor",
    [ 4] = "Kitchen regulator",
    [16] = "Corridor lights",
```

```
      }
]]
```

Time

Methods

`utils.time:fromISO(str)`

Converts an ISO 8601-like time string to Unix timestamp. Month, day, hour, minute and second have to be specified using 2 digits. Year has to be specified using 4 digits. Date components can be separated with dashes (`-`), time components can be separated with colons (`:`). Date and time have to be separated using character `T` .

The argument is interpreted as local time, unless a time zone offset specifier is present at the end. UTC can also be represented as `Z` .

Examples of valid time strings:

- `2024-01-01T12:00:00Z`
- `2024-01-01T13:00:00+0100`
- `2024-01-01T14:00:00+02:00`
- `20240101T140000+0200`

Returns:

- *integer* — Unix timestamp

Arguments:

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

Example:

```
utils.time:fromISO("1986-04-26T01:23:00+04:00")
-- 514848180
```

`utils.time:toISO(unix, mode)`

Converts Unix timestamp to ISO 8601-like time string. One of three modes can be selected with optional second argument:

- `utils.time.implicit` (default) — local time without UTC offset
- `utils.time.utc` — UTC time with `Z` suffix
- `utils.time.explicit` — local time with UTC offset

Dashes (`-`) and colons (`:`) are always used to separate date and time components.

Returns:

- *string* — ISO time string

Arguments:

- `unix` *(integer)* — Unix timestamp
- `mode` *(any)* — One of predefined constants

Example:

```
utils.time:toISO(1688554570)
-- 2023-07-05T12:56:10

utils.time:toISO(1688554570, utils.time.utc)
-- 2023-07-05T10:56:10Z

utils.time:toISO(1688554570, utils.time.explicit)
-- 2023-07-05T12:56:10+0200
```

utils.time:toTimeOfDay(timeString)

Converts *hour:minute* string to a TOD value (minutes since 00:00).

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
```

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:getAuthority(url)

Extracts authority component of the `url`.

Returns:

- authority *(string)*

Arguments:

- `url` *(string)* — a valid URL

Example:

```
local auth = utils.url:getAuthority("http://user:pass@localhost:9000/img.jpg")
-- auth = "user:pass@localhost:9000"
```

utils.url:getUserinfo(url)

Extracts userinfo subcomponent of the authority component.

Returns:

- userinfo *(string)*

Arguments:

- `url` *(string)* — a valid URL

```
local uinfo = utils.url:getUserinfo("http://user:pass@localhost:9000/img.jpg")
-- uinfo = "user:pass"
```

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:

```lua
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:

```lua
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:

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

`utils.url:stripQueryParams(url)`

Returns `url` with query and fragment components removed.

Returns:

- stripped url *(string)*

Arguments:

- `url` *(string)* — a valid URL

Example:

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

Unit conversion

A unit conversion system with SI prefix support.

Converter

The system resides inside `utils.unit` table. Unit category can be chosen by calling an appropriate method. Every function takes at least 2 arguments: source unit and target unit abbreviations. Any amount of extra arguments will be converted to the target unit. Specifying invalid unit abbrevation will cause an error.

Examples:

```lua
-- simple conversion from degrees Celsius to degrees Fahrenheit
local temp1 = utils.unit:therm('C', 'F', 45.3)

-- alternative unit abbreviations
local temp2 = utils.unit:therm('°F', '°C', 212)

-- conversion of multiple values in a single call:
local p1, p2, p3 = utils.unit:pressure('psi', 'bar', 14.7, 23, 30)

-- conversion with metric prefixes:
local energy = utils.unit:work('kWh', 'MJ', 700)
local power = utils.unit:power('PS', 'kW', 286)
```

Available metric prefixes

Every official SI prefix (as of 2023) is available. They can be used by simply prepending their symbol to a "prefixable" unit's abbreviation, for example:

- `hPa` for hectopascals (100 Pa)
- `mbar` for millibars ($\frac{1}{1000}$ bars)
- `MW` for megawatts (one million watts)

"micro" can be indicated by both `u` and `μ`.

Available groups and units

`power`

Power units:

Full name	Abbreviations	Prefixable
imperial horsepower	HP hp	no
metric horsepower (Pferdestärke)	PS	no
watt	W	yes

`pressure`

Pressure units:

Full name	Abbreviations	Prefixable
technical atmosphere	`at`	no
standard atmosphere	`atm`	no
bar	`bar`	yes
millimetre of mercury	`mmHg`	no
pound per square inch	`psi`	no
pascal	`Pa`	yes

`therm`

Temperature measurement (thermometric) scales:

Full name	Abbreviations			Prefixable
degree Celsius	`C`	`°C`		no
degree Fahrenheit	`F`	`°F`		no
kelvin	`K`			yes
degree Rankine	`R`	`°R`	`°Ra`	no

`work`

Energy, heat and work units:

Full name	Abbreviations	Prefixable
british thermal unit	`BTU`	no
calorie	`cal`	no
kilocalorie	`kcal`	no
joule	`J`	yes
watt-hour	`Wh`	yes

Network

Following objects are responsible for communication with other devices and allow to integrate with external services.

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.

Default values

HTTP client properties can't be changed via Lua scripts, but they will be used in some cases when sending requests.

- **URL**

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

- **Request body**

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

- **Headers**

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

  For example:

  ```lua
  local http = http_client[1]
  -- 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.

- **Query parameters**

  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[1]
  -- 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)`, `POST(url)`, `DELETE(url)`, `PATCH(url)`, `PUT(url)`

  Set the request method and URL (optional).

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

- `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)* — Positive values are response statuses received from server, e.g. 200 if request OK. Negative values represent internal error codes e.g. -101 for networking issues.

    - `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<string, string>)* — response headers in form of Lua table (key = header name, value = header value).

Deprecated methods

Old HTTP client API used internal response cache which made those methods behave unexpectedly when multiple requests were made. `onMessage` should be used in new applications instead.

- `hasResponse()`

  Checks whether client received response from server. Response is cached if available.

  Returns:

  - *(boolean)*

- `hasFailure()`

  Checks whether last cached client request failed, e.g. due to invalid URL. If request failed it caches the request result.

  Returns:

  - *(boolean)*

- `response()`

  Returns last cached response body received from server.

  Returns:

  - *(string)*

- `status()`

  Returns last cached response status received from server, e.g. 200 if request OK.

  Returns:

  - *(number)*

- `failureCause()`

  Returns the cause of last cached request failure.

  Returns:

  - *(string)*

Examples

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

Send GET requests to custom.server.com at 8:00 and 19:00

```
local http = http_client[8]

if dateTime:changed() then

    -- without chained calls
    if dateTime:getHours() == 8 and dateTime:getMinutes() == 0 then
        http:GET("https://custom.server.com/day")
        http:send()
    end
```

```lua
    -- with chained calls
    if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
        http:GET("https://custom.server.com/night"):send()
    end

end
```

POST data to custom.server.com at sunrise

```lua
local http = http_client[1]

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

if event.type == "sunset" then
    -- with chained calls
    http
        :POST("https://custom.server.com/night")
        :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`

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

Handle received response

```lua
http_client[1]:onMessage(function (status, responseBody, url, responseHeaders)
    print("URL requested:" .. url)

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

    local success = status // 100 == 2
```

```lua
    if success then
        print("Request succeeded with status " .. status)
        print("Response from server: " .. responseBody)
    elseif status >= 0 then
        print("Request failed with status " .. status)
        print("Response from server: " .. responseBody)
    else
        print("Internal error " .. status)
        print("Error message: " .. responseBody)
    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

- request and response headers

- any content type of request/response

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 doesn't need refreshing - is valid as long as exists in configuration. (Can be created using the API or web/mobile application)

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 handler for this URL / method wasn't declared.

- `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`.

Methods

- `on(method, path, handler)`

  Router hook. Attaches 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. e.g. when you request `/lua/http-server/my-endpoint/5` it will get forwarded as `/my-endpoint/5` URL.

  You can catch `5` as parameter (e.g. 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 executed when request is received. Handler should accept two arguments:

  Arguments:

  - • `request` *(HttpServerRequest, required)* — received request, see HTTP server request description below for details.

  - • `response` *(HttpServerResponse, required)* — used to generate response, see HTTP server response description below for details.

- • `tokens()`

  Returns collection (table) of configured static HTTP API tokens.

  Returns:

  - ∘ *{ name: string, value: string }[]*, i.e. *table* — sequence of objects:
    - • *table* — object with token info with following properties:
      - ∘ `name` *(string)* configured token name
      - ∘ `value` *(string)* generated token value

HttpServerRequest

This object (Lua table) is passed to handler and can be used to read incoming HTTP request data.

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

- `header(name)`

  Returns request header or nil if not found.

  > **Note**
  >
  > HTTP header names are case-insensitive, so there is no need to match the letter case of the name.

  Returns:

  ○ *string*

  Arguments:

  ○ `name` *(string, required)* — name of header 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 method is optional since calling `response:body(...)` automatically sets code as `200` if none was set.

Returns:

◦ *HttpServerResponse* — HTTP server response reference, for chained calls

Arguments:

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

- `header(name, value)`

  Sets response header. Automatically sets status code to 200 and `Content-Type` header
  to `application/json` if none was set.

  > **Note**
  >
  > HTTP header names are case-insensitive, so there is no need to match the letter
  > case of the name.

  Returns:

  ◦ *HttpServerResponse* HTTP server response reference, for chained calls

  Arguments:

  ◦ `name` *(string, required)* — name of header to set
  ◦ `value` *(string, required)* — value of header to set

- `body(content)`

  Sets HTTP response body (content). Automatically sets status code to 200 and
  `Content-Type` header to `application/json` if none was set. If you wish to change it,
  use appropriate methods: `status` or `header`

  Returns:

  ◦ *HttpServerResponse* HTTP server response reference, for chained calls

  Arguments:

  ◦ `content` *(string, required)* — String representation of body e.g. 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 HTTP headers

Sending request to `/api/v1/lua/http-server/hello` with headers `X-From: admin` and `X-To: sinum` will create response with message: `Hello sinum was sent by admin!` and response headers set to `X-From: sinum` and `X-To: admin`.

```lua
http_server:on("GET", "/hello", function(request, response)
    -- Http header names are case-insensitive,
    -- so there is no need to match the letter case of the name

    local from = request:header("x-from")
    local to = request:header("x-to")

    response
        :header("X-From", to)
        :header("X-To", from)
        :body("Hello " .. to .. " was sent by " .. from .. "!")
end)
```

Handle json body in request and response

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

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

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

The response will contain:

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

Code:

```lua
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)
```

```lua
    local responseBody = {
        name = "Sinum",
        data = {
            66,
            77,
            88,
            99
        },
        success = true,
        reason = nil
    }

    response:body(JSON:encode(responseBody))
end)
```

Handle other body data types in response

Request to `/api/v1/lua/http-server/html` will return simple static HTML page:

The response will contain:

```html
<html><body>Hello there!</body></html>
```

Request to `/api/v1/lua/http-server/xml` will return xml data:

The response will contain:

```xml
<user>admin</user>
<system>
  <name>sinum</name>
  <version>1.11.0</version>
</system>
```

Code:

```lua
http_server:on("GET", "/html", function(request, response)
    response
        :header("content-type", "text/html")
        :body("<html><body>Hello there!</body></html>")
end)

http_server:on("GET", "/xml", function(request, response)
    local data = {
        user = "admin",
        system = {
            name = "sinum",
            version = system:version().semver
        }
    }

    response
        :header("content-type", "application/xml")
        :body(XML:encode(data))
end)
```

Print all available static API tokens

```lua
for _, token in pairs(http_server:tokens()) do
    print("Token name ", token.name, " value ", token.value)
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 (e.g. 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, [1-30], default: 5)* - maximum waiting time for reply in seconds.

  ○ `dataSize` *(integer, optional, [1-256], 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' were 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,
  ttl)
    if success then
      print("Success!")
    else
      print("Failed! Reason: " .. errorMessage)
    end

    -- Print diagnostic data
    print("Elapsed:", elapsed)
    print("Destination:", destination)
    print("Reply From:", replyFrom)
    print("TTL:", ttl)

    -- 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)
```

Modbus client (master)

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.

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.

General properties

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

  Unique client identifier.

- `name` *(string)*

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

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

  Type of the client: `modbus_rtu`, `modbus_tcp`

Modbus RTU client properties

These properties are only available when client type is `modbus_rtu`.

- `slave_address` *(integer)*

  Address of slave device to which client will communicate.

  Value in range: 1 – 255 (above 247 is not standard compliant, but some devices may use it).

- `baud_rate` *(integer)*

  Baud rate used by the slave device.

  One of: `4800`, `9600`, `19200`, `38400`, `57600`, `115200`.

- `parity` *(string)*

  UART parity bit setting which slave device uses.

  One of values: `none`, `odd`, `even`.

- `stop_bits` *(string)*

  Count of UART stop bits which slave device uses.

  One of values: `one`, `two`.

- `associations.transceiver` *(device)*

  Reference to transceiver associated to client which will be used for communication. Has to be reference to **system_module** device of type: **modbus_transceiver** or **modbus_extender**.

Modbus TCP client properties

These properties are only available when client type is `modbus_tcp`.

- `ip_address` *(string)*

  IP address of the device the client communicates with.

- `port` *(integer)*

  TCP port used for communication. Default is 502.

- `device_id` *(integer)*

  Slave device ID. Used if target device is a gateway.

  Value in range: 0 – 255 (above 247 is not standard compliant, but some devices may use it).

Methods

- `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` *(table)* — integer values that should be written to consecutive holding registers starting with `startAddress`

- `readHoldingRegisterAsync(address)`

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

  Arguments:

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

- `readHoldingRegistersAsync(address, registersCount)`

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

  Arguments:

  - `address` *(integer)* — address of a holding register.
  - `registersCount` *(integer)* — number of a holding register to read

- `readInputRegisterAsync(address)`

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

  Arguments:

  - `address` *(integer)* — address of an input register.

- `readInputRegistersAsync(address, registersCount)`

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

  Arguments:

  - `address` *(integer)* — address of an input register.
  - `registersCount` *(integer)* — number of an input registers to read

- `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` *(table)* — boolean values that should be written to consecutive coils starting with `startAddress`

- `readCoilAsync(address)`

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

  Arguments:

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

- `readCoilsAsync(address, registersCount)`

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

  Arguments:

  - `address` *(integer)* — address of a coil.
  - `registersCount` *(integer)* — number of coils to read.

- `readDiscreteInputAsync(address)`

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

  Arguments:

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

- `readDiscreteInputsAsync(address, registersCount)`

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

  Arguments:

- ◦ `address` *(integer)* — address of a discrete input.
- ◦ `registersCount` *(integer)* — number of a discrete inputs to read.

- `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)* — 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` *(boolean|boolean[]|integer|integer[])* — value of read register/registers, type depends on register type and registers number read (see example below)

- `onRegisterAsyncWrite(callback)`

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

  Arguments:

  - ◦ `callback` *(function)* — 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/registers, type depends on register type and registers number write (see example below)

- `onAsyncRequestFailure(callback)`

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

  Arguments:

  - ◦ `callback` *(function)* — callback function which should be called

    Arguments:

    - • `requestType` *(string)* — type of request (one of values: `READ`, `WRITE`, `MULTIPLE_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` *(boolean|boolean[]|integer|integer[])* — value of register to write (`0`/`false` for read requests), type depends on register type

- `getAsyncQueueSize()`

  Returns the asynchronous queue size for transmitter associated with Modbus client.

- `clearAsyncQueue()`

  Clear the asynchronous queue size for transmitter associated with Modbus client.

Deprecated methods

Synchronous Modbus methods should not be used, as calls to them are blocking, thus slowing down execution of scripts.

- `writeHoldingRegister(address, value)`
- `writeHoldingRegisters(startAddress, values)`
- `readHoldingRegister(address)`
- `readInputRegister(address)`
- `writeCoil(address, bit_value)`
- `writeCoils(startAddress, bit_values)`
- `readCoil(address)`
- `readDiscreteInput(address)`

Examples

Read data from Modbus device using asynchronous read

```
modbus_client[1]:readHoldingRegisterAsync(104)
modbus_client[1]:readInputRegisterAsync(123)
modbus_client[1]:readCoilAsync(201)
modbus_client[1]:readDiscreteInputAsync(302)
```

Read multiple registers from Modbus device using asynchronous read

```
modbus_client[1]:readHoldingRegistersAsync(104, 4)
modbus_client[1]:readInputRegistersAsync(123, 5)
modbus_client[1]:readCoilsAsync(201, 2)
modbus_client[1]:readDiscreteInputsAsync(302, 2)
```

Write multiple values to holding registers and coils

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

Handle asynchronous read request

```
modbus_client[1]:onRegisterAsyncRead(function(registerType, address, value)
  print ("Successfully read from register:", registerType)

  -- check type of value to see if one or more values were read
  if type(value) == "table" then
    -- Read more than one value from Modbus registers
    print("Read values:")
    for i, val in pairs(value) do
      -- subtract 1 as first value of i is 1
      print("Register: ", address + i - 1, " value: ", val)
    end
  else
    -- only one value read
    print("Read value:", value, "from register: ", address)
  end
end)
```

Handle asynchronous write request

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

  -- check type of value to see if one or more values were written to register
  if type(value) == "table" then
    -- Written more than one value to Modbus registers
    print("Written values:")
    for i, val in pairs(value) do
      -- subtract 1 as first value of i is 1
      print("Register: ", address + i - 1, " value: ", val)
    end
  else
    -- only one value written
    print("Written value:", value, "from register: ", address)
  end

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)
```

Simple custom device with Modbus client

Required custom device setup:

- toggle switch element with name `power_switch` and callback `onPowerSwitch`
- correctly configured Modbus client with name `rtu_client`

```lua
-- Modbus register that controls the power switch
local powerSwitchReg <const> = 12

-- Custom device power switch handler
function CustomDevice:onPowerSwitch(on)
    -- cast boolean to integer
    on = on and 1 or 0

    -- send new switch state to the physical device
    self:getComponent('rtu_client'):writeHoldingRegisterAsync(powerSwitchReg,
        on)
end

-- Modbus data poll handler
function CustomDevice:onPoll(powerSwitch)
    -- cast integer to boolean
    powerSwitch = powerSwitch ~= 0

    -- set custom device switch to position indicated by read data
    -- with event propagation disabled, so that :onPowerSwitch won't be called
    -- causing an infinite loop
    self:getElement('power_switch'):setValue('value', powerSwitch, true)
end

function CustomDevice:onEvent()
    local rtu <const> = self:getComponent('rtu_client')

    -- poll device once a minute
    if dateTime:changed() then
        rtu:readHoldingRegisterAsync(powerSwitchReg)
    end

    -- handle read data
    rtu:onRegisterAsyncRead(function (kind, addr, value)
        self:setValue('status', 'online')

        if kind == 'HOLDING_REGISTERS' and addr == powerSwitchReg then
            self:onPoll(value)
        end
    end)

    -- handle written data
    rtu:onRegisterAsyncWrite(function ()
        self:setValue('status', 'online')
    end)

    -- handle errors
    rtu:onAsyncRequestFailure(function (request, err, kind, addr)
        utils:printf('Failed to %s %s (address %d): %s', request, kind, addr,
            err)

        -- these errors indicate a bad connection
        if err == 'TIMEOUT' or err == 'BAD_CRC' then
            self:setValue('status', 'offline')
        end
    end)
end
```

Modbus server (slave)

Global scope object which allow user to act as Modbus Slave to receive requests and generate responses using Lua automations or Custom Devices. In order to talk with Modbus Server, you need to connect to the central device at port `tcp/502` (default Modbus port) or set the build-in Modbus transceiver (or Modbus extender) into slave mode in application.

Server can handle multiple connections at once acting as multiple slaves.

Features:

- Multiple slaves support

- Multiple connections support

- Single/Multi read and write requests supported.

- Full control over response generation in Lua.

- Automatic response generation in case of invalid request or processing failure.

Server is exposed as object: `modbus_slave`.

Methods

- `onDiscreteInputRead(slave_id, handler)`

  Discrete Input Read Request handler, called when master requests to read discrete inputs for given slave.

  Returns:

  - `Modbus server reference`

  Arguments:

  - `slave_id` *(number, required)* - Slave ID for which handler should be called.

  - `handler` *(function, required)* - callback function which should be executed when request is received. Handler should accept two arguments:

    Arguments:

    - `request` *(ModbusSlaveRequest, required)* - received request, see Modbus Slave request description below for details.

    - `response` *(ModbusSlaveResponse, required)* - used to generate response, see Modbus Slave response description below for details.

- `onCoilRead(slave_id, handler)`

  Coil Read Request handler, called when master requests to read coils for given slave.

  Returns:

  - `Modbus server reference`

  Arguments:

- ○ `slave_id` *(number, required)* - See description above for `onDiscreteInputRead`.

  - ○ `handler` *(function, required)* - See description above for `onDiscreteInputRead`.

- • `onHoldingRegisterRead(slave_id, handler)`

  Holding Register Read Request handler, called when master requests to read holding registers for given slave.

  Returns:

  - ○ `Modbus server reference`

  Arguments:

  - ○ `slave_id` *(number, required)* - See description above for `onDiscreteInputRead`.

  - ○ `handler` *(function, required)* - See description above for `onDiscreteInputRead`.

- • `onInputRegisterRead(slave_id, handler)`

  Input Register Read Request handler, called when master requests to read input registers for given slave.

  Returns:

  - ○ `Modbus server reference`

  Arguments:

  - ○ `slave_id` *(number, required)* - See description above for `onDiscreteInputRead`.

  - ○ `handler` *(function, required)* - See description above for `onDiscreteInputRead`.

- • `onCoilWrite(slave_id, handler)`

  Coil Write Request handler, called when master requests to write coils for given slave.

  Returns:

  - ○ `Modbus server reference`

  Arguments:

  - ○ `slave_id` *(number, required)* - See description above for `onDiscreteInputRead`.

  - ○ `handler` *(function, required)* - See description above for `onDiscreteInputRead`.

- • `onHoldingRegisterWrite(slave_id, handler)`

  Holding Register Write Request handler, called when master requests to write holding registers for given slave.

  Returns:

  - ○ `Modbus server reference`

  Arguments:

  - ○ `slave_id` *(number, required)* - See description above for `onDiscreteInputRead`.

  - ○ `handler` *(function, required)* - See description above for `onDiscreteInputRead`.

ModbusSlaveRequest

This object (Lua table) is passed to handler and can be used to get Modbus request data.

Methods

- `address()`

  Returns requested address of coil, discrete input, holding register or input register.

  Returns:

  - *(number)*

- `size()`

  Returns request data size. For read request it returns number of requested coils, discrete inputs, holding registers or input registers. For write request it returns number of coils or holding registers to write.

  Returns:

  - *(number)*

- `data()`

  Returns request data. For read request it always returns nil. For single write request it returns value to write (boolean for coils, u16 number for holding registers). For multi write request it returns table with values to write (table of booleans for coils, table of u16 numbers for holding registers).

  Returns:

  - *(boolean/number/table/nil)*

ModbusSlaveResponse

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

Methods

- `success`

  Marks response as successful.

  Returns:

  - *(userdata)* Modbus response reference, for chained calls

- `exception(code)`

  Sets response exception code. Use this method to indicate that request was invalid or processing failed. Setting exception will clear any data set by `data` method.

  Returns:

  - *(userdata)* Modbus response reference, for chained calls

  Arguments:

- ◦ `code` *(number, required)* - exception code to set, one of:

  - 1 - Illegal Function
  - 2 - Illegal Data Address
  - 3 - Illegal Data Value
  - 4 - Slave Device Failure
  - 5 - Acknowledge
  - 6 - Slave Device Busy
  - 8 - Memory Parity Error
  - 10 - Gateway Path Unavailable
  - 11 - Gateway Target Device Failed to Respond

> **Note**
>
> Hex values are also supported.

- `data(data)`

  Sets Modbus response data. Available only when handling read request. When request is single read, data should be single value (boolean for coils/discrete inputs, u16 number for holding/input registers) or single value table.

  When request is multi read, data should be table with values (table of booleans for coils/discrete, table of u16 numbers for holding/input registers) with size equal to requested size.

  Setting valid data will clear any exception set by `exception` method and mark response as successful.

  Returns:

  - ◦ *(userdata)* Modbus response reference, for chained calls

  Arguments:

  - ◦ `data` *(boolean/number/table, required)* - data to set in response

Examples

Handle single coil read request for slave 1

```lua
modbus_slave:onCoilRead(1, function (request, response)
  print("Received request for coil " .. tostring(request:address()))
  response:success():data(true)
end)
```

Handle multiple discrete input read request for slave 2

```lua
modbus_slave:onDiscreteInputRead(2, function(request, response)
  local size = request:size()

  print("Received request for " .. tostring(size) .. " discrete inputs")

  local data = {}
  for i = 1, size do
    data[i] = true
```

```lua
    end

  response:success():data(data)
end)
```

Handle single holding register read request for slave 3

```lua
modbus_slave:onHoldingRegisterRead(3, function (request, response)
  response:data(1234) -- success is called automatically when valid data is set
end)
```

Handle multiple input register read request for slave 4

```lua
modbus_slave:onInputRegisterRead(4, function (request, response)
  local size = request:size()

  local data = {}
  for i = 1, size do
    data[i] = i
  end

  response:success():data(data)
end)
```

Handle single coil write request for slave 5

```lua
modbus_slave:onCoilWrite(5, function (request, response)
  local value = request:data()
  print("Received value: ", value)
  response:success() -- do not forget to call success
end)
```

Handle multiple holding register write request for slave 6

```lua
modbus_slave:onHoldingRegisterWrite(6, function (request, response)
  local data = request:data()
  local address = request:address()
  for i, value in pairs(data) do
    print("Received value " .. tostring(value) .. " for register " ..
       tostring(address + i - 1))
  end
  response:success()
end)
```

Multiple slaves at once

```
modbus_slave:onCoilRead(1, function (request, response)
  response:success():data(true)
end)

modbus_slave:onCoilRead(2, function (request, response)
  response:success():data(false)
end)

modbus_slave:onCoilRead(3, function (request, response)
  response:success():data(true)
end)
```

MQTT client

Global scope objects which allow user to exchange MQTT messages. (Currently, *MQTT over WebSockets* 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

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 client identifier.

- `name` *(string)*

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

- `broker` *(string)*

  Broker hostname or ip.

- `port` *(integer)*

  Broker port.

- `client_id` *(string)*

  MQTT client identifier. Cannot contain special characters except `-` `_`

- `username` *(string)*

  MQTT broker authorization username (optional, may be empty)

- `password` *(string, write-only)*

  MQTT broker authorization password (optional, may be empty). Cannot be read. Will throw script error when tried.

- `tls.enabled` *(boolean)*

  Enable or disable TLS connection.

- `tls.certificate` *(string, write-only)*

  TLS certificate in PEM format (optional, may be empty). Cannot be read. Will throw script error when tried.

- `tls.private_key` *(string, write-only)*

  TLS private key in PEM format (optional, may be empty). Cannot be read. Will throw script error when tried.

- `tls.ca_certificate` *(string, write-only)*

  TLS CA certificate in PEM format (optional, may be empty). Cannot be read. Will throw script error when tried.

- `last_will.topic` *(string)*

  Last will topic. Last Will message will not be sent if topic is empty.

- `last_will.payload` *(string)*

  Last will payload to be sent.

- `last_will.qos` *(0|1|2)*

  Last will qos configuration.

- `last_will.retain` *(boolean)*

  Last will retain flag configuration.

- `subscriptions` *({qos: 0|1|2, topic: string}[])*

  List of topics to subscribe, passed as array of object with keys `qos` *(integer, [0-2])* and `topic` *(string)*.

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)* — 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)* — topic on which message should be published.
  - `payload` *(string)* — message payload.
  - `qos` *(0|1|2)* — message QoS.
  - `retain` *(string)* — 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)* — 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 (e.g. due to network error).

  Returns:

  - *userdata* — MQTT client reference for chained calls

  Arguments:

  - `function` *(function)* — callback function which should be called on disconnect or error.

    Arguments:

    - `error` *(boolean)* — 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)* — 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

```lua
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)
```

Set subscriptions

```
mqtt_client[4]:setValue("subscriptions", {
    { qos = 1, topic = "sub-1" },
    { qos = 2, topic = "sub-2" }
})
```

Websocket client

Global scope objects which allow user to exchange Websocket messages. Currently only text messages are supported.

Clients are exposed in the key-based container of objects: `ws_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 Websocket Client** with **ID 4** you should use: `ws_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

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 client identifier.

- `name` *(string)*

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

- `url` *(string)*

  Url to server eg. `wss://echo.websocket.org/ws`

- `headers` *(table)*

  List of key-value pairs of headers used while connecting to server.

- `query_params` *(table)*

  List of key-value pairs of query parameters used while connecting to server.

Methods

- `isConnected()`

  Checks whether client successfully connected to server.

  Returns:

  - *boolean*

- `send(payload)`

  Sends message with desired payload.

  Returns:

    ◦ *userdata* — Websocket client reference for chained calls

  Arguments:

    ◦ `payload` *(string)* — message payload.

- `onConnected(function() end)`

  Callback hook. Calls function passed in argument on successful connection to server.

  Returns:

    ◦ *userdata* — Websocket client reference for chained calls

  Arguments:

    ◦ `function` *(function)* — 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 (e.g. due to network error).

  Returns:

    ◦ *userdata* — Websocket client reference for chained calls

  Arguments:

    ◦ `function` *(function)* — callback function which should be called on disconnect or error.

      Arguments:

        • `error` *(boolean)* — disconnection status - graceful (false) or error (true).

- `onMessage(function(payload) end)`

  Callback hook. Calls function passed in argument on message received.

  Returns:

    ◦ *userdata* — Websocket client reference for chained calls

  Arguments:

    ◦ `function` *(function)* — callback function used as message handler.

      Arguments:

        • `payload` *(string, required)* — received message payload.

Examples

All methods in **Websocket Client** which return Websocket client reference can be called successively without calling `ws_client` container every time.

Receive message

```
ws_client[4]:onMessage(function(payload)
  if payload == "ON" then
      wtp[68]:setValue("state", true)
  else
      wtp[68]:setValue("state", false)
  end
end)
```

Send message

```
if dateTime:changed() then
    ws_client[4]:send("I am still alive mate!")
end
```

Catch connect and disconnect

```
ws_client[4]:onConnected(function ()
    print("Client with ID 4 connected to server!")
end)

ws_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 server.")
    end
end)
```

Set headers and query parameters

```
ws_client[4]:setValue("headers", {
    ["Authorization"] = "Bearer 123456",
    ["X-My-Custom-Header"] = "custom_value"
})

ws_client[4]:setValue("query_params", {
    ["param1"] = "value1",
    ["param2"] = "value2"
})
```

TCP port knocking

Global scope utility which allow user to knock remote service at certain port, exposed as the `port_knock` object.

It may be used to check if tcp service is available (listens for connection). Main purpose of this utility is network/service diagnostics.

> **Note**
>
> This utility has throttling mechanism, to maximum 120 usages per hour. Once you have exhausted limit, Lua error will be thrown. Limit will refill for 1 use every 30 seconds, up to a maximum of 120 after an hour. To make sure your script won't be throttled, use this utility no more often than once every 30 seconds.
> Be aware that some services can only accept one connection at a time (e.g. some Modbus TCP devices). Connecting to such services while another service is active (e.g. Sinum Central, reading data) may disrupt the operation of the other service (e.g. disconnect both of connections)

Methods

- `begin(destination, port, timeout)`

  Attempts to open TCP connection to host at certain port, to check if remote side service is available.

  Returns:

  - *userdata* - `port_knock` object reference for chained calls

  Arguments:

  - `destination` *(string, required)* — hostname or ip of service.
  - `port` *(integer, required, [1-65535])* — service port.
  - `timeout` *(integer, optional, [1-5], default: 1)* — maximum waiting time for connection accept in seconds.

- `onDone(callback)`

  Callback hook. Calls function passed in argument on knock success or error.

  Returns:

  - *userdata* — `port_knock` object reference for chained calls

  Arguments:

  - `callback` *(fun(success: boolean, errorMsg: string, elapsed: integer, destination: string, port: integer)*

    Arguments:

    - `success` *(boolean)* — status flag, on successful knock equals `true`, on fail equals `false`.

- errorMessage *(string)* — error message, describes why knock failed. Empty on success.

- elapsed *(integer)* — time spent while processing knock in milliseconds, either successful or not.

- destination *(string)* — always equal to destination used in begin function. May be used to distinguish between many knocks at the same time.

- port *(integer)* — always equal to port used in begin function. May be used to distinguish between many knocks at the same time.

Examples

All methods in **TCP Port Knocking** which return port_knock object reference can be called successively without calling port_knock every time.

Knock local service at 19:00

```
if dateTime:changed() then
    if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
        port_knock:begin("192.168.1.200", 504)
    end
end
```

Knock two remote services every minute

```
if dateTime:changed() then
    port_knock:begin("192.168.1.1", 8000):begin("192.168.1.2", 7676)
end
```

Handle knock done with a callback

```
port_knock:onDone(function(success, errorMessage, elapsed, destination, port)
    if success then
        print("Success!")
    else
        print("Failed! Reason:")
        print(errorMessage)
    end

    -- Print diagnostic data
    print("Elapsed:", elapsed)
    print("Destination:", destination)
    print("Port:", port)

    -- You may use destination to distinguish responses from different hosts

    local devices = {
        inverter_modbus = '192.168.1.100',
        heatpump_modbus = '192.168.1.200'
    }
```

```lua
    if destination == devices.inverter_modbus then
        if success then
            print("TCP Modbus in Inverter is reachable.")
        else
            print("TCP Modbus in Inverter is not reachable.")
        end
    elseif destination == devices.heatpump_modbus then
        if success then
            print("TCP Modbus in HeatPump is reachable.")
        else
            print("TCP Modbus in HeatPump is not reachable.")
        end
    end
end)
```

Wake-on-LAN

Global scope utility which allow user to send Wake-on-LAN magic packet to wake up device from standby, exposed as: `wakeOnLan` object.

> **Note**
>
> Remote device needs to support this function and your router should allow sending WoL packets.
> WoL protocol does not support confirmations, so you can't check if device is turned on. ICMP Ping utility can be used to check that.

Properties

Wake-on-LAN utility doesn't have properties.

Methods

- `send(destination)`

  Sends WoL packet to destination.

  Returns:

  - *userdata* — `wakeOnLan` object reference for chained calls

  Arguments:

  - `destination` *(string, required)* — MAC address of destination device.

Examples

All methods in **Wake-on-LAN** which return `wakeOnLan` object reference can be called successively without fetching `wakeOnLan` global 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
```

Energy center

Following modules allow to access data of the energy center, allowing to implement various energy management algorithms. Scripts can use these modules to react to i.e. current and expected energy prices, photovoltaic inverter production, energy use of various appliances, &c.

Energy consumption

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 e.g. `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

Direct access to properties is not allowed. You can read values using the `getValue` method. An attempt at retrieving a nonexistent object property will cause 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: 1 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: 1 Wh

- `total.electrical_outlets_consumption` *(number, read-only)*

  Total energy consumption of electrical outlets. Represents computed value of energy consumption of devices that provide their power consumption.

  Unit: 1 Wh

- `total.car_chargers_consumption` *(number, read-only)*

  Total energy consumption of car chargers. Represents computed value of energy consumed for charging EV.

  Unit: 1 Wh

- `today.total_consumption` *(number, read-only)*

  Today summary of building energy consumption.

  Unit: 1 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: 1 Wh

- `today.electrical_outlets_consumption` *(number, read-only)*

  Today energy consumption of electrical outlets. Represents computed value of energy consumption of devices that provide their power consumption.

  Unit: 1 Wh

- `today.car_chargers_consumption` *(number, read-only)*

  Today energy consumption of electrical outlets. Represents computed value of energy consumption of devices that provide their power consumption.

  Unit: 1 Wh

Energy prices

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 e.g. `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, accessType)`

  Check if energy price for `hour` of current day 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` *(required, integer)* — hour in range 0–23
  ◦ `accessType` *(optional, string)* — allows selecting data source. One of: `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `isPriceAvailable(hour, minutes, accessType)`

  Check if energy price for `hour` and `minutes` of current day is available. Returns false if passed time is out of range or invalid, price for the given time is not yet set or downloaded or Energy Prices feature is disabled.

  This function is equivalent to `isHourPriceAvailable` with `minutes` set to 0.

  Returns:

  ◦ *boolean*

  Arguments:

  ◦ `hour` *(required, integer)* — hour in range 0–23

  ◦ `minutes` *(required, integer)* — minutes, a multiple of `prices_interval`, less than 60.

  ◦ `accessType` *(optional, string)* — allows selecting data source. One of: `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `getHourPrice(hour, accessType)`

  Returns energy price for `hour` of current day.

  Return:

  ◦ *number?* - energy price at `hour`. Return nil if `hour` is out of range, price for `hour` is not yet set or downloaded or Energy Prices feature is disabled.

  Arguments:

  ◦ `hour` *(required, integer)* — hour in range 0–23

  ◦ `accessType` *(optional, string)* — allows selecting data source. One of: `api`, `lua`. Falls back to configured `access_type` property if not provided.

- `getPrice(hour, minutes, accessType)`

  Returns energy price for `hour` and `minutes` of current day.

  This function is equivalent to `getHourPrice` with `minutes` set to 0.

Return:

- *number?* - energy price at given time. Returns nil if time is out of range or invalid, price for given time is not yet set or downloaded or energy prices feature is disabled.

Arguments:

- `hour` *(integer)* — hour in range 0–23

- `minutes` *(integer)* — minutes, a multiple of `prices_interval`, less than 60.

- `accessType` *(string?)* — allows selecting data source. One of `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `getPrices(accessType)`

  Returns energy prices table for current day.

  Return:

  - *number[]* - table of prices with 24–96 elements (for each time interval of the day)

  Arguments:

  - `accessType` *(optional, string)* — allows selecting data source. One of: `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `clearPrices()`

  Clears current day prices

  Return:

  - *boolean* - true if prices were cleared, false if prices didn't change (already cleared)

- `setStaticPrice(price)`

  Sets same energy price for every hour in a current day.

  Returns:

  - *userdata* - reference to Energy Prices object, for call chains

  Arguments:

  - `price` *(number)* — energy price to set

- `setHourPrice(hour, price)`

  Sets energy price for one `hour` of current day.

  Returns:

  - *userdata* - reference to Energy Prices object, for call chains

  Arguments:

  - `hour` *(integer)* — selected hour in range 0–23
  - `price` *(number)* — energy price to set

- `setHoursPrice(hours, price)`

  Sets single energy price for multiple `hours` of current day

Returns:

- *userdata* - reference to Energy Prices object, for call chains

Arguments:

- `hour` *(integer[])* — table of hours in range 0–23
- `price` *(number)* — energy price to set

- `setPrice(hour, minutes, price)`

  Sets energy price for the given time of current day.

  This function is equivalent to `setHourPrice` with `minutes` set to 0.

  Returns:

  - *userdata* — reference to Energy Prices object, for call chains

  Arguments:

  - `hour` *(integer)* — selected hour in range 0–23
  - `minutes` *(integer)* — minutes, multiplication of `prices_interval`, less than 60.
  - `price` *(number)* — energy price to set

- `setPrices(prices)`

  Sets prices for current day.

  Returns:

  - *userdata* — reference to Energy Prices object, for call chains

  Arguments:

  - `prices` *(number[])* — table of prices. If it's shorter than 24 elements only prices for first hours will be set.

- `isNextDayHourPriceAvailable(hour, accessType)`

  Check if energy price for `hour` in the next day 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
  - `accessType` *(string?)* — allows selecting data source. One of: `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `isNextDayPriceAvailable(hour, minutes, accessType)`

  Check if energy price for `hour` and `minutes` in the next day is available. Returns false if passed time is out of range or invalid, price for the given time is not yet set or downloaded or energy prices feature is disabled.

  This function is equivalent to `isNextDayHourPriceAvailable` with `minutes` set to 0.

Returns:

- *boolean*

Arguments:

- `hour` *(integer)* — hour in range 0–23
- `minutes` *(integer)* — minutes, a multiple of `prices_interval`, less than 60.
- `accessType` *(string?)* — allows selecting data source. One of: `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `getNextDayHourPrice(hour, accessType)`

  Returns energy price for `hour` in the next day.

  Return:

  - *number?* — energy price at `hour`. Returns nil if `hour` is out range, price for `hour` is not yet set or downloaded or Energy Prices feature is disabled.

  Arguments:

  - `hour` *(integer)* — hour in range 0–23
  - `accessType` *(string?)* — allows selecting data source. One of: `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `getNextDayPrice(hour, minutes, accessType)`

  Returns energy price for `hour` and `minutes` in the next day.

  This function is equivalent to `getgetNextDayHourPriceHourPrice` with `minutes` set to 0.

  Return:

  - *number?* — energy price at given time. Returns nil if time is out of range or invalid, price for given time is not yet set or downloaded or Energy Prices feature is disabled.

  Arguments:

  - `hour` *(integer)* — hour in range 0–23
  - `minutes` *(integer)* — minutes, multiplication of `prices_interval`, less than 60.
  - `accessType` *(string?)* — allows selecting data source. One of: `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `getNextDayPrices(accessType)`

  Returns energy prices table for the next day.

  > **Note**
  >
  > Returns `0` when price is not yet set, not downloaded or energy prices feature is disabled.

  Return:

  - *number[]* — table of prices with 24–96 elements (for each time interval of the day)

  Arguments:

- accessType *(string?)* — allows selecting data source. One of: `"api"`, `"lua"`. Falls back to configured `access_type` property if not provided.

- `clearNextDayPrices()`

  Clears next day prices

  Return:

  - *boolean* — true if prices were cleared, false if prices didn't change (already cleared)

- `setNextDayStaticPrice(price)`

  Sets same energy price for every hour in the next day.

  Returns:

  - *userdata* — reference to Energy Prices object, for call chains

  Arguments:

  - `price` *number* — energy price to set

- `setNextDayHourPrice(hour, price)`

  Sets energy price for one `hour` in the next day.

  Returns:

  - *userdata* — reference to Energy Prices object, for call chains

  Arguments:

  - `hour` *(integer)* — selected hour in range 0–23
  - `price` *(number)* — energy price to set

- `setNextDayHoursPrice(hours, price)`

  Sets single energy price for multiple `hours` in the next day

  Returns:

  - *userdata* — reference to Energy Prices object, for call chains

  Arguments:

  - `hour` *(table)* — table of hours in range 0–23
  - `price` *(number)* — energy price to set

- `setNextDayPrice(hour, minutes, price)`

  Sets energy price for the given time in the next day.

  This function is equivalent to `setNextDayHourPrice` with `minutes` set to 0.

  Returns:

  - *userdata* — reference to Energy Prices object, for call chains

  Arguments:

  - `hour` *(integer)* — selected hour in range 0 min – 23 min
  - `minutes` *(integer)* — minutes, a multiple of `prices_interval`, less than 60.

- ◦ `price` *(number)* — energy price to set

- `setNextDayPrices(prices)`

  Sets prices for the next day.

  Returns:

  - ◦ *userdata* — reference to Energy Prices object, for call chains

  Arguments:

  - ◦ `prices` *(number[])* — table of prices. If it's shorter than 24 elements only prices for first hours will be set.

- `moveNextDayPricesToCurrentDay()`

  Moves next day prices to current day, clears next day prices.

  Return:

  - ◦ *boolean* — true if prices changed

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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`.

- `prices_interval` *(integer)*

  The interval in which prices are given.

  Unit: 1 min.

  Can be one of values: 15, 30, 60.

  > **Note**
  >
  > Returns interval received from API when `access_type` is set to `api`.

Energy production

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 e.g. `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

Direct access to properties is not allowed. You can read values using the `getValue` method. An attempt at retrieving a nonexistent object property will cause a script error.

- `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: 1 Wh

- `total.energy_storage` *(number, read-only)*

  Total value of produced energy that was used to charge battery.

  Unit: 1 Wh

- `total.grid_export` *(number, read-only)*

  Total value of produced energy that was exported to grid.

  Unit: 1 Wh

- `today.autoconsumption` *(number, read-only)*

  Today value of produced energy that was autoconsumed by building.

  Unit: 1 Wh

- `today.energy_storage` *(number, read-only)*

  Today value of produced energy that was used to charge battery.

  Unit: 1 Wh

- `today.grid_export` *(number, read-only)*

  Today value of produced energy that was exported to grid.

  Unit: 1 Wh

Energy storage

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 e.g. `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

Direct access to properties is not allowed. You can read values using the `getValue` method. An attempt at retrieving a nonexistent object property will cause 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: 1 mW

- `energy_charged_today` *(number, read-only)*

  Daily sum of energy the battery was charged with.

  Unit: 1 Wh

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

  Daily sum of energy the battery was discharged with.

  Unit: 1 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: 1 %

Examples

Turn off relay when battery is discharging and level drops to 20%

```lua
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
```

Flow monitor

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 e.g. `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

Direct access to properties is not allowed. You can read values using the `getValue` method. An attempt at retrieving a nonexistent object property will cause 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: 1 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: 1 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: 1 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: 1 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: 1 %

- `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: 1 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: 1 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: 1 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: 1 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: 1 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: 1 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: 1 mW

- `building_consumption_details.by_devices` *(table, read-only)*

Represents collection of devices that provide their power consumption.

Unit: 1 mW

Examples

Turn off relay when you start importing power from grid

```lua
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

```lua
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
```

Schedules

Following modules allow accessing data of the Schedules, allowing to implement various control over time or control over external temperature algorithms. Scripts can use these modules to react to i.e. change of target temperature computed by schedule etc.

Thermal

Controls target temperature changes over time. The user can configure the target temperature for any time range (or ranges) during the day. Configuration can be done for each day of week separately.

Thermal schedule will match current time to configuration and compute target temperature. If no temperature range is configured for current time, it will set `fallback` as target temperature.

Thermostats can be assigned to thermal schedule in web application, to automatically handle target temperature updates.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

Available Properties

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

  Unique object identifier

- `name` *(string)*

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

- `icon` *(string)*

  User defined icon of schedule.

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

  Schedule type: `thermal`.

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

  Computed target temperature for current time if available or `fallback` value if not available.

  Unit: 0.1 °C.

- `fallback` *(integer)*

  This target temperature will be used when current time-point couldn't match any configured range.

  Unit: 0.1 °C.

  Range: 50 – 350 (5.0 °C – 35.0 °C)

- `monday` *(array-like table)*

  Collection of objects with time range ↔ target temperature configuration for monday.

  Schema of configuration object:

  - `start` *(integer)*

    Schedule start time-point in minutes of the day since 0:00. Must be lower than end.

    Range: 0 min – 1439 min (00:00 – 23:59)

  - `end` *(integer)*

    Schedule end timepoint in minutes of the day since 0:00. Must be greater than start.

    Range: 0 min – 1439 min (00:00 – 23:59)

  - `target_temperature` *(integer)*

    Target temperature for this time range.

    Unit: 0.1 °C

    Range: 50 – 350 (5.0 °C – 35.0 °C)

- `tuesday` *(array-like table)*

  Collection of objects with time range ↔ target temperature configuration for tuesday.

  Schema of configuration object: **see `monday` property for details**

- `wednesday` *(array-like table)*

  Collection of objects with time range ↔ target temperature configuration for wednesday.

  Schema of configuration object: **see `monday` property for details**

- `thursday` *(array-like table)*

  Collection of objects with time range ↔ target temperature configuration for thursday.

  Schema of configuration object: **see `monday` property for details**

- `friday` *(array-like table)*

  Collection of objects with time range ↔ target temperature configuration for friday.

  Schema of configuration object: **see `monday` property for details**

- `saturday` *(array-like table)*

  Collection of objects with time range ↔ target temperature configuration for saturday.

  Schema of configuration object: **see `monday` property for details**

- `sunday` *(array-like table)*

  Collection of objects with time range ↔ target temperature configuration for sunday.

  Schema of configuration object: **see `monday` property for details**

- `associations.thermostats` *(array of devices)*

  Reference to associated thermostats. Returns empty array when no devices associated.

- `associations.heat_pump_managers` *(array of devices)*

  Reference to associated heat pump managers. Returns empty array when no devices associated.

Commands

There are no commands available for this type of schedule.

Examples

Change friday configuration

```
schedule[4]:setValue("friday", {
    -- 25.5°C between 00:00 (0 minutes past midnight)
    --           and 10:00 (600 minutes past midnight)
    { ['target_temperature'] = 255, ['start'] =    0, ['end'] =  600 },

    -- 24.5°C between 20:00 (1200 minutes past midnight)
    --           and 22:00 (1320 minutes past midnight)
    { ['target_temperature'] = 245, ['start'] = 1200, ['end'] = 1320 },
})
```

> **Note**
>
> Unlike in many examples, time range configuration table should have escaped keys with `[' ']`, due to `end` being one of Lua keywords! Not escaping this key will result in script error!

Change schedule thermostats

```
local newThermostats = {
    { id = 12, class = "virtual" },
    { id = 13, class = "virtual" }
}
schedule[4]:setValue("associations.thermostats", newThermostats)
```

Temperature curve

Controls target temperature changes over different outdoor temperature (based on weather). The user can configure the target temperature for many outdoor temperature points.

Temperature curve will match current outdoor temperature to configuration points and compute target temperature. The target temperature is the result of interpolation between two outdoor temperature points.

If there is problem getting outdoor temperature, it will set `fallback` as target temperature.

Heat buffers, valves, pellet boilers and DHW can be assigned to temperature curve schedule in web application, to automatically handle target temperature updates.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

Available Properties

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

  Unique object identifier

- `name` *(string)*

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

- `icon` *(string)*

  User defined icon of schedule.

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

  Schedule type: `"temperature_curve"`.

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

  Computed target temperature for current outdoor temperature if available or `fallback` value if not available.

  Unit: 0.1 °C

- `outdoor_temperature_override` *(integer, nilable)*

  Override outdoor temperature for target temperature calculation. Override will last for 30 minutes since last `setValue`, then the temperature curve will use weather forecast data or associated temperature sensor data. If set to `nil`, the override will be cleared instantly.

Unit: 0.1 °C

Range: −500 – 500 (−50.0 °C – 50.0 °C)

- `fallback` *(integer)*

  This target temperature will be used when there is no valid outdoor temperature data from weather or points configuration is not correct.

  Unit: 0.1 °C

  Range: −500 – 1200 (−50.0 °C – 120.0 °C)

- `points` *(array-like table)*

  Collection of objects with outdoor temperature ↔ target temperature configuration.

  Schema of configuration object:

  - `outdoor_temperature` *(integer)*

    Related outdoor temperature for point on curve (X axis value).

    Unit: 0.1 °C.

    Range: −500 – 500 (−50.0 °C – 50.0 °C)

  - `target_temperature` *(integer)*

    Related target temperature for point on curve (Y axis value).

    Unit: 0.1 °C.

    Range: −500 – 1200 (−50.0 °C – 120.0 °C)

- `associations.heat_buffers` *(array of devices)*

  Reference to associated heat buffers. Returns empty array when no devices associated.

- `associations.valves` *(array of devices)*

  Reference to associated valves. Returns empty array when no devices associated.

- `associations.pellet_boilers` *(array of devices)*

  Reference to associated pellet boilers. Returns empty array when no devices associated.

- `associations.domestic_hot_waters` *(array of devices)*

  Reference to associated domestic hot waters. Returns empty array when no devices associated.

- `associations.temperature_sensor` *(device, nilable)*

  Reference to associated temperature sensor. Returns `nil` when no device associated. If temperature sensor is associated, the temperature curve will use this sensor data instead of weather forecast data.

Commands

There are no commands available for this type of schedule.

Examples

Change points configuration

```
schedule[4]:setValue("points", {
    { outdoor_temperature =    0, target_temperature = 255 },
    { outdoor_temperature = -100, target_temperature = 355 },
    { outdoor_temperature = -200, target_temperature = 455 },
    { outdoor_temperature = -300, target_temperature = 550 },
})
```

Change curve associations

```
local newHeatBuffers = {
    { id = 12, class = "tech" },
    { id = 13, class = "tech" }
}
schedule[4]:setValue("associations.heat_buffers", newHeatBuffers)

local newValves = {
    { id = 25, class = "tech" },
    { id = 26, class = "tech" }
}
schedule[4]:setValue("associations.valves", newValves)

local newPelletBoilers = {
    { id = 38, class = "tech" },
    { id = 44, class = "tech" }
}
schedule[4]:setValue("associations.pellet_boilers", newPelletBoilers)

local newDHWs = {
    { id = 66, class = "tech" },
    { id = 71, class = "tech" }
}
schedule[4]:setValue("associations.domestic_hot_waters", newDHWs)
```

Change temperature sensor association

```
-- add using association table
schedule[4]:setValue("associations.temperature_sensor", { id = 5, class = "tech"
   })

-- remove association using table
schedule[4]:setValue("associations.temperature_sensor", { id = 0, class = "" })

-- add using instance
schedule[4]:setValue("associations.temperature_sensor", tech[5])

-- remove using nil
schedule[4]:setValue("associations.temperature_sensor", nil)
```

Override outdoor temperature to 10.0°C

```
schedule[4]:setValue("outdoor_temperature_override", 100)
```

Relay control

Controls relay state changes over time. The user can configure the state for any time range (or ranges) during the day. Configuration can be done for each day of week separately.

Relay control schedule will match current time to configuration, compute state and set it accordingly to control policy (continous or once at start of time range).

A relay of any class can be assigned to schedule in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

Available Properties

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

  Unique object identifier

- `name` *(string)*

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

- `icon` *(string)*

  User defined icon of schedule.

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

  Schedule type: `relay_control`.

- `control_policy` *(string)*

  Policy of relay control. One of: `continuous`, `once_at_change`.

  - `continuous` means that the device state will be forced during specific time range - manual change will always be overridden.

  - `once_at_change` means that device state will be set only at time range start - manual change will not be overridden.

- `current.state` *(boolean, read-only)*

  Current state for the relays or `nil` if not available (day disabled).

- `monday.enabled` *(boolean)*

  Flag indicating if monday is enabled. If not enabled (`false`), no control will be applied for this day.

- `monday.configuration` *(array-like table)*

  Collection of objects with time range ↔ state configuration for monday, control will be applied according to configuration.

  > **Note**
  >
  > The configuration entries must not overlap!

  Schema of configuration object:

  - `start` *(integer)*

    Schedule start time-point in minutes of the day since 0:00. Must be lower than end.

    Range: 0 min – 1439 min (00:00 – 23:59)

  - `end` *(integer)*

    Schedule end timepoint in minutes of the day since 0:00. Must be greater than start.

    Range: 0 min – 1439 min (00:00 – 23:59)

  - `state` *(boolean)*

    Expected state for this time range.

- `tuesday.enabled` *(boolean)*

  Flag indicating if tuesday is enabled. If not enabled (`false`), no control will be applied for this day.

- `tuesday.configuration` *(array-like table)*

  Collection of objects with time range ↔ state configuration for tuesday, control will be applied according to configuration.

  > **Note**
  >
  > The configuration entries must not overlap!

  Schema of configuration object: **see `monday.configuration` property for details**

- `wednesday.enabled` *(boolean)*

  Flag indicating if wednesday is enabled. If not enabled (`false`), no control will be applied for this day.

- `wednesday.configuration` *(array-like table)*

  Collection of objects with time range ↔ state configuration for wednesday, control will be applied according to configuration.

  > **Note**
  >
  > The configuration entries must not overlap!

  Schema of configuration object: **see `monday.configuration` property for details**

- `thursday.enabled` *(boolean)*

Flag indicating if thursday is enabled. If not enabled (`false`), no control will be applied for this day.

- `thursday.configuration` *(array-like table)*

Collection of objects with time range ↔ state configuration for thursday, control will be applied according to configuration.

> **Note**
>
> The configuration entries must not overlap!

Schema of configuration object: **see `monday.configuration` property for details**

- `friday.enabled` *(boolean)*

Flag indicating if friday is enabled. If not enabled (`false`), no control will be applied for this day.

- `friday.configuration` *(array-like table)*

Collection of objects with time range ↔ state configuration for friday, control will be applied according to configuration.

> **Note**
>
> The configuration entries must not overlap!

Schema of configuration object: **see `monday.configuration` property for details**

- `saturday.enabled` *(boolean)*

Flag indicating if saturday is enabled. If not enabled (`false`), no control will be applied for this day.

- `saturday.configuration` *(array-like table)*

Collection of objects with time range ↔ state configuration for saturday, control will be applied according to configuration.

> **Note**
>
> The configuration entries must not overlap!

Schema of configuration object: **see `monday.configuration` property for details**

- `sunday.enabled` *(boolean)*

Flag indicating if sunday is enabled. If not enabled (`false`), no control will be applied for this day.

- `sunday.configuration` *(array-like table)*

Collection of objects with time range ↔ state configuration for sunday, control will be applied according to configuration.

> **Note**
>
> The configuration entries must not overlap!

Schema of configuration object: **see `monday.configuration` property for details**

- `associations.relays` *(array of devices)*

  Reference to associated relays. Returns empty array when no devices associated.

Commands

There are no commands available for this type of schedule.

Examples

Change friday configuration

```lua
schedule[4]:setValue("friday.configuration", {
    -- ON between 00:00 (0 minutes past midnight)
    --         and 10:00 (600 minutes past midnight)
    { ['state'] = true, ['start'] =    0, ['end'] =  600 },

    -- OFF between 20:00 (1200 minutes past midnight)
    --          and 24:00 (1440 minutes past midnight)
    { ['state'] = false, ['start'] = 1200, ['end'] = 1440 },
})
```

> **Note**
>
> Unlike in many examples, time range configuration table should have escaped keys
> with `[' ']`, due to `end` being one of Lua keywords! Not escaping this key will result
> in script error!

Change schedule relays

```lua
local newRelays = {
    { id = 12, class = "sbus" },
    { id = 2, class = "wtp" },
    { id = 13, class = "virtual" },
    { id = 66, class = "slink" }
}
schedule[4]:setValue("associations.relays", newRelays)
```

WTP devices

Devices using WTP radio communication protocol. A device has to be registered with Sinum central before use. Registration and device management can be performed via [REST API](#) or the web app.

WTP devices can be accessed from scripts by indexing global `wtp` container, e.g. `wtp[6]` returns WTP device with ID #6. This container is available in every execution context.

Properties can only be accessed via `setValue` and `getValue` methods.

Common WTP device properties

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

  Unique network address.

  > **Required label:** `"battery_powered"`
  >
  > - `battery` *(integer, read-only)*
  >
  >   Unit: 1%
  >
  >   Value range: ⟨0; 100⟩

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

  Signal status.

  Unit: 1%

  Value range: ⟨0; 100⟩

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

  Software name and version description.

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

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

  Value range: ⟨0; 10⟩

Air quality sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `pm1p0` *(number, read-only)*

  Measured concentration of PM1.0 value (particulate matter).

  Unit: $1\,\mu g/m^3$

- `pm2p5` *(number, read-only)*

  Measured concentration of PM2.5 value (particulate matter).

  Unit: $1\,\mu g/m^3$

- `pm4p0` *(number, read-only)*

  Measured concentration of PM4.0 value (particulate matter).

  Unit: $1\,\mu g/m^3$

- `pm10p0` *(number, read-only)*

  Measured concentration of PM10.0 value (particulate matter).

  Unit: $1\,\mu g/m^3$

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

  Descriptive name for air quality. Based on PM10.0 concetration.

Values ($1\,\mu g/m^3$)	Description
≤ 20	`very_good`
21 – 50	`good`
51 – 80	`moderate`
81 – 110	`poor`
111 – 150	`unhealthy`
> 150	`very_unhealthy`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"aq_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (**full spec**)

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

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

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

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

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

Blind controller

Controller opens and closes a roller shade, tilt blind or pergola.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_opening` *(number)*

  Desired setpoint opening, which device will try to achieve.

  Unit: 1 %

  > **Note**
  >
  > If device doesn't contain `percent_opening_control` label, target opening is limited to 0%, 50% or 100% (only these three).

- `current_opening` *(number, read-only)*

  Current opening value.

  Unit: 1 %

- `window_covering_type` *(string)*

  Defines the type of window covering the controller is connected to. Depending on the value of this parameter, the controller's behavior will change and some parameters may be unavailable.

  > **Note**
  >
  > Can be modified with values in `allowed_window_covering_types` property.

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

  List of available window covering types supported by the controller.

  Possible values: `roller_shade`, `tilt_blind`, `pergola`.

- `lift_control_mode` *(string)*

  Defines the control algorithm of lifting movement. Depending on the value of this parameter, the controller's behavior will change and some parameters may be unavailable.

  > **Note**
  >
  > Can be modified with values in `allowed_lift_control_modes` property.

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

  List of available lift control modes supported by the controller.

  Possible values: `current_detection`, `fixed_duration`.

Required label: `"percent_tilt_control"`

- `target_tilt` *(number, optional)*

  Desired tilt position.

  Unit: 1 %.

- `current_tilt` *(number, optional, read-only)*

  Current tilt position

  Unit: 1 %.

- `tilt_range` *(number, optional)*

  Determines tilt range.

  Unit: angle (degrees).

  > **Note**
  >
  > Can be modified when: `window_covering_type` is equal to `tilt_blind` or `pergola`.

  > **Note**
  >
  > When `window_covering_type` is equal to `tilt_blind` can be **only** set to 90 or 180, otherwise can be set to 0-180.

Required label: `"has_lift_duration"`

- `full_cycle_duration` *(number, optional)*

  Time required by motor to do full lift cycle from 100% to 0% or 0% to 100% (select larger). Proper full open or full close action is based on this value.

  Unit: seconds.

  > **Note**
  >
  > Can be modified when: `lift_control_mode` is equal to `fixed_duration`.

Required label: `"has_tilt_duration"`

- `tilt_duration` *(number, optional)*

  Time required by motor to do full tilt cycle.

  Unit: ms.

Required label: `"has_tilt_cycle_distance"`

- `tilt_cycle_distance` *(number, optional)*

  Number of motor steps a full tilt cycle takes.

Required label: `"has_motor_running_current_threshold"`

- `motor_running_current_threshold` *(integer, optional)*

  Current threshold that indicates motor is running.

  Unit: mA

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `auto`, `fixed`, `off`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `#FF00FF`

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active e.g. motor is working.

  Example: `#FFFF00`

Required label: `"button_inversion_support"`

- `buttons_inverted` *(boolean, optional)*

  Replace up and down buttons directions.

Required label: `"output_inversion_support"`

- `outputs_inverted` *(boolean, optional)*

  Replace up and down outputs directions.

Not available when labels contain: `"percent_opening_control"`

- `button_signal_type` *(string, optional)*

  Selected button specific behavior. e.g. impulse = on/off impulse is required to start action.

  Available values: `impulse`, `state_change`

- `output_signal_type` *(string, optional)*

  Selected output specific behavior. e.g. impulse = on/off impulse is required to start motor.

  Available values to set: `impulse`, `state_change`

Required label: `"has_backlight_brightness_sensor"`

- `ambient_light_intensity` *(number, read-only)*

  Measured ambient light intensity in percent.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"blind_controller"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties ([full spec](#))

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

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

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

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

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

Commands

- `open`

  Opens a blind to specific value in percent passed in argument.

  Argument:

  opening percentage *(number)*

- `up`

  Fully opens a blind.

  > **Note**
  >
  > Command is NOT available when `window_covering_type` is `pergola`.

- `down`

  Fully closes a blind.

  > **Note**
  >
  > Command is NOT available when `window_covering_type` is `pergola`.

- `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

```lua
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

```lua
if dateTime:changed() then
  if dateTime:getHours() == 12 and dateTime:getMinutes() == 0 then
    wtp[3]:call("open", 50)
  end
end
```

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

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `buttons_count` *(number, read-only)*

  Count of physical buttons.

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

  Last action performed by user. Lua patterns of available action kinds:

  `"button_(%d+)_clicked_(%d+)_times"`
  Given button has been clicked specified number of times.
  Examples:
    ◦ `"button_2_clicked_1_times"` — Single click was detected.
    ◦ `"button_1_clicked_8_times"` — A sequence of eight clicks was detected.

  `"button_(%d+)_hold_started"`
  Given button is being held down, starting from now.
  Example: `"button_1_hold_started"` — User just started holding the button down.

  `"button_(%d+)_held_(%d+)_seconds"`
  Given button has been released, after specified time of being held down.
  Example: `"button_1_held_8_seconds"` — User just released a button after holding it down for eight seconds.

- `buzzer` *(string)*

  Embedded buzzer (speaker) settings. One of following: `"on"`, `"off"`, `"unsupported"`

> **Required label:** `"single_click_mode_support"`

- `single_click_mode` *(boolean)*

  Enables single click mode. If enabled, only single click and hold actions will be reported, but it won't have report delay.

Required label: `"button_set_support"`

> **Note**
>
> A button set is a number displayed on the remote's LED screen (also called channels). It allows you to assign different actions to the same buttons depending on the currently selected set (channel)

- `button_set.total_count` *(number, read-only)*

  Total count of button sets. (Maximum available for the device)

- `button_set.available_count` *(number)*

  Available (configured as visible for user) count of button sets, cannot be larger than `button_set.total_count`.

- `button_set.current` *(number, read-only)*

  Indicates button set that is currently selected. Updated when button action is triggered.

Device properties (full spec)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"button"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (**full spec**)

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

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

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

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

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

Commands

- `emit`

  Emits button action event by software.

  Argument:

  packed arguments *(table)*:

  - `button_index` - *(number, required)* index of the button that the action event concerns, minimum: 1, maximum: value of `buttons_count`
  - `type` - *(string, required)* action that will be emitted, possible values: `click`, `hold_start`, `hold_end`
  - `duration` - *(number, optional)* duration of the hold in seconds or amount of clicks, ignored if the type is `hold_start`, minimum: 1, maximum: 600
  - `set` - *(number, optional)* the current button set (channel) that the action applies to (changes `button_set.current`), minimum: 1, maximum: value of `button_set.available_count`

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

```lua
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
```

Handle multi-set buttons (Remote Control with LED display)

```lua
-- Remote Control with LED display
local button <const> = wtp[9]

-- Get all blinds/pergolas in a table, in order they are displayed on LED
-- display (1, 2, 3, ...)
local blinds <const> = { wtp[5], wtp[6], wtp[7] }

-- Check if button action has been performed
if button:changedValue("action") then

    -- Get current set (displayed on LED display)
    local set <const> = button:getValue("button_set.current")

    if set > #blinds then
      -- Set is out of range, we do not have such amount of
      -- blinds/pergolas configured in table above, do nothing
      return
    end

    -- Handle actions, pass to corresponding blind or pergola
    local action <const> = button:getValue("action")
    if action == "button_1_clicked_1_times" then
        blinds[set]:call("up")
    elseif action == "button_2_clicked_1_times" then
        blinds[set]:call("stop")
    elseif "button_3_clicked_1_times" then
        blinds[set]:call("down")
    end

end
```

Emit button action

```lua
local button = wtp[9]

-- Emit 2 clicks on button 1
button:call("emit", { button_index = 1, type = "click", duration = 2 })

-- Emit hold start on button 2, change set (channel) to 3
button:call("emit", { button_index = 2, type = "hold_start", set = 3 })

-- Emit hold end after 10 secs on button 2 in set 3 (channel 3)
```

```lua
    -- NOTE: if the previous action was emitted by button in set 3, all subsequent actions w
-- until the set is changed by remote or by using `set` property of the command
button:call("emit", { button_index = 2, type = "hold_end", duration = 10 })
```

CO_2 sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `co2` *(number, read-only)*

  Measured CO_2 value.

  Unit: 1 PPM.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"co2_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

Dimmer

Device that controls light intensity of output LED.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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`). If it is set to 0, the dimmer will be turned off.

  Unit: Unit: 1 %.

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `"auto"`, `"fixed"`, `"off"`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `"#FF00FF"`

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active e.g. motor is working.

  Example: `"#FFFF00"`

Required label: `"has_backlight_brightness_sensor"`

- `ambient_light_intensity` *(number, read-only)*

  Measured ambient light intensity in percent.

Required label: `"has_output_temperature_sensor"`

- `output_temperature` *(number, read-only)*

  Measured temperature value of hardware output circuit. Unit: °C with one decimal number, multiplied by 10.

Required label: `"level_calibration_support"`

- `level_calibration.minimum` *(number)*

  Absolute minimum level that can be set on the device. Could not be larger than maximum. `target_level` will be internally scaled between `level_calibration.minimum` and `level_calibration.maximum`. Value range: ⟨0; 255⟩

- `level_calibration.maximum` *(number)*

  Absolute maximum level that can be set on the device. Could not be less than minimum. `target_level` will be internally scaled between `level_calibration.minimum` and `level_calibration.maximum`. Value range: ⟨0; 255⟩

Required label: `"switch_mode_support"`

- `switch_mode` *(string)*

  Depending on the value of this parameter, the swiches behavior will change. Can be modified with values in `allowed_switch_modes` property.

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

  List of available switch modes supported by the device.

  Possible combination of values:

  ○ `single_monostable` — (mode 1) Controls connected level control using single monostable input.

  ○ `single_bistable` — (mode 2) Controls connected level control using single bistable input.

    Click: toggle between SAVED and OFF level
    Hold: increase/decrease level

  ○ `up_and_down` — (mode 3) Controls connected level control using separate up and down inputs.

  ○ `single_monostable_dim_hotkey` — (mode 4) Controls connected level control using single monostable input.

    Click: toggle between SAVED and OFF level

> **Hold**: increase level
> **Click then hold hotkey**: decreases level
>
> ○ `single_monostable_dim_saved_hotkey` — (mode 5) Controls connected level control using single monostable input.
>
> **Click**: toggle between MAX and OFF level
> **Hold**: increase level
> **Click then hold hotkey**: decreases level
> **Double click hotkey**: goes to SAVED level

Device properties (full spec)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"dimmer"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

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
    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})
```

Dim or brighten lights while button is pressed (simple version)

Solution Drawback: it will always take constant time to move from 1% → 100%, 50% → 100%, 10% → 1% etc.

```lua
local dimmer = wtp[4]
local button = wtp[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_") or action:find("button_2_held_")
  then
    -- stop current moving action
    dimmer:call("stop")
  end
end
```

Dim or brighten lights while a button is pressed (advanced version)

This solution uses constant dimming rate instead of constant time.

```lua
local buttonUp, buttonDown = wtp[86], wtp[87]
local dimmer = wtp[126]

-- maximum fading time in 0.1s
local fadeTime = 50

-- calculate duration proportional to level difference
local function constantRateMove(currentLevel, desiredLevel)
  local diff = math.abs(desiredLevel - currentLevel)
  local time = utils.math:scale(0, 100, 1, fadeTime, diff)

  -- this table can be used directly as 'set_level' command argument
  return { desiredLevel, math.floor(time) }
end

if buttonUp:changedValue('action') then
  local action = buttonUp:getValue('action')
  if action == 'button_1_hold_started' then
    -- first button pressed, start moving towards full brightness
    local currentLevel = dimmer:getValue('target_level')
    dimmer:call('set_level', constantRateMove(currentLevel, 100))
  elseif action:find('button_1_held_', 1, true) then
    -- button released, stop the transition
    dimmer:call('stop')
  end
end

if buttonDown:changedValue('action') then
  local action = buttonDown:getValue('action')
  if action == 'button_1_hold_started' then
    -- second button pressed, start moving towards minimal brightness
    local currentLevel = dimmer:getValue('target_level')
    dimmer:call('set_level', constantRateMove(currentLevel,   1))
  elseif action:find('button_1_held_', 1, true) then
    -- button released, stop the transition
    dimmer:call('stop')
```

```
        end
    end
```

Energy meter

Energy meter is a device which can track and count consumed energy (total so far and daily) and sense voltage/current and active power.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `current` *(number, read-only)*

  Recent AC current measument.

  Unit: 1 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: 1 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: 1 mW.

- `energy_consumed_today` *(number, read-only)*

  Sum of energy used today.

  Unit: 1 Wh.

- `energy_consumed_yesterday` *(number, read-only)*

  Sum of energy used yesterday.

  Unit: 1 Wh.

- `energy_consumed_total` *(number, read-only)*

  Total sum of energy used Unit: 1 Wh.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties ([full spec](#))

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

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

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

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

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

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

```lua
-- 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
```

Fan control

Fan Control is a device which is used to control ventilation fan.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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 CO_2 thresholds specifying working in `automatic` state.

- `state_configuration.holiday.air_out_interval` *(number)*

  Interval for airing in `holiday` state.

  Unit: 1 day

- `state_configuration.hurricane.default_duration` *(number)*

  Default duration of `hurricane` state.

  Unit: 1 s

- `state_configuration.party.default_duration` *(number)*

  Default duration of `party` state.

  Unit: 1 s

- `state_configuration.hearth.default_duration` *(number)*

  Default duration of `hearth` state.

  Unit: 1 s

- `state_configuration.flaccid.default_duration` *(number)*

  Default duration of `flaccid` state.

  Unit: 1 s

- `computed_flow` *(number)*

  Value of computed flow passed from other devices.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"fan_control"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties ([full spec](#))

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

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

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

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

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

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 forbidded for permanent states and is optional for temporal states. If it is not passed default duration is used.

Flood sensor

Battery powered, flood sensor. Detects water leak on flat surfaces.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

  A flag representing the detection of flood / water leak by the sensor.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"flood_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (**full spec**)

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

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

- signal *(integer, read-only)*

- software_version *(string, read-only)*

- sub_id *(integer, read-only)*

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
```

Humidity sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `humidity` *(number, read-only)*

  Measured humidity value.

  Unit: RH% with one decimal number, multiplied by 10. (0.1 %)

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"humidity_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

IAQ sensor

Battery powered index of air quality sensor. Calculates air quality index based on various measures like CO_2 or particles level and relative humidity.

Sensors measure values only every few minutes to save battery.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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 ongoing
`high`	Calibration is done, now IAQ estimate achieves the 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`

Device properties (full spec)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"iaq_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

Light sensor

Battery powered light sensor. Measures light illuminance in lux and sends measurement to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

  Measured light illuminance value.

  Unit: 1 lx

Device properties (full spec)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"light_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

Motion sensor

Battery powered motion sensor. Based on custom configuration checks whether motion
was detected.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`
and `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting
wrong value type will cause a script error.*

- `enabled` *(boolean)*

  Enable or disable sensor. e.g. If you want sense only at night-time, you can set up an
  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.

  > **Note**
  >
  > The value will remain `true` all the time when subsequent motion detections occur
  > until motion stops.

  This parameter doesn't emit event when switch from `true` to `true` happens
  (subsequent motion detections). If you need to observe such action, you need to use
  `time_since_motion` parameter and check if `time_since_motion` equals to `0`.

- `time_since_motion` *(number, read-only)*

  Time since last motion detected. Value of -1 means there wasn't any motion since last
  system startup.

> **Note**
>
> The value will be `0` for each detected move, even if the previous one has not yet finished.

Unit: seconds.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"motion_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties ([full spec](#))

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

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

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

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

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

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)*, at least 1 s

Examples

Catching motion events

```lua
if wtp[4]:changedValue("motion_detected") then
    print("someone is moving around!")
end
```

```lua
if wtp[4]:changedValue("time_since_motion")
  then
  if wtp[4]:getValue("time_since_motion") == 0
  then
    print("someone is moving around!")
  end
end
```

Delayed action

```lua
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

```lua
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("time_since_motion") then
    if wtp[4]:getValue("time_since_motion") == 0 then
        wtp[60]:setValue("state", true)
        wtp[60]:setValueAfter("state", false, 5 * 60)
    end
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
```

Opening sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

  Opening sensor state. Open/Closed.

- `acknowledgment` *(string)*

  Newer sensors support communication protocol with acknowledgment. When enabled, sensor will try to 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`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"opening_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties ()

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

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

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

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

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

Examples

Catch open and close events

```lua
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
```

Pressure sensor

Battery powered pressure sensor. Measures pressure and sends measurement to central unit.

Sensors measure pressure only every few minutes to save battery.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

  Measured pressure value.

  Unit: hPa with one decimal number, multiplied by 10.

- `altitude` *(integer)*

  Setting the altitude compensates the atmospheric pressure reading to the pressure at mean sea level, that is normally given in weather reports. Possible range of altitude: *[0 – 8849]*

  Unit: msl, meters above sea level for your location.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"pressure_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

RGB controller

Device that controlls color and light intensity of output LED.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 %.

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

  HTML/Hex RGB color that device will set on its output led strip.

  Example: `"#00ff7f"`

- `white_temperature` *(number, read-only)*

  White temperature that device will set on its output led strip.

  Unit: 1 K

- `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.

Device properties (full spec)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"rgb_controller"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

Commands

- `turn_on`

  Turns on output.

  Argument

  Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `turn_off`

  Turns off output.

Argument

Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `toggle`

Changes state to opposite.

Argument

Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `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`.

- `stop`

  Calls RGB controller to stop current moving action. Does nothing if no action is in progress.

Examples

Turn on light to specific color at 19:00 and turn off at 21:00

```lua
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

```lua
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

```lua
local rgb = wtp[79]
local animation_id = 2

rgb:call("activate_animation", { id = animation_id })
```

Stop active animation

```lua
local rgb = wtp[79]
rgb:call("stop_animation")
```

Activate an animation by ID when device state changes

```lua
local rgb = wtp[79]
local animation_id = 3

if wtp[3]:changedValue("state") then
  rgb:call("activate_animation", { id = animation_id })
end
```

Radiator actuator

Battery powered radiator actuator. Controls valve opening based on e.g. temperature
regulator or thermostat state.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`
and `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting
wrong value type will cause a script error.*

- `opening` *(number)*

  Current valve opening level.

  Unit: 1 %

- `opening_minimum` *(number)*

  Lower valve opening level limit. Could not be greater than maximum. Setting minimum
  value above current opening value, will also change current opening value to minimum.

  Unit: 1 %

- `opening_maximum` *(number)*

  Upper valve opening level limit. Could not be less than minimum. Setting maximum
  value below current opening value, will also change current opening value to maximum.

  Unit: 1 %

- `valve_temperature` *(number, optional)*

  Measured valve temperature value.

  > **Note**
  >
  > Parameter is optional. Available when: `has_valve_temperature_sensor` label is
  > provided.

  Unit: 0.1 °C

- `emergency_opening` *(number, optional)*

  Emergency opening level when communication with central device is lost.

  > **Note**
  >
  > Parameter is optional. Available when: `emergency_opening_support` label is
  > provided.

  Unit: 1 %

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"radiator_actuator"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (**full spec**)

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

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

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

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

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

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 if actuator does not have `calibration_support` label!

Examples

Regulate valve based on room temperature

```lua
sensor = wtp[1]
valve = wtp[2]

if sensor:changedValue("temperature") then
    local reading = sensor:getValue("temperature")

    if reading > 220 then
        valve:call("open", 0)
    elseif reading > 200 then
        valve:call("open", 50)
    else
        valve:call("open", 100)
    end
end
```

Relay

Execution module that changes state depending on the control signal. Relay can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `state` *(boolean)*

  State of the output. On/Off.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `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.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `"auto"`, `"fixed"`, `"off"`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `"#FF00FF"`

- `backlight_active_color` *(string)*

HTML/Hex RGB representation of color when controller is active e.g. motor is working.

Example: `"#FFFF00"`

- `inverted` *(boolean)*

Indicates if physical state of relay should be the inversion of state shown in application.

- `time_since_state_change` *(number, read-only)*

Time since last relay state change. Unit: seconds.

Required label: `"relay_startup_state_support"`

- `startup_state` *(string)*

State of output that should be set on device after power restart. Available values: `"off"`, `"on"`, `"previous"`

Cannot be changed if device is assigned to thermostat or thermostat output group (virtual contact) / (has label `managed_by_thermostat` or `managed_by_tog`). Cannot be changed when relay `work_mode` is set to `alarm_siren`.

Required label: `"no_output_mode_support"`

- `no_output_mode` *(boolean)*

Indicates that relay is in no output mode - it takes state update as usual but hardware output remains off.

Required label: `"trigger_signal_config_support"`

- `trigger_signal_type` *(string)*

Available values:

`"impulse"`
   Monostable trigger — impulse signal toggles the output state
`"state_change"`
   Bistable trigger — when trigger state changes, output will be set equal to trigger signal

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

List of available trigger signal types supported by the device.

Possible combination of values:

  - `impulse`
  - `state_change`

> **Required label:** `"has_backlight_brightness_sensor"`
>
> - `ambient_light_intensity` *(number, read-only)*
>
>   Measured ambient light intensity in percent.

- `work_mode` *(string)*

  Relay work mode. One of: `standard`, `alarm_siren`.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"relay"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (**full spec**)

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

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

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

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

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

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

```lua
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

```lua
if wtp[4]:changedValue("motion_detected") then
  wtp[60]:setValue("state", true)
  wtp[60]:setValueAfter("state", false, 5 * 60)
end
```

Smoke sensor

Battery powered, optical smoke sensor. Detects smoke presence, high temperature (e.g. fire) and tamper.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 (e.g. fire) by the sensor.

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

  A flag representing the detection of tamper (e.g. 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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"smoke_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties ([full spec](#))

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

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

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

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

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

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

```lua
if wtp[5]:changedValue("smoke_detected") and wtp[5]:getValue("smoke_detected")
then
  print("Sensor detected smoke!!!")
end
```

```lua
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

```lua
-- 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

```lua
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
```

Temperature regulator

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.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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 doesn't 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_minimum` *(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). e.g. 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 or heat pump manager (label `managed_by_thermostat` and `managed_by_heat_pump_manager` not present).

  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.

Required label: `"user_menu_lock_support"`

- `user_menu_lock.enabled` *(boolean)*

  Indicates that it is required to enter pin code to access device's user menu.

- `user_menu_lock.pin_code` *(string)*

  Pin Code to access device's user menu. Has to be longer or equal to `user_menu_lock.pin_code_length_minimum` and shorter or equal to `user_menu_lock.pin_code_length_maximum` and contains characters from `user_menu_lock.allowed_characters`.

- `user_menu_lock.pin_code_length_minimum` *(integer, read-only)*

  Minimum length of user menu pin code.

- `user_menu_lock.pin_code_length_maximum` *(integer, read-only)*

  Maximum length of user menu pin code.

- `user_menu_lock.allowed_characters` *(integer, read-only)*

  Allowed characters of user menu pin code.

Device properties ([full spec]())

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_regulator"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties ([full spec](#))

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

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

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

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

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

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

```lua
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
```

Temperature sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

  Measured 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.

  > **Note**
  >
  > Parameter will be read-only if `factory_calibrated` label is present!

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

Throttle

Standalone radio controlled throttle.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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
  ```

Device properties (full spec)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"throttle"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties ([full spec](#))

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

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

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

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

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

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
```

Two-state input sensor

Boolean input sensor checks input state and send it to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"throttle"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

WTP device properties (full spec)

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

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

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

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

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

Fan coil

Fan coil turns on gears for heating and cooling with the hysteresis from target temperature.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `gear_1.state` *(boolean, read-only)*

  Current state of the gear 1 output.

- `gear_1.hysteresis` *(integer)*

  Hysteresis from target temperature for the gear output to turn on. Unit: °C with one decimal number, multiplied by 10.

- `gear_2.state` *(boolean, read-only)*

  Current state of the gear 2 output.

- `gear_2.hysteresis` *(integer)*

  Hysteresis from target temperature for the gear output to turn on. Unit: °C with one decimal number, multiplied by 10.

- `gear_3.state` *(boolean, read-only)*

  Current state of the gear 3 output.

- `gear_3.hysteresis` *(integer)*

  Hysteresis from target temperature for the gear output to turn on. Unit: °C with one decimal number, multiplied by 10.

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

  Timestamp of last device factory reset. Returns nil when no factory reset information received.

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

  State of the valve output.

IR Heater

IR foil heater controller. IR Heater can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(boolean)*

  State of the output. On/Off.

  Cannot be changed if device is assigned to thermostat (has label `managed_by_thermostat`).

- `foil_working_temperature` *(number)*

  Working temperature of the IR foil.

  Unit: °C with one decimal number, multiplied by 10.

- `foil_current_temperature` *(number, read-only)*

  Current IR foil temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `triac_current_temperature` *(number, read-only)*

  Current triac temperature.

  Unit: °C with one decimal number, multiplied by 10.

- `pwm_duty` *(number, read-only)*

  PWM duty of the IR foil controller.

  Unit: % with two decimal numbers, multiplied by 100.

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

  IR heater controller state. One of: `idle`, `heating_full_power`, `heating_temp_rampe_up`, `heating_keep_temp`, `cooling_temp_rampe_down`, `alarm`, `test`, `manual`.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"wtp"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"ir_heater"`

- `variant` *(string, read-only)* = `"simple_fan_coil"`

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

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

- `purpose` *(string)*

WTP device properties ([full spec](#))

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

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

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

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

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

Commands

- `turn_on`

  Turns on output.

- `turn_off`

  Turns off output.

- `toggle`

  Changes output to opposite.

- `reset_alarms`

  Reset current alarms in IR heater controller.

Examples

Turn on IR heater based on room temperature

```
sensor = wtp[1]
ir_heater = wtp[2]

if sensor:changedValue("temperature") then
    local temperature = sensor:getValue("temperature")

    if temperature >= 220 then
        valve:call("turn_off", 0)
    elseif temperature <= 215 then
        valve:call("turn_on", 50)
    end
end
```

TECH RS devices

Wired TECH RS devices connected to the central with 6P6C plugs.

A device may be added by registration using web application. Can be edited or deleted via [REST API](#) or the web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `tech` container e.g. `tech[6]` gives you access to device with **ID 6**. TECH devices have global scope and they are visible in all executions contexts.

Common TECH device properties

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

  TECH device unique address.

Common heat buffer

Device plugged into RS input in central unit. Heat buffer representation. Allows user to read and modify buffer parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `target_temperature` *(number)*

  Desired target (setpoint) temperature, which device will try to achieve.

  Unit: 1 °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 e.g. 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)*

  Measured temperature in lower part of buffer.

  Unit: °C with one decimal number, multiplied by 10.

  > **Note**
  >
  > Parameter is optional. Available when `temperature_down_available` label is present.

- `temperature_up` *(number, read-only)*

  Measured 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](#).

- `target_temperature_minimum` *(number, read-only)*

  Lower limit of the target temperature.

  Unit: 1 °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature.

  Unit: 1 °C

- `correction` *(number, read-only)*

  Target temperature correction resulting from some algorithms in valve controller. Unit: 1 °C

Device properties (full spec)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_heat_buffer"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (full spec)

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

Additional CH pump

Device plugged into RS input in central unit. Additional CH pump representation. Allows user to read and modify CH pump parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 the 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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"ch_pump_additional"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (full spec)

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

Common DHW

Device plugged into RS input in central unit. Common DHW representation. Allows user to read and modify DHW parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired target (setpoint) temperature, which device will try to achieve. Unit: 1 °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 e.g. 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: 1 °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature. Can't be less than minimum.

  Unit: 1 °C

- `correction` *(number, read-only)*

  Target temperature correction.

  Unit: 1 °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.

Device properties (full spec)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (full spec)

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

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
```

Additional DHW pump

Device plugged into RS input in central unit. Additional DHW Pump representation. Allows user to read and modify DHW Pump parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 °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: 1 °C

- `temperature_threshold` *(number, read-only)*

  Current threshold temperature.

  Unit: 1 °C

Device properties (**full spec**)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_additional"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties ([full spec](#))

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

Additional floor pump

Device plugged into RS input in central unit. Additional floor pump representation. Allows user to read pump parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 the 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: 1 °C

- `maximum_temperature` *(number, read-only)*

  Upper limit of the floor temperature.

  Unit: 1 °C

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_floor_pump_additional"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties ([full spec](#))

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

Heat pump

Device plugged into RS input in central unit. Heat pump representation. Allows user to read and modify heat pump parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `sub_id` *(number, read-only)*

  Unique (per device container) identifier that helps to distinguish same device types in one container.

- `tags` *(array-like table)*

  Collection of tags (array-like table of strings) 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: 1 W

- `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: 1 W

- `valve_buffer_state_text` *(number, read-only)*

  Current valve-buffer state text ID

- `ehome_work_mode` *(string)*

  Current heat pump work mode. One of `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: 1 L/h (-1 error)

- `current_power_consumption` *(number, read-only)*

  Current power consumption.

  Unit: 1 W

- `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: 1 Pa

- `compressor_last_work_time` *(number, read-only)*

  Current compressor last work time Unit: second.

- `temperature_return` *(number, read-only)*

  Current return temperature.

Unit: °C with one decimal number, multiplied by 10.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (**full spec**)

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

Commands

- `set_ehome_work_mode`

  Changes eHome work mode (device climate mode).

  Arguments:

  ○ *(string)* - eHome work mode (climate mode), one of: `auto`, `heating`, `cooling`

- `set_work_mode`

  Changes work mode.

  Arguments:

  ○ *(number)* - work mode ID, one of available in property `work_mode_list`

Humidity sensor

Device plugged into RS input in central unit. Measures humidity and sends measurement to central unit. Can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `humidity` *(number, read-only)*

  Measured humidity value.

  Unit: rH% with one decimal number, multiplied by 10.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"tech"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*
- `status` *(string, read-only)*
- `tags` *(string[])*
- `type` *(string, read-only)* = `"humidity_sensor"`
- `variant` *(string, read-only)* = `"generic"`
- `visible` *(boolean, read-only)*
- `voice_assistant_device_type` *(string, read-only)*
- `purpose` *(string)*

TECH device properties (**full spec**)

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

Pellet boiler

Device plugged into RS input in central unit. Pellet boiler representation. Allows user to read and modify boiler parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 %

- `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: 1 °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 e.g. computed by heat curve.

  > **Note**
  >
  > Can be changed only if device has a temperature curve associated.

  Available values: `fixed`, `heat_curve`.

  Default: `fixed`

- `target_temperature_minimum` *(number, read-only)*

  Lower limit of the target temperature.

  Unit: 1 °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature.

  Unit: 1 °C

- `correction` *(number, read-only)*

  Target temperature correction resulting from some algorithms in pellet controller.

  Unit: 1 °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` *(sequence of numbers, read-only)*

  Table of text IDs which show a cause of damping. ID text from TECH translations.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"pellet_boiler"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties ([full spec](#))

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

Examples

Stop pellet boiler when all thermostats reach their target temperatures

```lua
thermostats = { virtual[3], virtual[4], virtual[5], virtual[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
```

Main pellet CH

Device plugged into RS input in central unit. Pellet CH representation. Allows user to read and modify CH parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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`

Device properties (full spec)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"pellet_ch_main"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (full spec)

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

Examples

Change modes based on current season

```lua
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
```

Additional protection pump

Device plugged into RS input in central unit. Allows user to read parameters of the additional pump.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"pellet_ch_main"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (**full spec**)

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

Additional relay

Device plugged into RS input in central unit. Allows user to read additional relay parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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.

Device properties (full spec)

- `class` *(string, read-only)* = `"tech"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*
- `status` *(string, read-only)*
- `tags` *(string[])*
- `type` *(string, read-only)* = `"common_relay_additional"`
- `variant` *(string, read-only)* = `"generic"`
- `visible` *(boolean, read-only)*
- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

TECH device properties (full spec)

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

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(boolean)*

  State of the output. On/Off.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `timeout` *(number)*

  Protection functionality, that will set device state to off if there are communication problems.

  Unit: 1 min

- `timeout_enabled` *(boolean)*

  Parameter that indicates if timeout functionality is enabled.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `inverted` *(boolean)*

  Indicates if should invert physical state of relay compared to represented state in application.

- `time_since_state_change` *(number, read-only)*

  Time since last relay state change.

  Unit: 1 s

- `work_mode` *(string)*

  Relay work mode. One of: `standard`, `alarm_siren`.

Device properties (full spec)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"relay"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties ([full spec](#))

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

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
```

Temperature regulator

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.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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 whether the regulator works in `constant` mode with one setpoint, `time_limited` mode with a temporary setpoint, or according to a schedule in `schedule` mode, with target temperature changing in time.

  Parameter is read only, use commands to change target temperature mode! Parameter cannot be `schedule` if thermostat doesn't have the `has_schedule` label! When not associated with a 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, appropriate commands have to be used instead.

  Unit: minutes.

- `target_temperature_minimum` *(number)*

  Lower limit of the target temperature. Can't 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) e.g. an LED. 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.

Device properties (full spec)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_regulator"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (full spec)

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

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:

- *number* – target temperature in 0.1 °C

- `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:

- *number* – target temperature in 0.1 °C

- `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 a thermostat.

Argument:

- *table* – packed arguments:
  - *number* – remaining time in minutes
  - *number* – target temperature in 0.1 °C

- `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 a thermostat.

Examples

Raise target temperature between 15:00 and 20:00

```lua
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
```

Temperature sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `temperature` *(number, read-only)*

  Measured 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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (full spec)

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

Two state input sensor

Device plugged into RS input in central unit. Boolean input sensor checks input state and send it to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

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

  State of the input. On/Off.

- `inverted` *(boolean)*

  Indicates if physical state of input should be inverted compared to the state represented in system.

Device properties (full spec)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"two_state_input_sensor"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties (full spec)

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

Valve

Device plugged into RS input in central unit. Valve representation. Allows user to read and modify valve parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired target (setpoint) temperature, which device will try to achieve.

  Unit: 1 °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 e.g. computed by a 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.

  Unit: 1 °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature.

  Unit: 1 °C

- `correction` *(number, read-only)*

  Target temperature correction resulting from some algorithms in valve controller. Unit: 1 °C

- `temperature_valve` *(number, read-only)*

  Current valve temperature.

  Unit: 0.1 °C

- `open_percent` *(number, read-only)*

  Current open percentage.

Unit: 1%

- `state` *(number, read-only)*

  Valve working state identifier, with following meanings:

  1. Off
  2. Calibration
  3. Reserved
  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: 0.1 °C

- `temperature_central_heating` *(number, read-only)*

  Current central heating temperature.

  Unit: 0.1 °C

- `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: 0.1 °C

- `work_mode` *(string)*

  Current valve work mode (heating, cooling)

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_valve"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties ([full spec](#))

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

Examples

Close valve if thermostat reached target temperature

```lua
if dateTime:changed() then
  local thermostat = virtual[3]
  local valve = tech[3]

  local temperature_reached = not thermostat:getValue("state")
  valve:setValue("blockade", temperature_reached)
end
```

Ventilation

Device plugged into RS input in central unit. Allows user to read and modify valve parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 °C

- `target_temperature_minimum` *(number, read-only)*

  Lower limit of the target temperature.

  Unit: 1 °C

- `target_temperature_maximum` *(number, read-only)*

  Upper limit of the target temperature.

  Unit: 1 °C

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

  Current cooling state,

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

  Current preheating state.

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

  Current postheating 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. `"additional_temperature_supply_available"` label has to be present.

- `additional_temperature_outside` *(number, read-only)*

  Current additional_temperature_outside Unit: °C with one decimal number, multiplied by 10.

  > **Note**
  >
  > Parameter is optional. `"additional_temperature_outside_available"` label has to be present.

- `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: 1 %

- `co2ppm` *(number, read-only)*

  Current CO_2 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: $1\,\mathrm{m^3/h}$

- `extract_fan_flow` *(number, read-only)*

  Current extract fan flow.

  Unit: $1\,\mathrm{m^3/h}$

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

  Ventilation flow modeet If set to true ventilation working with flow settings

- `target_flow_supply` *(number)*

  Desired setpoint flow supply, which device will try to achieve.

  Unit: $1\,\mathrm{m^3/h}$

- `target_flow_extract` *(number)*

  Desired setpoint flow extract, which device will try to achieve.

  Unit: $1\,\mathrm{m^3/h}$

- `min_flow` *(number, read-only)*

  Lower limit of the `target target_flow`.

  Unit: $1\,\mathrm{m^3/h}$

- `max_flow` *(number, read-only)*

  Upper limit of the `target target_flow`.

  Unit: $1\,\mathrm{m^3/h}$

- `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: $1\,\%$

- `target_gear_extract` *(number)*

  Desired setpoint gear extract, which device will try to achieve.

  Unit: 1 %

- `bypass_work_mode` *(number)*

  The bypass current operating mode (text ID)

  > **Note**
  >
  > Parameter is optional. Available only when `bypass_available` label is present.

- `bypass_work_mode_list` *(number, read-only)*

  Available bypass work mode list (text ID)

- `gwc_work_mode` *(number)*

  The gwc current operating mode (text ID)

- `gwc_work_mode_list` *(number, read-only)*

  Available gwc work mode list (text ID)

Device properties (full spec)

- `class` *(string, read-only)* = `"tech"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"ventilation"`

- `variant` *(string, read-only)* = `"generic"`

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

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

- `purpose` *(string)*

TECH device properties ([full spec](#))

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

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)*

- `set_gwc_work_mode`

  Calls device to change gwc work mode

  Argument:

  gwc work mode text ID, one of available in property `gwc_work_mode_list` *(number)*

Modbus devices

Modbus devices connected to the central via RS-485 port.

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 e.g. `modbus[6]` gives you access to device with **ID 6**. Modbus devices have global scope and they are visible in all executions contexts.

Alpha-Innotec — Heat pump

Representation of Heat Pump related parameters of Alpha-Innotec device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `temperature_indoor` *(number, read-only)*

  Indoor temperature.

  Unit: 0.1 °C.

- `target_temperature_indoor` *(number, read-only)*

  Set indoor temperature.

  Unit: 0.1 °C.

- `fixed_heating_target_temperature` *(number)*

  Set temperature for heating in fixed temperature mode.

  Unit: 0.1 °C.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: 0.1 °C.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: 0.1 °C.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: 0.1 °C.

- `hot_gas_temperature` *(number, read-only)*

  Hot gas temperature.

  Unit: 0.1 °C.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: 0.1 °C.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: 0.1 °C.

- `overheating` *(number, read-only)*

  Overheating.

  Unit: 0.1 K.

- `lower_source_out_temperature` *(number, read-only)*

  Lower source out temperature.

  Unit: 0.1 °C.

- `lower_source_in_temperature` *(number, read-only)*

  Lower source in temperature.

  Unit: 0.1 °C.

- `heat_quantity_hot_water` *(number, read-only)*

  Heat quantity domestic hot water.

  Unit: 0.1 kWh.

- `heat_quantity_heating` *(number, read-only)*

  Heat quantity heating.

  Unit: 0.1 kWh.

- `heat_quantity_total` *(number, read-only)*

  Heat quantity total.

  Unit: 0.1 kWh.

- `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: 0.1 °C.

- `heat_curve_parallel_shift` *(number, read-only)*

  Heat curve parallel shift.

  Unit: 0.1 °C.

- `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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"alpha_innotec"`

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

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

- `purpose` *(string)*

Alpha-Innotec — Main DHW

Representation of DHW related parameters of Alpha-Innotec device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(integer)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °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: 0.1 °C.

- `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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"alpha_innotec"`

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

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

- `purpose` *(string)*

Alpha-Innotec — Temperature sensor

Representation of Temperature sensor related parameters of Alpha Innotec device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `temperature` *(number, read-only)*

  Measured temperature value.

  Unit: 0.1 °C.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"alpha_innotec"`

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

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

- `purpose` *(string)*

Ampowr Ampi Home 1 Phase — Inverter

Representation of Inverter related parameters of Ampowr Ampi Home 1 Phase device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: 1 mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(number, read-only)*

  Instantaneous current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(number, read-only)*

  Instantaneous current on second group of photovoltaic panels.

  Unit: 1 mA

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: 1 mW

- `energy_produced_total` *(number, read-only)*

  Amount of energy produced by photovoltaic panels over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_produced_month` *(number, read-only)*

  Amount of energy produced by photovoltaic panels this month.

Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_produced_year` *(number, read-only)*

  Amount of energy produced by photovoltaic panels this year.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by photovoltaic panels today.

  Unit: 1 Wh (with accuracy of 10 Wh)

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

  Inverter model.

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

  Current system state: `initialization`, `standby`, `hybrid_grid`, `off_network`, `mains_charging`, `pv_charging`, `mains_bypass`, `fault`, `debug`, `forced_charge`, `power_on_device_separately`, `dsp_burn`, `mcu_burn`, `permanent_error`.

- `radiator_temperature` *(number, read-only)*

  Current radiator temperature.

  Unit: 0.1 °C.

- `inverter_working_mode` *(string)*

  Inverter working mode: `self_consumption`, `peak_shift`, `battery_priority`.

- `pv_input_mode` *(string)*

  Photovoltaic input mode: `indenpendend`, `parallel`, `constant_voltage`.

- `power_control_enabled` *(boolean)*

  Allows to set power control settings.

- `active_power_limit` *(number)*

  Current power limit in percent. Can only be changed when `power_control_enabled` is set to `true`.

  Unit: 1 %

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"ampowr_ampi_home_1_phase"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter. (Set power limit to 100%)

- `turn_off`

  Turns off inverter. (Set power limit to 0%)

- `limit_active_power`

  Sets current active power limit.

  Argument:

  Active power limit.

  Unit: 1 % *(number)*

Ampowr Ampi Home 1 Phase — Battery

Representation of Battery related parameters of Ampowr Ampi Home 1 Phase device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_charged_total` *(number, read-only)*

  Amount of energy charged to the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_discharged_today` *(number, read-only)*

  Amount of energy consumed from the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_discharged_total` *(number, read-only)*

  Amount of energy consumed from the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `radiator_temperature` *(number, read-only)*

  Current radiator temperature.

  Unit: 0.1 °C.

- `depth_of_discharge_enabled` *(boolean)*

  Allows to change depth of discharge.

- `depth_of_discharge_bms_enabled` *(boolean)*

  Allows to change depth of discharge BMS.

- `off_grid_depth_of_discharge` *(number)*

  Sets off grid depth of discharge. Can only be changed when `depth_of_discharge_enabled` and `depth_of_discharge_bms_enabled` are set to `true`.

  Unit: 1 % Range: 5 % – 90 %

- `on_grid_depth_of_discharge` *(number)*

Sets on grid depth of discharge. Can only be changed when
`depth_of_discharge_enabled` and `depth_of_discharge_bms_enabled` are set to
`true`.

Unit: 1 %

Range: 10 % – 90 %

- `voltage` *(number, read-only)*

  Current battery voltage.

  Unit: 1 mV

- `current` *(number, read-only)*

  Current battery current.

  Unit: 1 mA

- `charge_power` *(number, read-only)*

  Current charing (positive number) or discharging (negative number) power.

  Unit: 1 mW

- `soc` *(number, read-only)*

  Current state of charge.

  Unit: 1 %

- `temperature` *(number, read-only)*

  Current battery temperature.

  Unit: 0.1 °C.

- `charge_voltage` *(number, read-only)*

  Current charging voltage.

  Unit: 1 mV

- `charge_current_limit` *(number, read-only)*

  Limit of charging current.

  Unit: 1 mA

- `discharge_current_limit` *(number, read-only)*

  Limit of discharging current.

  Unit: 1 mA

- `maximum_discharge_power` *(number)*

  Limiting discharging power.

  Unit: 1 %

- `maximum_charge_power` *(number)*

  Limiting charging power.

  Unit: 1 %

- `grid_max_soc_charge` *(number)*

  Limiting charging battery from grid.

  Unit: 1 %

  Range: 20 % – 100 %

- `forced_state` *(string)*

  Forced state of the battery. Set by user. Any of: `charge`, `discharge`, `none`.

  > **Note**
  >
  > Can only be changed when `inverter_working_mode` is `peak_shift`.'

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

  Inverter working mode: `self_consumption`, `peak_shift`, `battery_priority`.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"battery"`

- `variant` *(string, read-only)* = `"ampowr_ampi_home_1_phase"`

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

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

- `purpose` *(string)*

Commands

- `charge`

Sends request to force charge the battery.

> **Note**
>
> Can only be changed when `inverter_working_mode` is `peak_shift`. Device has label `battery_charge_support` when available.

- `discharge`

Sends request to force discharge the battery.

> **Note**
>
> Can only be changed when `inverter_working_mode` is `peak_shift`. Device has label `battery_charge_support` when available.

- `stop_forced_state`

Sends request to stop any forced state of the battery.

> **Note**
>
> Can only be changed when `inverter_working_mode` is `peak_shift`. Device has label `battery_charge_support` when available.

- `stop`

Sends request to stop any forced state of the battery.

> **Note**
>
> Can only be changed when `inverter_working_mode` is `peak_shift`. Device has label `battery_charge_support` when available.

Ampowr Ampi Home 1 Phase — Energy meter

setType(energy_meter)

Representation of Energy Meter related parameters of Ampowr Ampi Home 1 Phase device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `total_active_power` *(number, read-only)*

  Total active power on all phases.

  Unit: 1 mW

- `energy_consumed_today` *(number, read-only)*

  Amount of energy consumed from the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_month` *(number, read-only)*

  Amount of energy consumed from the power grid this month.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_year` *(number, read-only)*

  Amount of energy consumed from the power grid this year.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_total` *(number, read-only)*

  Amount of energy consumed from the power grid over lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_month` *(number, read-only)*

  Amount of energy fed to the power grid this month.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_year` *(number, read-only)*

  Amount of energy fed to the power grid this year.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `phase_1.active_power` *(number, read-only)*

  Current power on first phase of power grid.

  Unit: 1 mW

- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid.

  Unit: 1 mV

- `phase_1.current` *(number, read-only)*

  Current current on first phase of power grid.

  Unit: 1 mA

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"battery"`

- `variant` *(string, read-only)* = `"ampowr_ampi_home_1_phase"`

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

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

- `purpose` *(string)*

Ampowr Ampi Home 3 Phase — Inverter

Representation of Inverter related parameters of Ampowr Ampi Home 3 Phase device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: 1 mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: 1 mA

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: 1 mW

- `energy_produced_total` *(number, read-only)*

  Amount of energy produced by photovoltaic panels over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_produced_month` *(number, read-only)*

  Amount of energy produced by photovoltaic panels this month.

Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_produced_year` *(number, read-only)*

Amount of energy produced by photovoltaic panels this year.

Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_produced_today` *(number, read-only)*

Amount of energy produced by photovoltaic panels today.

Unit: 1 Wh (with accuracy of 10 Wh)

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

Inverter model.

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

Current system state: `initialization`, `standby`, `hybrid_grid`, `off_network`, `mains_charging`, `pv_charging`, `mains_bypass`, `fault`, `debug`, `forced_charge`, `power_on_device_separately`, `dsp_burn`, `mcu_burn`, `permanent_error`.

- `radiator_temperature` *(number, read-only)*

Current radiator temperature.

Unit: 0.1 °C.

- `inverter_working_mode` *(string)*

Inverter working mode: `self_consumption`, `peak_shift`, `battery_priority`.

- `pv_input_mode` *(string)*

Photovoltaic input mode: `indenpendend`, `parallel`, `constant_voltage`.

- `power_control_enabled` *(boolean)*

Allows to set power control settings.

- `active_power_limit` *(number)*

Current power limit in percent. Can only be changed when `power_control_enabled` is set to `true`.

Unit: 1 %

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"ampowr_ampi_home_3_phase"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter. (Set power limit to 100%)

- `turn_off`

  Turns off inverter. (Set power limit to 0%)

- `limit_active_power`

  Sets current active power limit.

  Argument:

  Active power limit.

  Unit: 1 % *(number)*

Ampowr Ampi Home 3 Phase — Battery

Representation of Battery related parameters of Ampowr Ampi Home 3 Phase device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_charged_total` *(number, read-only)*

  Amount of energy charged to the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_discharged_today` *(number, read-only)*

  Amount of energy consumed from the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_discharged_total` *(number, read-only)*

  Amount of energy consumed from the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `radiator_temperature` *(number, read-only)*

  Current radiator temperature.

  Unit: 0.1 °C.

- `depth_of_discharge_enabled` *(boolean)*

  Allows to change depth of discharge.

- `depth_of_discharge_bms_enabled` *(boolean)*

  Allows to change depth of discharge BMS.

- `off_grid_depth_of_discharge` *(number)*

  Sets off grid depth of discharge. Can only be changed when `depth_of_discharge_enabled` and `depth_of_discharge_bms_enabled` are set to `true`.

  Unit: 1 %

  Range: 5 % – 90 %

- `on_grid_depth_of_discharge` *(number)*

  Sets on grid depth of discharge. Can only be changed when `depth_of_discharge_enabled` and `depth_of_discharge_bms_enabled` are set to `true`.

  Unit: 1 %

  Range: 10 % – 90 %

- `voltage` *(number, read-only)*

  Current battery voltage.

  Unit: 1 mV

- `current` *(number, read-only)*

  Current battery current.

  Unit: 1 mA

- `charge_power` *(number, read-only)*

  Current charing (positive number) or discharging (negative number) power.

  Unit: 1 mW

- `soc` *(number, read-only)*

  Current state of charge.

  Unit: 1 %

- `temperature` *(number, read-only)*

  Current battery temperature.

  Unit: 0.1 °C.

- `charge_voltage` *(number, read-only)*

  Current charging voltage.

  Unit: 1 mV

- `charge_current_limit` *(number, read-only)*

  Limit of charging current.

  Unit: 1 mA

- `discharge_current_limit` *(number, read-only)*

  Limit of discharging current.

  Unit: 1 mA

- `maximum_discharge_power` *(number)*

  Limiting discharging power.

  Unit: 1 %

- `maximum_charge_power` *(number)*

  Limiting charging power.

  Unit: 1 %

- `grid_max_soc_charge` *(number)*

  Limiting charging battery from grid.

  Unit: 1 %

  Range: 20 % – 100 %

- `forced_state` *(string)*

  Forced state of the battery. Set by user. Any of: `charge`, `discharge`, `none`.

  > **Note**
  >
  > Can only be changed when `inverter_working_mode` is `peak_shift`.

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

  Inverter working mode: `self_consumption`, `peak_shift`, `battery_priority`.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"battery"`

- `variant` *(string, read-only)* = `"ampowr_ampi_home_3_phase"`

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

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

- `purpose` *(string)*

Commands

- `charge`

Sends request to force charge the battery.

> **Note**
>
> Can only be changed when `inverter_working_mode` is `peak_shift`. Device has label `battery_charge_support` when available.

- `discharge`

Sends request to force discharge the battery.

> **Note**
>
> Can only be changed when `inverter_working_mode` is `peak_shift`. Device has label `battery_charge_support` when available.

- `stop_forced_state`

Sends request to stop any forced state of the battery.

> **Note**
>
> Can only be changed when `inverter_working_mode` is `peak_shift`. Device has label `battery_charge_support` when available.

- `stop`

Sends request to stop any forced state of the battery.

> **Note**
>
> Can only be changed when `inverter_working_mode` is `peak_shift`. Device has label `battery_charge_support` when available.

Ampowr Ampi Home 3 Phase — Energy meter

Representation of Energy Meter related parameters of Ampowr Ampi Home 3 Phase device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `total_active_power` *(number, read-only)*

  Total active power on all phases.

  Unit: 1 mW

- `energy_consumed_today` *(number, read-only)*

  Amount of energy consumed from the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_month` *(number, read-only)*

  Amount of energy consumed from the power grid this month.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_year` *(number, read-only)*

  Amount of energy consumed from the power grid this year.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_total` *(number, read-only)*

  Amount of energy consumed from the power grid over lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_month` *(number, read-only)*

  Amount of energy fed to the power grid this month.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_year` *(number, read-only)*

  Amount of energy fed to the power grid this year.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `phase_1.active_power` *(number, read-only)*

  Current power on first phase of power grid.

  Unit: 1 mW

- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid.

  Unit: 1 mV

- `phase_1.current` *(number, read-only)*

  Current current on first phase of power grid.

  Unit: 1 mA

- `phase_2.active_power` *(number, read-only)*

  Current power on second phase of power grid.

  Unit: 1 mW

- `phase_2.voltage` *(number, read-only)*

  Current voltage on second phase of power grid.

  Unit: 1 mV

- `phase_2.current` *(number, read-only)*

  Current current on second phase of power grid.

  Unit: 1 mA

- `phase_3.active_power` *(number, read-only)*

  Current power on third phase of power grid.

  Unit: 1 mW

- `phase_3.voltage` *(number, read-only)*

  Current voltage on third phase of power grid.

  Unit: 1 mV

- `phase_3.current` *(number, read-only)*

  Current current on third phase of power grid.

  Unit: 1 mA

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"ampowr_ampi_home_3_phase"`

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

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

- `purpose` *(string)*

Daikin Altherma — Heat pump

Representation of Heat Pump related parameters of Daikin Altherma device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `work_mode` *(string)*

  Current work mode of the heat pump: `automatic`, `cooling`, `heating`

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: 0.1 °C.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: 0.1 °C.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: 0.1 °C.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: 0.1 °C.

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

  Informs if compressor is running.

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

  Informs if circulation pump is running.

- `heating_target_temperature` *(number)*

  Heating room target temperature.

  Unit: 0.1 °C.

- `cooling_target_temperature` *(number)*

  Cooling room target temperature.

  Unit: 0.1 °C.

- `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.

- `heating_water_target_temperature` *(number)*

  Heating water target temperature.

  Unit °C.

- `cooling_water_target_temperature` *(number)*

  Cooling water target temperature.

  Unit °C.

- `quiet_mode` *(boolean)*

  Indicates if quiet mode operation is enabled.

- `weather_dependent_mode` *(string)*

  Current weather dependent mode: `fixed`, `weather_dependent`, `fixed_scheduled`, `weather_dependent_scheduled`

- `weather_dependent_mode_heating_slope_offset` *(number)*

  Weather dependent mode leaving water temperature heating setpoint offset.

  Unit: 1 °C.

- `weather_dependent_mode_cooling_slope_offset` *(number)*

  Weather dependent mode leaving water temperature cooling setpoint offset.

  Unit: 1 °C.

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

  Indicates if electric booster heater is active.

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

  Indicates if desinfection operation is active.

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

  Indicates if defrost or startup operation in active.

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

  Indicates if hot start operation is active.

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

  State of 3-way valve: `space_heating`, `dhw`

- `operation_mode` *(number, read-only)*

  Current operation mode: `heating`, `cooling`

- `pre_heater_water_temperature` *(number, read-only)*

  Leaving water temperature pre backup heater.

  Unit: 0.1 °C.

- `flow_rate` *(number, read-only)*

  Water flow rate.

  Unit: 0.01 L/min.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"daikin_altherma"`

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

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

- `purpose` *(string)*

Daikin Altherma — Main DHW

Representation of DHW related parameters of Daikin Altherma device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °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: 0.1 °C.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `booster_mode_active` *(boolean)*

  Indicates if booster DHW booster mode is active.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"daikin_altherma"`

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

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

- `purpose` *(string)*

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```lua
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
```

Daikin Altherma — Temperature sensor

Representation of Temperature sensor related parameters of Daikin Altherma device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `temperature` *(number, read-only)*

  Measured temperature value.

  Unit: 0.1 °C.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"daikin_altherma"`

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

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

- `purpose` *(string)*

Eastron SDM630 — Energy meter

Representation of Energy Meter related parameters of Eastron SDM630 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `phase_1.active_power` *(number, read-only)*

  First phase active power.

  Unit: 1 mW

- `phase_1.voltage` *(number, read-only)*

  First phase voltage.

  Unit: 1 mV

- `phase_1.current` *(number, read-only)*

  First phase current.

  Unit: 1 mA

- `phase_1.apparent_power` *(number, read-only)*

  First phase apparent power.

  Unit: 1 mVA

- `phase_1.reactive_power` *(number, read-only)*

  First phase reactive power.

  Unit: 1 mvar

- `phase_1.energy_consumed_total` *(number, read-only)*

  Energy consumed lifetime on first phase.

  Unit: 1 Wh

- `phase_1.energy_consumed_today` *(number, read-only)*

  Energy consumed today on first phase.

  Unit: 1 Wh

- `phase_1.energy_fed_total` *(number, read-only)*

  Energy fed lifetime on first phase.

  Unit: 1 Wh

- `phase_1.energy_fed_today` *(number, read-only)*

  Energy fed today on first phase.

  Unit: 1 Wh

- `phase_1.energy_sum_total` *(number, read-only)*

  Energy sum (consumed + fed) lifetime on first phase.

  Unit: 1 Wh

- `phase_1.energy_sum_today` *(number, read-only)*

  Energy sum (consumed + fed) today on first phase.

  Unit: 1 Wh

- `phase_1.reactive_energy_consumed_total` *(number, read-only)*

  Reactive energy consumed lifetime on first phase.

  Unit: 1 varh

- `phase_1.reactive_energy_consumed_today` *(number, read-only)*

  Reactive energy consumed today on first phase.

  Unit: 1 varh

- `phase_1.reactive_energy_fed_total` *(number, read-only)*

  Reactive energy fed lifetime on first phase.

  Unit: 1 varh

- `phase_1.reactive_energy_fed_today` *(number, read-only)*

  Reactive energy fed today on first phase.

  Unit: 1 varh

- `phase_1.reactive_energy_sum_total` *(number, read-only)*

  Reactive energy sum (consumed + fed) lifetime on first phase.

  Unit: 1 varh

- `phase_1.reactive_energy_sum_today` *(number, read-only)*

  Reactive energy sum (consumed + fed) today on first phase.

  Unit: 1 varh

- `phase_2.active_power` *(number, read-only)*

  Second phase active power.

  Unit: 1 mW

- `phase_2.voltage` *(number, read-only)*

  Second phase voltage.

  Unit: 1 mV

- `phase_2.current` *(number, read-only)*

  Second phase current.

  Unit: 1 mA

- `phase_2.apparent_power` *(number, read-only)*

  Second phase apparent power.

  Unit: 1 mVA

- `phase_2.reactive_power` *(number, read-only)*

  Second phase reactive power.

  Unit: 1 mvar

- `phase_2.energy_consumed_total` *(number, read-only)*

  Energy consumed lifetime on second phase.

  Unit: 1 Wh

- `phase_2.energy_consumed_today` *(number, read-only)*

  Energy consumed today on second phase.

  Unit: 1 Wh

- `phase_2.energy_fed_total` *(number, read-only)*

  Energy fed lifetime on second phase.

  Unit: 1 Wh

- `phase_2.energy_fed_today` *(number, read-only)*

  Energy fed today on second phase.

  Unit: 1 Wh

- `phase_2.energy_sum_total` *(number, read-only)*

  Energy sum (consumed + fed) lifetime on second phase.

  Unit: 1 Wh

- `phase_2.energy_sum_today` *(number, read-only)*

  Energy sum (consumed + fed) today on second phase.

  Unit: 1 Wh

- `phase_2.reactive_energy_consumed_total` *(number, read-only)*

  Reactive energy consumed lifetime on second phase.

  Unit: 1 varh

- `phase_2.reactive_energy_consumed_today` *(number, read-only)*

  Reactive energy consumed today on second phase.

  Unit: 1 varh

- `phase_2.reactive_energy_fed_total` *(number, read-only)*

  Reactive energy fed lifetime on second phase.

  Unit: 1 varh

- `phase_2.reactive_energy_fed_today` *(number, read-only)*

  Reactive energy fed today on second phase.

  Unit: 1 varh

- `phase_2.reactive_energy_sum_total` *(number, read-only)*

  Reactive energy sum (consumed + fed) lifetime on second phase.

  Unit: 1 varh

- `phase_2.reactive_energy_sum_today` *(number, read-only)*

  Reactive energy sum (consumed + fed) today on second phase.

  Unit: 1 varh

- `phase_3.active_power` *(number, read-only)*

  Third phase active power.

  Unit: 1 mW

- `phase_3.voltage` *(number, read-only)*

  Third phase voltage.

  Unit: 1 mV

- `phase_3.current` *(number, read-only)*

  Third phase current.

  Unit: 1 mA

- `phase_3.apparent_power` *(number, read-only)*

  Third phase apparent power.

  Unit: 1 mVA

- `phase_3.reactive_power` *(number, read-only)*

  Third phase reactive power.

  Unit: 1 mvar

- `phase_3.energy_consumed_total` *(number, read-only)*

  Energy consumed lifetime on third phase.

  Unit: 1 Wh

- `phase_3.energy_consumed_today` *(number, read-only)*

  Energy consumed today on third phase.

  Unit: 1 Wh

- `phase_3.energy_fed_total` *(number, read-only)*

  Energy fed lifetime on third phase.

  Unit: 1 Wh

- `phase_3.energy_fed_today` *(number, read-only)*

  Energy fed today on third phase.

  Unit: 1 Wh

- `phase_3.energy_sum_total` *(number, read-only)*

  Energy sum (consumed + fed) lifetime on third phase.

  Unit: 1 Wh

- `phase_3.energy_sum_today` *(number, read-only)*

  Energy sum (consumed + fed) today on third phase.

  Unit: 1 Wh

- `phase_3.reactive_energy_consumed_total` *(number, read-only)*

  Reactive energy consumed lifetime on third phase.

  Unit: 1 varh

- `phase_3.reactive_energy_consumed_today` *(number, read-only)*

  Reactive energy consumed today on third phase.

  Unit: 1 varh

- `phase_3.reactive_energy_fed_total` *(number, read-only)*

  Reactive energy fed lifetime on third phase.

  Unit: 1 varh

- `phase_3.reactive_energy_fed_today` *(number, read-only)*

  Reactive energy fed today on third phase.

  Unit: 1 varh

- `phase_3.reactive_energy_sum_total` *(number, read-only)*

  Reactive energy sum (consumed + fed) lifetime on third phase.

  Unit: 1 varh

- `phase_3.reactive_energy_sum_today` *(number, read-only)*

  Reactive energy sum (consumed + fed) today on third phase.

  Unit: 1 varh

- `total_active_power` *(number, read-only)*

  Total active power on all phases.

  Unit: 1 mW

- `total_apparent_power` *(number, read-only)*

  Total apparent power on all phases.

  Unit: 1 mVA

- `total_reactive_power` *(number, read-only)*

  Total reactive power on all phases.

  Unit: 1 mvar

- `energy_sum_total` *(number, read-only)*

  Energy sum (consumed + fed) lifetime on all phases.

  Unit: 1 Wh

- `energy_sum_today` *(number, read-only)*

  Energy sum (consumed + fed) today on all phases.

  Unit: 1 Wh

- `reactive_energy_sum_total` *(number, read-only)*

  Reactive energy sum (consumed + fed) lifetime on all phases.

  Unit: 1 varh

- `reactive_energy_sum_today` *(number, read-only)*

  Reactive energy sum (consumed + fed) today on all phases.

  Unit: 1 varh

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"eastron_sdm630"`

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

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

- `purpose` *(string)*

EcoAir — Heat pump

Representation of Heat Pump related parameters of EcoAir device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 0.1 °C.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: 0.1 °C.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: 0.1 °C.

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: 0.1 bar

- `hot_gas_temperature` *(number, read-only)*

  Hot Gas temperature.

  Unit: 0.1 °C.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: 0.1 °C.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: 0.1 °C.

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

  Unit: 1 h

- `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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"eco_air"`

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

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

- `purpose` *(string)*

Commands

- `reset_alarms`

  Sends request to heat pump device to reset alarms.

EcoAir — Main DHW

Representation of DHW related parameters of EcoAir device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °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: 0.1 °C

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"eco_air"`

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

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

- `purpose` *(string)*

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```lua
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
```

EcoGeo — Heat pump

Representation of Heat Pump related parameters of EcoGeo device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 0.1 °C

- `brine_out_temperature` *(number, read-only)*

  Brine out temperature.

  Unit: 0.1 °C

- `brine_in_temperature` *(number, read-only)*

  Brine in temperature.

  Unit: 0.1 °C

- `brine_pressure` *(number, read-only)*

  Brine pressure.

  Unit: 0.1 bar

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: 0.1 °C

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: 0.1 °C

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: 0.1 bar

- `hot_gas_temperature` *(number, read-only)*

  Hot Gas temperature.

  Unit: 0.1 °C

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: 0.1 °C

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: 0.1 °C

- `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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"eco_geo"`

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

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

- `purpose` *(string)*

Commands

- `reset_alarms`

  Sends request to heat pump device to reset alarms.

EcoGeo — Main DHW

Representation of DHW related parameters of EcoGeo device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °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: 0.1 °C

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*
- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"eco_geo"`

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

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

- `purpose` *(string)*

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```lua
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
```

EcoGeo HighPower — Heat pump

Representation of Heat Pump related parameters of EcoGeo HighPower device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 0.1 °C

- `brine_out_temperature` *(number, read-only)*

  Brine out temperature.

  Unit: 0.1 °C

- `brine_in_temperature` *(number, read-only)*

  Brine in temperature.

  Unit: 0.1 °C

- `brine_pressure` *(number, read-only)*

  Brine pressure.

  Unit: 0.1 bar

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: 0.1 °C

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: 0.1 °C

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: Bar with one decimal number, multiplied by 10.

- `electric_heater_emergency` *(boolean)*

  Indicates electric heater emergency state.

- `electric_heater_active` *(boolean)*

  Indicates electric heater activation state.

- `fixed_heating_target_temperature` *(number)*

  Fixed heating target temperature.

  Unit: 0.1 °C

- `fixed_cooling_target_temperature` *(number)*

  Fixed cooling target temperature.

  Unit: 0.1 °C

- `target_temperature_mode` *(string)*

  Target temperature mode: `fixed`, `heat_curve`

- `heat_curve_base_point` *(number, read-only)*

  Heat curve base point.

  Unit: 0.1 °C

- `heat_curve_end_point` *(number, read-only)*

  Heat curve end point.

  Unit: 0.1 °C

- `heat_curve_end_point_outside` *(number, read-only)*

  Heat curve end point outside.

  Unit: 0.1 °C

- `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.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"eco_geo_high_power"`

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

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

- `purpose` *(string)*

EcoGeo HighPower — Main DHW

Representation of DHW related parameters of EcoGeo High Power device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °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: 0.1 °C

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"eco_geo_high_power"`

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

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

- `purpose` *(string)*

Galmet Prima — Heat pump

Representation of Heat Pump related parameters of Galmet Prima device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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_minimum` *(number, read-only)*

  Minimum value of `fixed_target_temperature` parameter.

  Unit: 1 °C.

- `fixed_target_temperature_maximum` *(number, read-only)*

  Maximum value of `fixed_target_temperature` parameter.

  Unit: 1 °C.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: 0.1 °C

- `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: 0.1 °C

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: 0.1 °C

- `water_inlet_temperature` *(number, read-only)*

  Water inlet temperature.

  Unit: 0.1 °C

- `water_outlet_temperature` *(number, read-only)*

  Water outlet temperature.

  Unit: 0.1 °C

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `electric_heater_active` *(boolean)*

  Indicates electric heater desired state.

- `zone_1.heat_demand` *(boolean)*

  Informs device that heat is demanded or not for zone 1.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `zone_1.cool_demand` *(boolean)*

  Informs device that cool is demanded or not for zone 1.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `zone_1.fixed_target_temperature` *(number)*

  Set temperature for heating or cooling in fixed temperature mode for zone 1.

  Unit: 1 °C.

- `zone_1.heat_curve` *(number)*

  Current set heat curve ID (1-9 for) for zone 1.

- `zone_1.heat_curve_target_temperature` *(number, read-only)*

  Current target temperature set by heat curve for zone 1.

  Unit: 1 °C.

- `zone_1.heat_curve_enabled` *(boolean)*

  Indicator if heat curve mode is enabled for zone 1.

- `zone_2.heat_demand` *(boolean)*

  Informs device that heat is demanded or not for zone 2..

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `zone_2.cool_demand` *(boolean)*

  Informs device that cool is demanded or not for zone 2..

> **Note**
> Cannot be modified when device is associated with Heat Pump Manager.

- `zone_2.fixed_target_temperature` *(number)*

  Set temperature for heating or cooling in fixed temperature mode for zone 2..

  Unit: 1 °C.

- `zone_2.heat_curve` *(number)*

  Current set heat curve ID (1-9) for zone 2.

- `zone_2.heat_curve_target_temperature` *(number, read-only)*

  Current target temperature set by heat curve for zone 2..

  Unit: 1 °C.

- `zone_2.heat_curve_enabled` *(boolean)*

  Indicator if heat curve mode is enabled for zone 2.

- `work_frequency` *(number, read-only)*

  Compressor operating frequency.

  Unit: 1 Hz

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

  Actual work mode of the heat pump outdoor unit. *off, cooling, heating*.

- `fan_speed` *(number, read-only)*

  Fan speed.

  Unit: revolutions per minute.

- `t1_water_outlet_temperature` *(number, read-only)*

  T1 temperature. Total water outlet temperature.

  Unit: 0.1 °C

- `t2_temperature` *(number, read-only)*

  T2 temperature. Temperature on the liquid coolant side.

  Unit: 0.1 °C

- `device_power` *(number, read-only)*

  Heat pump max power.

  Unit: 1 Wh

- `energy_used_total` *(number, read-only)*

  Total energy used by heat pump.

  Unit: 1 Wh

- `energy_generated_total` *(number, read-only)*

  Total energy generated by heat pump.

  Unit: 1 Wh

- `outdoor_unit_capacity` *(number, read-only)*

  Outdoor unit power capacity.

  Unit: 1 W

- `water_flow` *(number, read-only)*

  Water flow in instalation.

  Unit: 1 L/h

- `buffer_up_temperature` *(number, read-only)*

  Measured temperature in upper part of buffer.

  Unit: 0.1 °C

- `buffer_down_temperature` *(number, read-only)*

  Measured temperature in lower part of buffer.

  Unit: 0.1 °C

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

  Internal water circulation pump (P_i) state.

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

  External water circulation pump (P_o) state.

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

  DHW water pump (P_d) state.

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

  Water pump of the solar collector system (P_s) state.

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

  Mixed water pump (P_c) state.

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

  Actual electric heater state.

- `sv_1_state` *(boolean, read-only)*

  Three-way solenoid valve (SV1) state.

- `sv_2_state` *(boolean, read-only)*

  Two-way solenoid valve (SV2) state.

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

  Indicator of current defrost state.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"galmet_prima"`

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

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

- `purpose` *(string)*

Commands

- `set_work_mode`

  Change work mode of heat pump.

  > **Note**
  >
  > Cannot be executed when heat pump is associated with Heat pump manager.

  Argument:

  Work mode *(string)*. One of: `automatic`, `cooling`, `heating`.

- `set_fixed_target_temperature`

  Set fixed target temperature for requested zone.

  Arguments:

  packed arguments *(table)*:

  - `zone` - zone number *(number)*:
    - minumum: 1
    - maximum: 2
  - `value` - target temperature *(number)*:
    - minimum: `fixed_target_temperature_minimum`
    - maximum: `fixed_target_temperature_maximum`
    - unit: °C

- `set_heat_demand`

  Set heat demand for requested zone.

  > **Note**
  >
  > Cannot be executed when heat pump is associated with Heat pump manager.

  Arguments:

  packed arguments *(table)*:

  - `zone` - zone number *(number)*:
    - minumum: 1
    - maximum: 2
  - `value` - demand *(boolean)*:

- `set_cool_demand`

  Set cool demand for requested zone.

  > **Note**
  >
  > Cannot be executed when heat pump is associated with Heat pump manager.

  Arguments:

  packed arguments *(table)*:

  - `zone` - zone number *(number)*:
    - minumum: 1
    - maximum: 2
  - `value` - demand *(boolean)*:

- `set_heat_curve`

  Set active heat curve for requested zone.

  > **Note**
  >
  > this only changes active heat curve number. To enable heat curve mode user has to use `set_heat_curve_enabled` command.

  Arguments:

  packed arguments *(table)*:

  - `zone` - zone number *(number)*:
    - minumum: 1
    - maximum: 2
  - `value` - heat curve number *(number)*:
    - minumum: 1
    - maximum: 9

- `set_heat_curve_enabled`

  Enable or disable heat curve mode for requested zone.

> **Note**
>
> Cannot be executed when heat pump is associated with Heat pump manager.

Arguments:

packed arguments *(table)*:

- `zone` - zone number *(number)*:
  - minumum: 1
  - maximum: 2
- `value` - enabled *(boolean)*:

- `set_electric_heater_active`

  Activate or disactivate electric heater.

  Argument:

  Active *(boolean)*

Examples

Set cooling work mode

```
modbus[1]:call("set_work_mode", "cooling")
```

Set heating work mode

```
modbus[1]:call("set_work_mode", "heating")
```

Set fixed target temperature for both zones

```
modbus[1]:call("set_fixed_target_temperature", { zone=1, value=26 })
modbus[1]:call("set_fixed_target_temperature", { zone=2, value=28 })
```

Turn on heat demand and turn off cool demand for both zones

```
modbus[1]:call("set_heat_demand", { zone=1, value=true })
modbus[1]:call("set_heat_demand", { zone=2, value=true })

modbus[1]:call("set_cool_demand", { zone=1, value=false })
modbus[1]:call("set_cool_demand", { zone=2, value=false })
```

Enable and set heat curve id for both zones

```
modbus[1]:call("set_heat_curve_enabled", { zone=1, value=true })
modbus[1]:call("set_heat_curve", { zone=1, value=2 })

modbus[1]:call("set_heat_curve_enabled", { zone=2, value=true })
modbus[1]:call("set_heat_curve", { zone=2, value=3 })
```

Activate electric heater

```
modbus[1]:call("set_electric_heater_active", true)
```

Galmet Prima — Main DHW

Representation of DHW related parameters of Galmet Prima device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 1 °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: 0.1 °C.

- `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 desired state.

- `circulation_pump_enabled` *(boolean)*

  Indicates if curculation pump work is enabled.

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

  Actual electric heater state.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"galmet_prima"`

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

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

- `purpose` *(string)*

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```lua
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
```

Galmet Prima — Temperature sensor

Representation of Temperature sensor related parameters of Galmet Prima device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `temperature` *(number, read-only)*

  Measured temperature value.

  Unit: 0.1 °C

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"galmet_prima"`

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

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

- `purpose` *(string)*

GoodWe MT / SMT — Inverter

Representation of Inverter related parameters of GoodWe MT/SMT device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 mW

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: 1 mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: 1 mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

Unit: 1 mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: 1 mA

- `pv_3.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_3.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_3.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: 1 mA

- `pv_4.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_4.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_4.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: 1 mA

- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid.

  Unit: 1 mV

- `phase_1.current` *(number, read-only)*

  Current current on first phase of power grid.

  Unit: 1 mA

- `phase_2.voltage` *(number, read-only)*

  Current voltage on second phase of power grid.

  Unit: 1 mV

- `phase_2.current` *(number, read-only)*

  Current current on second phase of power grid.

  Unit: 1 mA

- `phase_3.voltage` *(number, read-only)*

  Current voltage on third phase of power grid.

  Unit: 1 mV

- `phase_3.current` *(number, read-only)*

  Current current on third phase of power grid.

  Unit: 1 mA

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"goodwe_mt_smt"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter. (Set power limit to 100 %)

- `turn_off`

  Turns off inverter. (Set power limit to 0 %)

- `limit_active_power`

  Sets current active power limit.

  Argument:

  Active power limit *(number)*.

  Unit: 1 %

GoodWe SDT / MS / DNS / XS — Inverter

Representation of Inverter related parameters of GoodWe SDT/MS/DNS/XS device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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: 1 mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: 1 mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: 1 mA

- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid.

  Unit: 1 mV

- `phase_1.current` *(number, read-only)*

  Instantaneous current current on first phase of power grid.

  Unit: 1 mA

- `phase_2.voltage` *(number, read-only)*

  Current voltage on second phase of power grid.

  Unit: 1 mV

- `phase_2.current` *(number, read-only)*

  Instantaneous current on second phase of power grid.

  Unit: 1 mA

- `phase_3.voltage` *(number, read-only)*

  Current voltage on third phase of power grid.

  Unit: 1 mV

- `phase_3.current` *(number, read-only)*

  Instantaneous current on third phase of power grid.

  Unit: 1 mA

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"goodwe_sdt_ms_dns_xs"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter. (Set power limit to 100 %)

- `turn_off`

  Turns off inverter. (Set power limit to 0 %)

- `limit_active_power`

  Sets current active power limit.

  Argument:

  Active power limit *(number)*.

  Unit: 1 %

Heatcomp — Heat pump

Representation of Heat Pump related parameters of Heatcomp device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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: 0.1 °C

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: 0.1 °C

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: 0.1 °C

- `hot_gas_temperature` *(number, read-only)*

  Hot Gas temperature.

  Unit: 0.1 °C

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: 0.1 °C

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: 0.1 °C

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `compressor_percentage` *(number, read-only)*

  Compresor percentage.

  > **Note**
  >
  > Parameter deprecated, replaced with `max_compressor_frequency`.

  Unit: 1 %/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: 0.1 °C

- `fixed_cooling_target_temperature` *(number)*

  Fixed cooling target temperature.

  Unit: 0.1 °C

- `target_temperature_mode` *(string)*

  Target temperature mode: `fixed`, `heat_curve`

- `heat_curve_slope` *(number)*

  Heat curve slope.

  Unit: 0.1 °C

- `heat_curve_offset` *(number)*

  Heat curve offset.

  Unit: 0.1 °C

- `min_compressor_frequency` *(number)*

  Minimum compressor operating frequency.

  Unit: 1 Hz

- `max_compressor_frequency` *(number)*

  Maximum compressor operating frequency.

  Unit: 1 Hz

- `buffer_temperature` *(number, read-only)*

  Current buffer temperature.

  Unit: 0.1 °C

- `installation_base_temperature` *(string)*

  Selected installation source sensor for heat pump work. One of:
  `"supply_temperature"`, `"room_temperature"`, `"buffer_temperature"`,
  `"return_temperature"`, `"unknown"`.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"heatcomp"`

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

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

- `purpose` *(string)*

Heatcomp — Main DHW

Representation of DHW related parameters of Heatcomp device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °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: 0.1 °C

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"heatcomp"`

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

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

- `purpose` *(string)*

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```lua
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
```

Heatcomp HC-EV01 — Car charger

Representation of Car Charger related parameters of Heatcomp HC-EV01 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

  Current car charger state: `initializing`, `waiting`, `waiting_for_delayed_charging`, `connection_status`, `charging`, `failure`, `end_charging`, `unknown`

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

  Current car charger error: `leakage_error`, `communication_abnormality`, `too_high_temperature`, `under_voltage_l1`, `under_voltage_l2`, `under_voltage_l3`, `over_voltage_l1`, `over_voltage_l2`, `over_voltage_l3`, `emergency_stop`, `none`

- `voltage` *(number, read-only)*

  Charging voltage.

  Unit: 1 mV.

- `current` *(number, read-only)*

  Charging current.

  Unit: 1 mA.

- `charge_power` *(number, read-only)*

  Current charing (positive number) power. Unit: 1 mW

- `energy_charged_total` *(number, read-only)*

  Amount of energy charged to the car over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the car today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `temperature` *(number, read-only)*

  Car charger temperature.

  Unit: 0.1 °C

- `charge_time` *(number, read-only)*

  Car charger charging time.

  Unit: 1 min

- `current_limit` *(number, read-only)*

  Limit of charging current.

  Unit: 1 mA

  Range: 0 mA – 32 000 mA

- `postpone_time` *(number, read-only)*

  Time to postpone charging.

  Unit: 1 h

  Range: 0 h – 15 h.

- `charge_time_left` *(number, read-only)*

  Time to stop charging in time.

  Unit: 1 h

  Range: 0 h – 15 h.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"heatcomp"`

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

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

- `purpose` *(string)*

Commands

- `start_charge`

  Start charging.

- `stop_charge`

  Stop charging.

- `set_current_limit`

  Calls car charger to set limit of charging current.

  Argument:

  Limit in 1 mA

  Range: 0 mA – 32 000 mA.

- `set_postpone_time`

  Calls car charger to postone charging.

  Argument:

  Hours to postpone charging

  Range: 0 h – 15 h.

- `set_charge_time_left`

  Calls car charger to set limit of charging time.

  Argument:

  Hours to stop charging.

  Range: 0 h – 15 h.

Heatcomp inverter — Inverter

Representation of Inverter related parameters of HeatcompInverter device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

  Inverter current run mode. Available values are: `standby`, `initialization`, `ac_power_operation`, `inverter_operation`

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: 1 mW

- `energy_produced_total` *(number, read-only)*

  Total amount of energy produced by PV over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: 1 mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_total` *(number, read-only)*

  Amount of energy consumed from the power grid over a lifetime.

  Unit: 1 Wh (with accuracy of 10 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: 1 mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: 1 mA

- `grid_total_active_power` *(number, read-only)*

  Current total power on the grid.

  Unit: 1 mW

- `phase_1.active_power` *(number, read-only)*

  Current power on first phase of power grid.

  Unit: 1 mW

- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid.

  Unit: 1 mV

- `phase_1.current` *(number, read-only)*

  Current current on first phase of power grid.

  Unit: 1 mA

- `phase_2.active_power` *(number, read-only)*

  Current power on second phase of power grid.

Unit: 1 mW

- `phase_2.voltage` *(number, read-only)*

  Current voltage on second phase of power grid.

  Unit: 1 mV

- `phase_2.current` *(number, read-only)*

  Current current on second phase of power grid.

  Unit: 1 mA

- `phase_3.active_power` *(number, read-only)*

  Current power on third phase of power grid.

  Unit: 1 mW

- `phase_3.voltage` *(number, read-only)*

  Current voltage on third phase of power grid.

  Unit: 1 mV

- `phase_3.current` *(number, read-only)*

  Current current on third phase of power grid.

  Unit: 1 mA

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"heatcomp_inverter"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns device on.

- `turn_off`

  Turns device off.

Heatcomp inverter — Battery

Representation of Battery related parameters of HeatcompInverter device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `soc` *(number, read-only)*

  Current state of charge.

  Unit: 1 %

- `charge_current` *(number, read-only)*

  Charging current.

  Unit: 1 mA

- `charge_current_limit` *(number, read-only)*

  Charging current limit.

  Range: 0 mA – 150 000 mA.

  Unit: 1 mA

- `voltage` *(number, read-only)*

  Battery voltage.

  Unit: 1 mV

- `charge_power` *(number, read-only)*

  Current charging (positive number) or discharging (negative number) power.

  Unit: 1 mW

- `energy_charged_total` *(number, read-only)*

  Amount of energy charged to the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_discharged_total` *(number, read-only)*

  Amount of energy consumed from the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_discharged_today` *(number, read-only)*

  Amount of energy consumed from the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `scheduled_charge.active` *(boolean, read-only)*

  Indicates if scheduled charge is activated on the device.

- `scheduled_charge.enabled` *(boolean)*

  Indicates if scheduled battery charge is enabled. Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_charge.start_time_1` *(integer)*

  First period charge start time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_charge.end_time_1` *(integer)*

  First period charge end time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_charge.start_time_2` *(integer)*

  Second period charge start time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_charge.end_time_2` *(integer)*

  Second period charge end time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_charge.start_time_3` *(integer)*

  Third period charge start time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_charge.end_time_3` *(integer)*

  Third period charge end time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_discharge.active` *(boolean, read-only)*

  Indicates if scheduled discharge is activated on the device.

- `scheduled_discharge.enabled` *(boolean)*

  Indicates if scheduled battery discharge is enabled. Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_discharge.start_time_1` *(integer)*

  First period discharge start time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_discharge.end_time_1` *(integer)*

  First period discharge end time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_discharge.start_time_2` *(integer)*

  Second perioddis charge start time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_discharge.end_time_2` *(integer)*

  Second perioddis charge end time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_discharge.start_time_3` *(integer)*

  Third period discharge start time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `scheduled_discharge.end_time_3` *(integer)*

  Third period discharge end time. In minutes of the day (maximum 1439). Can only be changed when `charge_work_mode` is set to `schedule`.

- `max_grid_charge_power` *(integer)*

  Maximum charge power from grid. Unit: W

- `grid_active_power_set` *(integer)*

  Maximum power discharged to the grid. Unit: W

- `charge_work_mode` *(string)*

  Defines if battery charges or discharges according to schedule or manual control. Any of: *schedule*, *manual*.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"battery"`

- `variant` *(string, read-only)* = `"heatcomp_inverter"`

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

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

- `purpose` *(string)*

Commands

- `set_charge_current_limit`

  Calls battery to set charging current limit.

  Argument: *(number)*

  Sets charging current limit (Range: 0 mA – 150 000 mA).

  Unit: 1 mA

- `charge`

  Calls battery to charge during given period of time.

  > **Note**
  >
  > Can only be changed when `charge_work_mode` is `manual`. Device has label `battery_charge_support` when available.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 0
    - maximum: 65535000
    - changes `scheduled_charge.active` and `max_grid_charge_power` parameters

- `discharge`

  Calls battery to discharge during given period of time.

  > **Note**
  >
  > Can only be changed when `charge_work_mode` is `manual`. Device has label `battery_charge_support` when available.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 0
    - maximum: 65000000
    - changes `scheduled_discharge.active` and `grid_active_power_set` parameters

- `stop`

  Stops the forced charging/discharging of the device.

  > **Note**
  >
  > Can only be changed when `charge_work_mode` is `manual`. Device has label
  > `battery_charge_support` when available.

Examples

Turn on battery charging with 1kW at 1:00PM

```lua
if dateTime:changed() then
  if dateTime:getHours() == 13 and dateTime:getMinutes() == 0 then
    modbus[2]:call("charge", { 1000000 })
  end
end
```

HeatEco — Heat pump

Representation of Heat Pump related parameters of HeatEco device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 0.1 °C.

- `fixed_cooling_target_temperature` *(integer)*

  Target cooling temperature.

  Unit: 0.1 °C.

- `bottom_hysteresis` *(integer)*

  Damper factor, which will protect from continuous on/off switching when current temperature is below target value.

  Unit: 0.1 °C.

- `top_hysteresis` *(integer)*

  Damper factor, which will protect from continuous on/off switching when current temperature is above target value.

  Unit: 0.1 °C.

- `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: 0.1 °C.

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

  Water outlet temperature.

  Unit: 0.1 °C.

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

  Outdoor temperature. Unit: 0.1 °C.

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

  Discharge Gas temperature.

  Unit: 0.1 °C.

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

  Suction Gas temperature.

  Unit: 0.1 °C.

- `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: 0.1 °C.

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

  Evaporation temperature.

  Unit: 0.1 °C.

- `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: 0.1 %

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

  Current pump output value.

  Unit: 0.1 %

- `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: 1 %.

- `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.

HeatEco — Main DHW

Representation of DHW related parameters of HeatEco device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(integer)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 1 °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: 0.1 °C.

- `bottom_hysteresis` *(integer)*

  Damper factor, which will protect from continuous on/off switching when current temperature is below target value.

  Unit: 0.1 °C.

- `top_hysteresis` *(integer)*

  Damper factor, which will protect from continuous on/off switching when current temperature is above target value.

  Unit: 0.1 °C.

- `dhw_demand` *(boolean)*

  Domestic Hot Water demand.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"heat_eco"`

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

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

- `purpose` *(string)*

Huawei SUN2000 — Battery

Representation of Battery related parameters of Huawei SUN2000 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 Wh (with accuracy of 100 Wh)

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

  Amount of energy charged to the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh)

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

  Amount of energy consumed from the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh)

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

  Amount of energy consumed from the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh)

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

  Current charing (positive number) or discharging (negative number) power.

  Unit: 1 mW

- `maximum_charging_power` *(integer)*

  Maximum charging power.

  Unit: 1 mW.

- `maximum_discharging_power` *(integer)*

  Maximum charging power.

  Unit: 1 mW.

- `charging_cutoff_capacity` *(integer)*

  Charging cutoff capacity.

Unit: 0.1 %

- `discharge_cutoff_capacity` *(integer)*

  Discharge cutoff capacity.

  Unit: 0.1 %

Required label: `"battery_charge_support"`

- `forced_charging_duration` *(read-only, optional)*

  Forced charge/discharge duration.

  Unit: 1 min.
- `forced_charging_power` *(read-only, optional)*

  Forced charge (positive) or discharge (negative) power.

  Unit: 1 W.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"battery"`

- `variant` *(string, read-only)* = `"huawei_sun_2000"`

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

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

- `purpose` *(string)*

Commands

- `charge`

  Calls battery to charge during given period of time.

  > **Note**
  >
  > Available when device has label `battery_charge_support`.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 0
    - maximum: from parameter `maximum_charging_power`
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 86400
    - parameter optional, maximum - 86400 is used when not provided
    - device accepts only full minutes so value less than 60 will not affect the battery

- `discharge`

  Calls battery to discharge during given period of time.

  > **Note**
  >
  > Available when device has label `battery_charge_support`.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 0
    - maximum: from parameter `maximum_discharging_power`
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 86400
    - parameter optional, maximum - 86400 is used when not provided
    - device accepts only full minutes so value less than 60 will not affect the battery

- `stop`

  Stops the forced charging/discharging of the device.

  > **Note**
  >
  > Available when device has label `battery_charge_support`.

Examples

Turn on battery charging with 1kW for 1 hour at 1:00PM

```lua
if dateTime:changed() then
  if dateTime:getHours() == 13 and dateTime:getMinutes() == 0 then
    modbus[2]:call("charge", { 1000000, 3600 })
  end
end
```

Huawei SUN2000 — Inverter

Representation of Inverter related parameters of Huawei SUN2000 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 mW

- `pv_1.voltage` *(integer, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(integer, read-only)*

  Instantaneous current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(integer, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(integer, read-only)*

  Current voltage on second group of photovoltaic panels. Unit: 1 mV

- `pv_2.current` *(integer, read-only)*

  Instantaneous current on second group of photovoltaic panels.

  Unit: 1 mA

- `pv_3.active_power` *(integer, read-only)*

  Current power produced by third group of photovoltaic panels.

  Unit: 1 mW

- `pv_3.voltage` *(integer, read-only)*

  Current voltage on third group of photovoltaic panels.

  Unit: 1 mV

- `pv_3.current` *(integer, read-only)*

  Instantaneous current on third group of photovoltaic panels.

  Unit: 1 mA

- `pv_4.active_power` *(integer, read-only)*

  Current power produced by fourth group of photovoltaic panels.

  Unit: 1 mW

- `pv_4.voltage` *(integer, read-only)*

  Current voltage on fourth group of photovoltaic panels.

  Unit: 1 mV

- `pv_4.current` *(integer, read-only)*

  Instantaneous current on fourth group of photovoltaic panels.

  Unit: 1 mA

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

  Current total power produced by all photovoltaic panels.

  Unit: 1 mW

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

  Total amount of energy produced by PV over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

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

  Amount of energy produced by PV today.

  Unit: 1 Wh (with accuracy of 100 Wh).

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

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: 1 mW

- `phase_1.voltage` *(integer, read-only)*

  First phase voltage.

  Unit: 1 mV

- `phase_1.current` *(integer, read-only)*

  First phase current.

  Unit: 1 mA

- `phase_2.voltage` *(integer, read-only)*

  Second phase voltage.

  Unit: 1 mV

- `phase_2.current` *(integer, read-only)*

  Second phase current.

  Unit: 1 mA

- `phase_3.voltage` *(integer, read-only)*

  Third phase voltage.

  Unit: 1 mV

- `phase_3.current` *(integer, read-only)*

  Third phase current.

  Unit: 1 mA

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

  Numeric model ID read from the device.

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

  Descriptive model of the connected inverter.

- `active_power_limit` *(integer)*

  Inverter output power limit.

  > **Note**
  >
  > Available when function is supported - check if `power_control_support` label is provided.

- `active_power_control_mode` *(string)*

  Inverter output power control mode: `unknown`, `unlimited`, `di_active_scheduling`, `zero_power_grid_connection`, `power_limited_grid_connection_kw`, `power_limited_grid_connection_percent`.

  > **Note**
  >
  > Available when function is supported - check if `power_control_mode_support` label is provided.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"huawei_sun_2000"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter.

  > **Note**
  >
  > Available when function is supported - check if `power_control_support` label is provided.

- `turn_off`

  Turns off inverter.

  > **Note**
  >
  > Available when function is supported - check if `power_control_support` label is provided.

- `limit_active_power`

  Sets current active power limit.

  > **Note**
  >
  > Available when function is supported - check if `power_control_support` label is provided.

  Argument:

  Active power limit. Unit: 1 % *(integer)*

- `set_power_control_mode`

  Sends request to change power control mode of inverter.

  > **Note**
  >
  > Available when function is supported - check of `power_control_mode_support`
  > label is provided.

  Argument:

  Power control mode *(string)*: `unlimited`, `di_active_scheduling`,
  `zero_power_grid_connection`, `power_limited_grid_connection_kw`,
  `power_limited_grid_connection_percent`

- `use_power_limit_register`

  Set the Modbus register to use by inverter implementation as power limit. Register
  should store inverter output power limit in 0.1%, e.g. 320 is 32 %.

  > **Warning**
  >
  > Use only with confirmed registers by manufacturer. Sinum implementation already
  > uses a set of registers for power control and this command should be used only in
  > emergency situations. Register writes can be send a lot, depending on different
  > automations or custom devices.

  Argument:

  Register number. *(integer)*

Huawei SUN2000 — Energy meter

Representation of Energy meter related parameters of Huawei SUN2000 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

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

  Current power fed to (negative number) or consumed from (positive number) the power grid.

  Unit: 1 mW

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"huawei_sun_2000"`

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

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

- `purpose` *(string)*

Itho — Heat pump

Representation of Heat Pump related parameters of Itho device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(string)*

  State of the heat pump: `on`, `off`

- `fixed_heating_target_temperature` *(number)*

  Set temperature for heating in fixed temperature mode.

  Unit: 1 °C.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: 0.1 °C.

- `temperature_indoor` *(number, read-only)*

  Indoor temperature.

  Unit: 0.1 °C.

- `target_temperature_indoor` *(number, read-only)*

  Set indoor temperature.

  Unit: 0.1 °C.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: 0.1 °C.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: 0.1 °C.

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: 0.1 bar.

- `hot_gas_temperature` *(number, read-only)*

  Hot gas temperature.

  Unit: 0.1 °C.

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: 0.1 °C.

- `evaporation_temperature` *(number, read-only)*

  Evaporation temperature.

  Unit: 0.1 °C.

- `brine_out_temperature` *(number, read-only)*

  Brine out temperature.

  Unit: 0.1 °C.

- `brine_in_temperature` *(number, read-only)*

  Brine in temperature.

  Unit: 0.1 °C.

- `energy_used_for_hot_water` *(number, read-only)*

  Energy used for hot water.

  Unit: 1 kWh.

- `energy_used_for_heating` *(number, read-only)*

  Energy used for heating.

  Unit: 1 kWh.

- `energy_used_for_cooling` *(number, read-only)*

  Energy used for cooling.

  Unit: 1 kWh.

- `energy_used_in_stand_by` *(number, read-only)*

  Energy used in stand-by.

  Unit: 1 kWh.

- `energy_used_total` *(number, read-only)*

  Energy used total.

  Unit: 1 kWh.

- `source_supply_energy` *(number, read-only)*

  Energy in source supply.

  Unit: 10 kWh

- `source_return_energy` *(number, read-only)*

  Energy in source return.

  Unit: 10 kWh

- `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: 0.1 °C.

- `heat_curve_base_point` *(number, read-only)*

  Heat curve base point.

  Unit: 0.1 °C.

- `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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"itho"`

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

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

- `purpose` *(string)*

Itho — Main DHW

Representation of DHW related parameters of Itho device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °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: 0.1 °C.

- `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 continuous on/off switching when current temperature is near target value.

  Unit: 0.1 °C.

  > **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: 0.1 °C.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`
- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"itho"`

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

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

- `purpose` *(string)*

Itho — Temperature sensor

Representation of Temperature sensor related parameters of Itho device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `temperature` *(number, read-only)*

  Measured temperature value.

  Unit: 0.1 °C.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"itho"`

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

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

- `purpose` *(string)*

Kaisai KHC — Heat pump

Representation of Heat Pump related parameters of Kaisai KHC device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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_minimum` *(number, read-only)*

  Minimum value of `fixed_target_temperature` parameter.

  Unit: 1 °C.

- `fixed_target_temperature_maximum` *(number, read-only)*

  Maximum value of `fixed_target_temperature` parameter.

  Unit: 1 °C.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: 0.1 °C

- `heating_system_pressure` *(number, read-only)*

  Heating System pressure.

  Unit: 0.1 bar.

- `hot_gas_temperature` *(number, read-only)*

  Hot gas temperature.

  Unit: 0.1 °C

- `condensation_temperature` *(number, read-only)*

  Condensation temperature.

  Unit: 0.1 °C

- `water_inlet_temperature` *(number, read-only)*

  Water inlet temperature.

  Unit: 0.1 °C

- `water_outlet_temperature` *(number, read-only)*

  Water outlet temperature.

  Unit: 0.1 °C

- `running_hours` *(number, read-only)*

  Hours heat pump is working.

- `electric_heater_active` *(boolean)*

  Indicates electric heater desired state.

- `zone_1.heat_demand` *(boolean)*

  Informs device that heat is demanded or not for zone 1.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `zone_1.cool_demand` *(boolean)*

  Informs device that cool is demanded or not for zone 1.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `zone_1.fixed_target_temperature` *(number)*

  Set temperature for heating or cooling in fixed temperature mode for zone 1.

  Unit: 1 °C.

- `zone_1.heat_curve` *(number)*

  Current set heat curve ID (1-9 for) for zone 1.

- `zone_1.heat_curve_target_temperature` *(number, read-only)*

  Current target temperature set by heat curve for zone 1.

  Unit: 1 °C.

- `zone_1.heat_curve_enabled` *(boolean)*

  Indicator if heat curve mode is enabled for zone 1.

- `zone_2.heat_demand` *(boolean)*

  Informs device that heat is demanded or not for zone 2..

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `zone_2.cool_demand` *(boolean)*

  Informs device that cool is demanded or not for zone 2..

> **Note**
>
> Cannot be modified when device is associated with Heat Pump Manager.

- `zone_2.fixed_target_temperature` *(number)*

  Set temperature for heating or cooling in fixed temperature mode for zone 2..

  Unit: 1 °C.

- `zone_2.heat_curve` *(number)*

  Current set heat curve ID (1-9) for zone 2.

- `zone_2.heat_curve_target_temperature` *(number, read-only)*

  Current target temperature set by heat curve for zone 2..

  Unit: 1 °C.

- `zone_2.heat_curve_enabled` *(boolean)*

  Indicator if heat curve mode is enabled for zone 2.

- `work_frequency` *(number, read-only)*

  Compressor operating frequency.

  Unit: 1 Hz

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

  Actual work mode of the heat pump outdoor unit. *off, cooling, heating*.

- `fan_speed` *(number, read-only)*

  Fan speed.

  Unit: revolutions per minute.

- `t1_water_outlet_temperature` *(number, read-only)*

  T1 temperature. Total water outlet temperature.

  Unit: 0.1 °C

- `t2_temperature` *(number, read-only)*

  T2 temperature. Temperature on the liquid coolant side.

  Unit: 0.1 °C

- `device_power` *(number, read-only)*

  Heat pump max power.

  Unit: 1 Wh

- `energy_used_total` *(number, read-only)*

  Total energy used by heat pump.

  Unit: 1 Wh

- `energy_generated_total` *(number, read-only)*

  Total energy generated by heat pump.

  Unit: 1 Wh

- `outdoor_unit_capacity` *(number, read-only)*

  Outdoor unit power capacity.

  Unit: 1 W

- `water_flow` *(number, read-only)*

  Water flow in instalation.

  Unit: 1 L/h

- `buffer_up_temperature` *(number, read-only)*

  Measured temperature in upper part of buffer.

  Unit: 0.1 °C

- `buffer_down_temperature` *(number, read-only)*

  Measured temperature in lower part of buffer.

  Unit: 0.1 °C

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

  Internal water circulation pump (P_i) state.

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

  External water circulation pump (P_o) state.

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

  DHW water pump (P_d) state.

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

  Water pump of the solar collector system (P_s) state.

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

  Mixed water pump (P_c) state.

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

  Actual electric heater state.

- `sv_1_state` *(boolean, read-only)*

  Three-way solenoid valve (SV1) state.

- `sv_2_state` *(boolean, read-only)*

  Two-way solenoid valve (SV2) state.

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

  Indicator of current defrost state.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"kaisai_khc"`

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

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

- `purpose` *(string)*

Commands

- `set_work_mode`

  Change work mode of heat pump.

  > **Note**
  >
  > Cannot be executed when heat pump is associated with Heat pump manager.

  Argument:

  Work mode *(string)*. One of: `automatic`, `cooling`, `heating`.

- `set_fixed_target_temperature`

  Set fixed target temperature for requested zone.

  Arguments:

  packed arguments *(table)*:

  - `zone` - zone number *(number)*:
    - minumum: 1
    - maximum: 2
  - `value` - target temperature *(number)*:
    - minimum: `fixed_target_temperature_minimum`
    - maximum: `fixed_target_temperature_maximum`
    - unit: °C

- `set_heat_demand`

  Set heat demand for requested zone.

  > **Note**
  >
  > Cannot be executed when heat pump is associated with Heat pump manager.

  Arguments:

  packed arguments *(table)*:

  - `zone` - zone number *(number)*:
    - minumum: 1
    - maximum: 2
  - `value` - demand *(boolean)*:

- `set_cool_demand`

  Set cool demand for requested zone.

  > **Note**
  >
  > Cannot be executed when heat pump is associated with Heat pump manager.

  Arguments:

  packed arguments *(table)*:

  - `zone` - zone number *(number)*:
    - minumum: 1
    - maximum: 2
  - `value` - demand *(boolean)*:

- `set_heat_curve`

  Set active heat curve for requested zone.

  > **Note**
  >
  > this only changes active heat curve number. To enable heat curve mode user has to use `set_heat_curve_enabled` command.

  Arguments:

  packed arguments *(table)*:

  - `zone` - zone number *(number)*:
    - minumum: 1
    - maximum: 2
  - `value` - heat curve number *(number)*:
    - minumum: 1
    - maximum: 9

- `set_heat_curve_enabled`

  Enable or disable heat curve mode for requested zone.

> **Note**
>
> Cannot be executed when heat pump is associated with Heat pump manager.

Arguments:

packed arguments *(table)*:

- `zone` - zone number *(number)*:
  - minumum: 1
  - maximum: 2
- `value` - enabled *(boolean)*:

- `set_electric_heater_active`

  Activate or disactivate electric heater.

  Argument:

  Active *(boolean)*

Examples

Set cooling work mode

```
modbus[1]:call("set_work_mode", "cooling")
```

Set heating work mode

```
modbus[1]:call("set_work_mode", "heating")
```

Set fixed target temperature for both zones

```
modbus[1]:call("set_fixed_target_temperature", { zone=1, value=26 })
modbus[1]:call("set_fixed_target_temperature", { zone=2, value=28 })
```

Turn on heat demand and turn off cool demand for both zones

```
modbus[1]:call("set_heat_demand", { zone=1, value=true })
modbus[1]:call("set_heat_demand", { zone=2, value=true })

modbus[1]:call("set_cool_demand", { zone=1, value=false })
modbus[1]:call("set_cool_demand", { zone=2, value=false })
```

Enable and set heat curve id for both zones

```
modbus[1]:call("set_heat_curve_enabled", { zone=1, value=true })
modbus[1]:call("set_heat_curve", { zone=1, value=2 })

modbus[1]:call("set_heat_curve_enabled", { zone=2, value=true })
modbus[1]:call("set_heat_curve", { zone=2, value=3 })
```

Activate electric heater

```
modbus[1]:call("set_electric_heater_active", true)
```

Kaisai KHC — Main DHW

Representation of DHW related parameters of Kaisai KHC device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 1 °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: 0.1 °C.

- `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 desired state.

- `circulation_pump_enabled` *(boolean)*

  Indicates if curculation pump work is enabled.

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

  Actual electric heater state.

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"kaisai_khc"`

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

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

- `purpose` *(string)*

Examples

Set target temperature to 45 in cooling work mode and 55 in other

```lua
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
```

Kaisai KHC — Temperature sensor

Representation of Temperature sensor related parameters of Kaisai KHC device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `temperature` *(number, read-only)*

  Measured temperature value.

  Unit: 0.1 °C.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"kaisai_khc"`

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

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

- `purpose` *(string)*

Mitsubishi Ecodan — Heat pump

Representation of Heat Pump related parameters of Mitsubishi Ecodan device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(string)*

  State of the heat pump: `on`, `off`

- `work_mode` *(string)*

  Work mode of the heat pump: `off`, `dhw`, `heating`, `cooling`, `buffer`, `freeze_stat`, `legionella`, `heating_eco`, `mode_1`, `mode_2`, `mode_3`, `heating_up`

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

  Outdoor temperature.

  Unit: 0.1 °C.

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

  Heating supply temperature.

  Unit: 0.1 °C.

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

  Heating return temperature.

  Unit: 0.1 °C.

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

  Hours heat pump is working.

- `zone1.target_temperature` *(integer)*

  Target temperature at first zone.

  Unit: 0.1 °C.

- `zone1.current_temperature` *(integer, read-only)*

  Current temperature at first zone.

  Unit: 0.1 °C.

- `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.

- `zone1.heating_supply` *(integer, read-only)*

  Current heating supply temperature at first zone.

  Unit: 0.1 °C.

- `zone1.heating_return` *(integer, read-only)*

  Current heating return temperature at first zone.

  Unit: 0.1 °C.

- `zone2.target_temperature` *(integer)*

  Target temperature at second zone.

  Unit: 0.1 °C.

- `zone2.current_temperature` *(integer, read-only)*

  Current temperature at second zone.

  Unit: 0.1 °C.

- `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.

- `zone2.heating_supply` *(integer, read-only)*

  Current heating supply temperature at second zone.

  Unit: 0.1 °C.

- `zone2.heating_return` *(integer, read-only)*

  Current heating return temperature at second zone.

  Unit: 0.1 °C.

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

  Informs that heating is demanded.

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

  Informs that cooling is demanded.

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

  Current deforst mode: `normal`, `standby`, `defrost`, `waiting_restart`.

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

  Residual heat removal: `normal`, `prepared`, `residual_heat_removal`.

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

  Frequency of master device.

  Unit: 1 Hz

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

  Refrigerant liquid temperature.

  Unit: 0.1 °C.

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

  Energy used for heating.

  Unit: 1 Wh

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

  Energy used for cooling.

  Unit: 1 Wh

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

  Energy used for domestic hot water.

  Unit: 1 Wh

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

  Energy produced for heating.

  Unit: 1 Wh

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

  Energy produced for cooling.

  Unit: 1 Wh

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

  Energy produced for domestic hot water.

  Unit: 1 Wh

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

  Flow rate.

  Unit: 0.01 L/min

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"mitsubishi_ecodan"`

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

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

- `purpose` *(string)*

Mitsubishi Ecodan — Main DHW

Representation of DHW related parameters of Mitsubishi Ecodan device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °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: 0.1 °C.

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

  Domestic Hot Water demand.

- `work_mode` *(string)*

  Domestic Hot Water work mode: `normal`, `eco`.

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

  Temperature drop.

  Unit: 0.1 °C.

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

  Heating supply temperature for domestic hot water

  Unit: 0.1 °C.

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

  Heating return temperature for domestic hot water.

  Unit: 0.1 °C.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"mitsubishi_ecodan"`

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

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

- `purpose` *(string)*

Remeha Elga ACE — Heat pump

Representation of Heat Pump related parameters of Remeha Elga ACE device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 0.1 °C.

- `target_temperature_indoor` *(number, read-only)*

  Set indoor temperature.

  Unit: 0.1 °C.

- `central_heating_target_temperature` *(number, read-only)*

  Central heating set temperature.

  Unit: 0.1 °C.

- `fixed_heating_target_temperature` *(number)*

  Set temperature for heating in fixed temperature mode.

  Unit: 0.1 °C.

- `temperature_outdoor` *(number, read-only)*

  Outdoor temperature.

  Unit: 0.1 °C.

- `heating_supply` *(number, read-only)*

  Heating supply temperature.

  Unit: 0.1 °C.

- `heating_return` *(number, read-only)*

  Heating return temperature.

  Unit: 0.1 °C.

- `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: 0.1 kWh.

- `current_power` *(number, read-only)*

  Current relative power produced.

  Unit: 0.1 kW.

- `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.

- `smart_grid_state` *(string)*

  Smart grid working state. One of: *not_set, standard_work, heat_pump_blocked, preheating*.

  > **Note**
  >
  > Cannot be modified when smart grid is not turned on (does not have label `smart_grid_support`).

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"remeha_elga_ace"`

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

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

- `purpose` *(string)*

Commands

- `reset_alarms`

  Sends request to heat pump device to reset alarms.

- `enable_smart_grid`

  Enable smart grid work mode support.

Examples

Enable smart grid constantly every minute if it is not enabled, set heat pump blocked mode (gas boiler only) if energy price is too high or standard_work otherwise

```lua
local pump = modbus[3]
local tooHighPrice = 0.5

if dateTime:changed() then

    if not pump:hasLabel("smart_grid_support") then
        pump:call("enable_smart_grid")
```

```lua
    end

    local price = energy_prices:getHourPrice(dateTime:getHours())

    if price >= tooHighPrice then
        pump:setValue("smart_grid_state", "heat_pump_blocked")
    else
        pump:setValue("smart_grid_state", "standard_work")
    end
end
```

Remeha Elga ACE — Temperature sensor

Representation of Temperature sensor related parameters of Remeha Elga ACE device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `temperature` *(number, read-only)*

  Measured temperature value.

  Unit: 0.1 °C.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"remeha_elga_ace"`

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

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

- `purpose` *(string)*

SolarEdge with MTTP extension model — Inverter

Representation of Inverter related parameters of SolarEdge device with MTTP Extension Model.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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: 1 Wh (with accuracy of 100 Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: 1 mW

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: 1 mW

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: 1 mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(number, read-only)*

  Instantaneous current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(number, read-only)*

  Instantaneous current on second group of photovoltaic panels.

  Unit: 1 mA

- `pv_3.active_power` *(number, read-only)*

  Current power produced by third group of photovoltaic panels.

  Unit: 1 mW

- `pv_3.voltage` *(number, read-only)*

  Current voltage on third group of photovoltaic panels.

  Unit: 1 mV

- `pv_3.current` *(number, read-only)*

  Instantaneous current on third group of photovoltaic panels.

  Unit: 1 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: 1 %

> **Note**
>
> Requires `advanced_power_control_enabled` to be set to `true` and `reactive_power_config` to `rrcr`.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"solar_edge_multiple"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter (set power limit to maximum).

  > **Note**
  >
  > Requires `advanced_power_control_enabled` to be set to `true` and `reactive_power_config` to `rrcr`.

- `turn_off`

  Turns off inverter (set power limit to 0).

  > **Note**
  >
  > Requires `advanced_power_control_enabled` to be set to `true` and `reactive_power_config` to `rrcr`.

SolarEdge — Inverter

Representation of Inverter related parameters of SolarEdge device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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: 1 Wh (with accuracy of 100 Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `power_to_grid` *(number, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid. Unit: 1 mW

- `pv.active_power` *(number, read-only)*

  Current power produced by photovoltaic panels.

  Unit: 1 mW

- `pv.voltage` *(number, read-only)*

  Current voltage on photovoltaic panels.

  Unit: 1 mV

- `pv.current` *(number, read-only)*

  Instantaneous current on photovoltaic panels.

  Unit: 1 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: 1 %

  > **Note**
  >
  > Requires `advanced_power_control_enabled` to be set to `true` and `reactive_power_config` to `rrcr`.

- `pv_total_active_power` *(number, read-only)*

  Current total power produced by all photovoltaic panels.

  Unit: 1 mW

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"solar_edge_single"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter (set power limit to maximum).

> **Note**
>
> Requires `advanced_power_control_enabled` to be set to `true` and `reactive_power_config` to `rrcr`.

- `turn_off`

  Turns off inverter (set power limit to 0).

  > **Note**
  >
  > Requires `advanced_power_control_enabled` to be set to `true` and `reactive_power_config` to `rrcr`.

Solax X1 — Battery

Representation of Battery related parameters of Solax X1 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `soc` *(number, read-only)*

  Current state of charge.

  Unit: 1 %

- `energy_charged_total` *(number, read-only)*

  Amount of energy charged to the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_discharged_total` *(number, read-only)*

  Amount of energy consumed from the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh)

- `energy_discharged_today` *(number, read-only)*

  Amount of energy consumed from the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh)

- `charge_power` *(number, read-only)*

  Current charing (positive number) or discharging (negative number) power.

  Unit: 1 mW

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

  Parameter describing maximum value for charge power. Unit: 1 mW

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

  Parameter describing maximum value for discharge power. Unit: 1 mW

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"battery"`

- `variant` *(string, read-only)* = `"solax_x1"`

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

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

- `purpose` *(string)*

Commands

- `charge`

  Calls battery to charge during given period of time.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 10000
    - maximum: from parameter `maximum_charge_power`
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 65535
    - parameter optional, maximum - 65535 is used when not provided

- `discharge`

  Calls battery to discharge during given period of time.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 10000
    - maximum: from parameter `maximum_discharge_power`
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 65535

- parameter optional, maximum - 65535 is used when not provided

- `stop`

  Disables the remote control of battery.

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
```

Solax X1 — Inverter

Representation of Inverter related parameters of Solax X1 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 mW

- `energy_produced_total` *(number, read-only)*

  Total amount of energy produced by PV over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: 1 mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(number, read-only)*

  Instantaneous current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(number, read-only)*

  Instantaneous current on second group of photovoltaic panels.

  Unit: 1 mA

- `active_power_limit` *(integer)*

  Percent of max power at which inverter is going to work.

  Unit: 1 %

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"solax_x1"`

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

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

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter (set power limit to maximum).

- `turn_off`

  Turns off inverter (set power limit to 0).

Solax X1 — Energy meter

Representation of Energy meter related parameters of Solax X1 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `meter_number` *(number, read-only)*

  Connected meter number. Solax can have 2 meters connected which has different parameters.

- `total_active_power` *(number, read-only)*

  Current power fed to (negative number) or consumed from (positive number) the power grid.

  Unit: 1 mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_total` *(number, read-only)*

  Amount of energy consumed from the power grid over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `energy_consumed_today` *(number, read-only)*

  Amount of energy consumed from the power grid over today.

  Unit: 1 Wh (with accuracy of 10 Wh)

- `grid.active_power` *(number, read-only)*

  Current power on the power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mW

- `grid.voltage` *(number, read-only)*

  Current voltage on the power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mV

- `grid.current` *(number, read-only)*

  Current current on the power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mA

- `grid.frequency` *(number, read-only)*

  Current AC frequency. This parameter is available only when `meter_number` is equal to 1.

  Unit: 0.01 Hz

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

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

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"solax_x1"`

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

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

- `purpose` *(string)*

Solax X3 — Battery

Representation of Battery related parameters of Solax X3 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `soc` *(number, read-only)*

  Current state of charge.

  Unit: 1 %

- `energy_charged_total` *(number, read-only)*

  Amount of energy charged to the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh)

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh)

- `energy_discharged_total` *(number, read-only)*

  Amount of energy consumed from the battery over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh)

- `energy_discharged_today` *(number, read-only)*

  Amount of energy consumed from the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh)

- `charge_power` *(number, read-only)*

  Current charing (positive number) or discharging (negative number) power.

  Unit: 1 mW

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

  Parameter describing maximum value for charge power. Unit: 1 mW

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

  Parameter describing maximum value for discharge power. Unit: 1 mW

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

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

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

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

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"battery"`

- `variant` *(string, read-only)* = `"solax_x3"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Commands

- `charge`

  Calls battery to charge during given period of time.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 10000
    - maximum: from parameter `maximum_charge_power`
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 65535
    - parameter optional, maximum - 65535 is used when not provided

- `discharge`

  Calls battery to discharge during given period of time.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 10000
    - maximum: from parameter `maximum_discharge_power`
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 65535

- parameter optional, maximum - 65535 is used when not provided

- `stop`

  Disables the remote control of battery.

Examples

Turn on battery charging with 1kW for 1 hour at 1:00PM

```lua
if dateTime:changed() then
  if dateTime:getHours() == 13 and dateTime:getMinutes() == 0 then
    modbus[2]:call("charge", { 1000000, 3600 })
  end
end
```

Solax X3 — Inverter

Representation of Inverter related parameters of Solax X3 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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: 1 mW

- `energy_produced_total` *(number, read-only)*

  Total amount of energy produced by PV over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_produced_today` *(number, read-only)*

  Amount of energy produced by PV today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `pv_1.active_power` *(number, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: 1 mW

- `pv_1.voltage` *(number, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(number, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(number, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(number, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(number, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: 1 mA

- `active_power_limit` *(integer)*

  Percent of max power at which inverter is going to work.

  Unit: 1 %

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"solax_x3"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter (set power limit to maximum).

- `turn_off`

  Turns off inverter (set power limit to 0).

Solax X3 — Energy meter

Representation of Energy meter related parameters of Solax X3 device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `total_active_power` *(number, read-only)*

  Current power fed to (negative number) or consumed from (positive number) the power grid.

  Unit: 1 mW

- `energy_fed_total` *(number, read-only)*

  Amount of energy fed to the power grid over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh).

- `energy_fed_today` *(number, read-only)*

  Amount of energy fed to the power grid today.

  Unit: 1 Wh (with accuracy of 10 Wh).

- `energy_consumed_total` *(number, read-only)*

  Amount of energy consumed from the power grid over a lifetime.

  Unit: 1 Wh (with accuracy of 10 Wh).

- `energy_consumed_today` *(number, read-only)*

  Amount of energy consumed from the power grid over today.

  Unit: 1 Wh (with accuracy of 10 Wh).

- `phase_1.active_power` *(number, read-only)*

  Current power on first phase of power grid.

  Unit: 1 mW

- `phase_1.voltage` *(number, read-only)*

  Current voltage on first phase of power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mV

- `phase_1.current` *(number, read-only)*

  Current current on first phase of power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mA

- `phase_1.frequency` *(number, read-only)*

  Current AC frequency on first phase. This parameter is available only when `meter_number` is equal to 1.

  Unit: Hz with two decimal numbers

- `phase_2.active_power` *(number, read-only)*

  Current power on second phase of power grid.

  Unit: 1 mW

- `phase_2.voltage` *(number, read-only)*

  Current voltage on second phase of power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mV

- `phase_2.current` *(number, read-only)*

  Current current on second phase of power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mA

- `phase_2.frequency` *(number, read-only)*

  Current AC frequency on second phase. This parameter is available only when `meter_number` is equal to 1.

  Unit: Hz with two decimal numbers

- `phase_3.active_power` *(number, read-only)*

  Current power on third phase of power grid.

  Unit: 1 mW

- `phase_3.voltage` *(number, read-only)*

  Current voltage on third phase of power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mV

- `phase_3.current` *(number, read-only)*

  Current current on third phase of power grid. This parameter is available only when `meter_number` is equal to 1.

  Unit: 1 mA

- `phase_3.frequency` *(number, read-only)*

  Current AC frequency on third phase. This parameter is available only when `meter_number` is equal to 1.

  Unit: 0.01 Hz

Device properties (full spec)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"solax_x3"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Solis — Inverter

Representation of Inverter related parameters of Solis device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 mW

- `grid_total_active_power` *(integer, read-only)*

  Current total power on the grid.

  Unit: 1 mW

- `energy_produced_total` *(integer, read-only)*

  Total amount of energy produced by PV over a lifetime.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `energy_produced_today` *(integer, read-only)*

  Amount of energy produced by PV today.

  Unit: 1 Wh (with accuracy of 100 Wh).

- `power_to_grid` *(integer, read-only)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: 1 mW

- `pv_1.active_power` *(integer, read-only)*

  Current power produced by first group of photovoltaic panels.

  Unit: 1 mW

- `pv_1.voltage` *(integer, read-only)*

  Current voltage on first group of photovoltaic panels.

  Unit: 1 mV

- `pv_1.current` *(integer, read-only)*

  Current current on first group of photovoltaic panels.

  Unit: 1 mA

- `pv_2.active_power` *(integer, read-only)*

  Current power produced by second group of photovoltaic panels.

  Unit: 1 mW

- `pv_2.voltage` *(integer, read-only)*

  Current voltage on second group of photovoltaic panels.

  Unit: 1 mV

- `pv_2.current` *(integer, read-only)*

  Current current on second group of photovoltaic panels.

  Unit: 1 mA

- `pv_3.active_power` *(integer, read-only)*

  Current power produced by third group of photovoltaic panels.

  Unit: 1 mW

- `pv_3.voltage` *(integer, read-only)*

  Current voltage on third group of photovoltaic panels.

  Unit: 1 mV

- `pv_3.current` *(integer, read-only)*

  Current current on third group of photovoltaic panels.

  Unit: 1 mA

- `pv_4.active_power` *(integer, read-only)*

  Current power produced by fourth group of photovoltaic panels.

  Unit: 1 mW

- `pv_4.voltage` *(integer, read-only)*

  Current voltage on fourth group of photovoltaic panels.

  Unit: 1 mV

- `pv_4.current` *(integer, read-only)*

  Current current on fourth group of photovoltaic panels.

  Unit: 1 mA

- `phase_1.voltage` *(integer, read-only)*

  Current voltage on first phase of power grid.

  Unit: 1 mV

- `phase_1.current` *(integer, read-only)*

  Current current on first phase of power grid.

  Unit: 1 mA

- `phase_2.voltage` *(integer, read-only)*

  Current voltage on second phase of power grid.

  Unit: 1 mV

- `phase_2.current` *(integer, read-only)*

  Current current on second phase of power grid.

  Unit: 1 mA

- `phase_3.voltage` *(integer, read-only)*

  Current voltage on third phase of power grid.

  Unit: 1 mV

- `phase_3.current` *(integer, read-only)*

  Current current on third phase of power grid.

  Unit: 1 mA

- `active_power_limit` *(integer)*

  Percent of max power at which inverter is going to work.

  Unit: 1 %

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"solis"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Commands

- `turn_on`

  Turns on inverter (set power limit to maximum).

- `turn_off`

  Turns off inverter (set power limit to 0).

TECH LE-3x230mb — Energy meter

Representation of Energy Meter related parameters of TECH LE-3x230mb device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `phase_1.active_power` *(number, read-only)*

  First phase active power.

  Unit: 1 mW

- `phase_1.voltage` *(number, read-only)*

  First phase voltage.

  Unit: 1 mV

- `phase_1.current` *(number, read-only)*

  First phase current.

  Unit: 1 mA

- `phase_1.apparent_power` *(number, read-only)*

  First phase apparent power.

  Unit: 1 mVA

- `phase_1.reactive_power` *(number, read-only)*

  First phase reactive power.

  Unit: 1 mvar

- `phase_2.active_power` *(number, read-only)*

  Second phase active power.

  Unit: 1 mW

- `phase_2.voltage` *(number, read-only)*

  Second phase voltage.

  Unit: 1 mV

- `phase_2.current` *(number, read-only)*

  Second phase current.

  Unit: 1 mA

- `phase_2.apparent_power` *(number, read-only)*

  Second phase apparent power.

  Unit: 1 mVA

- `phase_2.reactive_power` *(number, read-only)*

  Second phase reactive power.

  Unit: 1 mvar

- `phase_3.active_power` *(number, read-only)*

  Third phase active power.

  Unit: 1 mW

- `phase_3.voltage` *(number, read-only)*

  Third phase voltage.

  Unit: 1 mV

- `phase_3.current` *(number, read-only)*

  Third phase current.

  Unit: 1 mA

- `phase_3.apparent_power` *(number, read-only)*

  Third phase apparent power.

  Unit: 1 mVA

- `phase_3.reactive_power` *(number, read-only)*

  Third phase reactive power.

  Unit: 1 mvar

- `total_active_power` *(number, read-only)*

  Total active power on all phases.

  Unit: 1 mW

- `total_apparent_power` *(number, read-only)*

  Total apparent power on all phases.

  Unit: 1 mVA

- `total_reactive_power` *(number, read-only)*

  Total reactive power on all phases.

  Unit: 1 mvar

- `energy_consumed_total` *(number, read-only)*

  Energy consumed lifetime on all phases.

  Unit: 1 Wh

- `energy_consumed_today` *(number, read-only)*

  Energy consumed today on all phases.

  Unit: 1 Wh

- `energy_fed_total` *(number, read-only)*

  Energy fed lifetime on all phases.

  Unit: 1 Wh

- `energy_fed_today` *(number, read-only)*

  Energy fed today on all phases.

  Unit: 1 Wh

- `reactive_energy_consumed_total` *(number, read-only)*

  Reactive energy consumed lifetime on all phases.

  Unit: 1 varh

- `reactive_energy_consumed_today` *(number, read-only)*

  Reactive energy consumed today on all phases.

  Unit: 1 varh

- `reactive_energy_fed_total` *(number, read-only)*

  Reactive energy fed lifetime on all phases.

  Unit: 1 varh

- `reactive_energy_fed_today` *(number, read-only)*

  Reactive energy fed today on all phases.

  Unit: 1 varh

Device properties (**full spec**)

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"tech_le3x230mb"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

KM1 energy meter converter

Representation of KM1 Energy Meter converter device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `tariff_1.energy_consumed_total` *(integer, read-only)*

  Total energy delivered to client in tariff 1.

  Unit: 1 Wh
- `tariff_2.energy_consumed_total` *(integer, read-only)*

  Total energy delivered to client in tariff 2.

  Unit: 1 Wh
- `tariff_1.energy_fed_total` *(integer, read-only)*

  Total energy delivered by client in tariff 1.

  Unit: 1 Wh
- `tariff_2.energy_fed_total` *(integer, read-only)*

  Total energy delivered by client in tariff 2.

  Unit: 1 Wh
- `tariff_indicator` *(integer, read-only)*

  Electricity tariff indicator.
- `power_to_grid` *(integer, read-only)*

  Current power fed to the power grid.

  Unit: 1 mW
- `power_from_grid` *(integer, read-only)*

  Current power consumed from the power grid.

  Unit: 1 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: 1 mV

- `phase_1.current` *(integer, read-only)*

  First phase current.

  Unit: 1 mA

- `phase_1.active_power` *(integer, read-only)*

  First phase active power.

  Unit: 1 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: 1 mV

- `phase_2.current` *(integer, read-only)*

  Second phase current.

  Unit: 1 mA

- `phase_2.active_power` *(integer, read-only)*

  Second phase active power.

  Unit: 1 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: 1 mV

- `phase_3.current` *(integer, read-only)*

  Third phase current.

  Unit: 1 mA

- `phase_3.active_power` *(integer, read-only)*

  Third phase active power.

  Unit: 1 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.

- `energy_consumed_total` *(integer, read-only)*

  Total energy delivered to client for meter without tariffs.

  Unit: 1 Wh

- `energy_fed_total` *(integer, read-only)*

  Total energy delivered by client for meter without tariffs.

  Unit: 1 Wh

- `p1_meter_no_communication_time` *(integer, read-only)*

  Number of minutes without P1 communication. 0 if connection is good.

- `parsing_errors` *(integer, read-only)*

  Number of parsing errors the converter encountered.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"modbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"km_1"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Commands

- `reset_tariff_energy_data`

  Reset all tariff total energy data to 0.

- `reset_energy_data`

  Reset non tariff total energy data to 0.

Virtual devices

Virtual devices are used to implement functionality that requires connecting a couple of physical devices working together. Custom devices can be configured and programmed by the user. Devices can be added and modified via [REST API](#) or the web app.

WTP devices can be accessed from scripts by indexing global `virtual` container, e.g. `virtual[7]` returns virtual device with ID #7. This container is available in every execution context.

Properties can only be accessed via `setValue` and `getValue` methods.

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

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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`.

  Minimum from parameter `target_temperature_minimum`, maximum from parameter `target_temperature_maximum`.

  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 doesn't 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_minimum` *(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.

- `target_temperature_heating` *(integer)*

  Target temperature for heating mode.

  Minimum from parameter `target_temperature_heating_minimum`, maximum from parameter `target_temperature_heating_maximum`.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_heating_minimum` *(integer)*

  Lower limit of the heating target temperature. Could not be greater than maximum. Setting minimum value above target value, will also change heating target value to minimum.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_heating_maximum` *(integer)*

  Upper limit of the heating target temperature. Could not be less than minimum. Setting maximum below target, will also change heating target value to maximum.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_cooling` *(integer)*

  Target temperature for heating mode.

  Minimum from parameter `target_temperature_cooling_minimum`, maximum from parameter `target_temperature_cooling_maximum`.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_cooling_minimum` *(integer)*

  Lower limit of the cooling target temperature. Could not be greater than maximum. Setting minimum value above target value, will also change cooling target value to minimum.

  Unit: °C with one decimal number, multiplied by 10.

- `target_temperature_cooling_maximum` *(integer)*

  Upper limit of the cooling target temperature. Could not be less than minimum. Setting maximum below target, will also change cooling target value to maximum.

  Unit: °C with one decimal number, multiplied by 10.

- `hysteresis` *(integer)*

  Damper factor, which will protect from continuous 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`

- `mode_mutable` *(boolean)*

  Blocks `set_mode` command when set to `false`.

- `schedule_id` *(integer, read-only, nilable)*

  ID of the schedule used by thermostat. If no schedule is used, it will return `nil`.

- `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.

- `cooling_control.maximum_humidity` *(integer)*

  This parameter controls how high the relative humidity can be when cooling is active, once relative humidity exceeds this value cooling will stop to prevent condensation.

  Unit: rH% with one decimal number, multiplied by 10.

- `cooling_control.dew_point_control.enabled` *(boolean)*

  Indicates if dew point control in cooling is enabled. If enabled cooling will be stopped if supply temperature drops below calculated dew point temperature plus offset. Maximum humidity control works if dew point control is disabled.

- `cooling_control.dew_point_control.offset` *(integer)*

  Offset which is added to dew point temperature to set threshold for supply temperature below which cooling is disabled.

  Unit: °C with one decimal number, multiplied by 10.

- `cooling_control.dew_point_control.hysteresis` *(integer)*

  Hysteresis for dew point control threshold.

  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.

  Unit: %

- `radiator_control.use_for_heating` *(boolean)*

  Configures whether radiator actuators should be controlled (opening and closing) while heating or not (always closed).

- `radiator_control.use_for_cooling` *(boolean)*

Configures whether radiator actuators should be controlled (opening and closing) while cooling or not (always closed).

- `radiator_control.emergency_opening` *(integer)*

Configures emergency opening level while temperature sensor is broken or not available e.g offline.

Unit: %

- `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 continuous 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 `antifrost_protection_temperature`.

- `antifrost_protection_temperature` *(integer)*

Threshold for antifrost protection algorithm.

Unit: °C with one decimal number, multiplied by 10.

- `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.

- `dew_point` *(integer, read-only)*

  Dew point calculated based on room temperature sensor and humidity. Available when room temperature sensor and humidity sensor assigned and online with valid values.

  Unit: °C with one decimal number, multiplied by 10.

- `floor_dew_point` *(integer, read-only)*

  Dew point calculated based on floor temperature sensor and humidity. Available when floor temperature sensor and humidity sensor assigned and online with valid values.

  Unit: °C with one decimal number, multiplied by 10.

- `associations.room_temperature_sensor` *(device, read-only)*

  Reference to associated room temperature sensor. Returns `nil` when not associated.

- `associations.floor_temperature_sensor` *(device, read-only)*

  Reference to associated floor temperature sensor. Returns `nil` when not associated.

- `associations.humidity_sensor` *(device, read-only)*

  Reference to associated humidity sensor. Returns `nil` when not associated.

- `associations.temperature_regulator` *(device, read-only)*

  Reference to associated temperature regulator. Returns `nil` when not associated.

- `associations.radiator_actuators` *(array of devices, read-only)*

  Reference to associated radiator actuators. Returns empty array when no devices associated.

- `associations.relays` *(array of devices, read-only)*

  Reference to associated relays. Returns empty array when no devices associated.

- `associations.heating_relays` *(array of devices, read-only)*

  Reference to associated heating relays. Returns empty array when no devices associated.

- `associations.cooling_relays` *(array of devices, read-only)*

  Reference to associated cooling relays. Returns empty array when no devices associated.

- `associations.opening_sensors` *(array of devices, read-only)*

  Reference to associated opening sensors. Returns empty array when no devices associated.

- `associations.two_state_input_sensors` *(array of devices, read-only)*

  Reference to associated two state input sensors. Returns empty array when no devices associated.

- `associations.supply_temperature_sensor` *(device, read-only)*

  Reference to associated supply temperature sensor used for dew point control in cooling. Returns `nil` when not associated.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"virtual"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"thermostat"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

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 doesn't 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
```

Thermostat output group (virtual contact)

The virtual thermostat output group controls the output devices based on the readings from the virtual thermostats e.g. 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

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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 e.g. 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.

- `pump_antistop.enabled` *(boolean)*

  Indicates if antistop function is turned on. Function to automatically turn on circuit pump when it was not turned on for a long time.

- `pump_antistop.work_time` *(number)*

  Turn on time for pump if antistop active. Unit: seconds.

- `pump_antistop.pause_time` *(number)*

  Tells how long pump should be turned off before turned on by antistop. Unit: hours

- `associations.two_state_input_sensor` *(device)*

  Reference to associated two state input sensor. Returns `nil` when not associated.

- `associations.thermostats` *(array of devices)*

  Reference to associated thermostats. Returns empty array when no devices associated.

- `associations.relays` *(array of devices)*

  Reference to associated relays. Returns empty array when no devices associated.

- `associations.heating_relays` *(array of devices)*

  Reference to associated heating relays. Returns empty array when no devices associated.

- `associations.cooling_relays` *(array of devices)*

  Reference to associated cooling relays. Returns empty array when no devices associated.

- `associations.pellet_boilers` *(array of devices)*

  Reference to associated pellet boilers. Returns empty array when no devices associated.

- `associations.heat_pumps` *(array of devices)*

  Reference to associated heat pumps. Returns empty array when no devices associated.

- `associations.circuit_pumps` *(array of devices)*

  Reference to associated circuit pumps. Returns empty array when no devices associated.

- `associations.valves` *(array of devices)*

  Reference to associated valves. Returns empty array when no devices associated.

Device properties (full spec)

- `class` *(string, read-only)* = `"virtual"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"thermostat_output_group"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Commands

- `enable`

  Enables device.

- `disable`

  Disables device.

- `set_propagation_delay` *(number)*

  Sets desired propagation delay for outputs.

  Argument:

  delay in seconds *(number)*

- `set_antistop_enabled` *(boolean)*

  Calls Thermostat Output Group to enable/disable antistop function.

  Argument:

  Enable/disable.

- `set_antistop_work_time` *(number)*

  Calls Thermostat Output Group to set antistop work time to desired value.

  Argument:

  Work time in seconds.

- `set_antistop_pause_time` *(boonumberlean)*

  Calls Thermostat Output Group to set antistop pause time to desired value.

  Argument:

  Pause time in hours.

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

```lua
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
```

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

Direct access to properties is not allowed. You can read or change values using `getValue`
and `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting
wrong value type will cause a script error.*

- `state` *(boolean)*

  Current relays output state.

- `blind_duration` *(integer)*

  Duration when integrator ignores consecutive relay state changes.

  Unit: 1 ms

  Range: 0 ms – 10 000 ms.

- `associations.relays` *(array of devices)*

  Reference to associated relays. Returns empty array when no devices associated.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"virtual"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"relay_integrator"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Commands

- `turn_on`

  Changes the `state` of integrator and all associated relays to `true`.

- `turn_off`

  Changes the `state` of integrator and all associated relays to `false`.

- `toggle`

  Changes the `state` of all associated relays to the opposite of integrators `state`.

Examples

Turn on all assigned relays when motion sensor triggers

```lua
if wtp[3]:changedValue("motion_detected") then
  virtual[5]:setValue("state", true)
end
```

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

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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.

- `lift_position_description` *(string, read-only)*

  Describes lift position summary of associated blind controllers.

- `associations.blind_controllers` *(array of devices)*

  Reference to associated blind controllers. Returns empty array when no devices associated.

> **Required label:** `"percent_tilt_control"`
>
> - `last_set_target_tilt` *(number, read-only)*
>
>   Contains last set target tilt via integrator.
> - `tilt_in_progress` *(boolean, read-only)*
>
>   Indicates if tilt action requested via integrator is in progress.
> - `tilt_position_description` *(array of devices)*
>
>   Describes tilt position summary of associated blind controllers.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"virtual"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"blind_controller_integrator"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Commands

- `open`

  Sets the `target_opening` of all associated blind controllers to the value of the argument.

  Argument:

  target opening in percent *(number)*

- `up`

  Sets the `target_opening` of all associated blind controllers to 100%.

- `down`

  Sets the `target_opening` of all associated blind controllers to 0%.

- `stop`

  Call `stop` command on all associated blind controllers.

- `tilt`

  Sets the `target_tilt` of all associated blind controllers to the value of the argument.

  Argument:

  target tilt in percent *(number)*

Examples

Open all assigned blinds at sunrise and close at sunset

```lua
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

```lua
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

```lua
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
```

Custom device

Custom device is a special type of device in which the user can design the layout of the elements 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 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 element 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 [controls][Controls] chapter for more information.

User can choose from a couple of variants of custom device. That way user can add integration to specific type of device and integrate it with internal Sinum algorithms.

Device and elements may be added, edited or removed via REST API

or a web application served through the central unit server.

Methods

This is an extension of the methods available in standard devices.

- `getElement(element_name)`

  Returns reference to control or nil if it doesn't exist.

  Returns:

  - *any* — depends on element type

  Arguments:

  - `element_name` *(string)* — name of element configured by user

- `getComponent(component_name)`

  Returns reference to component or nil if it doesn't exist.

  Returns:

  - *any* — depends on component type

  Arguments:

  - `component_name` *(string)* — name of component configured by user

- `changedVariantDeviceValue(property_name)`

  Checks if specific property of device variant data has recently changed (thus is source of event).

  Returns:

  ○ *boolean*

  Arguments:

  ○ `property_name` *(string)* — name of variant data property which should be checked

- `getVariantDeviceValue(property_name)`

  Returns value of object variant data property.

  Returns:

  ○ *any* — depends on property type

  Arguments:

  ○ `property_name` *(string)* — name of variant data property

- `setVariantDeviceValue(property_name, property_value, stop_propagation)`

  Sets value for object variant data property.

  Returns:

  ○ *userdata* — reference to device object, for chained calls

  Arguments:

  ○ `property_name` *(string)* — name of variant data property
  ○ `property_value` *(any)* — 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 (e.g. `setTargetTemperature`) won't be executed after value change. This may help reduce callback propagation infinite loops - see explanation below.

- `setVariantDeviceValueAfter(property_name, property_value, seconds_after)`

  Sets value for object variant data property after certain time.

  Returns:

  ○ *userdata* — reference to device object, used for chained calls

  Arguments:

  ○ `property_name` *(string)* — name of variant data 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

- `addWarning(text, title_id)`

  Add warning message with specified text and translation text ID to device.

  Returns:

- *userdata* — reference to device object, used for chained calls

  Arguments:

  - `text` *(string)* — text that will be displayed in warning message
  - `title_id` *(number)* — translation text ID used for warning message title, parameter is optional, when no provied the warning title will `Warning`

- `removeWarning(text, title_id)`

  Remove warning message with specified text and translation text ID from device.

  Returns:

  - *userdata* — reference to device object, used for chained calls

  Arguments:

  - `text` *(string)* — text that is displayed in warning message
  - `title_id` *(number)* — translation text ID used for warning message title, parameter is optional, when no provied the warning title `Warning` assumed

- `hasWarning(text, title_id)`

  Checks if warning message with specified text and translation text ID exist in device.

  Returns:

  - *boolean* — true if warning message exists in devices

  Arguments:

  - `text` *(string)* — text that is displayed in warning message
  - `title_id` *(number)* — translation text ID used for warning message title, parameter is optional, when no provied the warning title `Warning` assumed

- `updateWarning(text, title_id, condition)`

  Add or remove warning message with specified text and translation text ID to device based on provided condition.

  Returns:

  - *userdata* — reference to device object, used for chained calls

  Arguments:

  - `text` *(string)* — text that will be displayed in warning message
  - `title_id` *(number)* — translation text ID used for warning message title, parameter is optional, can be `nil`, when no provied the warning title will `Warning`
  - `condition` *(boolean)* — indicates if warning should be added or removed

- `addError(text, title_id)`

  Add error message with specified text and translation text ID to device.

  Returns:

  - *userdata* — reference to device object, used for chained calls

  Arguments:

- ◦ `text` *(string)* — text that will be displayed in error message
  - ◦ `title_id` *(number)* — translation text ID used for error message title, parameter is optional, when no provied the error title will `Error`

- `removeError(text, title_id)`

  Remove error message with specified text and translation text ID from device.

  Returns:

  - ◦ *userdata* — reference to device object, used for chained calls

  Arguments:

  - ◦ `text` *(string)* — text that is displayed in error message
  - ◦ `title_id` *(number)* — translation text ID used for error message title, parameter is optional, when no provied the error title `Error` assumed

- `hasError(text, title_id)`

  Checks if error message with specified text and translation text ID exist in device.

  Returns:

  - ◦ *boolean* — true if error message exists in device

  Arguments:

  - ◦ `text` *(string)* — text that is displayed in error message
  - ◦ `title_id` *(number)* — translation text ID used for error message title, parameter is optional, when no provied the error title `Error` assumed

- `updateError(text, title_id, condition)`

  Add or remove error message with specified text and translation text ID to device based on provided condition.

  Returns:

  - ◦ *userdata* — reference to device object, used for chained calls

  Arguments:

  - ◦ `text` *(string)* — text that will be displayed in error message
  - ◦ `title_id` *(number)* — translation text ID used for error message title, parameter is optional, can be `nil`, when no provied the error title will `Error`
  - ◦ `condition` *(boolean)* — indicates if error should be added or removed

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.

- `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.

- `status` *(string)*

  Current device status. Available values: `unknown`, `online`, `offline`.

  > **Note**
  >
  > For `generic` variant it is set to `online` on application startup out of the box, for other variants the initial value is set to `unknown` and it is up to custom device developer to set proper status during runtime or in `onInit` handler.

- `messages` *(array-like table, read-only)*

  Collection of device specific messages. Contains device error/warning details.

- `labels` *(array-like table, read-only)*

  Collection of device specific labels. Contains device specification and additional flags.

- `tags` *(array-like table)*

  Collection of tags (array-like table of strings) 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: `"#00ffff"`

- `visible` *(boolean)*

  Indicates if device is visible in application.

- `enabled` *(boolean)*

  Current device state (*enabled* = `true` / *disabled* = `false`). When disabled, Lua code won't be executed.

- `blocked` *(boolean, read-only)*

  Indicates whether the device is blocked by its author. Blocked device cannot be modified.

- `uuid` *(string, read-only)*

  Unique identifier of the device in the marketplace.

- `version` *(string)*

  Version of custom device. It has to fulfill format `a.b.c`. Example: `"1.0.0"`

- `blockade_pin_code_enabled` *(boolean, read-only)*

  Indicates if the device is blocked by pin code. Device blocked by pin cannot be modifed or deleted.

- `integration_model` *(string)* Model of the device connected to the custom device.

- `integration_version` *(string)* Version of the device connected to the custom device.

- `integration_uid` *(string)* Uid of the device connected to the custom device.

- `logo` *(string)*

  Logo displayed on the widget. Possible values: `blank`, `four_eco`, `galmet`, `econtro`, `kaisai`, `hymon`, `kolton`, `eplucon`, `wanas`, `surma`, `defro`.

- `bannable` *(boolean)*

  Indicates if the device can be set as banned.

- `banned` *(boolean, read-only)*

  Similiar to `enabled` property but set by system. Defines if custom device failed and is excluded (not able to execute) when condition `error_counter >= max_errors` is met.

- `ban_reason` *(string, read-only)*

  Information about error which caused custom device to get banned.

- `error_counter` *(integer, read-only)*

  Error counter counts error on every fail of custom device e.g. syntax error or exceeding execution time.

- `max_errors` *(integer)*

  Maximum possible errors counted before custom device gets banned.

  Range: 1 – 10

- `max_execution_time` *(integer)*

  Maximum possible execution time in seconds before custom device will get terminated with error.

  Unit: 1 s

  Range: 1 s – 120 s

Commands

- `enable`

  Enables device.

- `disable`

  Disables device.

- More...

  Depending on the custom device variant device can support more commands. For more information check the specific <u>variant</u>. Custom commands can be also handled by the Lua script.

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 `onCommand` callback handler for commands and write its logic in Custom Device - see <u>Lua reference</u> for more details.

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

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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.

- `schedule_id` *(integer, read-only, nilable)*

ID of the schedule used by thermostat. If no schedule is used, it will return `nil`.

- `hysteresis.heating` *(number)*

Damper factor, which will protect from continuous 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 continuous 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 continuous 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 continuous 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.

Required label: `"has_schedule"`

- `target_temperature_mode` *(string)*

> Device target temperature mode. Specifies if manager is working in `constant` mode with mode dependent temperature or according to schedule in `schedule` mode with many target temperatures in time, configured by user.

- `associations.heat_pump` *(device)*

  Reference to associated heat pump. Returns `nil` when not associated.

- `associations.domestic_hot_water` *(device)*

  Reference to associated domestic hot water. Returns `nil` when not associated.

- `associations.temperature_sensors` *(array of devices)*

  Reference to associated temperature sensors. Returns empty array when no devices associated.

- `associations.temperature_regulator` *(device)*

  Reference to associated domestic hot water. Returns `nil` when not associated.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"virtual"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump_manager"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

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)*

- `enable_schedule`

  Calls Heat Pump Manager to change `target_temperature_mode` to `constant`.
  Available only when device has label `has_schedule`.

- `disable_schedule`

  Calls Heat Pump Manager to change `target_temperature_mode` to `schedule`.
  Available only when device has label `has_schedule`.

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
```

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, close gate or stop when moving depending on current `state` / `stopped_state` by sending on/off impulse for 500ms.

- `partial_open_close_output`, commands the gate controller device to partially open, close gate or stop when moving depending on current `state` / `stopped_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 (e.g. wall switch impulse, or RC remote output impulse) and trigger `full_move` and `stop` actions

- `button`, this device can be used to trigger `full_move`, `partial_move` and `stop` actions

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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.

- `stopped_state` *(string, read-only, optional, nilable)*

  The state in which the gate was stopped. Can be `closing`, `opening` or `nil`.

- `partial_movement` *(boolean)*

  Indicates if the current `state` is the result of partial move/open command (`true`) or not (`false`).

- `operating_logic` *(string)*

  Defines the operating logic of the gate controller with close status sensor. Cannot be changed if close status sensor is not associated.

  One of: `full_step_by_step`, `half_step_by_step`.

- ◦ `full_step_by_step` - The gate controller moves in sequence: `opening` -> `stop (open)` -> `closing` -> `stop (open)` -> `opening`

- ◦ `half_step_by_step` - The gate controller moves in sequence: `opening` -> `stop (open)` -> `closing` -> `opening`

- `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.

- `associations.full_open_close_output` *(device)*

  Reference to associated full open/close output. Returns `nil` when not associated.

- `associations.partial_open_close_output` *(device)*

  Reference to associated partial open/close output. Returns `nil` when not associated.

- `associations.close_status_sensor` *(device)*

  Reference to associated close status sensor. Returns `nil` when not associated.

- `associations.trigger_sensors` *(array of devices)*

  Reference to associated trigger sensors. Returns empty array when no devices associated.

- `associations.buttons` *(array of devices)*

  Reference to associated buttons. Returns empty array when no devices associated.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"virtual"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"gate"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

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!

  The command is accepted only if:

  - the gate is in `closed` state
  - the gate is in `open` state while the `stopped_state` is `closing`

  Otherwise, the command is ignored.

- `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!

  The command is accepted only if:

  - the gate is in `closed` state

  Otherwise, the command is ignored.

- `close`

  Commands the gate controller to do the close action. This command requires the `close_status_sensor` to be associated!

  The command is accepted only if:

  - the gate is in `open` state without `stopped_state` being `closing`

  Otherwise, the command is ignored.

- `stop`

  Commands the gate controller to stop current move action.

  The command is accepted only if:

  - the gate is in `moving` state
  - the gate is in `closing` state
  - the gate is in `opening` state

  Otherwise, the command is ignored.

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
```

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 (e.g. wall switch impulse, or RC remote output impulse) and trigger `unlock` action

- `button`, this device can be used to trigger `unlock`, `lock` actions

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(string, read-only)*

  Current state of wicket: `locked`, `unlocked`, `closed`, `open`. Open states are 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 s – 45 s.

- `associations.electric_strike_output` *(device)*

  Reference to associated electric strike output. Returns `nil` when not associated.

- `associations.close_status_sensor` *(device)*

  Reference to associated close status sensor. Returns `nil` when not associated.

- `associations.trigger_sensors` *(array of devices)*

  Reference to associated trigger sensors. Returns empty array when no devices associated.

- `associations.buttons` *(array of devices)*

  Reference to associated buttons. Returns empty array when no devices associated.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"virtual"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"wicket"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Commands

- `lock`

  Locks (turns off) electric strike if it's already unlocked.

- `unlock`

  Unlocks (turns on) electric strike if it's already locked.

Examples

Unlock wicket when smoke sensor detects smoke

```lua
if wtp[5]:changedValue("smoke_detected") and wtp[5]:getValue("smoke_detected")
then
  print("Sensor detected smoke!!! Opening gates!")

  virtual[5]:call("unlock")
end
```

Custom devices

This part describes features specific to [custom devices](#).

Lua reference

Property modification is possible via REST API, web app or directly from scripts (excluding adding/removing controls and changing positions) using `virtual` container e.g. `virtual[6]` gives you access to virtual device with **ID 6**. VIRTUAL devices have global scope and they are visible in all executions contexts.

Callbacks

These methods may be called by the system when various events occur, making it possible to react to them. If a custom device does not need to react to a certain callback, it can simply be not defined at all.

- `onEvent`

  Executed when any event in system occurs. Allows user to react to other parts of the system.

  Arguments:

  - `event_object` *(object)* — Current system event. See [Events](#) section for more details about events

- `onInit`

  Executed when custom device added/imported or at central unit startup. Allows user to set up initial state of custom device, for example setup connections to external integrations.

  NOTE: For variants other than `generic` you may need to set custom device `status` property here or later on during runtime.

  Arguments:

  - `reason` *(string)* — reason for executing this callback, one of:
    - `"restart"` - callback called after central unit startup
    - `"add"` - callback called after creating new custom device
    - `"import"` - callback called after importing a custom device

- `onConfigStarted`

  Executed when custom device configuration popup is opened by user in web app. Allows user to set up configuration popup with some dynamic data.

- `onConfigFinished`

  Executed when custom device configuration popup is closed by user in web app. Allows user to set up custom device when configuration is finished.

- `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 e.g. to control multiple elements at once from automations or scenes.

Arguments:

- ◦ `command` *(string)* — Name of command to handle
- ◦ `arg` *(any)* — Argument passed by user to command. Argument can be anything but a table or userdata.

Example:

```lua
function CustomDevice:onCommand(command, arg)
  if command == "my_command_1" then
    print("You called first command without arg.")
  elseif command == "my_command_2_with_arg" then
    print("You called second command with arg: " .. tostring(arg))
  end
end
```

Use in other scripts:

```lua
virtual[5]:call("my_command_1")

virtual[5]:call("my_command_2_with_arg", 77)
```

- `onRebuildInstance`

  Executed after the `CustomDevice` object has been rebuilt. It can be used to initialize some fields:

```lua
function CustomDevice:onRebuildInstance()
    self.rtu = self:getComponent("rtu_client")
end

function CustomDevice:onPress()
    self.rtu:writeCoilAsync(1, true)
end
```

Elements

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 a specific event occurs.

> **Warning**
>
> Element properties modified during custom device run time (by a user or Lua script) are *not saved* in database, unless stated otherwise. Central device restart may restore configuration which was saved in the editor or via REST API.

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 changed by user from the Lua context and the widget/option context.

- `device_selector` - Control for selecting any other device in the system. Accepted device's classes and types can be set. Every setting can be changed by user from Lua context and the widget/option context. Selected device **is saved in the database**.

- `color_picker` - Control for selecting color. Color can be set by user from Lua context and the widget/option context.

- `schedule_selector` - Control for selecting schedule. Schedule can be set by user from Lua context and the widget/option context. Selected shedule **is saved in the database**.

- `time_picker` - Control for selecting time. Time can be set by user from Lua context and the widget/option context.

- `date_picker` - Control for selecting date. Date can be set by user from Lua context and the widget/option context.

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.

Every element

These 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 (e.g. `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

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

- `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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
      Element is visible
  `"hidden_gap"`
      Element is invisible and there's gap in its place.
  `"hidden_adjust"`
      Element is invisible and other element is adjusted.

- `value` *(string)*

  User defined text value. Max 32 characters. Value change will trigger `on_change` callback.

- `text_id` *(number)*

  Text identifier used to reference text from translation database.

- `icon` *(string)*

  User defined icon value. Max 64 characters.

- `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

- `set_value`

  Sets new text value. Value change will trigger `on_change` callback.

  Arguments:

  - `string`

- `set_text_id`

  Sets new text identifier used to reference text from translation database.

  Arguments:

  - `number`

- `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`

- `set_icon`

  Sets new icon value.

  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`

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

- `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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
  > Element is visible
  `"hidden_gap"`
  > Element is invisible and there's gap in its place.
  `"hidden_adjust"`
  > Element is invisible and other element is adjusted.

- `text` *(string)*

  User defined text value. Max 32 characters.

- `text_id` *(number)*

  Text identifier used to reference text from translation database.

- `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

- `press`

  Emits press event and executes callback if attached.

- `set_text`

  Sets new text value.

  Arguments:

  - `string`

- `set_text_id`

  Sets new text identifier used to reference text from translation database.

  Arguments:

  - `number`

- `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`

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

- `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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
  : Element is visible
  `"hidden_gap"`
  : Element is invisible and there's gap in its place.
  `"hidden_adjust"`
  : Element is invisible and other element is adjusted.

- `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

- `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)*

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

- `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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
      Element is visible
  `"hidden_gap"`
      Element is invisible and there's gap in its place.
  `"hidden_adjust"`
      Element is invisible and other element is adjusted.

- `value` *(number)*

  User defined value between 0 (empty bar) and 100 (filled bar). 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

- `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)*

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

- `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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
  : Element is visible
  `"hidden_gap"`
  : Element is invisible and there's gap in its place.
  `"hidden_adjust"`
  : Element is invisible and other element is adjusted.

- `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, read-only)*

  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.

- `label_text` *(string)*

  Text that will be displayed as slider label. Used when slider with label used.

- `label_text_id` *(integer)*

  Text identifier from translation database of text that will be displayed as slider label. Used when slider with label used.

- `unit` *(string)*

  Unit text that will be displayed for slider value. Used when slider with label used.

Commands

- `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.

- `alter`

  Changes bounds and value at once.

  Argument:

  { [1]: number, [2]: number, [3]: number } — A sequence with minimum, new value and maximum.

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)*

Combo box

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

- `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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
      Element is visible
  `"hidden_gap"`
      Element is invisible and there's gap in its place.
  `"hidden_adjust"`
      Element is invisible and other element is adjusted.

- `value` *(string)*

  Selected value. Has to be one of the available values stored in `options`. Value change will trigger `on_change` callback.

- `available_options` *(table, read-only)*

  List of tables (objects) with `label`, `text_id` and `value` properties representing all available options in combo box.

- `options` *(string, read-only)*

  Usage not recommended, see `available_options` as more efficient replacement

  JSON formatted string with list of objects with fields `label`, `text_id` 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

- `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-like table` of size 2 where first field is `label` and second is `value` or size 3 where first field is `label` and second is `value` and third is `text_id`.

- `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`

- `remove_option_by_text_id`

  Removes one of the available option from `options` where `text_id` is equal to passed argument. If currently selected value is removed `value` is changed to empty string and `on_change` callback is triggered.

  Arguments:

  ◦ `number`

- `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`

Device selector

This element represents a control for selecting any device in the system via Lua context or the widget / options. It is possible to attach a Lua callback which will be executed when selected device changes. Accepted device's classes and types can be changed from the Lua.

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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  "visible"
    Element is visible
  "hidden_gap"
    Element is invisible and there's gap in its place.
  "hidden_adjust"
    Element is invisible and other element is adjusted.

- `accepted_classes` *(table, read-only)*

  List of accepted classes for selected device. Empty table means all classes are accepted.

- `accepted_types` *(table, read-only)*

  List of accepted types for selected device. Empty table means all types are accepted.

- `allow_multiple` *(boolean, read-only)*

  Indicates if device selector allows to select multiple devices.

- `associations.selected` *(device)*

  If `allow_multiple` is set to `false` it is reference to device which is selected. Returns `nil` when no device is selected.

  If `allow_multiple` is set to `true` it is table of references to devices which are selected.

This association **is saved in the database**, thus is persistent across central device reboots.

- `on_change` *(string, read-only)*

Name of method (function) from CustomDevice Lua code which will be executed on selected device change. Should not contain `CustomDevice:` prefix, only name.

Commands

- `select_device`

Sets new selected device. Device change will trigger `on_change` callback. Device has to be one of class stored in `accepted_classes` and type stored in `accepted_types`. If `allow_multiple` is set to `false` it replaces currently selected device, otherwise it adds passed device to selected devices table.

Arguments:

- `device`

- `select_devices`

Sets new selected devices. It replaces currently selected devices with passed ones. Devices change will trigger `on_change` callback. Devices has to be one of class stored in `accepted_classes` and type stored in `accepted_types`. If `allow_multiple` is set to `false` it allows to set only one device.

Arguments:

- `array-like table of devices`

- `set_accepted_classes`

Sets accepted device's classes. Values has to be unique. Empty table means all classes are accepted.

Arguments:

- `array-like table`

- `set_accepted_types`

Sets accepted device's types. Values has to be unique. Empty table means all types are accepted.

Arguments:

- `array-like table`

Lua Callback signature

- `on_change`

Executed on selected device change. Takes string with JSON formatted object with fields `class` and `id` of selected device and reference to element as arguments.

Arguments:

- `string`
- `element_reference`

Color picker

This element represents a control for selecting a color via Lua context or the widget / options. It is possible to attach a Lua callback which will be executed when color or state changed.

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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
  > Element is visible
  `"hidden_gap"`
  > Element is invisible and there's gap in its place.
  `"hidden_adjust"`
  > Element is invisible and other element is adjusted.

- `state` *(boolean)*

  Indicates the state of the build-in switcher.

- `available_color_modes` *(table)*

  List of available color modes. Possible modes are: `temperature`, `rgb`, `gradient`.

- `gradient_size_maximum` *(integer)*

  Maximum amount of colors which can be applied to gradient. Available only when `gradient` is in `available_color_modes`.

- `color_mode` *(string, read-only)*

  Currently selected color mode. One of: `temperature`, `rgb`, `gradient`.

- `brightness` *(integer)*

Current brightness value.

Unit: 1 %

Range: 1 % – 100 %

- `temperature` *(integer)*

  Current white color temperature value. Available only when `color_mode` is *temperature*

  Unit: 1 K

  Range: 1000 K – 40 000 K.

- `color` *(string)*

  Current HTML/Hex RGB color. Available only when `color_mode` is `rgb`.

- `gradient` *(table)*

  List of HTML/Hex RGB colors. Length of the list cannot be higher than `gradient_size_maximum`. Available only when `color_mode` is `gradient`.

- `on_state_change` *(string, read-only)*

  Name of method (function) from CustomDevice Lua code which will be executed on `state` change. Should not contain CustomDevice: prefix, only name.

- `on_brightness_change` *(string, read-only)*

  Name of method (function) from CustomDevice Lua code which will be executed on `brightness` change. Should not contain CustomDevice: prefix, only name.

- `on_color_change` *(string, read-only)*

  Name of method (function) from CustomDevice Lua code which will be executed on any color related property change. Should not contain CustomDevice: prefix, only name.

Commands

- `toggle`

  Sets opposite state. State change will trigger `on_state_change` callback.

- `set_state`

  Sets switcher state. State change will trigger `on_state_change` callback.

  Arguments:

  - `boolean`

- `set_brightness`

  Sets current brightness level, triggering `on_color_change` callback.

  Unit: 1 %

  Range: 1 % – 100 %

  Arguments:

- ○ `integer`

- `set_temperature`

  Sets current white color temperature. Available only when `temperature` is in `available_color_modes`. Temperature change will trigger `on_color_change` callback.

  Unit: 1 K

  Range: 1000 K – 40 000 K

  Arguments:

  - ○ `integer`

- `set_color`

  Sets current RGB color in HTML/Hex format. Available only when `"rgb"` is in `available_color_modes`. Color change will trigger `on_color_change` callback.

  Arguments:

  - ○ `string`

- `set_gradient`

  Sets current gradient color as list of HTML/Hex colors. Available only when `gradient` is in `available_color_modes`. Gradient change will trigger `on_color_change` callback.

  Arguments:

  - ○ `array-like table`

Lua Callback signature

- `on_state_change`

  Executed on `state` change. Takes boolean and reference to element as arguments.

  Arguments:

  - ○ `boolean`
  - ○ `element_reference`

- `on_brightness_change`

  Executed on `brightness` change. Takes integer and reference to element as arguments.

  Arguments:

  - ○ `integer`
  - ○ `element_reference`

- `on_color_change`

Executed when any color related property changes. Takes object with `color_mode` and one of: (depends on current `color_mode`) `temperature`, `color` or `gradient` property values and reference to element as arguments.

Arguments:

- `object-like table`
- `element_reference`

Schedule selector

This element represents a control for selecting any schedule in the system via Lua context or the widget / options. It is possible to attach a Lua callback which will be executed when selected schedule changes. Accepted schedule's types can be changed from the Lua.

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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
    Element is visible
  `"hidden_gap"`
    Element is invisible and there's gap in its place.
  `"hidden_adjust"`
    Element is invisible and other element is adjusted.

- `accepted_types` *(table, read-only)*

  List of accepted types for selected schedule. Empty table means all types are accepted.

- `schedule_id` *(number)*

  Selected schedule ID. Returns `0` when no device is selected.

  This value **is saved in the database**, thus is persistent across central device reboots.

- `on_change` *(string, read-only)*

  Name of method (function) from CustomDevice Lua code which will be executed on selected schedule change. Should not contain `CustomDevice:` prefix, only name.

Commands

- `select_schedule_id`

Sets new selected schedule ID. Schedule ID change will trigger `on_change` callback. Schedule has to be one of type stored in `accepted_types`.

Arguments:

- `number`

- `set_accepted_types`

Sets accepted schedule's types. Values has to be unique. Empty table means all types are accepted.

Arguments:

- `array-like table`

Lua Callback signature

- `on_change`

Executed on selected schedule ID change. Takes schedule ID and reference to element as arguments.

Arguments:

- `number`
- `element_reference`

Time picker

This element represents a control for selecting time via Lua context or the widget / options. It is possible to attach a Lua callback which will be executed when selected time changes.

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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
  : Element is visible
  `"hidden_gap"`
  : Element is invisible and there's gap in its place.
  `"hidden_adjust"`
  : Element is invisible and other element is adjusted.

- `minimum` *(number)*

  Minimum time count to set.

- `maximum` *(number)*

  Maximum time count to set.

- `units` *(array-like table, read-only)*

  Available time units. Can be any combination of values: `days`, `hours`, `minutes`, `seconds`.

- `time` *(object-like table)*

  Object-like table with values of all available units. E.g if `units` are `[days,hours]` it will have `days` and `hours` keys.

- `time.count` *(number)*

Time in lowest available unit. E.g. when units are `[hours, minutes]` `time.count` will be in minutes.

Has to be in range of `minimum` - `maximum`.

- `time.days` *(number)*

Selected day. Available only when `days` is present in `units`.

- `time.hours` *(number)*

Selected hour. Available only when `hours` is present in `units`.

- `time.minutes` *(number)*

Selected minutes. Available only when `minutes` is present in `units`.

- `time.seconds` *(number)*

Selected seconds. Available only when `seconds` is present in `units`.

- `on_change` *(string, read-only)*

Name of method (function) from CustomDevice Lua code which will be executed on selected time change. Should not contain `CustomDevice:` prefix, only name.

Commands

- `set_time`

Sets new selected time. Time change will trigger `on_change` callback. Time shall be passed as array-like table with elements holding values of every available unit starting from highest one.

Arguments:

- `array-like table`

- `set_count`

Sets new selected time count. Time change will trigger `on_change` callback.

Arguments:

- `number`

Lua Callback signature

- `on_change`

Executed on selected time change. Takes object-like table with properties `days`, `hours`, `minutes`, `seconds` (only units available in `units` are present) and `count` and reference to element as arguments.

Arguments:

- `object-like table`

- `element_reference`

Date picker

This element represents a control for selecting date via Lua context or the widget / options. It is possible to attach a Lua callback which will be executed when selected date changes.

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.

- `visibility` *(string)*

  Element visibility. Can be one of these values:

  `"visible"`
  : Element is visible
  `"hidden_gap"`
  : Element is invisible and there's gap in its place.
  `"hidden_adjust"`
  : Element is invisible and other element is adjusted.

- `date_type` *(string, read-only)*

  Type of stored date. Can be one of values `single` or `range`.

- `date` *(object-like table)*

  Currently selected date. It is object with fields: `year`. `month` and `day`. Available only when `date_type` is `single`.

- `range` *(object-like table)*

  Currently selected range. It is object with fields `from`, `to`, which are objects with fields: `year`. `month` and `day` Available only when `date_type` is `range`.

- `on_change` *(string, read-only)*

  Name of method (function) from CustomDevice Lua code which will be executed on selected date change. Should not contain `CustomDevice:` prefix, only name.

Commands

- `set_date`

  Sets new selected date. Date change will trigger `on_change` callback. Date shall be passed as object-like table with keys `year`, `month` and `day`. Available only when `date_type` is `single`.

  Arguments:

  - `object-like table`

- `set_range`

  Sets new selected date range. Range change will trigger `on_change` callback. Range shall be passed as object-like table with keys `from` and `to` which are object-like tables with keys `year`, `month` and `day`. Available only when `date_type` is `range`.

  Arguments:

  - `object-like table`

Lua Callback signature

- `on_change`

  Executed on selected date change. Takes object-like table with properties `year`, `month` and `day` if `date_type` is `single` or object-like table with properties `from` and `to` which are objects with properties `year`, `month` and `day` if `date_type` is `range` and reference to element as arguments.

  Arguments:

  - `object-like table`
  - `element_reference`

Components

Components are connection clients, variables and timers that are available only within given custom device. They are not available from global scope. They have the same functionality as they globally accessed equivalents.

You can access components in Lua code using `:getComponent()` method. For example: `self:getComponent("variable_0")`.

Following component types are available:

- `http_client` - handles HTTP and HTTPS connections. For detailed documentation check out [HTTP client](#).

- `mqtt_client` - handles MQTT connections. For detailed documentation check out [MQTT client](#).

- `ws_client` - handles Websocket connections. For detailed documentation check out [Websocket client](#).

- `modbus_rtu_client` - handles Modbus RTU connections. For detailed documentation check out [Modbus client](#).

- `modbus_tcp_client` - handles Modbus TCP connections. For detailed documentation check out [Modbus client](#).

- `variable_boolean` - allows holding boolean values. For detailed documentation check out [variables](#).

- `variable_integer` - allows holding integer values. For detailed documentation check out [variables](#).

- `variable_string` - allows holding string values. For detailed documentation check out [variables](#).

- `variable_table` - allows holding simple Lua table values. For detailed documentation check out [variables](#).

- `timer` - allows handling delays or time counting. For detailed documentation check out [timers](#).

> **Note**
>
> Variable components' values are preserved when custom device is updated via marketplace.

Variants

To enable more flexible integrations custom device can be defined with specific `variant` type. That way user can add their own integration with device from another system and defining `variant` enable it to integrate with Sinum internal algorithms. For example, user can add Custom device with variant `temperature_sensor` and later associate this custom temperature sensor with Sinum thermostat or a heat pump manager. User needs to set all variant properties and define callbacks for proper integration.

Available variant types are: `"generic"`, `"battery"`, `"common_dhw_main"`, `"energy_meter"`, `"heat_pump"`, `"inverter"`, `"relay"`, `"temperature_regulator"`, `"temperature_sensor"`, `"two_state_input_sensor"`, `"humidity_sensor"`, `"analog_input"`, `"car_charger"`.

Direct access to variant device properties is not allowed — `setVariantDeviceValue` and `getVariantDeviceValue` methods have to be used. An attempt at retrieving a nonexistent object property, or setting value with a wrong type will result in a script error.

Most variants **require some callbacks** in custom device Lua code. They are required by internal Sinum algorithms and used for communication from Sinum to integrated external device.

> **Warning**
>
> Callbacks will be called when parameter changes from Lua code too. In order to prevent that user needs to call `setVariantDeviceValue` with `stop_propagation` argument set to `true`.

Statistics are not stored by default for custom device. If user wants to store devices' statistics, it has to be done manually in Lua script using `statistics:addPoint` method. For more information see chapter about [statistics](#).

When custom device has different `variant` than `"generic"` its `status` set to `"unknown"` by default. Integrators *ignore* devices which are not `"online"`. User needs to set statuses for custom device based on e.g. connection with integrated device. When custom device is associated with other devices or Sinum features like Energy center setting this `status` to `"online"` is essential for proper work. It is recommended to set this status in `onInit` callback of custom device and later periodically check and set this status in `onEvent`. That way is recommended because some features for example statistics chart in Energy center will not be set properly without `online` status. See example of setting `status` in **Setting custom device status** example below.

Adding a variant can add some commands to custom device. They are called as regular commands for the device using `call` method. See the specific variant section to see the commands that are supported.

Battery

Battery device variant is set when `variant` is `"battery"`.

Associations

Device can be associated with **Energy center**. When associated with Energy center, Battery will be used to

- display power flow to and from battery in *flow monitor* widget
- display charge power and stored/discharged today energy in *energy storage* widget

Parameters values must be set from Lua code for proper widgets work (see parameters description below).

Status of device must be set to `"online"` for proper widgets work. Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available properties

- `charge_power` *(number)*

  Current charge power. Used to display power flow power in Flow monitor and charge power in Energy storage widget.

  When positive — battery is charging. When negative — battery is discharging.

  Unit: 1 mW
  Range: −2 147 483 647 mW – 2 147 483 647 mW

- `soc` *(number)*

  Current state of charge. Used to display state of charge in Energy storage widget.

  Unit: 1 %

  Range: 1 % – 100 %

- `energy_charged_today` *(number)*

  Amount of energy charged to the battery today. Used to display energy charged today in Energy storage widget.

  Unit: 1 Wh

  Range: 0 Wh – 4 294 967 295 Wh

- `energy_discharged_today` *(number)*

  Amount of energy consumed from the battery today. Used to display energy discharged today in Energy storage widget.

  Unit: 1 Wh

  Range: 0 Wh – 4 294 967 295 Wh

- `battery_charge_support` *(boolean)*

  Indicates if battery charge commands should be supported. Adds or removes the label `battery_charge_support` for the module. This parameter is just informational to keep compatibility with other battery device types.

Example

```lua
-- Battery integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:onEvent(event)
    local battery = self:getClient()

    if dateTime:changed() then
        -- read registers 10 — 13
        battery:readHoldingRegistersAsync(10, 4)
    end

    battery:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    battery:onRegisterAsyncRead(function (kind, address, values)
        -- the device responded, so it is online
        self:setValue("status", "online")

        -- convert received data
        local charge_power = asInt16(values[1])
        local soc = values[2]
        local charged_today = values[3]
        local discharged_today = values[4]

        self:setVariantDeviceValue("charge_power", charge_power, true)
        self:setVariantDeviceValue("soc", soc, true)
        self:setVariantDeviceValue("energy_charged_today", charged_today, true)
        self:setVariantDeviceValue("energy_discharged_today", discharged_today,
            true)
    end)
end
```

Domestic hot water

Domestic Hot Water device variant is set when `variant` is `"common_dhw_main"`.

Associations

Device can be associated with **Heat pump manager**. When associated with Heat pump manager device will be turned on and off based on difference between `current_temperature` and `target_temperature`. Device will be controled by changing `heat_demand` parameter by Heat pump manager. Heat pump manager will also change `target_temperature` and `hysteresis` to values set in Heat pump manager.

Parameters values must be set from Lua code for proper work (see parameters description below). Callbacks must be added to Lua code for proper work. (see required callbacks below)

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available properties

- `current_temperature` *(number)*

  Current domestic hot water temperature. When below `target_temperature` heat pump manager will set work demand.

  Unit: 0.1 °C

  Range: −400 – 1500 (−40.0 °C – 150.0 °C)

- `target_temperature` *(number)*

  Target hot water temperature.

  Unit: 0.1 °C

  Range: 0 – 1000 (0.0 °C – 100.0 °C)

- `hysteresis` *(number)*

  Target hot water temperature hysteresis.

  Unit: 0.1 K

  Range: 0 – 200 (0.0 K – 20.0 K)

- `heat_demand` *(boolean)*

  Domestic Hot Water heat demand. State of domestic hot water device work.

Required callbacks

- `setTargetTemperature(temperature)`

  Called when target temperature is changed. Should set target temperature in integrated external device. Parameter `target_temperature` is changed prior to callback call.

Argument:

number — target temperature in 0.1 °C

- `setHysteresis(value)`

Called when hysteresis is changed. Should set hysteresis in integrated external device. Parameter `hysteresis` is changed prior to callback call.

Argument:

number — hysteresis value in 0.1 °C

- `setHeatDemand(value)`

Called when heat demand is changed. Should enable/disable heat demand in integrated external device based on argument value. Parameter `heat_demand` is changed prior to callback call.

Argument:

boolean — heat demand

Commands

- `set_target_temperature`

Set target temperature to the desired value. Command will call callback `setTargetTemperature`.

Argument:

target temperature in 0.1°C *(integer)*

Example

```lua
-- DHW integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:setTargetTemperature(value)
    -- send value to Modbus device
    self:getClient():writeHoldingRegisterAsync(12, value)
end

function CustomDevice:setHysteresis(value)
    -- send value to Modbus device
    self:getClient():writeHoldingRegisterAsync(13, value)
end

function CustomDevice:setHeatDemand(demand)
    -- send value to Modbus device
    self:getClient():writeHoldingRegisterAsync(10, demand and 1 or 0)
        -- cast boolean to 0/1
end

function CustomDevice:onEvent(event)
    local dhw = self:getClient()
```

```lua
    if dateTime:changed() then
        -- read registers 10 – 13
        dhw:readHoldingRegistersAsync(10, 4)
    end

    dhw:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    dhw:onRegisterAsyncWrite(function ()
        -- the device acknowledged transfer, so it is online
        self:setValue("status", "online")
    end)

    dhw:onRegisterAsyncRead(function (_, address, values)
        -- the device responded, so it is online
        self:setValue("status", "online")

        -- convert received data
        local current_temperature = values[2]
        local target_temperature = values[3]
        local hysteresis = values[4]
        local heat_demand = values[1] == 1 -- cast 0/1 to boolean

        -- set with stop_propagation flag set to true
        -- to avoid callbacks call and sending it back
        self:setVariantDeviceValue("current_temperature", current_temperature,
            true)
        self:setVariantDeviceValue("target_temperature", target_temperature,
            true)
        self:setVariantDeviceValue("hysteresis", hysteresis, true)
        self:setVariantDeviceValue("heat_demand", heat_demand, true)
    end)
end
```

Car Charger

Car Charger device variant is set when `variant` is `"car_charger"`.

Associations

Device will be automatically displayed in **Energy center** and will be used to

- display power in detailed view of *flow monitor* widget
- display energy consumption in *energy consumption* widget

Parameters values must be set from Lua code for proper widgets work (see parameters description below)

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available properties

- `charge_power` *(positive number)*

  Instantaneous power used by charger to charge the EV.

  Unit: 1 mW

  Range: 0 mW – 2 147 483 647 mW

- `energy_charged_total` *(number)*

  Total energy charged during device lifetime. Used to show energy consumption for car chargers in Energy consumption widget. Today's energy is calculated based on changes of this parameter.

  Unit: 1 Wh

  Range: 0 Wh – 4 294 967 295 Wh

Example

```lua
-- Car charger integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:onEvent(event)
    local meter = self:getClient()

    if dateTime:changed() then
        -- read registers 10 — 11
        meter:readHoldingRegistersAsync(10, 2)
    end

    meter:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
```

```lua
    end)

    meter:onRegisterAsyncRead(function (_, _, values)
        -- the device responded, so it is online
        self:setValue("status", "online")

        local charge_power = values[1]
        local energy_charged_total = values[2]

        self:setVariantDeviceValue("charge_power", charge_power, true)
        self:setVariantDeviceValue("energy_charged_total", energy_charged_total,
            true)
    end)
end
```

Energy meter

Energy meter device variant is set when `variant` is `"energy_meter"`.

Associations

Device can be associated with **Energy center**. When associated with Energy center, Energy meter will be used to

- display power flow to and from grid in *flow monitor*
- display power in detailed view of *flow monitor* (if `uses_energy_of_building == true`)
- display energy consumption in *energy consumption* widget, for electrical sockets or household (based on `uses_energy_of_building` parameter)

When energy meter has `uses_energy_of_building` set to `true` it will automatically be included in Energy center widgets. Only main energy meter with `uses_energy_of_building` set to `false` need to be added to Energy center associations.

Parameters values must be set from Lua code for proper widgets work (see parameters description below)

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available properties

- `total_active_power` *(number)*

  Active power measured by energy meter. Used for power flow to or from grid in Flow monitor. When positive number - power is received from the grid. And then calculated for flow based on battery measurements and pv inverter. When negative number - power is transfered to the grid. Active power is also used to calculate energy used for the household in Energy consumption widget. Energy is calculated based on `total_active_power` changes through the time.

  Unit: 1 mW

  Range: $-2\,147\,483\,647$ mW – $2\,147\,483\,647$ mW

- `energy_sum_total` *(number)*

  Total energy consumed of all phases lifetime. Used to show energy consumption for electrical sockets in Energy consumption widget. Today's energy is calculated based on changes of this parameter. Parameter is only used for displaying electrical sockets' energy, so only if `uses_energy_of_building` is `true`.

  Unit: 1 Wh

  Range: 0 Wh – $4\,294\,967\,295$ Wh

- `uses_energy_of_building` *(boolean)*

  Indicates if energy meter is used to measure energy of device inside the building. (Not main energy meter). When set to `true` it is automatically included in Energy center widgets calculations (Flow monitor and Energy consumption). Should be set to `false` if

it is used as main energy meter. In this case to be included in Energy center widgets calculations user needs to add this energy meter as **Main energy meter** in energy center.

Example

```lua
-- Energy meter integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:onEvent(event)
    local meter = self:getClient()

    if dateTime:changed() then
        -- read registers 10 - 11
        meter:readHoldingRegistersAsync(10, 2)
    end

    meter:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    meter:onRegisterAsyncRead(function (_, _, values)
        -- the device responded, so it is online
        self:setValue("status", "online")

        local total_active_power = values[1]
        local energy_sum_total = values[2]

        self:setVariantDeviceValue("total_active_power", total_active_power,
            true)
        self:setVariantDeviceValue("energy_sum_total", energy_sum_total, true)

        -- set value to false always, this is main energy meter
        self:setVariantDeviceValue("uses_energy_of_building", false, true)
    end)
end
```

Heat pump

Heat Pump device variant is set when `variant` is `heat_pump`.

Associations

Device can be associated with **Heat pump manager** and **Thermostat output group (Virtual Contact)**. When associated with Heat pump manager or Virtual Contact device will be turned on and off based on their algorithms

- for Heat pump manager: difference between current and target
- for Virtual Contact: based on associated thermostats state

Device will be controled by changing `thermal_demand` parameter.

Parameters values must be set from Lua code for proper work. (see parameters description below) Callbacks must be added to Lua code for proper work. (see required callbacks below)

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available properties

- `enabled` *(boolean)*

  Indicates if heat pump is enabled. When associated with Heat pump manger or Virtual contact, `thermal_demand` will be set to `none` if this parameter is set to `false`.

  Otherwise, `thermal_demand` will be changed based on these devices algorithms.

- `thermal_demand` *(string)*

  Heat pump thermal demand. Available values: `none`, `heat`, `cool`. Main parameter used to control heat pump.

  - `none` - pump is not working
  - `heat` - pump is working and heating
  - `cool` - pump is working and cooling

- `electric_heater_active` *(boolean)*

  Indicates electric heater activation state in heat pump. Used when associated with Heat pump manger to inform used if electric heater is active.

- `target_temperature_indoor` *(integer)*

  Room target temperature. When associated with Heat pump manager it will be get as target temperature in manager or changed by manager if user changes target temperature via Heat pump manager (change will be indicated via callback `setTargetTemperatureIndoor`).

  Unit: 0.1 °C

  Range: 100 – 350 (10.0 °C – 35.0 °C)

Required callbacks

- `setThermalDemand(demand)`

  Called when thermal demand for heat pump is changed by algorithms in Heat pump manager or Virtual Contact. Should set integrated heat pump in given state to fullfill the requirement

  - `none` : heat pump should turn off heating and cooling
  - `heat` : heat pump should turn on heating
  - `cool` : heat pump should turn on cooling

  Argument:

  Thermal demand for device, one of: `none`, `heat`, `cool` *(string)*

- `setTargetTemperatureIndoor(temperature)`

  Called when target temperature is changed. Should set target temperature in integrated external device. Parameter `target_temperature_indoor` is changed prior to callback call.

  Argument:

  number — target temperature in 0.1 °C

Example

```lua
-- Heat Pump integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:setThermalDemand(value)
    local heatpump = self:getClient()

    if value == "heat" then
        -- send value to Modbus device
        heatpump:writeHoldingRegisterAsync(100, 1)
    elseif value == "cool" then
        -- send value to Modbus device
        heatpump:writeHoldingRegisterAsync(100, 2)
    else
        -- send value to Modbus device
        heatpump:writeHoldingRegisterAsync(100, 0)
    end
end

function CustomDevice:onEvent(event)
    local heatpump = self:getClient()

    if dateTime:changed() then
        -- read registers 100 – 102
        heatpump:readHoldingRegistersAsync(100, 3)
    end

    heatpump:onAsyncRequestFailure(function (_, err)
        if err == 'TIMEOUT' then
            -- the device is not responding
```

```lua
            self:setValue('status', 'offline')
        end
    end)

    heatpump:onRegisterAsyncWrite(function ()
        -- the device acknowledged transfer, so it is online
        self:setValue("status", "online")
    end)

    heatpump:onRegisterAsyncRead(function (_, _, values)
        -- the device responded, so it is online
        self:setValue("status", "online")

        local enabled = values[2] == 1
        local thermal_demand = values[1]
        local electric_heater_active = values[3] == 1

        -- set with stop_propagation flag set to true,
        -- to avoid callbacks call and sending it back
        self:setVariantDeviceValue("enabled", enabled, true)
        self:setVariantDeviceValue("electric_heater_active",
            electric_heater_active, true)

        if thermal_demand == 1 then
            self:setVariantDeviceValue("thermal_demand", "heat", true)
        elseif thermal_demand == 2 then
            self:setVariantDeviceValue("thermal_demand", "cool", true)
        else
            self:setVariantDeviceValue("thermal_demand", "none", true)
        end
    end)
end
```

Inverter

Inverter device variant is set when `variant` is `inverter`.

Associations

Device can be associated with **Energy center**. When associated with Energy center, inverter will be used to

- display power flow to grid, battery and house in Flow monitor
- display energy distribution in Energy consumption widget, energy used for auto-consumption, energy storage and grid.

Parameters values must be set from Lua code for proper widgets work. (see parameters description below)

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available properties

- `pv_total_active_power` *(number)*

  Current total power produced by all photovoltaic panels. Parameter will be used to calculate power flow to grid, battery and house in Flow monitor.

  Unit: 1 mW

  Range: 0 mW – 4 294 967 295 mW

- `energy_produced_total` *(number)*

  Total amount of energy produced by PV over a lifetime. Parameter will be used to display energy distibution in Energy consumption widget. Energy used for auto-consumption, energy storage and grid. Distribution is calculated using this parameter and ratio based on flow monitor power flow.

  Unit: 1 Wh

  Range: 0 Wh – 4 294 967 295 Wh

- `power_to_grid` *(number)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: 1 mW

  Range: −2 147 483 647 mW – 2 147 483 647 mW

Example

```lua
-- Inverter integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end
```

```lua
function CustomDevice:onEvent(event)
    local inverter = self:getClient()

    if dateTime:changed() then
        -- read registers 11 – 13
        inverter:readHoldingRegistersAsync(11, 3)
    end

    inverter:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    inverter:onRegisterAsyncRead(function (_, _, values)
        -- the device responded, so it is online
        self:setValue("status", "online")

        local pv_total_active_power = values[1]
        local power_to_grid = asInt16(values[2])
        local energy_produced_total = values[3]

        self:setVariantDeviceValue("pv_total_active_power",
            pv_total_active_power, true)
        self:setVariantDeviceValue("power_to_grid", power_to_grid, true)
        self:setVariantDeviceValue("energy_produced_total",
            energy_produced_total, true)
    end)
end
```

Temperature sensor

Temperature sensor device variant is set when `variant` is `temperature_sensor`.

Associations

Device can be associated with **Thermostat** and **Heat pump manager**. Can be used as outdoor sensor for **Weather** and added to sensor screen in **SBus Control Panel** or **SLink Control Panel**.

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available Properties

- `temperature` *(number)*

    Measured temperature sensor value.

    Unit: 0.1 °C

    Range: −1000 – 3000 (−100.0 °C – 300.0 °C)

Example

```lua
-- Temperature sensor integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:onEvent(event)
    local sensor = self:getClient()

    if dateTime:changed() then
        -- read register 0801
        sensor:readInputRegisterAsync(801)
    end

    sensor:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    sensor:onRegisterAsyncRead(function (_, _, value)
        -- the device responded, so it is online
        self:setValue("status", "online")

        local temperature = asInt16(value) * 10
        self:setVariantDeviceValue("temperature", temperature, true)
    end)
 end
```

Relay

Relay device variant is set when `variant` is `relay`.

Associations

Device can be associated with **Thermostat**, **Thermostat output group (Virtual Contact)**, **Gate**, **Wicket** and **Relay integrator**. Device can be added to bistable button in **SBus Control Panel** or **SLink Control Panel**.

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available properties

- `state` *(boolean)*

  State of the output. On/Off.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `timeout` *(number)*

  Protection functionality, remote device should be turned off when no communication for time set in this parameter. Remote device should have that functionality if user wants to use this, it is of device responsibility to turn off after this time. When user wants to use this feature, custom device should send state to remote device periodically in `onEvent` callback, for example every minute.

  Unit: 1 min
  Range: 5 min – 60 min

- `timeout_enabled` *(boolean)*

  Parameter that indicates if timeout functionality is enabled.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `time_since_state_change` *(number, read-only)*

  Time since last relay state change.

  Unit: 1 s.

Required callbacks

- `setState(state)`

  Called when relay state is changed. Should turn on/off integrated device based on argument.

  Argument:

boolean — desired state of device

- `setTimeout(timeout)`

Called when relay timeout value is changed. If device has timeout functionality should set timeout for required minutes from argument.

Argument:

number — desired timeout value in minutes

- `setTimeoutEnabled(enabled)`

Called when relay timeout function state is changed. If device has timeout functionality should enable it and set timeout for required minutes from `timeout` parameter.

Argument:

boolean — function should be enabled

Commands

- `turn_on`

Change the relay state to on. Command will call the `setState` callback.

- `turn_off`

Change the relay state to off. Command will call the `setState` callback.

Example

```lua
-- Relay integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:setState(state)
    -- send value to modbus device
    self:getClient():writeCoilAsync(141, state)
end

function CustomDevice:onEvent(event)
    local relay = self:getClient()

    if dateTime:changed() then
        -- read register 141
        relay:readCoilAsync(141)
    end

    relay:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    relay:onRegisterAsyncWrite(function ()
        -- the device acknowledged transfer, so it is online
        self:setValue("status", "online")
```

```lua
    end)

    relay:onRegisterAsyncRead(function (_, _, value)
        -- the device responded, so it is online
        self:setValue("status", "online")

        -- set with stop_propagation flag set to true,
        -- to avoid callbacks call and sending it back
        local state = values
        self:setVariantDeviceValue("state", state, true)
    end)
end
```

Temperature regulator

Temperature regulator device variant is set when `variant` is `temperature_regulator`.

Associations

Device can be associated with **Thermostat** or **Heat pump manager**. When associated, it synchronizes target temperature and target temperature modes between the device and **Thermostat** or **Heat pump manager**.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Parameters values must be set from Lua code for proper work. (see parameters description below) Callbacks must by added to Lua code for proper work. (see required callbacks below)

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available properties

- `target_temperature` *(number)*

  Desired setpoint temperature.

  Unit: 0.1 °C

  Min: from parameter `target_temperature_minimum`
  Max: from parameter `target_temperature_maximum`

- `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 doesn't 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: 1 min.

- `target_temperature_minimum` *(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: 0.1 °C

Min: 50 (5.0 °C)
Max: from parameter `target_temperature_maximum`

- `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: 0.1 °C

Min: from parameter `target_temperature_minimum` Max: 350

- `target_temperature_reached` *(boolean)*

Controls device's algorithm state indicator (available on some regulators). e.g. LED Diode. May be controlled by external algorithms or devices such as thermostat or heat pump manager (when thermostat is active, indicator should blink)

- `system_mode` *(string)*

Indicates external system work mode.

May only be changed if device is not assigned to thermostat or heat pump manager (label `managed_by_thermostat` and `managed_by_heat_pump_manager` not present).

Available values: `"off"`, `"heating"`, `"cooling"`.

Default: `"heating"`

- `confirm_time_mode` *(boolean, read-only)*

Indicates if device should ask for timed mode target temperature set when changing target temperature. Parameter controlled by Virtual Thermostat and Heat pump manager.

Required callbacks

- `setTargetTemperature(temperature)`

Called when target temperature is changed. Should set target temperature in integrated external device. Parameter `target_temperature` is changed prior to callback call.

Argument:

number — target temperature in 0.1 °C

- `setTargetTemperatureMinimum(temperature)`

Called when target temperature minimum is changed. Should set target temperature minimum in integrated external device. Parameter `target_temperature_minimum` is changed prior to callback call.

Argument:

number — target temperature minimum in 0.1 °C

- `setTargetTemperatureMaximum(temperature)`

  Called when target temperature maximum is changed. Should set target temperature maximum in integrated external device. Parameter `target_temperature_maximum` is changed prior to callback call.

  Argument:

  number — target temperature maximum in 0.1 °C

- `setTargetTemperatureReached(value)`

  Called when target temperature reached indicator is changed. Should set indicator in integrated external device based on argument value. Parameter `target_temperature_reached` is changed prior to callback call.

  Argument:

  boolean — target temperature reached

- `setSystemMode(mode)`

  Called when system mode is changed. Should set system mode in integrated external device based on argument value. Parameter `system_mode` is changed prior to callback call.

  Argument:

  string — system mode, one of: `"off"`, `"heating"`, `"cooling"`

- `setTargetTemperatureMode(value)`

  Called when target temperature mode is changed. Should set mode in integrated external device based on argument value. Parameter `target_temperature_mode.current` is changed prior to callback call.

  Argument:

  string — target temperature mode, one of: `constant`, `schedule`, `time_limited`

- `setTargetTemperatureRemainingTime(value)`

  Called when target temperature remaining time is changed. Should set remaining time in integrated external device. Parameter `target_temperature_mode.remaining_time` is changed prior to callback call.

  Argument:

  number — remaining time in minutes

- `setConfirmTimeMode(value)`

  Called when confirm time mode indicator is changed. Should set indicator in integrated external device based on argument value. Parameter `confirm_time_mode` is changed prior to callback call.

  Argument:

  boolean — confirm time mode indicator

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`.

  Command will call the `setTargetTemperature` callback. Depending on the target temperature mode command can call `setTargetTemperatureMode` mode.

  Argument:

  number — target temperature in 0.1°C

- `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.

  Command will call the `setTargetTemperature` callback. Depending on the target temperature mode command can call `setTargetTemperatureMode` mode.

  > **Note**
  >
  > Cannot be executed when regulator is not associated with a thermostat.

  Argument:

  number — target temperature in 0.1°C

- `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.

  Command will call `setTargetTemperature` and `setTargetTemperatureRemainingTime` callbacks if values changed. Depending on the target temperature mode command can call `setTargetTemperatureMode` mode.

  > **Note**
  >
  > Cannot be executed when regulator is not associated with a 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.

  Command will call `setTargetTemperatureMode` and `setTargetTemperature` callbacks if values changed.

  > **Note**
  >
  > Cannot be executed when regulator is not associated with a thermostat.

Example

```lua
-- Temperature regulator integrated using an HTTP client
-- In order to work, HTTP client component must be added with name
-- "http_client"
function CustomDevice:getClient()
    return self:getComponent("http_client")
end

function CustomDevice:setTargetTemperature(value)
    local body = {
        target_temperature = value
    }
    self:getClient():POST('/update'):body(JSON:encode(body)):send()
end

function CustomDevice:setTargetTemperatureMinimum(value)
    local body = {
        target_temperature_minimum = value
    }
    self:getClient():POST('/update'):body(JSON:encode(body)):send()
end

function CustomDevice:setTargetTemperatureMaximum(value)
    local body = {
        target_temperature_maximum = value
    }
    self:getClient():POST('/update'):body(JSON:encode(body)):send()
end

function CustomDevice:setTargetTemperatureReached(value)
    local body = {
        target_temperature_reached = value
    }
    self:getClient():POST('/update'):body(JSON:encode(body)):send()
end

function CustomDevice:setSystemMode(value)
    local body = {
        system_mode = value
    }
    self:getClient():POST('/update'):body(JSON:encode(body)):send()
end

function CustomDevice:setTargetTemperatureMode(value)
    local body = {
        timed_mode = (value == 'time_limited')
    }
```

```lua
    self:getClient():POST('/update'):body(JSON:encode(body)):send()
end

function CustomDevice:setTargetTemperatureRemainingTime(value)
    local body = {
        timed_mode_remaining_time = value
    }
    self:getClient():POST('/update'):body(JSON:encode(body)):send()
end

function CustomDevice:setConfirmTimeMode(value)
    local body = {
        time_mode_request_needed = value
    }
    self:getClient():POST('/update'):body(JSON:encode(body)):send()
end

function CustomDevice:onEvent(event)
    if dateTime:changed() then
        -- send status request every minute
        self:getClient():GET('/status')
    end

    self:getClient():onMessage(function (status, responseBody, url,
        responseHeaders)

        local success = status // 100 == 2
        if success then

            -- check if response from status request and parse
            if url:find('/status') then
                local response = JSON:decode(responseBody)

                -- set parameters according to read data
                -- adding `true` as the last argument will prevent
                -- running callbacks for these parameters
                self:setVariantDeviceValue('target_temperature_minimum',
                    response['target_temperature_minimum'], true)
                self:setVariantDeviceValue('target_temperature_maximum',
                    response['target_temperature_maximum'], true)

                -- check if system mode can be changed
                -- (device is not associated with a thermostat or
                -- a heat pump manager)
                if not self:hasLabel('managed_by_thermostat') and not
                    self:hasLabel('managed_by_heat_pump_manager') then
                     self:setVariantDeviceValue('system_mode',
                        response['system_mode'], true)
                end

                -- set target temperature using commands or parameter,
                -- based on association with thermostat
                if self:hasLabel('managed_by_thermostat') then

                    -- if thermsotat call proper command
                    if response["timed_mode"] == true then
                        -- enable time mode for time from integrated device
                        self:call('enable_time_limited_mode',
                            {response['timed_mode_remaining_time'],
                            response['target_temperature']})
                    else
                        self:call('enable_constant_mode',
                            response['target_temperature'])
                    end
```

```lua
            else

                -- set parameter directly if not associated with thermostat
                self:setVariantDeviceValue('target_temperature',
                    response['target_temperature'], true)

            end

        end

    elseif status >= 0 then

        print("Request failed with status " .. status)
        print("Response from server: " .. responseBody)

    else

        print("Internal error " .. status)
        print("Error message: " .. responseBody)

    end
end)
end
```

Two state input sensor

Two state input sensor device variant is set when `variant` is `two_state_input_sensor`.

Associations

Device can be associated with *Thermostat* or *Thermostat output group (Virtual Contact)*.

When associated with Thermostat device acts as opening sensor stopping the thermostat if `state` is `false`. When associated with Virtual Contact device can change the mode of associated thermostats. If `state` is `true` it will set `cooling` mode, and if `state` is `false` it will set `heating` mode.

Status of device must be set to `online` for proper work with associated devices/widgets. Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available Properties

- `state` *(boolean)*

  State of the input.

Example

```lua
-- Two state input sensor integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:onEvent(event)
    local sensor = self:getClient()

    if dateTime:changed() then
        -- read register 1
        sensor:readInputRegisterAsync(1)
    end

    sensor:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    sensor:onRegisterAsyncRead(function (_, _, value)
        -- the device responded, so it is online
        self:setValue("status", "online")

        local state = value == 1
        self:setVariantDeviceValue("state", state, true)
    end)
end
```

Humidity sensor

Humidity sensor device variant is set when `variant` is `humidity_sensor`.

Associations

Device can be associated with **Thermostat** and added to sensor screen in **SBus Control Panel** or **SLink Control Panel**.

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available Properties

- `humidity` *(number)*

  Measured humidity sensor value.

  Unit: 0.1 %

  Range: 0 – 1000 (0.0 % – 100.0 %)

Example

```lua
-- Humidity sensor integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:onEvent(event)
    local sensor = self:getClient()

    if dateTime:changed() then
        -- read register 0802
        sensor:readInputRegisterAsync(802)
    end

    sensor:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    sensor:onRegisterAsyncRead(function (_, _, value)
        -- the device responded, so it is online
        self:setValue("status", "online")

        local humidity = asInt16(value) * 10
        self:setVariantDeviceValue("humidity", humidity, true)
    end)
end
```

Analog Input

Analog input device variant is set when `variant` is `analog_input`. Can be used to represent any analog input value like voltage, current, pressure etc.

Associations

Device can be added to sensor screen in **SBus Control Panel** or **SLink Control Panel**.

Status of device must be set to `online` for proper work with associated devices/widgets.

Can be set using `self:setValue("status", "online")` in Lua code. See examples below.

Available Properties

- `value` *(number)*

  Measured analog input value.

- `unit` *(string)*

  Unit of the measured value.

  Range: maximum 16 characters

Example

```lua
-- Analog input integrated using a Modbus client
-- In order to work, Modbus client component must be added with name
-- "modbus_client"
function CustomDevice:getClient()
    return self:getComponent("modbus_client")
end

function CustomDevice:onEvent(event)
    local sensor = self:getClient()

    if dateTime:changed() then
        -- read register 0803
        sensor:readInputRegisterAsync(803)
    end

    sensor:onAsyncRequestFailure(function (_, err)
        if err == "TIMEOUT" then
            -- the device is not responding
            self:setValue("status", "offline")
        end
    end)

    sensor:onRegisterAsyncRead(function (_, _, value)
        -- the device responded, so it is online
        self:setValue("status", "online")

        local voltage = asInt16(value) * 1.23
        self:setVariantDeviceValue("value", voltage, true)
        self:setVariantDeviceValue("unit", "mV", true)
    end)
end
```

Modules

To enable more flexible integrations custom device can create its own modules. Modules will appear in the system as separate devices with class `custom_device_module`. User can create various types of modules and then associate them with Sinum internal algoritms, such as Virtual Thermostat, Energy management etc.

All integration must be defined and implemented in Custom device lua code, so user must set all the parameters of the modules in the lua code.

Each module device provides two callback functions - for parameters change: `params_callback_function` - this callback will be executed when parameter in device changes using `setValue` function from lua of from application via REST update - for command execution: `command_callback_function` - this callback will be executed after command call for the device, in function arguments user can get the command arguments and the list of parameters that were changed due to command call.

Available module types

- `relay`
- `temperature_sensor`
- `temperature_regulator`
- `two_state_input_sensor`
- `opening_sensor`
- `humidity_sensor`
- `dimmer`
- `rgb_controller`
- `radiator_actuator`
- `blind_controller`
- `inverter`
- `battery`
- `car_charger`
- `energy_meter`
- `heat_pump`
- `common_dhw_main`
- `co2_sensor`
- `flood_sensor`
- `smoke_sensor`
- `motion_sensor`
- `aq_sensor`
- `iaq_sensor`
- `light_sensor`

- `pressure_sensor`
- `analog_input`
- `analog_output`

Module related methods

Methods called in custom device

- `addModules(modules)`

  Adds the modules defined in the parameter to the custom device.

  Arguments:

  - `modules` *(array)* - array of module definition objects. (see `addModule` function for module definition schema).

- `addModule(module)`

  Add a single module device with the data defined in parameter.

  Arguments:

  - `module` *(object)* - module definition with given properties:
    - `uuid` *(string) required* - unique identifier of the module, must be unique in the scope of one custom device, second call of addModule with the same uuid will be ignored. `uuid` is later used to access the module in the custom device. UUID must consists of alphanumerical characters or `-` only. Maximum uuid length is 64 characters.
    - `type` *(string) required* - type of the device module that should be created
    - `name` *(string) required* - name of the device module
    - `params_callback_function` *(string)* - name of the callback function that will be called after the parameters change. Parameter is optional, when not provided, default name will be used: `onModuleChanged`
    - `command_callback_function` *(string)* - name of the callback function that will be called after some commands called on the module device. Parameter is optional, when not provided, default name will be used: `onModuleCommand`
    - `data` *(object)* - initial data with which the device should be created. Data varies depending on the module device type. Parameter is optional.

- `getModule(uuid)`

  Gets the module device when available in the custom device.

  Arguments:

  - `uuid` *(string)* - module device uuid

  Returns:

  (userdata) - Module device object or nil when no module device with given uuid exists.

- `getModules()`

  Gets the list of all module devices created for the custom device.

Arguments:

Returns:

(array) - List of all module devices.

Methods called on the module

- `updateProperties(data)`

Updates the properties of the module device. Can update the same properties with the same restrictions as `setValue` method. **Note:** Does not call parameters callback.

Method can be used only on the module retrieved using `getModule` method in custom device.

Arguments:

- `data` *(object)* - parameters update data in the form of map: parameter - value

Returns:

(userdata) - reference to device object, for call chains

- `callInternal(command_name, arg)`

Runs a device module command. Can be used as standard `call` method. **Note:** Does not call command call callback. Method can be used only on the module retrieved using `getModule` method in custom device.

Arguments:

- `command_name` *(string)* - name of command available for device
- `arg` *(any, optional)* - argument for command

Returns:

- *(userdata)* - reference to device object, for call chains

Examples

Create module devices for the heat pump integration after adding or importing the custom device and prepare the callback functions

```lua
function CustomDevice:onInit(reason)
    if reason == "add" or reason == "import" then

        -- add heat pump, DHW, energy meter, temperature sensor and relay modules

        self:addModules({
            {
                type = "heat_pump",
                uuid = "heat-pump-module",
                name = "Heat pump module",
                params_callback_function = "heatPumpParameterChanged"
            },
            {
                type = "common_dhw_main",
```

```lua
            uuid = "DhwUuid",
            name = "Domestic hot water device",
            params_callback_function = "dhwParameterChanged"
        },
        {
            type = "energy_meter",
            uuid = "energy-meter-module",
            name = "Main energy meter"
        },
        {
            type = "temperature_sensor",
            uuid = "temperature-module",
            name = "Heat pump temperature sensor"
        },
        {
            type = "relay",
            uuid = "relay-module",
            name = "Heat pump start relay",
            params_callback_function = "relayParameterChanged",
            command_callback_function = "onRelayModuleCommand"
        }
    })
    end
end

-- prepare callback functions for modules
function CustomDevice:heatPumpParameterChanged(uuid, parameter, value)
    print("Module " .. uuid .. " parameter: " .. parameter .. " changed to: " ..
      tostring(value))
end

function CustomDevice:dhwParameterChanged(uuid, parameter, value)
    print("Module " .. uuid .. " parameter: " .. parameter .. " changed to: " ..
      tostring(value))
end

function CustomDevice:relayParameterChanged(uuid, parameter, value)
    print("Module " .. uuid .. " parameter: " .. parameter .. " changed to: " ..
      tostring(value))
end

function CustomDevice:onRelayModuleCommand(uuid, cmd, commandParameters,
  changedParameters)
    print("Module " .. uuid .. " command: " .. cmd .. " called with arguments: "
      .. tostring(commandParameters))
    if #changedParameters == 0 then
        print("No parameters changed in module")
    else
        print("Parameters changed in module:")
        for param, value in pairs(changedParameters) do
            print(param .. ": " .. tostring(value))
        end
    end
end
```

Create dimmer modules using single addModule method and get the module in the callback to update the remote device

```lua
function CustomDevice:onEvent(event)

    -- integrated using HTTP api
    self.getComponent("http-client"):onMessage(function (status, payload, url)
```

```lua
        if status == 200 then
            local path = utils.url:getPath(url)

            -- create modules based on the HTTP discovery request
            if path == "/api/dimmers/discovery" then
                local dimmers = JSON:decode(payload)

                for i, dimmer in pairs(dimmers) do
                    self:addModule({
                        uuid = dimmer["id"],
                        name = "Dimmer " .. tostring(i),
                        type = "dimmer",
                        params_callback_function = "onModuleParameterChange",
                        command_callback_function = "onModuleCommand"
                    })
                end
            end

            -- update the states based on state request
            if path == "/api/dimmers/state" then
                local dimmers = JSON:decode(payload)

                for _, dimmer in pairs(dimmers) do

                    local module = self:getModule(dimmer["id"])

                    module:updateProperties({
                        target_level = dimmer["brightness"],
                        state = dimmer["state"]
                    })
                end
            end
        end
    end)
end

-- prepare callbacks for module devices to synchronize the remote device state


    -- common callback which will be invoked when any of the dimmers changes parameter
function CustomDevice:onModuleParameterChange(uuid, parameter, value)
    print("Module " .. uuid .. " parameter: " .. parameter .. " changed to: " ..
        tostring(value))

    -- get the module and prepare the data for request
    local module = self:getModule(uuid)

    local request = {
        id = uuid,
        state = module:getValue("state"),
        brightness = module:getValue("target_level")
    }

    -- send the data
    self:getComponent("http-client"):POST("/api/dimmers/update"):body(JSON:
        encode(request)):send()
end


    -- common callback which will be invoked when any of the dimmers receives the command ca
function CustomDevice:onModuleCommand(uuid, cmd, commandParameters,
    changedParameters)

    if cmd == "set_level" then
```

```lua
        -- set target brightness with transition time provided in command

        local request = {
            id = uuid,
            brightness = commandParameters[1],
            transitionTime = commandParameters[2]
        }

        self:getComponent("http-client"):POST("/api/dimmers/update"):body(JSON:
            encode(request)):send()

    elseif cmd == "stop" then

        local request = {
            id = uuid,
            command = stop
        }

        self:getComponent("http-client"):POST("/api/dimmers/update"):body(JSON:
            encode(request)):send()

    end
end
```

Examples

Using callbacks

Assuming the user created a device with one of each type of control e.g.:

(Example device structure)

```json
{
  "components": [
    {
      "type": "http_client",
      "name": "my_http_client",
      ...
    },
    {
      "type": "mqtt_client",
      "name": "my_mqtt_client",
      ...
    },
    {
      "type": "ws_client",
      "name": "my_ws_client",
      ...
    }
  ],
  "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"
    },
    {
      "type": "device_selector",
      "name": "my_device_selector",
      ...
      "on_change": "onMyDeviceSelectorValueChange"
    },
  ],
  ...
}
```

(Custom Device logic)

```lua
-- 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")
  }

  self:getComponent("my_http_client")
    :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
```

```lua
  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
  self:getComponent("my_mqtt_client"):publish("progress_bar/value",
    tostring(newValue), 0, false)

  -- send to websocket
  self:getComponent("my_ws_client"):send(tostring(newValue))
end

-- on_change callback handler for device_selector
function CustomDevice:onMyDeviceSelectorValueChange(newValue, element)
  -- print new value (an object with `class` and `id` fields, encoded in JSON)
  print("Value changed in element " .. element:getValue("name") .. " to " ..
    newValue)

  -- get device and work on it
  local device = element:getValue("associations.selected")
  if device ~= nil then -- protect against change when device is removed
    -- assume we have relay as accepted_types, so we can turn it off here
    device:call("turn_off")
  end
end

function CustomDevice:onMyDeviceSelectorMultiValueChange(newValue, element)
  -- print new value (an array of objects with `class` and `id` fields,
  -- encoded in JSON)
  print("Value changed in element " .. element:getValue("name") .. " to " ..
    newValue)

  -- when device selector allow_multiple is set to true,
  -- we get an array of devices
  local devices = element:getValue("associations.selected")
  for i = 1, #devices do
    -- assuming that 'relay' is the only accepted type,
    -- devices are relays and can be turned off like that
    devices[i]:call("turn_off")
  end
end

-- color_picker callbacks
function CustomDevice:onMyColorPickerStateChange(newValue, element)
  -- print new value (boolean value of new switcher state)
  print("State changed in color picker " .. element:getValue("name") .. " to "
    .. tostring(newValue))

  if newValue then
    sbus[9]:call("turn_on")
```

```lua
    else
      sbus[9]:call("turn_off")
    end
  end

  function CustomDevice:onMyColorPickerBrightnessChange(newValue, element)
    -- print new value (integer value of brightness)
    print ("Brightness changed in color picker " .. element:getValue("name") ..
        " to " .. tostring(newValue))

    sbus[9]:call("set_brightness", newValue)
  end

  function CustomDevice:onMyColorPickerColorChanged(newValue, element)
    -- print new value (object with new color data)
    print ("Color changed in color picker " .. element:getValue("name") .. " to "
        .. JSON:encode(newValue))

    if (newValue.colorMode = "temperature")
      sbus[9]:call("set_temperature", newValue.temperature)
    elseif (newValue.colorMode = "rgb")
      sbus[9]:call("set_color", newValue.color)
    end
  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 switches to off at 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
    self:getComponent("my_http_client"):onMessage(function (status, payload)
        local msg = ""
        if status == 200 then
            local decoded = JSON:decode(payload)
            msg = decoded.data
        else
            msg = string.format("Error: %d", status)
        end
        self:getElement("my_text_field"):call("set_value", msg)
    end)
  end

  -- onCommand callback, can catch custom commands executed
  function CustomDevice:onCommand(command, arg)

    if command == "modify_elements" then
      utils:printf("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
      utils:printf("Command %s not implemented!", command)
    end

  end
```

Change element values/call commands from scene or automation

```lua
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

```lua
-- 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)
```

`onInit` callback

Send login request via HTTP client and set initial texts in custom device elements.

```lua
local http = http_client[9]

function CustomDevice:onInit()
    -- set device to online
    self:setValue("status", "online")

    -- set initial texts
    self:getElement("state_text"):setValue("value", "Logging..")
    self:getElement("device_temperature_text"):setValue("value", "-- °C")

    -- send login request
    local data = { login = "admin", password = "password" }
    http:POST("/login")
        :body(JSON:encode(data))
        :send()
end
```

`onInit` callback with source argument, set text element value based on source of callback

```lua
function CustomDevice:onInit(source)
    if source == "restart" then
        -- load value from variable stored in database
        local storedValue = self:getComponent("storage"):getValue()
        self:getElement("value_text"):setValue("value", tostring(storedValue))
    else
        -- "import" or "add":
        -- in that case we do not have stored any value yet, so start with 0
        self:getElement("value_text"):setValue("value", "0")
    end
end
```

Setting custom device status and update the warning message about no connection

```lua
function CustomDevice:onInit()
    -- set online by default
    self:setValue("status", "online")

    -- configure the rest of needed parameters
end

function CustomDevice:onEvent(event)
    -- update every minute
    if dateTime:changed() then

        if self:isDeviceConnected() then
            self:setValue("status", "online")
        else
            self:setValue("status", "offline")
        end

        self:updateWarning("No connection with the device!", nil, not
            self:isDeviceConnected())

        -- update the rest of needed parameters
    end
end

function CustomDevice:isDeviceConnected()
    -- return device status as bool
end
```

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"))
  self:getComponent("my_mqtt_client"):publish(
    "cmnd/" .. tasmotaName .. "/POWER",
    utils:ternary(newValue, "ON", "OFF"), 0, false )
end

function CustomDevice:onEvent(event)
 -- message from remote device received
 self:getComponent("my_mqtt_client"):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 the switcher 2-3 times from REST API / Web or Mobile app in a row, you may end up with infinite 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"))
  self:getComponent("my_mqtt_client"):publish(
    "cmnd/" .. tasmotaName .. "/POWER",
    utils:ternary(newValue, "ON", "OFF"),
    0,
    false )
end

function CustomDevice:onEvent(event)
  self:getComponent("my_mqtt_client"):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.

CustomDeviceModule devices

Device modules created in custom devices.

Custom device modules can be accessed from scripts by indexing global `custom_device_module` container, e.g. `custom_device_module[6]` returns device module with ID #6. This container is available in every execution context.

Custom device modules can be also accessed from containing custom device using `self:getModule(uuid)`.

Properties can only be accessed via methods: `getValue`, `setValue`, and `updateProperties` (`updateProperties` only after `getModule` call).

Common Custom device module device properties

- `uuid` *(string, read-only)*

  Unique ID if the module in custom device.

  > **Required label:** `"battery_powered"`
  >
  > - `battery` *(integer, read-only)*
  >
  >   Unit: 1%
  >
  >   Value range: ⟨0; 100⟩

- `signal` *(integer, read-only)*

  Signal status if available, `nil` instead.

  Unit: 1%

  Value range: ⟨0; 100⟩

- `software_version` *(string, read-only)*

  Software name and version description.

- `params_callback_function` *(string)*

  Name of function that will be executed in containing custom device when parameters change.

  - Callback signature arguments
    - uuid *(string)* — uuid of the module in which parameter changed
    - param *(string)* — changed parameter name
    - value *(vary)* — new value of the parameter

- `command_callback_function` *(string)*

Name of function that will be executed in containing custom device when specific
command in device is executed.

- ◦ Callback signature arguments
  - uuid *(string)* — uuid of the module for which the command was called
  - cmd *(string)* — name of the command
  - commandParameters *(vary)* — parameters with which the command was called
  - changedParameters *(object)* — parameters list that were changed in module prior
    to the command callback, in key-value format

- `visible` *(boolean)*

  Indicates if device is visible in application.

Relay

Execution module that changes state depending on the control signal. Relay can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(boolean)*

  State of the output. On/Off.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `timeout` *(number)*

  Protection functionality, that will set device state to off if there are communication problems.

  Unit: 1 min

- `timeout_enabled` *(boolean)*

  Parameter that indicates if timeout functionality is enabled.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `inverted` *(boolean)*

  Indicates if physical state of relay should be the inversion of state shown in application.

- `time_since_state_change` *(number, read-only)*

  Time since last relay state change.

  Unit: 1 s

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"relay"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Commands

- `turn_on`

  Turns on relay output.

- `turn_off`

  Turns off relay output.

- `toggle`

  Changes relay output to opposite.

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "relay",
                uuid = "relay-module-1",
                name = "Relay",
                params_callback_function = "onRelayModuleChanged",
                command_callback_function = "onRelayCommand"
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onRelayModuleChanged(uuid, param, value)
    if param == "state" then
        local state = value
        -- send state change to remote device
    end
end

function CustomDevice:onRelayCommand(uuid, cmd, commandParameters,
   changedParameters)
    if cmd == "turn_on" then
        -- send the update to remote device
    elseif cmd == "turn_off" then
        -- send the update to remote device
    elseif cmd == "toggle" then
        local currentState = changedParameters["state"]
        -- send the update to remote device
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentState = false -- read from device

    self:getModule("relay-module-1"):updateProperties({
        state = currentState
    })
end
```

Turn on relay between 19:00 and 21:00

```lua
if dateTime:changed() then
    if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
      custom_device_module[4]:call("turn_on")
    elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
      custom_device_module[4]:call("turn_off")
    end
end
```

Temperature sensor

Can be assigned to virtual thermostat in web application as room or floor sensor.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`*,* `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `temperature` *(integer, read-only)*

  Measured temperature value.

  Unit: °C with one decimal number, multiplied by 10.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "temperature_sensor",
                uuid = "temperature-sensor-module-1",
                name = "Heat pump"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentTemperature = 0 -- read from device

    self:getModule("temperature-sensor-module-1"):updateProperties({
        temperature = currentTemperature
    })
end
```

Temperature regulator

Temperature regulator notifies when desired temperature is reached in room. Can be assigned to virtual thermostat in web application.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 doesn't 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_minimum` *(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). e.g. 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 or heat pump manager (label `managed_by_thermostat` and `managed_by_heat_pump_manager` not present).

  Available values: `off`, `heating`, `cooling`. Default: `heating`

- `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.

Device properties (full spec)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_regulator"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

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

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "temperature_regulator",
                uuid = "regulator-module-1",
                name = "Temperature regulator",
                params_callback_function =
                    "onTemperatureRegulatorModuleChanged",
                command_callback_function = "onRegulatorCommand"
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onTemperatureRegulatorModuleChanged(uuid, param, value)
    if param == "target_temperature" then
        -- send target_temperature change to remote device
    elseif param == "system_mode" then
        -- send system_mode change to remote device
    end
end

function CustomDevice:onRegulatorCommand(uuid, cmd, commandParameters,
    changedParameters)
    if cmd == "set_target_temperature" then
        local targetTemperature = commandParameters
        -- send the update to remote device

    elseif cmd == "enable_constant_mode" then
        local targetTemperature = commandParameters
        -- send the update to remote device

    elseif cmd == "enable_time_limited_mode" then
        local changeMinutes = commandParameters[1]
        local targetTemperature = commandParameters[2]
        -- send the update to remote device

    elseif cmd == "disable_time_limited_mode" then
        local targetTemperatureChanged = changedParameters["target_temperature"]
        local modeChanged = changedParameters["target_temperature_mode.current"]
        local remainingTimeChanged =
            changedParameters["target_temperature_mode.remaining_time"]

        if targetTemperatureChanged then

            -- send target temperature change from parameter targetTemperatureChanged
        end

        if modeChanged then
            -- send mode change from parameter modeChanged
        end
```

```lua
        if remainingTimeChanged then
            -- send remaining time change from parameter remainingTimeChanged
        end
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local targetTemperature = 0 -- read from device
    local targetTemperatureMin = 0 -- read from device
    local targetTemperatureMax = 0 -- read from device
    local mode = 0 -- read from device

    self:getModule("regulator-module-1"):updateProperties({
        target_temperature = targetTemperature,
        target_temperature_min = targetTemperatureMin,
        target_temperature_max = targetTemperatureMax,
        system_mode = (mode == 0 and 'heating' or 'cooling')
    })
end
```

Raise target temperature between 15:00 and 20:00

```lua
if dateTime:changed() then
  if dateTime:getHours() == 15 and dateTime:getMinutes() == 0 then
    custom_device_module[5]:call("set_target_temperature", 220)
  elseif dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
    custom_device_module[5]:call("set_target_temperature", 190)
  end
end
```

Humidity sensor

Can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (*`updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `humidity` *(number, read-only)*

  Measured humidity value.

  Unit: rH% with one decimal number, multiplied by 10.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"humidity_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "humidity_sensor",
                uuid = "humidity-sensor-module-1",
                name = "Humidity sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentHumidity = 0 -- read from device

    self:getModule("humidity-sensor-module-1"):updateProperties({
        humidity = currentHumidity
    })
end
```

Two-state input sensor

Boolean input sensor checks input state.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`*,* `setValue` *and* `updateProperties` *methods (*`updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(boolean, read-only)*

  State of the input.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"two_state_input_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "two_state_input_sensor",
                uuid = "two-state-sensor-module-1",
                name = "Two state sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentState = false -- read from device

    self:getModule("two-state-sensor-module-1"):updateProperties({
        state = currentState
    })
end
```

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`*,* `setValue` *and* `updateProperties` *methods (*`updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `open` *(boolean, read-only)*

  Opening sensor state. Open/Closed.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"opening_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- uuid *(string, read-only)*

- battery *(integer, read-only)*

- signal *(integer, read-only)*

- software_version *(string, read-only)*

- params_callback_function *(string)*

- command_callback_function *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "opening_sensor",
                uuid = "opening-sensor-module-1",
                name = "Opening sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentState = false -- read from device

    self:getModule("opening-sensor-module-1"):updateProperties({
        open = currentState
    })
end
```

Radiator actuator

Controls valve opening based on e.g. temperature regulator or thermostat state.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`*,* `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `opening` *(number)*

  Current valve opening level.

  Unit: 1 %

- `opening_minimum` *(number)*

  Lower valve opening level limit. Could not be greater than maximum. Setting minimum value above current opening value, will also change current opening value to minimum.

  Unit: 1 %

  Note: Does not call parameter change callback as it does not affect device directly.

- `opening_maximum` *(number)*

  Upper valve opening level limit. Could not be less than minimum. Setting maximum value below current opening value, will also change current opening value to maximum.

  Unit: 1 %

  Note: Does not call parameter change callback as it does not affect device directly.

- `emergency_opening` *(number)*

  Emergency opening level when communication with central device is lost.

  Unit: 1 %

Device properties (full spec)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"radiator_actuator"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (**full spec**)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

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. Calls a command callback. No action is actualy performed on the device parameters.

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "radiator_actuator",
```

```lua
                uuid = "radiator-module-1",
                name = "Radiator actuator",
                params_callback_function = "onRadiatorModuleChanged",
                command_callback_function = "onRadiatorCommand"
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onRadiatorModuleChanged(uuid, param, value)
    if param == "opening" then
        local opening = value
        -- send opening change to remote device
    elseif param == "emergency_opening" then
        local emergencyOpening = value
        -- send emergency opening change to remote device
    end
end

function CustomDevice:onRadiatorCommand(uuid, cmd, commandParameters,
    changedParameters)
    if cmd == "open" then
        local opening = commandParameters
        -- send the update to remote device
    elseif cmd == "calibration" then
        -- send the calibration request to remote device
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentState = false -- read from device

    self:getModule("relay-module-1"):updateProperties({
        state = currentState
    })
end
```

Regulate valve based on room temperature

```lua
local sensor = wtp[1]
local valve = custom_device_module[2]

if sensor:changedValue("temperature") then
    local reading = sensor:getValue("temperature")

    if reading > 220 then
        valve:call("open", 0)
    elseif reading > 200 then
        valve:call("open", 50)
    else
        valve:call("open", 100)
    end
end
```

Blind controller

Controller opens and closes a roller shade or tilt blind.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`*,* `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_opening` *(number)*

  Desired setpoint opening, which device will try to achieve.

  Unit: 1 %

  > **Note**
  >
  > If device doesn't contain `percent_opening_control` label (i.e. `percent_control_supported` is set to `false`), target opening is limited to 0%, 50% or 100% (only these three).

- `current_opening` *(number, read-only)*

  Current opening value.

  Unit: 1 %

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Required label: `"percent_tilt_control"`

- `target_tilt` *(number, optional)*

  Desired tilt position.

  Unit: 1 %.

- `current_tilt` *(number, optional, read-only)*

  Current tilt position

  Unit: 1 %.

Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `percent_control_supported` *(boolean, read-only)*

  Indicates if percent control is supported for the controller. Adds the `percent_opening_control` label if set to `true`. Default: `true`.

Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `tilt_control_supported` *(boolean, read-only)*

  Indicates if tilt control is supported for the controller. Adds the `percent_tilt_control` label if set to `true`. Default: `true`.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (full spec)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"blind_controller"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

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. Calls the callback only without affecting the device state.

- `calibration`

  Starts blind calibration cycle. Calls the callback only without affecting the device state.

- `tilt`

  Calls tilt to the desired value.

  Argument:

  tilt percentage *(number)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "blind_controller",
                uuid = "blind-module-1",
                name = "Blind controller",
                params_callback_function = "onBlindModuleChanged",
                command_callback_function = "onBlindCommand",
                data = {
                    percent_control_supported = true,
                    tilt_control_supported = true
                }
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onBlindModuleChanged(uuid, param, value)
    if param == "target_opening" then
        local targetOpening = value
        -- send target_opening change to remote device
    elseif param == "target_tilt" then
```

```lua
        local targetTilt = value
        -- send target_tilt change to remote device
    end
end

function CustomDevice:onBlindCommand(uuid, cmd, commandParameters,
   changedParameters)
    if cmd == "open" then
        local targetOpening = commandParameters
        -- send the update to remote device

    elseif cmd == "tilt" then
        local targetTilt = commandParameters
        -- send the update to remote device

    elseif cmd == "stop" then
        -- send the stop request to remote device

    elseif cmd == "calibration" then
        -- send the calibration request to remote device

    elseif cmd == "up" then
        -- send the update to remote device - target opening 100%

    elseif cmd == "down" then
        -- send the update to remote device - target opening 0%
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentOpening = 0 -- read from device
    local targetOpening = 0 -- read from device
    local currentTilt = 0 -- read from device
    local targetTilt = 0 -- read from device

    self:getModule("blind-module-1"):updateProperties({
        target_opening = targetOpening,
        current_opening = currentOpening,
        target_tilt = targetTilt,
        current_tilt = currentTilt
    })
end
```

Open blind at sunrise and close at sunset

```lua
if event.type == "sunrise" then
  custom_device_module[3]:call("up")
elseif event.type == "sunset" then
  custom_device_module[3]:call("down")
end
```

Set blind to half-open at noon

```
if dateTime:changed() then
  if dateTime:getHours() == 12 and dateTime:getMinutes() == 0 then
    custom_device_module[3]:call("open", 50)
  end
end
```

Dimmer

Device that controlls light intensity of output LED.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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: 1%.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"dimmer"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (**full spec**)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Commands

- `turn_on`

  Turns on output state.

- `turn_off`

  Turns off output state.

- `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 a command callback. No action is actualy performed on the device parameters.

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "dimmer",
```

```lua
                uuid = "dimmer-module-1",
                name = "Dimmer",
                params_callback_function = "onDimmerModuleChanged",
                command_callback_function = "onDimmerCommand"
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onDimmerModuleChanged(uuid, param, value)
    if param == "state" then
        local state = value
        -- send state change to remote device
    elseif param == "target_level" then
        local brightness = value
        -- send brightness change to remote device
    end
end

function CustomDevice:onDimmerCommand(uuid, cmd, commandParameters,
    changedParameters)
    if cmd == "turn_on" then
        local transitionTime = 0
        -- possible transition time passed in parameters
        if type(commandParameters) == "number" then
            transitionTime = commandParameters
        end
        -- send the update to remote device

    elseif cmd == "turn_off" then
        local transitionTime = 0
        -- possible transition time passed in parameters
        if type(commandParameters) == "number" then
            transitionTime = commandParameters
        end
        -- send the update to remote device

    elseif cmd == "toggle" then
        local transitionTime = 0
        -- possible transition time passed in parameters
        if type(commandParameters) == "number" then
            transitionTime = commandParameters
        end
        -- send the update to remote device

    elseif cmd == "set_level" then
        local targetLevel = commandParameters[1]
        local transitionTime = #commandParameters > 1 and commandParameters[2]
            or 50

        -- send the update to remote device
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentState = 0 -- read from device
    local currentBrightness = 0 -- read from device

    self:getModule("dimmer-module-1"):updateProperties({
        target_level = currentBrightness,
        state = currentState
    })
end
```

Turn on light at 19:00 and turn off at 21:00

```lua
if dateTime:changed() then
  if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
    custom_device_module[4]:call("turn_on")
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    custom_device_module[4]:call("turn_off")
  end
end
```

Set the light intensity to 75% during 2 minutes

```lua
custom_device_module[4]:call("set_level", {75, 1200})
```

Dim or brighten lights while button is pressed (simple version)

Solution Drawback: it will always take constant time to move from 1% → 100%, 50% → 100%, 10% → 1% etc.

```lua
local dimmer = custom_device_module[4]
local button = wtp[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_") or action:find("button_2_held_")
  then
    -- stop current moving action
    dimmer:call("stop")
  end
end
```

RGB controller

Device that controlls color and light intensity of output LED.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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: 1 %.

- `led_color` *(string, read-only)*

  HTML/Hex RGB color that device will set on its output led strip.

  Example: `"#00ff7f"`

- `white_temperature` *(number, read-only)*

  White temperature that device will set on its output led strip.

  Unit: 1 K

- `color_mode` *(string, read-only)*

  Color mode that device is set on. One of: `"rgb"`, `"temperature"`.

Device properties (full spec)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"rgb_controller"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

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)*

- `stop`

  Calls a command callback. No action is actualy performed on the device parameters.

Examples

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "rgb_controller",
                uuid = "rgb-module-1",
                name = "RGB light",
                params_callback_function = 'onRGBModuleChanged',
                command_callback_function = 'onRGBCommand'
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onRGBModuleChanged(uuid, param, value)
```

```lua
    if param == "state" then
        local state = value
        -- send state change to remote device
    end
end


    -- command callback function to update remote module after command call on inverter in S
function CustomDevice:onRGBCommand(uuid, cmd, commandParameters,
    changedParameters)
    if cmd == "set_color" then
        local rgbColor = commandParameters[1]
        local transitionTime = #commandParameters > 1 and commandParameters[2]
            or 50
        -- send color change to rgb with given parameters

    elseif cmd == "set_temperature" then
        local temperature = commandParameters[1]
        local transitionTime = #commandParameters > 1 and commandParameters[2]
            or 50
        -- send temperature change with given parameters

    elseif cmd == "set_brightness" then
        local brightness = commandParameters[1]
        local transitionTime = #commandParameters > 1 and commandParameters[2]
            or 50
        -- send brightness change with given parameters

    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentState = false -- read from device
    local currentMode = 0 -- read from device
    local color = "#123456" -- read from device
    local temperature = 3000 -- read from device

    if currentState then
        if currentMode == 0 then
            -- assume 0 is RGB mode, otherwise temperature mode
            self:getModule("rgb-module-1"):callInternal("set_color", {color})
        else
            self:getModule("rgb-module-1"):callInternal("set_temperature",
                {temperature})
        end
    else
        self:getModule("rgb-module-1"):updateProperties({
            state = currentState
        })
    end
end
```

Turn on light to specific color at 19:00 and turn off at 21:00

```lua
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
```

```lua
    rgb:call("turn_off")
  end
end
```

Tune color temperature based on the time of day

```lua
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

```lua
local rgb = wtp[79]
local animation_id = 2

rgb:call("activate_animation", { id = animation_id })
```

Stop active animation

```lua
local rgb = wtp[79]
rgb:call("stop_animation")
```

Activate an animation by ID when device state changes

```lua
local rgb = wtp[79]
local animation_id = 3

if wtp[3]:changedValue("state") then
  rgb:call("activate_animation", { id = animation_id })
end
```

Battery

Representation of Battery related parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `charge_power` *(number, read-only)*

  Current charing (positive number) or discharging (negative number) power.

  Unit: 1 mW

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `soc` *(number, read-only)*

  Current state of charge.

  Unit: 1 %

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `energy_charged_today` *(number, read-only)*

  Amount of energy charged to the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh).

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `energy_discharged_today` *(number, read-only)*

  Amount of energy consumed from the battery today.

  Unit: 1 Wh (with accuracy of 100 Wh)

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `battery_charge_support` *(boolean, read-only)*

  Indicates if battery charge commands should be supported. Adds or removes the label `battery_charge_support` for the module.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Commands

- `charge`

  Calls battery to charge during given period of time. Calls a command callback. No action is actualy performed on the device parameters.

  > **Note**
  >
  > Available when device has label `battery_charge_support`. Parameter `battery_charge_support` must be set to `true`.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 0
    - maximum: 50000000
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 86400
    - parameter optional

- `discharge`

  Calls battery to discharge during given period of time. Calls a command callback. No action is actualy performed on the device parameters.

  > **Note**
  >
  > Available when device has label `battery_charge_support`. Parameter `battery_charge_support` must be set to `true`.

  Argument:

  packed arguments *(table)*

  - active power in mW *(number)*
    - minumum: 0
    - maximum: 50000000
  - duration time in seconds *(number)*
    - minumum: 0
    - maximum: 86400
    - parameter optional

- `stop`

  Stops the forced charging/discharging of the device. Calls a command callback. No action is actualy performed on the device parameters.

  > **Note**
  >
  > Available when device has label `battery_charge_support`. Parameter `battery_charge_support` must be set to `true`.

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "battery",
                uuid = "battery-module-1",
                name = "Battery",
                command_callback_function = 'onBatteryCommand'
                data = {
                    battery_charge_support = true
                }
            })
    end
end


    -- command callback function to update remote module after command call on battery in Si
function CustomDevice:onBatteryCommand(uuid, cmd, value)
    if cmd == "charge" then
        local power = value[1]
        local duration = value[2]
        -- duration parameter is optional
        if duration == nil then
            duration = 3600
        end
        -- send update to device
    elseif cmd == "discharge" then
        local power = value[1]
        local duration = value[2]
        -- duration parameter is optional
        if duration == nil then
            duration = 3600
        end
        -- send update to device
    elseif cmd == "stop" then
        -- send update to device
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentChargePower = 0 -- read from device
    local currentSoc = 0 -- read from device
    local currentEnergyCharged = 0 -- read from device
    local currentEnergyDischarged = 0 -- read from device

    self:getModule("battery-module-1"):updateProperties({
        charge_power = currentChargePower,
        soc = currentSoc,
        energy_charged_today = currentEnergyCharged,
        energy_discharged_today = currentEnergyDischarged
    })
end
```

Domestic Hot Water

Representation of DHW related parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `current_temperature` *(number, read-only)*

  Current domestic hot water temperature.

  Unit: 0.1 °C.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °C.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `hysteresis` *(number)*

  Damper factor, which will protect from continuous on/off switching when current temperature is near target value.

  Unit: 0.1 °C.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

- `heat_demand` *(boolean)*

  Domestic Hot Water demand.

  > **Note**
  >
  > Cannot be modified when device is associated with Heat Pump Manager.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_dhw_main"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Commands

- `set_target_temperature`

  Changes target temperature for domestic hot water.

  Argument:

  target temperature in 0.1°C *(number)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "common_dhw_main",
                uuid = "dhw-module-1",
                name = "Domestic hot water",
                params_callback_function = 'onDHWModuleChanged'
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onDHWModuleChanged(uuid, param, value)
    if param == "target_temperature" then
        -- send target_temperature change to remote device
    elseif param == "hysteresis" then
        -- send hysteresis change to remote device
    elseif param == "heat_demand" then
        -- send heat_demand change to remote device
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentTemperature = 0 -- read from device
    local targetTemperature = 0 -- read from device
    local currentHysteresis = 0 -- read from device
    local currentDemand = 0 -- read from device

    self:getModule("dhw-module-1"):updateProperties({
        current_temperature = currentTemperature,
        target_temperature = targetTemperature,
        hysteresis = currentHysteresis,
        heat_demand = currentDemand
    })
end
```

Car Charger

Representation of Car Charger related parameters.

Device will be automatically displayed in **Energy center** and will be used to

- display power in detailed view of *flow monitor* widget
- display energy consumption in *energy consumption* widget

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `charge_power` *(positive number, read-only)*

  Instantaneous power used by charger to charge the EV.

  Unit: 1 mW

  Range: 0 mW – 2 147 483 647 mW

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `energy_charged_total` *(number)*

  Total energy charged during device lifetime. Used to show energy consumption for car chargers in Energy consumption widget. Today's energy is calculated based on changes of this parameter.

  Unit: 1 Wh

  Range: 0 Wh – 4 294 967 295 Wh

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"car_charger"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties ([full spec](#))

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Commands

- `start_charge`

  Starts charging the EV (with optional charge power limit).

  Argument:

  charge power in 1 mW *(number, optional)*

  - minumum: 1
  - maximum: 350000000

- `stop_charge`

  Stops charging the EV.

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then
```

```lua
        self:addModule({
                type = "car_charger",
                uuid = "car-charger-module-1",
                name = "Car Charger"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentPower = 0 -- read from device
    local currentEnergyTotal = 0 -- read from device

    self:getModule("car-charger-module-1"):updateProperties({
        charge_power = currentPower,
        energy_charged_total = currentEnergyTotal,
    })
end
```

Energy meter

Representation of Energy meter related parameters.

Device can be associated with **Energy center**. When associated with Energy center, Energy meter will be used to

- display power flow to and from grid in *flow monitor*
- display power in detailed view of *flow monitor* (if `uses_energy_of_building == true`)
- display energy consumption in *energy consumption* widget, for electrical sockets or household (based on `uses_energy_of_building` parameter)

When energy meter has `uses_energy_of_building` set to `true` it will automatically be included in Energy center widgets. Only main energy meter with `uses_energy_of_building` set to `false` need to be added to Energy center associations.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `total_active_power` *(integer, read-only)*

  Active power measured by energy meter.

  Unit: 1 mW

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `energy_sum_total` *(number)*

  Total energy consumed of all phases lifetime. Used to show energy consumption for electrical sockets in Energy consumption widget. Today's energy is calculated based on changes of this parameter. Parameter is only used for displaying electrical sockets' energy, so only if `uses_energy_of_building` is `true`.

  Unit: 1 Wh

  Range: 0 Wh – 4 294 967 295 Wh

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `uses_energy_of_building` *(boolean)*

  Indicates if energy meter is used to measure energy of device inside the building. (Not main energy meter). When set to `true` it is automatically included in Energy center widgets calculations (Flow monitor and Energy consumption). Should be set to `false` if it is used as main energy meter. In this case to be included in Energy center widgets calculations user needs to add this energy meter as **Main energy meter** in energy center.

Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (**full spec**)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "energy_meter",
                uuid = "energy-meter-module-1",
                name = "Energy meter"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentPower = 0 -- read from device
    local currentEnergyTotal = 0 -- read from device

    self:getModule("energy-meter-module-1"):updateProperties({
        total_active_power = currentPower,
        energy_sum_total = currentEnergyTotal,
        uses_energy_of_building = false   -- main energy meter
    })
end
```

Heat pump

Representation of Heat Pump related parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`*,* `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `enabled` *(boolean)*

  Indicates if heat pump is enabled. When associated with Heat pump manger or Virtual contact, `thermal_demand` will be set to `none` if this parameter is set to `false`. Otherwise, `thermal_demand` will be changed based on these devices algorithms.

- `thermal_demand` *(string)*

  Heat pump thermal demand. Available values: `none`, `heat`, `cool`. Main parameter used to control heat pump.

  - `none` - pump is not working
  - `heat` - pump is working and heating
  - `cool` - pump is working and cooling

- `electric_heater_active` *(boolean)*

  Indicates electric heater activation state in heat pump. Used when associated with Heat pump manger to inform used if electric heater is active.

- `target_temperature_indoor` *(integer)*

  Room target temperature. When associated with Heat pump manager it will be get as target temperature in manager or changed by manager if user changes target temperature via Heat pump manager (change will be indicated via callback `params_callback_function`).

  Unit: 0.1 °C

  Range: 100 – 350 (10.0 °C – 35.0 °C)

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"heat_pump"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "heat_pump",
                uuid = "heat-pump-module-1",
                name = "Heat pump",
                params_callback_function = 'onHeatPumpModuleChanged'
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onHeatPumpModuleChanged(uuid, param, value)
    if param == "enabled" then
        -- send enabled change to remote device
```

```lua
    elseif param == "thermal_demand" then
        -- send thermal_demand change to remote device
    elseif param == "target_temperature_indoor" then
        -- send target_temperature_indoor change to remote device
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local isEnabled = true -- read from device
    local mode = 0 -- read from device
    local targetTemperatureIndoor = 0 -- read from device

    self:getModule("heat-pump-module-1"):updateProperties({
        enabled = isEnabled,
        thermal_demand = (mode == 0 and 'heat' or 'none'),
        target_temperature_indoor = targetTemperatureIndoor
    })
end
```

Inverter

Representation of Inverter related parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `pv_total_active_power` *(number)*

  Current total power produced by all photovoltaic panels. Parameter will be used to calculate power flow to grid, battery and house in Flow monitor.

  Unit: 1 mW

  Range: 0 mW – 4 294 967 295 mW

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `energy_produced_total` *(number)*

  Total amount of energy produced by PV over a lifetime. Parameter will be used to display energy distibution in Energy consumption widget. Energy used for auto-consumption, energy storage and grid. Distribution is calculated using this parameter and ratio based on flow monitor power flow.

  Unit: 1 Wh

  Range: 0 Wh – 4 294 967 295 Wh

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `power_to_grid` *(number)*

  Current power fed to (positive number) or consumed from (negative number) the power grid.

  Unit: 1 mW

  Range: −2 147 483 647 mW – 2 147 483 647 mW

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"inverter"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (**full spec**)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Commands

- `turn_on`

  Calls a command callback. No action is actualy performed on the device parameters.

- `turn_off`

  Calls a command callback. No action is actualy performed on the device parameters.

- `limit_active_power`

  Calls a command callback. No action is actualy performed on the device parameters.

  Argument:

  Active power limit.

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "inverter",
                uuid = "inverter-module-1",
                name = "Inverter",
                command_callback_function = 'onInverterCommand'
            })
    end
end


    -- command callback function to update remote module after command call on inverter in S
function CustomDevice:onInverterCommand(uuid, cmd, value)
    if cmd == "turn_on" then
        -- send turn on command to inverter
    elseif cmd == "turn_off" then
        -- send turn off command to inverter
    elseif cmd == "limit_active_power" then
        -- send active power limit to inverter with value
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
    local totalPower = 0 -- read from device
    local energyToday = 0 -- read from device
    local powerToGrid = 0 -- read from device

    self:getModule("inverter-module-1"):updateProperties({
        pv_total_active_power = totalPower,
        energy_produced_total = energyToday,
        power_to_grid = powerToGrid
    })
end
```

Flood sensor

Detects water leak on flat surfaces.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `flood_detected` *(boolean, read-only)*

  A flag representing the detection of flood / water leak by the sensor.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"flood_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "flood_sensor",
                uuid = "flood-sensor-module-1",
                name = "Flood sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local floodDetected = false -- read from device

    self:getModule("flood-sensor-module-1"):updateProperties({
        flood_detected = floodDetected
    })
end
```

Light sensor

Measures light illuminance in lux and sends measurement to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `illuminance` *(integer, read-only)*

  Measured light illuminance value.

  Unit: 1 lx

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"light_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- uuid *(string, read-only)*

- battery *(integer, read-only)*

- signal *(integer, read-only)*

- software_version *(string, read-only)*

- params_callback_function *(string)*

- command_callback_function *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "light_sensor",
                uuid = "light-sensor-module-1",
                name = "Light sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local measuredValue = 0 -- read from device

    self:getModule("light-sensor-module-1"):updateProperties({
        illuminance = measuredValue
    })
end
```

Motion sensor

Based on custom configuration checks whether motion was detected.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `enabled` *(boolean)*

  Enable or disable sensor. e.g. If you want sense only at night-time, you can set up an automation to enable/disable sensor.

- `motion_detected` *(boolean, read-only)*

  Holds latest motion detection state.

  This parameter doesn't emit event when switch from `true` to `true` happens (subsequent motion detections). If you need to observe such action, you need to use `time_since_motion` parameter and check if `time_since_motion` equals to `0`.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"motion_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (**full spec**)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

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.

  Note: Does not call the module command callback as it does not affect the device directly.

  Argument:

  event reemission delay in seconds *(number)*, at least 1 s

Examples

Create the module and update properties in custom device code

```
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
```

```lua
                    type = "motion_sensor",
                    uuid = "motion-sensor-module-1",
                    name = "Motion sensor",
                    params_callback_function = "onMotionSensorModuleChanged",
                    command_callback_function = "onMotionSensorCommand"
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onMotionSensorModuleChanged(uuid, param, value)
    if param == "enabled" then
        local enabled = value
        -- send change to remote device
    end
end

function CustomDevice:onMotionSensorCommand(uuid, cmd, commandParameters,
    changedParameters)
    if cmd == "enable" then
        -- send the update to remote device
    elseif cmd == "disable" then
        -- send the update to remote device
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local motionDetected = false -- read from device

    self:getModule("motion-sensor-module-1"):updateProperties({
        motion_detected = motionDetected
    })
end
```

Catching motion events

```lua
if custom_device_module[4]:changedValue("motion_detected") then
    print("someone is moving around!")
end
```

```lua
if custom_device_module[4]:changedValue("time_since_motion") then
  if custom_device_module[4]:getValue("time_since_motion") == 0 then
    print("someone is moving around!")
  end
end
```

Delayed action

```lua
if dateTime:changed() then
  -- add 30 second delay
  custom_device_module[4]:call("add_time_since_motion_event", 30)
end

if custom_device_module[4]:changedValue("time_since_motion") then
  if custom_device_module[4]:getValue("time_since_motion") == 30 then
    print("someone was here 30 seconds ago")
```

```
    end
end
```

Pressure sensor

Pressure sensor measures pressure and sends measurement to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `pressure` *(number, read-only)*

  Measured pressure value.

  Unit: hPa with one decimal number, multiplied by 10 (10 Pa).

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"pressure_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "pressure_sensor",
                uuid = "pressure-sensor-module-1",
                name = "Pressure sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentPressure = 0 -- read from device

    self:getModule("pressure-sensor-module-1"):updateProperties({
        pressure = currentPressure
    })
end
```

Smoke sensor

Detects smoke presence.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `smoke_detected` *(boolean, read-only)*

  A flag representing the detection of smoke by the sensor.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"smoke_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (**full spec**)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Commands

- `test`

  Starts device self-test. Calls a command callback. No action is actualy performed on the device parameters.

- `reset`

  Resets current device alarms. Calls a command callback. No action is actualy performed on the device parameters.

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "smoke_sensor",
                uuid = "smoke-sensor-module-1",
                name = "Smoke sensor",
                command_callback_function = "onSmokeSensorCommand"
            })
    end
end

function CustomDevice:onSmokeSensorCommand(uuid, cmd, commandParameters,
    changedParameters)
    if cmd == "test" then
        -- send test request to remote device
    elseif cmd == "reset" then
        -- send reset request to remote device
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local smokeDetected = false -- read from device

    self:getModule("smoke-sensor-module-1"):updateProperties({
        smoke_detected = smokeDetected
```

```
    })
  end
```

Reacting to smoke

```lua
local fan = wtp[2]
local siren = wtp[4]
local smokeSensor = custom_device_module[8]

if smokeSensor:changedValue("smoke_detected") and
   smokeSensor:getValue("smoke_detected") then
  fan:call("turn_on")
  siren:call("turn_on")
end
```

CO_2 sensor

Measures CO_2 concentration in the air and sends measurement to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `co2` *(number, read-only)*

  Measured CO_2 value.

  Unit: 1 PPM.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"custom_device_module"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*
- `status` *(string, read-only)*
- `tags` *(string[])*
- `type` *(string, read-only)* = `"co2_sensor"`
- `variant` *(string, read-only)* = `"generic"`
- `visible` *(boolean, read-only)*
- `voice_assistant_device_type` *(string, read-only)*
- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "co2_sensor",
                uuid = "co2-sensor-module-1",
                name = "CO2 sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentCo2ppm = 0 -- read from device

    self:getModule("co2-sensor-module-1"):updateProperties({
        co2 = currentCo2ppm
    })
end
```

Analog input

Measures value from analog input and sends it to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `value` *(number, read-only)*

  Measured analog value.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `unit` *(string)*

  Unit of the value, used for displaying and statistics. Can contain only alphanumeric characters and `-`, `_`. Pattern: `[a-zA-Z0-9\\-_]+`. Example: `mV`

- `value_valid` *(boolean)*

  Indicates if the measured value is valid. Will show error on device if set to `false`

Note: Parameter can be changed only using `updateProperties` from containing custom device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"analog_input"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "analog_input",
                uuid = "analog-input-module-1",
                name = "Analog input",
                data = {
                    unit = "mV"
                }
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local currentValue = 0 -- read from device

    self:getModule("analog-input-module-1"):updateProperties({
        value = currentValue,
        value_valid = (value > 0 and value < 10000)
    })
end
```

Analog Output

Analog output representation. Set desired value to output in device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`*,* `setValue` *and* `updateProperties` *methods (*`updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `value` *(integer)*

  Value of analog output in unit specified in unit property. Minimum value is set in property value_minimum and maximum value is set in property value_maximum.

- `value_minimum` *(integer, read-only)*

  Lower limit of value.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `value_maximum` *(integer, read-only)*

  Upper limit of value.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `unit` *(string, read-only)*

  Unit of the value, used for displaying and statistics. Can contain only alphanumeric characters and `-` , `_` . Pattern: `[a-zA-Z0-9\\-_]+` . Example: `mV`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"analog_output"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties (full spec)

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "analog-output",
                uuid = "analog-output-module-1",
                name = "Analog output ",
                params_callback_function = "onModuleChanged",
                data = {
                    value_minimum = 100,
                    value_maximum = 1500,
                    unit = "mA"
                }
            })
    end
end

-- callback function to update remote module after changes in Sinum
function CustomDevice:onModuleChanged(uuid, param, value)
    if param == "value" then
        -- send value change to remote device
    end
end

-- update module state in event loop with values read from device
function CustomDevice:onEvent(event)
    -- read the data from remote device
```

```lua
    local currentValue = 0 -- read from device

    self:getModule("analog-output-module-1"):updateProperties({
        value = currentValue
    })
end
```

Set the value between 19:00 and 21:00

```lua
if dateTime:changed() then
    if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
      custom_device_module[4]:setValue("value", 800)
    elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
      custom_device_module[4]:setValue("value", 300)
    end
end
```

Air quality sensor

Can check PM (particulate matter): 1.0, 2.5, 4.0, 10.0 concentration in the air.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` and `updateProperties` methods (`updateProperties` only after `getModule` call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `pm1p0` *(number)*

  Measured concentration of PM1.0 value (particulate matter).

  Note: Parameter can be changed only using `updateProperties` from containing custom device. Will not be available when never set.

  Unit: $1\,\mu g/m^3$

- `pm2p5` *(number)*

  Measured concentration of PM2.5 value (particulate matter).

  Note: Parameter can be changed only using `updateProperties` from containing custom device. Will not be available when never set.

  Unit: $1\,\mu g/m^3$

- `pm4p0` *(number)*

  Measured concentration of PM4.0 value (particulate matter).

  Note: Parameter can be changed only using `updateProperties` from containing custom device. Will not be available when never set.

  Unit: $1\,\mu g/m^3$

- `pm10p0` *(number)*

  Measured concentration of PM10.0 value (particulate matter).

  Note: Parameter can be changed only using `updateProperties` from containing custom device. Will not be available when never set.

  Unit: $1\,\mu g/m^3$

- `air_quality` *(string, read-only)*

  Descriptive name for air quality. Based on PM10.0 concetration (if available, if not 4.0 used or 2.5 or 1.0).

Values ($1\,\mu g/m^3$)	Description
≤ 20	`very_good`
21 – 50	`good`

Values ($1\,\mu g/m^3$)	Description
51 – 80	`moderate`
81 – 110	`poor`
111 – 150	`unhealthy`
> 150	`very_unhealthy`

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"aq_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties ([full spec](#))

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "aq_sensor",
                uuid = "aq-sensor-module-1",
                name = "Air quality sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local pm1 = 0 -- read from device
    local pm4 = 0 -- read from device
    local pm2_5 = 0 -- read from device
    local pm10 = 0 -- read from device

    self:getModule("aq-sensor-module-1"):updateProperties({
        pm1p0 = pm1,
        pm2p5 = pm4,
        pm4p0 = pm2_5,
        pm10p0 = pm10
    })
end
```

IAQ sensor

Index of air quality sensor. Calculates air quality index based on various measures like CO_2 or particles level and relative humidity.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`, `setValue` *and* `updateProperties` *methods (* `updateProperties` *only after* `getModule` *call). An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `iaq` *(integer, read-only)*

  Calculated Index of Air Quality.

  Note: Parameter can be changed only using `updateProperties` from containing custom device.

- `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`

Device properties (full spec)

- `class` *(string, read-only)* = `"custom_device_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"iaq_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Custom device modules properties ([full spec](#))

- `uuid` *(string, read-only)*

- `battery` *(integer, read-only)*

- `signal` *(integer, read-only)*

- `software_version` *(string, read-only)*

- `params_callback_function` *(string)*

- `command_callback_function` *(string)*

Examples

Create the module and update properties in custom device code

```lua
function CustomDevice:onInit(reason)
    -- create only when added or imported
    if reason == "add" or reason == "import" then

        self:addModule({
                type = "iaq_sensor",
                uuid = "iaq-sensor-module-1",
                name = "IAQ sensor"
            })
    end
end

function CustomDevice:onEvent(event)
    -- read the data from remote device
    local iaqMeasured = 0 -- read from device

    self:getModule("iaq-sensor-module-1"):updateProperties({
        iaq = iaqMeasured
    })
end
```

SBUS devices

Wired devices.

Device may be added by registration using web application. Can be edited or deleted via [REST API](#) or the web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using `sbus` container e.g. `sbus[6]` gives you access to device with **ID 6**. SBUS devices have global scope and they are visible in all executions contexts.

Common SBUS device properties

- `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.

- `software_version` *(string, read-only)*

  Software name and version description.

Analog input

Analog input sensor representation. Measures value from analog input and sends it to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"analog_input"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Analog output

Analog output representation. Set desired value to output in device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `value` *(integer)*

  Value of analog output in unit specified in unit property. Minimum value is set in property value_minimum and maximum value is set in property value_maximum.

- `value_minimum` *(integer, read-only)*

  Lower limit of value.

- `value_maximum` *(integer, read-only)*

  Upper limit of value.

- `raw_value` *(integer, read-only)*

  Raw value that is sent to analog output. Calculated automatically when changed value, based on minimum and maximum values.

- `raw_value_minimum` *(integer, read-only)*

  Lower limit of raw value.

- `raw_value_maximum` *(integer, read-only)*

  Upper limit of raw value.

- `unit` *(string, read-only)*

  Value unit used by analog output.

  Example: `mV`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"analog_output"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `set_value`

  Sets value of analog output.

  Argument:

  value *(integer)*

Examples

Set value of analog output

```
sbus[3]:call("set_value", 3000)
```

Blind controller

Controller opens and closes a roller shade, tilt blind or pergola.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_opening` *(number)*

  Desired setpoint opening, which device will try to achieve.

  Unit: 1 %

  > **Note**
  >
  > If device doesn't contain `percent_opening_control` label, target opening is limited to 0%, 50% or 100% (only these three).

- `current_opening` *(number, read-only)*

  Current opening value.

  Unit: 1 %

- `window_covering_type` *(string)*

  Defines the type of window covering the controller is connected to. Depending on the value of this parameter, the controller's behavior will change and some parameters may be unavailable.

  > **Note**
  >
  > Can be modified with values in `allowed_window_covering_types` property.

- `allowed_window_covering_types` *(table, read-only)*

  List of available window covering types supported by the controller.

  Possible values: `roller_shade`, `tilt_blind`, `pergola`.

- `lift_control_mode` *(string)*

  Defines the control algorithm of lifting movement. Depending on the value of this parameter, the controller's behavior will change and some parameters may be unavailable.

  > **Note**
  >
  > Can be modified with values in `allowed_lift_control_modes` property.

- `allowed_lift_control_modes` *(table, read-only)*

  List of available lift control modes supported by the controller.

  Possible values: `current_detection`, `fixed_duration`.

Required label: `"percent_tilt_control"`

- `target_tilt` *(number, optional)*

  Desired tilt position.

  Unit: 1 %.

- `current_tilt` *(number, optional, read-only)*

  Current tilt position

  Unit: 1 %.

- `tilt_range` *(number, optional)*

  Determines tilt range.

  Unit: angle (degrees).

  > **Note**
  >
  > Can be modified when: `window_covering_type` is equal to `tilt_blind` or `pergola`.

  > **Note**
  >
  > When `window_covering_type` is equal to `tilt_blind` can be **only** set to 90 or 180, otherwise can be set to 0-180.

Required label: `"has_lift_duration"`

- `full_cycle_duration` *(number, optional)*

  Time required by motor to do full lift cycle from 100% to 0% or 0% to 100% (select larger). Proper full open or full close action is based on this value.

  Unit: seconds.

  > **Note**
  >
  > Can be modified when: `lift_control_mode` is equal to `fixed_duration`.

Required label: `"has_tilt_duration"`

- `tilt_duration` *(number, optional)*

  Time required by motor to do full tilt cycle.

  Unit: ms.

Required label: `"has_tilt_cycle_distance"`

- `tilt_cycle_distance` *(number, optional)*

  Number of motor steps a full tilt cycle takes.

Required label: `"has_motor_running_current_threshold"`

- `motor_running_current_threshold` *(integer, optional)*

  Current threshold that indicates motor is running.

  Unit: mA

Required label: `"has_motor_overload_current_threshold"`

- `motor_overload_current_threshold` *(integer, optional)*

  Current threshold that indicates motor is overloaded / stalled.

  Unit: mA

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `auto`, `fixed`, `off`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `#FF00FF`

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active e.g. motor is working.

  Example: `#FFFF00`

Device properties (full spec)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"blind_controller"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `open`

  Opens a blind to specific value in percent passed in argument.

  > **Note**
  >
  > Command is NOT available when `window_covering_type` is `pergola`.

  Argument:

  opening percentage *(number)*

- `up`

  Fully opens a blind.

  > **Note**
  >
  > Command is NOT available when `window_covering_type` is `pergola`.

- `down`

  Fully closes a blind.

  > **Note**
  >
  > Command is NOT available when `window_covering_type` is `pergola`.

- `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

```lua
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

```lua
if dateTime:changed() then
   if dateTime:getHours() == 12 and dateTime:getMinutes() == 0 then
     sbus[3]:call("open", 50)
   end
end
```

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

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `buttons_count` *(number, read-only)*

  Count of physical buttons.

- `action` *(string, read-only)*

  Last action performed by user. Lua patterns of available action kinds:

  `"button_(%d+)_clicked_(%d+)_times"`
    Given button has been clicked specified number of times.
    Examples:
      ◦ `"button_2_clicked_1_times"` — Single click was detected.
      ◦ `"button_1_clicked_8_times"` — A sequence of eight clicks was detected.
  `"button_(%d+)_hold_started"`
    Given button is being held down, starting from now.
    Example: `"button_1_hold_started"` — User just started holding the button down.
  `"button_(%d+)_held_(%d+)_seconds"`
    Given button has been released, after specified time of being held down.
    Example: `"button_1_held_8_seconds"` — User just released a button after holding it down for eight seconds.

- `endpoints_used` *(table, read-only)*

  List of used communication endpoints for all handled buttons.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"button"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `emit`

  Emits button action event by software.

  Argument:

  packed arguments *(table)*:

  - `button_index` - *(number, required)* index of the button that the action event concerns, minimum: 1, maximum: value of `buttons_count`
  - `type` - *(string, required)* action that will be emitted, possible values: `click`, `hold_start`, `hold_end`
  - `duration` - *(number, optional)* duration of the hold in seconds or amount of clicks, ignored if the type is `hold_start`, minimum: 1, maximum: 600
  - `set` - *(number, optional)* the current button set (channel) that the action applies to (changes `button_set.current`), minimum: 1, maximum: value of `button_set.available_count`

Examples

Turn on lights when button clicked once

```lua
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

```lua
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
```

Emit button action

```lua
local button = sbus[9]

-- Emit 2 clicks on button 1
button:call("emit", { button_index = 1, type = "click", duration = 2 })

-- Emit hold start on button 2, change set (channel) to 3
button:call("emit", { button_index = 2, type = "hold_start", set = 3 })

-- Emit hold end after 10 secs on button 2 in set 3 (channel 3)

  -- NOTE: if the previous action was emitted by button in set 3, all subsequent actions w
-- until the set is changed by remote or by using `set` property of the command
button:call("emit", { button_index = 2, type = "hold_end", duration = 10 })
```

CO_2 sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `co2` *(number, read-only)*

  Measured CO_2 value. Unit: PPM.

Device properties (full spec)

- `class` *(string, read-only)* = `"sbus"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*
- `status` *(string, read-only)*
- `tags` *(string[])*
- `type` *(string, read-only)* = `"button"`
- `variant` *(string, read-only)* = `"generic"`
- `visible` *(boolean, read-only)*
- `voice_assistant_device_type` *(string, read-only)*
- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Dimmer

Device that controls light intensity of output LED.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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`). If it is set to 0, the dimmer will be turned off.

  Unit: Unit: 1 %

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `"auto"`, `"fixed"`, `"off"`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `"#FF00FF"`

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active e.g. motor is working.

  Example: `"#FFFF00"`

Required label: `"has_backlight_brightness_sensor"`

- `ambient_light_intensity` *(number, read-only)*

  Measured ambient light intensity in percent.

Required label: `"has_output_temperature_sensor"`

- `output_temperature` *(number, read-only)*

  Measured temperature value of hardware output circuit. Unit: °C with one decimal number, multiplied by 10.

Required label: `"level_calibration_support"`

- `level_calibration.minimum` *(number)*

  Absolute minimum level that can be set on the device. Could not be larger than maximum. `target_level` will be internally scaled between `level_calibration.minimum` and `level_calibration.maximum`. Value range: $\langle 0; 255 \rangle$

- `level_calibration.maximum` *(number)*

  Absolute maximum level that can be set on the device. Could not be less than minimum. `target_level` will be internally scaled between `level_calibration.minimum` and `level_calibration.maximum`. Value range: $\langle 0; 255 \rangle$

Required label: `"switch_mode_support"`

- `switch_mode` *(string)*

  Depending on the value of this parameter, the swiches behavior will change. Can be modified with values in `allowed_switch_modes` property.

- `allowed_switch_modes` *(table, read-only)*

  List of available switch modes supported by the device.

  Possible combination of values:

  - `single_monostable` — (mode 1) Controls connected level control using single monostable input.

  - `single_bistable` — (mode 2) Controls connected level control using single bistable input.

    Click: toggle between SAVED and OFF level
    Hold: increase/decrease level

  - `up_and_down` — (mode 3) Controls connected level control using separate up and down inputs.

  - `single_monostable_dim_hotkey` — (mode 4) Controls connected level control using single monostable input.

    Click: toggle between SAVED and OFF level

Hold: increase level
Click then hold hotkey: decreases level

- `single_monostable_dim_saved_hotkey` — (mode 5) Controls connected level control using single monostable input.

  Click: toggle between MAX and OFF level
  Hold: increase level
  Click then hold hotkey: decreases level
  Double click hotkey: goes to SAVED level

Device properties (full spec)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"dimmer"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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: it will always take constant time to move from 1% → 100%, 50% → 100%, 10% → 1% etc.

```lua
local dimmer = sbus[4]
local button = sbus[98]
local fadeTime = 50 -- 5s / 5000ms

if button:changedValue("action") then
  local action = button:getValue("action")

  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 1% from current target level
    dimmer:call("set_level", {   1, fadeTime })
  elseif action:find("button_1_held_") or action:find("button_2_held_") then
    -- stop current moving action
    dimmer:call("stop")
  end
end
```

Dim or brighten lights while a button is pressed (advanced version)

This solution uses constant dimming rate instead of constant time.

```lua
local buttonUp, buttonDown = sbus[86], sbus[87]
local dimmer = sbus[126]

-- maximum fading time in 0.1s
local fadeTime = 50

-- calculate duration proportional to level difference
local function constantRateMove(currentLevel, desiredLevel)
  local diff = math.abs(desiredLevel - currentLevel)
  local time = utils.math:scale(0, 100, 1, fadeTime, diff)

  -- this table can be used directly as 'set_level' command argument
  return { desiredLevel, math.floor(time) }
end

if buttonUp:changedValue('action') then
  local action = buttonUp:getValue('action')
  if action == 'button_1_hold_started' then
    -- first button pressed, start moving towards full brightness
    local currentLevel = dimmer:getValue('target_level')
    dimmer:call('set_level', constantRateMove(currentLevel, 100))
  elseif action:find('button_1_held_', 1, true) then
    -- button released, stop the transition
    dimmer:call('stop')
  end
end

if buttonDown:changedValue('action') then
  local action = buttonDown:getValue('action')
  if action == 'button_1_hold_started' then
    -- second button pressed, start moving towards minimal brightness
    local currentLevel = dimmer:getValue('target_level')
    dimmer:call('set_level', constantRateMove(currentLevel,   1))
  elseif action:find('button_1_held_', 1, true) then
    -- button released, stop the transition
    dimmer:call('stop')
  end
end
```

Energy meter

Energy meter is a device which can track and count consumed energy (total so far and daily) and sense voltage/current and active power.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `current` *(number, read-only)*

  Recent AC current measument.

  Unit: 1 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: 1 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: 1 mW.

- `energy_consumed_today` *(number, read-only)*

  Sum of energy used today.

  Unit: 1 Wh.

- `energy_consumed_yesterday` *(number, read-only)*

  Sum of energy used yesterday.

  Unit: 1 Wh.

- `energy_consumed_total` *(number, read-only)*

  Total sum of energy used Unit: 1 Wh.

Device properties (full spec)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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

```lua
-- 2.5 kW = 2500 W = 2500000 mW
local threshold = 2500000
if sbus[10]:changedValue("active_power")
then
  if sbus[10]:getValue("active_power") > treshold
  then
    notify:error("Power usage too high!", "Check your device!")
  end
end
```

Flood sensor

Device that detects water leak on flat surfaces.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue`
and `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting
wrong value type will cause a script error.*

- `flood_detected` *(boolean, read-only)*

  A flag representing the detection of flood / water leak by the sensor.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"flood_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (**full spec**)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Examples

Catching alarms

```lua
if sbus[5]:changedValue("flood_detected") and sbus[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

```lua
local valve, siren, floodSensor = sbus[1], sbus[2], sbus[3]

if floodSensor:changedValue("flood_detected") and
   floodSensor:getValue("flood_detected")
then
  valve:call("turn_off")
  siren:call("turn_on")
end
```

Humidity sensor

Humidity sensor. Measures humidity and sends measurement to central unit. Can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `humidity` *(number, read-only)*

  Measured humidity value.

  Unit: rH% with one decimal number, multiplied by 10 (0.1 %).

Device properties (**full spec**)

- `class` *(string, read-only)* = `"sbus"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*
- `status` *(string, read-only)*
- `tags` *(string[])*
- `type` *(string, read-only)* = `"humidity_sensor"`
- `variant` *(string, read-only)* = `"generic"`
- `visible` *(boolean, read-only)*
- `voice_assistant_device_type` *(string, read-only)*
- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

IAQ sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 value	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`

Device properties (full spec)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"iaq_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Light sensor

Light sensor measures light illuminance in lux and sends measurement to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `illuminance` *(number, read-only)*

  Measured light illuminance value.

  Unit: 1 lx

Device properties (full spec)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"light_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Motion sensor

Motion sensor based on custom configuration checks whether motion was detected.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `enabled` *(boolean)*

  Enable or disable sensor. e.g. 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.

  > **Note**
  >
  > The value will remain `true` all the time when subsequent motion detections occur until motion stops.
  > This parameter doesn't emit event when switch from `true` to `true` happens (subsequent motion detections). If you need to observe such action, you need to use `time_since_motion` parameter and check if `time_since_motion` equals to `0`.

- `time_since_motion` *(number, read-only)*

  Time since last motion detected. Value of -1 means there wasn't any motion since last system startup.

> **Note**
>
> The value will be `0` for each detected move, even if the previous one has not yet finished.

Unit: seconds.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"motion_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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)*, at least 1 s.

Examples

Catching motion events

```lua
if sbus[4]:changedValue("motion_detected") then
  print("someone is moving aroung!")
end
```

```lua
if
  sbus[4]:changedValue("time_since_motion")
  and
  sbus[4]:getValue("time_since_motion") == 0
then
  print("someone is moving aroung!")
end
```

Delayed action

```lua
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

```lua
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

```lua
if sbus[4]:changedValue("time_since_motion") then
  if sbus[4]:getValue("time_since_motion") == 0 then
    sbus[60]:setValue("state", true)
    sbus[60]:setValueAfter("state", false, 5 * 60)
  end
end
```

Reconfigure thermostat when motion detected

```lua
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
```

Pressure sensor

Pressure sensor measures pressure and sends measurement to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `pressure` *(number, read-only)*

  Measured pressure value.

  Unit: hPa with one decimal number, multiplied by 10 (10 Pa).

- `altitude` *(integer)*

  Setting the altitude compensates the atmospheric pressure reading to the pressure at mean sea level, that is normally given in weather reports. Possible range of altitude: `0 - 8849`

  Unit: 1 msl, meters above sea level for your location.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"pressure_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Pulse width modulation

Pulse width modulation device representation. Set desired signal in device output. User can set desired frequency of signal and duty cycle i.e. the amount of time the digital signal is in the "active" state relative to the period of the signal.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `frequency` *(integer)*

  Frequency of pulse width modulation signal. Minimum value is set in property `frequency_minimum` and maximum value is set in property `frequency_maximum`.

  Unit: 1 Hz

- `frequency_minimum` *(integer, read-only)*

  Lower limit of frequency.

- `frequency_maximum` *(integer, read-only)*

  Upper limit of frequency.

- `raw_value` *(integer)*

  Pulse width modulation duty cycle. The amount of time the digital signal is in the "active" state relative to the period of the signal.

  Unit: 1 %

Commands

- `set_frequency`

  Calls PWM device to set output frequency.

  Argument:

  frequency *(integer)*

- `set_duty_cycle`

  Calls PWM device to set output duty cycle.

  Argument:

  duty_cycle *(integer)*

Examples

Set frequency and duty cycle of PWM

```
sbus[3]:call("set_frequency", 3000)
sbus[3]:call("set_duty_cycle", 40)
```

RGB controller

Device that controlls color and light intensity of output LED.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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: 1 %

- `led_color` *(string, read-only)*

  HTML/Hex RGB color that device will set on its output led strip.

  Example: `"#00ff7f"`

- `white_temperature` *(number, read-only)*

  White temperature that device will set on its output led strip.

  Unit: 1 K

- `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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"rgb_controller"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `turn_on`

  Turns on output.

  Argument

  Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `turn_off`

  Turns off output.

  Argument

  Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `toggle`

  Changes state to opposite.

  Argument

  Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `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`.

- `stop`

  Calls RGB controller to stop current moving action. Does nothing if no action is in progress.

Examples

Turn on light to specific color at 7:00 and turn off at 8:00

```lua
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

```lua
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

```lua
local rgb = sbus[79]
local animation_id = 2

rgb:call("activate_animation", { id = animation_id })
```

Stop active animation

```lua
local rgb = sbus[79]
rgb:call("stop_animation")
```

Activate an animation by ID when device state changes

```lua
local rgb = sbus[79]
local animation_id = 3

if wtp[3]:changedValue("state") then
  rgb:call("activate_animation", { id = animation_id })
end
```

Relay

Execution module that changes state depending on the control signal. Relay can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(boolean)*

  State of the output. On/Off.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `timeout` *(number)*

  Protection functionality, that will set device state to off if there are communication problems.

  Unit: 1 min

- `timeout_enabled` *(boolean)*

  Parameter that indicates if timeout functionality is enabled.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `inverted` *(boolean)*

  Indicates if physical state of relay should be the inversion of state shown in application.

- `time_since_state_change` *(number, read-only)*

  Time since last relay state change.

  Unit: 1 s

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `"auto"`, `"fixed"`, `"off"`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `"#FF00FF"`

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active e.g. motor is working.

  Example: `"FFFF00"`

Required label: `"relay_startup_state_support"`

- `startup_state` *(string)*

  State of output that should be set on device after power restart. Available values: `"off"`, `"on"`, `"previous"`

  Cannot be changed if device is assigned to thermostat or thermostat output group (virtual contact) / (has label `managed_by_thermostat` or `managed_by_tog`). Cannot be changed when relay `work_mode` is set to `alarm_siren`.

- `work_mode` *(string)*

  Relay work mode. One of: `standard`, `alarm_siren`.

Required label: `"trigger_signal_config_support"`

- `trigger_signal_type` *(string)*

  Available values:

  "impulse"
  > Monostable trigger — impulse signal toggles the output state

  "state_change"
  > Bistable trigger — when trigger state changes, output will be set equal to trigger signal

- `allowed_trigger_signal_types` *(table, read-only)*

  List of available trigger signal types supported by the device.

  Possible combination of values:

  - `impulse`
  - `state_change`

Device properties (full spec)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"relay"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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

```lua
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
```

Temperature regulator

Temperature regulator notifies when desired temperature is reached in room. Can be assigned to virtual thermostat in web application.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °C

- `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 doesn't 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_minimum` *(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: 0.1 °C

- `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: 0.1 °C

- `target_temperature_reached` *(boolean)*

  Controls device's algorithm state indicator (available on some regulators). e.g. 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 or heat pump manager (label `managed_by_thermostat` and `managed_by_heat_pump_manager` not present).

  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.

Required label: `"user_menu_lock_support"`

- `user_menu_lock.enabled` *(boolean)*

  Indicates that it is required to enter pin code to access device's user menu.

- `user_menu_lock.pin_code` *(string)*

  Pin Code to access device's user menu. Has to be longer or equal to `user_menu_lock.pin_code_length_minimum` and shorter or equal to `user_menu_lock.pin_code_length_maximum` and contains characters from `user_menu_lock.allowed_characters`.

- `user_menu_lock.pin_code_length_minimum` *(integer, read-only)*

  Minimum length of user menu pin code.

- `user_menu_lock.pin_code_length_maximum` *(integer, read-only)*

  Maximum length of user menu pin code.

- `user_menu_lock.allowed_characters` *(integer, read-only)*

  Allowed characters of user menu pin code.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_regulator"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (**full spec**)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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 a 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 a 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 a thermostat.

Examples

Raise target temperature between 15:00 and 20:00

```lua
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
```

Temperature sensor

Temperature sensor. Measures temperature and sends measurement to central unit. Can be assigned to virtual thermostat in web application as room or floor sensor.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `temperature` *(number, read-only)*

  Measured 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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_regulator"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Two state input sensor

Boolean input sensor checks input state and send it to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"two_state_input_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Valve

Valve representation. Allows user to read and modify valve parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `enabled` *(boolean)*

  Indicates if valve is turned on.

- `state` *(string, read-only)*

  Valve current working state. One of: `"off"`, `"calibration"`, `"return_protection"`, `"heat_source_protection"`, `"work"`, `"blockade"`, `"alarm"`, `"manual_work"`, `"synchronization"`

- `open_percent` *(integer, read-only)*

  Current opening percent. Unit: %, with 2 decimal numbers, multiplied by 100.

- `temperature_valve` *(integer, read-only)*

  Current valve temperature. Unit: °C with one decimal number, multiplied by 10.

- `opening_direction` *(string)*

  Valve opening direction. One of: `"left"`, `"right"`

- `max_single_move` *(integer)*

  The maximum percentage that the valve can open in one interval. Unit: % Min: 1 Max: 99

- `minimal_opening` *(integer)*

  Minimum opening below which the valve cannot close further in its normal operation. Unit: % Min: 1 Max: 99

- `pause_time` *(integer)*

  The pause time the valve takes after opening/closing during normal operation. Unit: second Min: 1 Max: 900

- `hysteresis` *(integer)*

  Valve temperature hysteresis. Unit: °C with one decimal number, multiplied by 10. Min: 2 Max: 20

- `proportional_gain` *(integer)*

  Specifies the response strength to temperature error. Min: 1 Max: 10

- `opening_time` *(integer)*

  The time it takes for the valve to open. Unit: seconds Min: 10 Max: 1500

- `calibration_time` *(integer)*

  The time it takes for the valve to open in calibration. Unit: seconds Min: 10 Max: 1500

Required label: `"has_opening_time_mode"`

- `opening_time_mode` *(string)*

  Determines whether the calibration time differs from valve opening time in work. One of: `"linked"`, `"splitted"`.

- `floor_overheating_temperature` *(integer)*

  The temperature after which the valve will close in floor heating mode. Unit: °C with one decimal number, multiplied by 10. Min: 250 Max: 550

Required label: `"heat_source_sensor_available"`

- `heat_source_protection` *(boolean)*

  Turns on/off the heating source protection. Requires a source temperature sensor.

- `valve_closes_from_pump_threshold` *(boolean)*

  Closes the valve when the pump operating conditions are not met.

- `heat_source_protection_temperature` *(boolean)*

  The temperature after which the valve will open if heat source protection is enabeld. Unit: °C with one decimal number, multiplied by 10. Min: 600 Max: 1000

Required label: `"return_sensor_available"`

- `return_protection` *(boolean)*

  Turns on/off the return protection. Requires a return temperature sensor.

- `return_protection_temperature` *(integer)*

  The temperature below which the valve will close if return protection is enabeld. Unit: °C with one decimal number, multiplied by 10. Min: 100 Max: 800

- `central_heating_target_temperature` *(integer)*

  Temperature setpoint that the device should maintain in heating as central heating valve. Unit: °C with one decimal number, multiplied by 10. Min: 100 Max: 990

- `floor_heating_target_temperature` *(integer)*

Temperature setpoint that the device should maintain in heating as floor heating valve. Unit: °C with one decimal number, multiplied by 10. Min: 100 Max: 500

- `cooling_target_temperature` *(integer)*

Temperature setpoint that the device should maintain in cooling mode. Unit: °C with one decimal number, multiplied by 10. Min: 50 Max: 990

- `valve_type` *(string)*

Mixing valve type. One of: `"central_heating"`, `"floor_heating"`.

- `work_mode` *(string)*

Specify whether valve is used to heating or cooling. One of: `"heating"`, `"cooling"`.

- `emergency_behaviour` *(string)*

Specify valve behavior after communication loss. One of: `"close"`, `"open"`, `"safe_opening_percentage"`, `"standalone_work"`.

- `blockade` *(boolean)*

Specify whether valve blockade is on or off.

- `blockade_reasons` *(array of string, read-only)*

Specify the reason why the valve entered blockade state. Possible values: `"floor_overheat"`, `"floor_heating_disabled"`, `"summer_mode"`, `"boiler_priority"`, `"device_standby_mode"`, `"closing_if_threshold_not_met"`, `"room_regulator"`, `"external_blockade"`.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"common_valve"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `calibration`

  Start valve calibration.

Valve Pump

Valve pump representation. Allows user to read and modify valve parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(boolean, read-only)*

  Specify whether pump is working or not.

- `relay_control_mode` *(string)*

  Specify whether the pump relay belongs to the internal valve algorithm or external entity.
  One of: `"internal"`, `"external"`.

- `emergency_behaviour` *(string)*

  Specify pump behavior after communication loss if the pump relay control mode is internal.
  One of: `"turn_off"`, `"turn_on"`.

- `antistop` *(boolean)*

  Turns on the pump for 5 hours every 10 days if the pump relay control mode is internal.

Required label: `"heat_source_sensor_available"`

- `work_mode` *(string)*

  Specify pump work mode if the pump relay control mode is internal.
  One of: `"always_on"`, `"always_off"`, `"temperature_control"`.
  Label is required to set `temperature_control` mode.

- `temperature_threshold_heating` *(integer)*

  The pump will operate after exceeding the temperature threshold.
  Unit: 0.1 °C.
  Range: 100 – 800 (10.0 °C – 80.0 °C).

- `temperature_threshold_cooling` *(integer)*

  The pump will operate below the temperature threshold.
  Unit: 0.1 °C.
  Range: 100 – 550 (10.0 °C – 55.0 °C).

> **Required label:** `"pump_work_in_calibration_support"`
>
> - `pump_work_in_calibration` *(boolean)*
>
>   Allows the pump to operate in valve calibration.

- `blockade` *(boolean)*

  Specify whether pump blockade is on or off.

Device properties (full spec)

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"valve_pump"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Valve Analog PID

0–10V valve with PID calculation representation. Allows user to read and modify valve parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `heating_target_temperature` *(integer)*

  Temperature setpoint that the device should maintain in heating.
  Unit: 0.1 °C.
  Min: from parameter `heating_target_temperature_minimum`.

  Max: from parameter `heating_target_temperature_maximum`.

- `heating_target_temperature_minimum` *(integer, read-only)*

  Minimum value of temperature setpoint in heating.
  Unit: 0.1 °C.

- `heating_target_temperature_maximum` *(integer, read-only)*

  Maximum value of temperature setpoint in heating.
  Unit: 0.1 °C.

- `cooling_target_temperature` *(integer)*

  Temperature setpoint that the device should maintain in cooling.
  Unit: 0.1 °C.
  Min: from parameter `heating_target_temperature_minimum`

  Max: from parameter `heating_target_temperature_maximum`

- `cooling_target_temperature_minimum` *(integer, read-only)*

  Minimum value of temperature setpoint in cooling.
  Unit: 0.1 °C.

- `cooling_target_temperature_maximum` *(integer, read-only)*

  Maximum value of temperature setpoint in cooling.
  Unit: 0.1 °C.

- `work_mode` *(string)*

  Specify valve work mode. One of: `"heating"`, `"cooling"`, `"off"`.

- `current_temperature` *(integer, read-only)*

  Current valve water temperature.
  Unit: 0.1 °C.

- `proportional_gain` *(integer)*

  Proportional gain parameter for PID calculation.
  Unit: 0.1 (no unit, parameter with one decimal number, multiplied by 10.)
  Range: 0 – 100 (0.0 – 10.0)

- `integral_gain` *(integer)*

  Integral gain parameter for PID calculation.
  Unit: 0.1 (no unit, parameter with one decimal number, multiplied by 10.)
  Range: 0 – 100 (0.0 – 10.0)

- `derivative_gain` *(integer)*

  Derivative gain parameter for PID calculation.
  Unit: 0.1 (no unit, parameter with one decimal number, multiplied by 10.)
  Range: 0 – 100 (0.0 – 10.0)

- `calculated_pid_value` *(integer, read-only)*

  Current calculated PID value.

- `output_value` *(integer, read-only)*

  Current valve output voltage.
  Unit: 1 mV

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"sbus"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"valve_analog_pid"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SBUS device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `reset_pid`

  Resets the PID calculations.

SLINK devices

Wireless devices.

Device may be added by registration using web application. Can be edited or deleted via
[REST API](#) or the web application served through the central unit server.

Property modification is possible via REST API, web app or directly from scripts using
`slink` container e.g. `slink[6]` gives you access to device with **ID 6**. SLINK devices have
global scope and they are visible in all executions contexts.

Common SLINK device properties

- `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.

- `software_version` *(string, read-only)*

  Software name and version description.

Analog input

Analog input sensor representation. Measures value from analog input and sends it to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"analog_input"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Analog output

Analog output representation. Set desired value to output in device.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `value` *(integer)*

  Value of analog output in unit specified in unit property. Minimum value is set in property value_minimum and maximum value is set in property value_maximum.

- `value_minimum` *(integer, read-only)*

  Lower limit of value.

- `value_maximum` *(integer, read-only)*

  Upper limit of value.

- `raw_value` *(integer, read-only)*

  Raw value that is sent to analog output. Calculated automatically when changed value, based on minimum and maximum values.

- `raw_value_minimum` *(integer, read-only)*

  Lower limit of raw value.

- `raw_value_maximum` *(integer, read-only)*

  Upper limit of raw value.

- `unit` *(string, read-only)*

  Value unit used by analog output.

  Example: `mV`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"analog_output"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `set_value`

  Sets value of analog output.

  Argument:

  value *(integer)*

Examples

Set value of analog output

```
slink[3]:call("set_value", 3000)
```

Blind controller

Controller opens and closes a roller shade, tilt blind or pergola.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `target_opening` *(number)*

  Desired setpoint opening, which device will try to achieve.

  Unit: 1 %

  > **Note**
  >
  > If device doesn't contain `percent_opening_control` label, target opening is limited to 0%, 50% or 100% (only these three).

- `current_opening` *(number, read-only)*

  Current opening value.

  Unit: 1 %

- `window_covering_type` *(string)*

  Defines the type of window covering the controller is connected to. Depending on the value of this parameter, the controller's behavior will change and some parameters may be unavailable.

  > **Note**
  >
  > Can be modified with values in `allowed_window_covering_types` property.

- `allowed_window_covering_types` *(table, read-only)*

  List of available window covering types supported by the controller.

  Possible values: `roller_shade`, `tilt_blind`, `pergola`.

- `lift_control_mode` *(string)*

  Defines the control algorithm of lifting movement. Depending on the value of this parameter, the controller's behavior will change and some parameters may be unavailable.

  > **Note**
  >
  > Can be modified with values in `allowed_lift_control_modes` property.

- `allowed_lift_control_modes` *(table, read-only)*

  List of available lift control modes supported by the controller.

  Possible values: `current_detection`, `fixed_duration`.

Required label: `"percent_tilt_control"`

- `target_tilt` *(number, optional)*

  Desired tilt position.

  Unit: 1 %.

- `current_tilt` *(number, optional, read-only)*

  Current tilt position

  Unit: 1 %.

- `tilt_range` *(number, optional)*

  Determines tilt range.

  Unit: angle (degrees).

  > **Note**
  >
  > Can be modified when: `window_covering_type` is equal to `tilt_blind` or `pergola`.

  > **Note**
  >
  > When `window_covering_type` is equal to `tilt_blind` can be **only** set to 90 or 180, otherwise can be set to 0-180.

Required label: `"has_lift_duration"`

- `full_cycle_duration` *(number, optional)*

  Time required by motor to do full lift cycle from 100% to 0% or 0% to 100% (select larger). Proper full open or full close action is based on this value.

  Unit: seconds.

  > **Note**
  >
  > Can be modified when: `lift_control_mode` is equal to `fixed_duration`.

Required label: `"has_tilt_duration"`

- `tilt_duration` *(number, optional)*

  Time required by motor to do full tilt cycle.

  Unit: ms.

Required label: `"has_tilt_cycle_distance"`

- `tilt_cycle_distance` *(number, optional)*

  Number of motor steps a full tilt cycle takes.

Required label: `"has_motor_running_current_threshold"`

- `motor_running_current_threshold` *(integer, optional)*

  Current threshold that indicates motor is running.

  Unit: mA

Required label: `"has_motor_overload_current_threshold"`

- `motor_overload_current_threshold` *(integer, optional)*

  Current threshold that indicates motor is overloaded / stalled.

  Unit: mA

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `auto`, `fixed`, `off`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `#FF00FF`

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active e.g. motor is working.

  Example: `#FFFF00`

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"blind_controller"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `open`

  Opens a blind to specific value in percent passed in argument.

  > **Note**
  >
  > Command is NOT available when `window_covering_type` is `pergola`.

  Argument:

  opening percentage *(number)*

- `up`

  Fully opens a blind.

  > **Note**
  >
  > Command is NOT available when `window_covering_type` is `pergola`.

- `down`

  Fully closes a blind.

  > **Note**
  >
  > Command is NOT available when `window_covering_type` is `pergola`.

- `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
   slink[3]:call("up")
elseif event.type == "sunset" then
   slink[3]:call("down")
end
```

Set blind to half-open at noon

```
if dateTime:changed() then
   if dateTime:getHours() == 12 and dateTime:getMinutes() == 0 then
     slink[3]:call("open", 50)
   end
end
```

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

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `buttons_count` *(number, read-only)*

  Count of physical buttons.

- `action` *(string, read-only)*

  Last action performed by user. Lua patterns of available action kinds:

  `"button_(%d+)_clicked_(%d+)_times"`
  Given button has been clicked specified number of times.
  Examples:
  - `"button_2_clicked_1_times"` — Single click was detected.
  - `"button_1_clicked_8_times"` — A sequence of eight clicks was detected.

  `"button_(%d+)_hold_started"`
  Given button is being held down, starting from now.
  Example: `"button_1_hold_started"` — User just started holding the button down.

  `"button_(%d+)_held_(%d+)_seconds"`
  Given button has been released, after specified time of being held down.
  Example: `"button_1_held_8_seconds"` — User just released a button after holding it down for eight seconds.

- `endpoints_used` *(table, read-only)*

  List of used communication endpoints for all handled buttons.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"button"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `emit`

  Emits button action event by software.

  Argument:

  packed arguments *(table)*:

  - `button_index` - *(number, required)* index of the button that the action event concerns, minimum: 1, maximum: value of `buttons_count`
  - `type` - *(string, required)* action that will be emitted, possible values: `click`, `hold_start`, `hold_end`
  - `duration` - *(number, optional)* duration of the hold in seconds or amount of clicks, ignored if the type is `hold_start`, minimum: 1, maximum: 600
  - `set` - *(number, optional)* the current button set (channel) that the action applies to (changes `button_set.current`), minimum: 1, maximum: value of `button_set.available_count`

Examples

Turn on lights when button clicked once

```lua
local button = slink[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

```lua
local button = slink[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
```

Emit button action

```lua
local button = slink[9]

-- Emit 2 clicks on button 1
button:call("emit", { button_index = 1, type = "click", duration = 2 })

-- Emit hold start on button 2, change set (channel) to 3
button:call("emit", { button_index = 2, type = "hold_start", set = 3 })

-- Emit hold end after 10 secs on button 2 in set 3 (channel 3)

  -- NOTE: if the previous action was emitted by button in set 3, all subsequent actions w
-- until the set is changed by remote or by using `set` property of the command
button:call("emit", { button_index = 2, type = "hold_end", duration = 10 })
```

CO_2 sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `co2` *(number, read-only)*

  Measured CO_2 value. Unit: PPM.

Device properties (full spec)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"button"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Dimmer

Device that controls light intensity of output LED.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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`). If it is set to 0, the dimmer will be turned off.

  Unit: Unit: 1 %

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `"auto"`, `"fixed"`, `"off"`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `"#FF00FF"`

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active e.g. motor is working.

  Example: `"#FFFF00"`

Required label: `"has_backlight_brightness_sensor"`

- `ambient_light_intensity` *(number, read-only)*

  Measured ambient light intensity in percent.

Required label: `"has_output_temperature_sensor"`

- `output_temperature` *(number, read-only)*

Measured temperature value of hardware output circuit. Unit: °C with one decimal number, multiplied by 10.

Required label: `"level_calibration_support"`

- `level_calibration.minimum` *(number)*

Absolute minimum level that can be set on the device. Could not be larger than maximum. `target_level` will be internally scaled between `level_calibration.minimum` and `level_calibration.maximum`. Value range: ⟨0; 255⟩

- `level_calibration.maximum` *(number)*

Absolute maximum level that can be set on the device. Could not be less than minimum. `target_level` will be internally scaled between `level_calibration.minimum` and `level_calibration.maximum`. Value range: ⟨0; 255⟩

Required label: `"switch_mode_support"`

- `switch_mode` *(string)*

Depending on the value of this parameter, the swiches behavior will change. Can be modified with values in `allowed_switch_modes` property.

- `allowed_switch_modes` *(table, read-only)*

List of available switch modes supported by the device.

Possible combination of values:

  - `single_monostable` — (mode 1) Controls connected level control using single monostable input.

  - `single_bistable` — (mode 2) Controls connected level control using single bistable input.

    Click: toggle between SAVED and OFF level
    Hold: increase/decrease level

  - `up_and_down` — (mode 3) Controls connected level control using separate up and down inputs.

  - `single_monostable_dim_hotkey` — (mode 4) Controls connected level control using single monostable input.

    Click: toggle between SAVED and OFF level

Hold: increase level
Click then hold hotkey: decreases level

○ `single_monostable_dim_saved_hotkey` — (mode 5) Controls connected level control using single monostable input.

Click: toggle between MAX and OFF level
Hold: increase level
Click then hold hotkey: decreases level
Double click hotkey: goes to SAVED level

Device properties (full spec)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"dimmer"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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
    slink[4]:call("turn_on")
  elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
    slink[4]:call("turn_off")
  end
end
```

Set the light intensity to 75% during 2 minutes

```
slink[4]:call("set_level", { 75, 1200 })
```

Dim or brighten lights while button is pressed (simple version)

Solution drawback: it will always take constant time to move from 1% → 100%, 50% → 100%, 10% → 1% etc.

```lua
local dimmer = slink[4]
local button = slink[98]
local fadeTime = 50 -- 5s / 5000ms

if button:changedValue("action") then
  local action = button:getValue("action")

  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 1% from current target level
    dimmer:call("set_level", {   1, fadeTime })
  elseif action:find("button_1_held_") or action:find("button_2_held_") then
    -- stop current moving action
    dimmer:call("stop")
  end
end
```

Dim or brighten lights while a button is pressed (advanced version)

This solution uses constant dimming rate instead of constant time.

```lua
local buttonUp, buttonDown = slink[86], slink[87]
local dimmer = slink[126]

-- maximum fading time in 0.1s
local fadeTime = 50

-- calculate duration proportional to level difference
local function constantRateMove(currentLevel, desiredLevel)
  local diff = math.abs(desiredLevel - currentLevel)
  local time = utils.math:scale(0, 100, 1, fadeTime, diff)

  -- this table can be used directly as 'set_level' command argument
  return { desiredLevel, math.floor(time) }
end

if buttonUp:changedValue('action') then
  local action = buttonUp:getValue('action')
  if action == 'button_1_hold_started' then
    -- first button pressed, start moving towards full brightness
    local currentLevel = dimmer:getValue('target_level')
    dimmer:call('set_level', constantRateMove(currentLevel, 100))
  elseif action:find('button_1_held_', 1, true) then
    -- button released, stop the transition
    dimmer:call('stop')
  end
end

if buttonDown:changedValue('action') then
  local action = buttonDown:getValue('action')
  if action == 'button_1_hold_started' then
    -- second button pressed, start moving towards minimal brightness
    local currentLevel = dimmer:getValue('target_level')
    dimmer:call('set_level', constantRateMove(currentLevel,   1))
  elseif action:find('button_1_held_', 1, true) then
    -- button released, stop the transition
    dimmer:call('stop')
  end
end
```

Energy meter

Energy meter is a device which can track and count consumed energy (total so far and daily) and sense voltage/current and active power.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `current` *(number, read-only)*

  Recent AC current measument.

  Unit: 1 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: 1 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: 1 mW.

- `energy_consumed_today` *(number, read-only)*

  Sum of energy used today.

  Unit: 1 Wh.

- `energy_consumed_yesterday` *(number, read-only)*

  Sum of energy used yesterday.

  Unit: 1 Wh.

- `energy_consumed_total` *(number, read-only)*

  Total sum of energy used Unit: 1 Wh.

Device properties (full spec)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"energy_meter"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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

```lua
-- 2.5 kW = 2500 W = 2500000 mW
local threshold = 2500000
if slink[10]:changedValue("active_power")
then
  if slink[10]:getValue("active_power") > treshold
  then
    notify:error("Power usage too high!", "Check your device!")
  end
end
```

Flood sensor

Device that detects water leak on flat surfaces.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `flood_detected` *(boolean, read-only)*

  A flag representing the detection of flood / water leak by the sensor.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"flood_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Examples

Catching alarms

```
if slink[5]:changedValue("flood_detected") and
   slink[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 = slink[1], slink[2], slink[3]

if floodSensor:changedValue("flood_detected") and
   floodSensor:getValue("flood_detected")
then
  valve:call("turn_off")
  siren:call("turn_on")
end
```

Humidity sensor

Humidity sensor. Measures humidity and sends measurement to central unit. Can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `humidity` *(number, read-only)*

  Measured humidity value.

  Unit: rH% with one decimal number, multiplied by 10 (0.1 %).

Device properties (full spec)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"humidity_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

IAQ sensor

Battery powered Index of Air Quality sensor. Calculates Air Quality Index based on various measures like CO_2 or particles level and relative humidity. Sensors measure values only every few minutes to save battery.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 value	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`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"iaq_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (**full spec**)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Light sensor

Light sensor measures light illuminance in lux and sends measurement to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `illuminance` *(number, read-only)*

  Measured light illuminance value.

  Unit: 1 lx

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"light_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (**full spec**)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Motion sensor

Motion sensor based on custom configuration checks whether motion was detected.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `enabled` *(boolean)*

  Enable or disable sensor. e.g. 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.

  > **Note**
  >
  > The value will remain `true` all the time when subsequent motion detections occur until motion stops.
  > This parameter doesn't emit event when switch from `true` to `true` happens (subsequent motion detections). If you need to observe such action, you need to use `time_since_motion` parameter and check if `time_since_motion` equals to `0`.

- `time_since_motion` *(number, read-only)*

  Time since last motion detected. Value of -1 means there wasn't any motion since last system startup.

> **Note**
>
> The value will be `0` for each detected move, even if the previous one has not yet finished.

Unit: seconds.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"motion_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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)*, at least 1 s.

Examples

Catching motion events

```lua
if slink[4]:changedValue("motion_detected") then
  print("someone is moving around!")
end
```

```lua
if
  slink[4]:changedValue("time_since_motion")
  and
  slink[4]:getValue("time_since_motion") == 0
then
  print("someone is moving around!")
end
```

Delayed action

```lua
if dateTime:changed() then
  slink[4]:call("add_time_since_motion_event", 30)
end

if
  slink[4]:changedValue("time_since_motion")
  and
  slink[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

```lua
if event.type == "sunrise" then
  slink[3]:call("disable")
elseif event.type == "sunset" then
  slink[3]:call("enable")
end
```

Enable a light for 5 minutes on motion detection

```lua
if slink[4]:changedValue("time_since_motion") then
  if slink[4]:getValue("time_since_motion") == 0 then
    slink[60]:setValue("state", true)
    slink[60]:setValueAfter("state", false, 5 * 60)
  end
end
```

Reconfigure thermostat when motion detected

```lua
if slink[4]:changedValue("motion_detected") then
  -- time limited to 2 hours, temperature 23.5°C
  virtual[1]:call("enable_time_limited_mode", {120, 235})
end
```

Pressure sensor

Pressure sensor measures pressure and sends measurement to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `pressure` *(number, read-only)*

  Measured pressure value.

  Unit: hPa with one decimal number, multiplied by 10 (10 Pa).

- `altitude` *(integer)*

  Setting the altitude compensates the atmospheric pressure reading to the pressure at mean sea level, that is normally given in weather reports. Possible range of altitude: `0 - 8849`

  Unit: 1 msl, meters above sea level for your location.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"pressure_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (**full spec****)**

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Pulse width modulation

Pulse width modulation device representation. Set desired signal in device output. User can set desired frequency of signal and duty cycle i.e. the amount of time the digital signal is in the "active" state relative to the period of the signal.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `frequency` *(integer)*

  Frequency of pulse width modulation signal. Minimum value is set in property `frequency_minimum` and maximum value is set in property `frequency_maximum`.

  Unit: 1 Hz

- `frequency_minimum` *(integer, read-only)*

  Lower limit of frequency.

- `frequency_maximum` *(integer, read-only)*

  Upper limit of frequency.

- `raw_value` *(integer)*

  Pulse width modulation duty cycle. The amount of time the digital signal is in the "active" state relative to the period of the signal.

  Unit: 1 %

Commands

- `set_frequency`

  Calls PWM device to set output frequency.

  Argument:

  frequency *(integer)*

- `set_duty_cycle`

  Calls PWM device to set output duty cycle.

  Argument:

  duty_cycle *(integer)*

Examples

Set frequency and duty cycle of PWM

```
slink[3]:call("set_frequency", 3000)
slink[3]:call("set_duty_cycle", 40)
```

RGB controller

Device that controlls color and light intensity of output LED.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `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: 1 %

- `led_color` *(string, read-only)*

  HTML/Hex RGB color that device will set on its output led strip.

  Example: `"#00ff7f"`

- `white_temperature` *(number, read-only)*

  White temperature that device will set on its output led strip.

  Unit: 1 K

- `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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"rgb_controller"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `turn_on`

  Turns on output.

  Argument

  Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `turn_off`

  Turns off output.

  Argument

  Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `toggle`

  Changes state to opposite.

  Argument

  Optional: transition time in 0.1s (default 5 - 500ms) *(number)*

- `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`.

- `stop`

  Calls RGB controller to stop current moving action. Does nothing if no action is in progress.

Examples

Turn on light to specific color at 7:00 and turn off at 8:00

```lua
local rgb = slink[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

```lua
local rgb = slink[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

```lua
local rgb = slink[79]
local animation_id = 2

rgb:call("activate_animation", { id = animation_id })
```

Stop active animation

```lua
local rgb = slink[79]
rgb:call("stop_animation")
```

Activate an animation by ID when device state changes

```lua
local rgb = slink[79]
local animation_id = 3

if wtp[3]:changedValue("state") then
  rgb:call("activate_animation", { id = animation_id })
end
```

Relay

Execution module that changes state depending on the control signal. Relay can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(boolean)*

  State of the output. On/Off.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `timeout` *(number)*

  Protection functionality, that will set device state to off if there are communication problems.

  Unit: 1 min

- `timeout_enabled` *(boolean)*

  Parameter that indicates if timeout functionality is enabled.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `inverted` *(boolean)*

  Indicates if physical state of relay should be the inversion of state shown in application.

- `time_since_state_change` *(number, read-only)*

  Time since last relay state change.

  Unit: 1 s

Required label: `"has_backlight"`

- `backlight_mode` *(string)*

  Buttons backlight mode. Available values: `"auto"`, `"fixed"`, `"off"`

- `backlight_brightness` *(number)*

  Buttons backlight brightness in percent.

- `backlight_idle_color` *(string)*

  HTML/Hex RGB representation of color when controller is in idle (inactive).

  Example: `"#FF00FF"`

- `backlight_active_color` *(string)*

  HTML/Hex RGB representation of color when controller is active e.g. motor is working.

  Example: `"FFFF00"`

Required label: `"relay_startup_state_support"`

- `startup_state` *(string)*

  State of output that should be set on device after power restart. Available values: `"off"`, `"on"`, `"previous"`

  Cannot be changed if device is assigned to thermostat or thermostat output group (virtual contact) / (has label `managed_by_thermostat` or `managed_by_tog`). Cannot be changed when relay `work_mode` is set to `alarm_siren`.

- `work_mode` *(string)*

  Relay work mode. One of: `standard`, `alarm_siren`.

Required label: `"trigger_signal_config_support"`

- `trigger_signal_type` *(string)*

  Available values:

  "impulse"
  > Monostable trigger — impulse signal toggles the output state

  "state_change"
  > Bistable trigger — when trigger state changes, output will be set equal to trigger signal

- `allowed_trigger_signal_types` *(table, read-only)*

  List of available trigger signal types supported by the device.

  Possible combination of values:

  - `impulse`
  - `state_change`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"relay"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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

```lua
if dateTime:changed() then
    if dateTime:getHours() == 19 and dateTime:getMinutes() == 0 then
      slink[4]:call("turn_on")
    elseif dateTime:getHours() == 21 and dateTime:getMinutes() == 0 then
      slink[4]:call("turn_off")
    end
end
```

Temperature regulator

Temperature regulator notifies when desired temperature is reached in room. Can be assigned to virtual thermostat in web application.

Normally works in `constant` temperature mode only, but additional modes (`time_limited` and `schedule`) can be unlocked when associated with Virtual Thermostat.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `target_temperature` *(number)*

  Desired setpoint temperature, which device will try to achieve.

  Unit: 0.1 °C

- `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 doesn't 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_minimum` *(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: 0.1 °C

- `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: 0.1 °C

- `target_temperature_reached` *(boolean)*

  Controls device's algorithm state indicator (available on some regulators). e.g. 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 or heat pump manager (label `managed_by_thermostat` and `managed_by_heat_pump_manager` not present).

  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.

Required label: `"user_menu_lock_support"`

- `user_menu_lock.enabled` *(boolean)*

  Indicates that it is required to enter pin code to access device's user menu.

- `user_menu_lock.pin_code` *(string)*

  Pin Code to access device's user menu. Has to be longer or equal to `user_menu_lock.pin_code_length_minimum` and shorter or equal to `user_menu_lock.pin_code_length_maximum` and contains characters from `user_menu_lock.allowed_characters`.

- `user_menu_lock.pin_code_length_minimum` *(integer, read-only)*

  Minimum length of user menu pin code.

- `user_menu_lock.pin_code_length_maximum` *(integer, read-only)*

  Maximum length of user menu pin code.

- `user_menu_lock.allowed_characters` *(integer, read-only)*

  Allowed characters of user menu pin code.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_regulator"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (**full spec**)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

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 a 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 a 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 a thermostat.

Examples

Raise target temperature between 15:00 and 20:00

```lua
if dateTime:changed() then
  if dateTime:getHours() == 15 and dateTime:getMinutes() == 0 then
    slink[5]:call("set_target_temperature", 220)
  elseif dateTime:getHours() == 20 and dateTime:getMinutes() == 0 then
    slink[5]:call("set_target_temperature", 190)
  end
end
```

Temperature sensor

Temperature sensor. Measures temperature and sends measurement to central unit. Can be assigned to virtual thermostat in web application as room or floor sensor.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `temperature` *(number, read-only)*

  Measured 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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_regulator"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Two state input sensor

Boolean input sensor checks input state and send it to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"two_state_input_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

Valve

Valve representation. Allows user to read and modify valve parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `enabled` *(boolean)*

  Indicates if valve is turned on.

- `state` *(string, read-only)*

  Valve current working state. One of: `"off"`, `"calibration"`, `"return_protection"`, `"heat_source_protection"`, `"work"`, `"blockade"`, `"alarm"`, `"manual_work"`, `"synchronization"`

- `open_percent` *(integer, read-only)*

  Current opening percent. Unit: %, with 2 decimal numbers, multiplied by 100.

- `temperature_valve` *(integer, read-only)*

  Current valve temperature. Unit: °C with one decimal number, multiplied by 10.

- `opening_direction` *(string)*

  Valve opening direction. One of: `"left"`, `"right"`

- `max_single_move` *(integer)*

  The maximum percentage that the valve can open in one interval. Unit: % Min: 1 Max: 99

- `minimal_opening` *(integer)*

  Minimum opening below which the valve cannot close further in its normal operation. Unit: % Min: 1 Max: 99

- `pause_time` *(integer)*

  The pause time the valve takes after opening/closing during normal operation. Unit: second Min: 1 Max: 900

- `hysteresis` *(integer)*

  Valve temperature hysteresis. Unit: °C with one decimal number, multiplied by 10. Min: 2 Max: 20

- `proportional_gain` *(integer)*

  Specifies the response strength to temperature error. Min: 1 Max: 10

- `opening_time` *(integer)*

  The time it takes for the valve to open. Unit: seconds Min: 10 Max: 1500

- `calibration_time` *(integer)*

  The time it takes for the valve to open in calibration. Unit: seconds Min: 10 Max: 1500

Required label: `"has_opening_time_mode"`

- `opening_time_mode` *(string)*

  Determines whether the calibration time differs from valve opening time in work. One of: `"linked"`, `"splitted"`.

- `floor_overheating_temperature` *(integer)*

  The temperature after which the valve will close in floor heating mode. Unit: °C with one decimal number, multiplied by 10. Min: 250 Max: 550

Required label: `"heat_source_sensor_available"`

- `heat_source_protection` *(boolean)*

  Turns on/off the heating source protection. Requires a source temperature sensor.

- `valve_closes_from_pump_threshold` *(boolean)*

  Closes the valve when the pump operating conditions are not met.

- `heat_source_protection_temperature` *(boolean)*

  The temperature after which the valve will open if heat source protection is enabeld. Unit: °C with one decimal number, multiplied by 10. Min: 600 Max: 1000

Required label: `"return_sensor_available"`

- `return_protection` *(boolean)*

  Turns on/off the return protection. Requires a return temperature sensor.

- `return_protection_temperature` *(integer)*

  The temperature below which the valve will close if return protection is enabeld. Unit: °C with one decimal number, multiplied by 10. Min: 100 Max: 800

- `central_heating_target_temperature` *(integer)*

  Temperature setpoint that the device should maintain in heating as central heating valve. Unit: °C with one decimal number, multiplied by 10. Min: 100 Max: 990

- `floor_heating_target_temperature` *(integer)*

Temperature setpoint that the device should maintain in heating as floor heating valve. Unit: °C with one decimal number, multiplied by 10. Min: 100 Max: 500

- `cooling_target_temperature` *(integer)*

Temperature setpoint that the device should maintain in cooling mode. Unit: °C with one decimal number, multiplied by 10. Min: 50 Max: 990

- `valve_type` *(string)*

Mixing valve type. One of: `"central_heating"`, `"floor_heating"`.

- `work_mode` *(string)*

Specify whether valve is used to heating or cooling. One of: `"heating"`, `"cooling"`.

- `emergency_behaviour` *(string)*

Specify valve behavior after communication loss. One of: `"close"`, `"open"`, `"safe_opening_percentage"`, `"standalone_work"`.

- `blockade` *(boolean)*

Specify whether valve blockade is on or off.

- `blockade_reasons` *(array of string, read-only)*

Specify the reason why the valve entered blockade state. Possible values: `"floor_overheat"`, `"floor_heating_disabled"`, `"summer_mode"`, `"boiler_priority"`, `"device_standby_mode"`, `"closing_if_threshold_not_met"`, `"room_regulator"`, `"external_blockade"`.

Device properties (full spec)

- `class` *(string, read-only)* = `"slink"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*
- `status` *(string, read-only)*
- `tags` *(string[])*
- `type` *(string, read-only)* = `"common_valve"`
- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties ([full spec](#))

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `calibration`

  Start valve calibration.

Valve Pump

Valve pump representation. Allows user to read and modify valve parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `state` *(boolean, read-only)*

  Specify whether pump is working or not.

- `relay_control_mode` *(string)*

  Specify whether the pump relay belongs to the internal valve algorithm or external entity.
  One of: `"internal"`, `"external"`.

- `emergency_behaviour` *(string)*

  Specify pump behavior after communication loss if the pump relay control mode is internal.
  One of: `"turn_off"`, `"turn_on"`.

- `antistop` *(boolean)*

  Turns on the pump for 5 hours every 10 days if the pump relay control mode is internal.

Required label: `"heat_source_sensor_available"`

- `work_mode` *(string)*

  Specify pump work mode if the pump relay control mode is internal.
  One of: `"always_on"`, `"always_off"`, `"temperature_control"`.
  Label is required to set `temperature_control` mode.

- `temperature_threshold_heating` *(integer)*

  The pump will operate after exceeding the temperature threshold.
  Unit: 0.1 °C.
  Range: 100 – 800 (10.0 °C – 80.0 °C).

- `temperature_threshold_cooling` *(integer)*

  The pump will operate below the temperature threshold.
  Unit: 0.1 °C.
  Range: 100 – 550 (10.0 °C – 55.0 °C).

> **Required label:** `"pump_work_in_calibration_support"`
>
> - `pump_work_in_calibration` *(boolean)*
>
>   Allows the pump to operate in valve calibration.

- `blockade` *(boolean)*

  Specify whether pump blockade is on or off.

Device properties (full spec)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"valve_pump"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Valve Analog PID

0–10V valve with PID calculation representation. Allows user to read and modify valve parameters.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `heating_target_temperature` *(integer)*

  Temperature setpoint that the device should maintain in heating.
  Unit: 0.1 °C.
  Min: from parameter `heating_target_temperature_minimum`.
  Max: from parameter `heating_target_temperature_maximum`.

- `heating_target_temperature_minimum` *(integer, read-only)*

  Minimum value of temperature setpoint in heating.
  Unit: 0.1 °C.

- `heating_target_temperature_maximum` *(integer, read-only)*

  Maximum value of temperature setpoint in heating.
  Unit: 0.1 °C.

- `cooling_target_temperature` *(integer)*

  Temperature setpoint that the device should maintain in cooling.
  Unit: 0.1 °C.
  Min: from parameter `heating_target_temperature_minimum`
  Max: from parameter `heating_target_temperature_maximum`

- `cooling_target_temperature_minimum` *(integer, read-only)*

  Minimum value of temperature setpoint in cooling.
  Unit: 0.1 °C.

- `cooling_target_temperature_maximum` *(integer, read-only)*

  Maximum value of temperature setpoint in cooling.
  Unit: 0.1 °C.

- `work_mode` *(string)*

  Specify valve work mode. One of: `"heating"`, `"cooling"`, `"off"`.

- `current_temperature` *(integer, read-only)*

  Current valve water temperature.
  Unit: 0.1 °C.

- `proportional_gain` *(integer)*

  Proportional gain parameter for PID calculation.
  Unit: 0.1 (no unit, parameter with one decimal number, multiplied by 10.)
  Range: 0 – 100 (0.0 – 10.0)

- `integral_gain` *(integer)*

  Integral gain parameter for PID calculation.
  Unit: 0.1 (no unit, parameter with one decimal number, multiplied by 10.)
  Range: 0 – 100 (0.0 – 10.0)

- `derivative_gain` *(integer)*

  Derivative gain parameter for PID calculation.
  Unit: 0.1 (no unit, parameter with one decimal number, multiplied by 10.)
  Range: 0 – 100 (0.0 – 10.0)

- `calculated_pid_value` *(integer, read-only)*

  Current calculated PID value.

- `output_value` *(integer, read-only)*

  Current valve output voltage.
  Unit: 1 mV

Device properties (**full spec**)

- `class` *(string, read-only)* = `"slink"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"valve_analog_pid"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

SLINK device properties (full spec)

- `address` *(integer, read-only)*

- `endpoint` *(integer, read-only)*

- `software_version` *(string, read-only)*

Commands

- `reset_pid`

  Resets the PID calculations.

Alarm system

Following devices allow managing an alarm system. Zones can be armed and disarmed.

Device or 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 e.g. `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.

Common alarm system device properties

- `parent_id` *(string, read-only)*

  ID of parent device which device belongs to.

- `software_version` *(string, read-only)*

  Software name and version description.

- `sub_id` *(integer, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

Satel — Alarm zone

Object from Satel central unit. Zone from Satel central unit representation.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `armed` *(boolean, read-only)*

  Indicates if zone is armed.

- `violated` *(boolean, read-only)*

  Zone violation state.

- `zone_status` *(string, read-only)*

  Zone status. One of: `armed`, `disarmed`, `entry_time`, `exit_time`, `violated`, `fire`, `emergency`, `flooding`

Device properties (**full spec**)

- `class` *(string, read-only)* = `"alarm_system"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"alarm_zone"`

- `variant` *(string, read-only)* = `"satel"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Alarm system properties (**full spec**)

- `parent_id` *(string, read-only)*

- `software_version` *(string, read-only)*

- `sub_id` *(integer, read-only)*

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

```lua
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

```lua
alarm_system[3]:call("arm_in_mode", {pin_code="1234", mode=1})
```

Send notification when zone violated

```lua
local zone = alarm_system[1]

if zone:changedValue("violated") and zone:getValue("violated") then
    notify:error("Zone violation!", "Zone " .. zone:getValue("name") ..
        " violated")
end
```

Turn on the light when zone entry detected

```lua
local zone = alarm_system[1]
local light = wtp[3]

if zone:changedValue("zone_status") and zone:getValue("zone_status") ==
    "entry_time" then
    light:setValue("state", true)
end
```

Satel — Two state input sensor

Device from Satel central unit.

Boolean input sensor state (violation) is read from Satel central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"alarm_system"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"two_state_input_sensor"`

- `variant` *(string, read-only)* = `"satel"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Alarm system properties (full spec)

- `parent_id` *(string, read-only)*

- `software_version` *(string, read-only)*

- `sub_id` *(integer, read-only)*

Satel — Two state output

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `state` *(boolean, read-only)*

  State of the output. On/Off.

Device properties (full spec)

- `class` *(string, read-only)* = `"alarm_system"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"two_state_output"`

- `variant` *(string, read-only)* = `"satel"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

Alarm system properties (**full spec**)

- `parent_id` *(string, read-only)*

- `software_version` *(string, read-only)*

- `sub_id` *(integer, read-only)*

Commands

- `turn_on`

  Turns on output.

  > **Note**
  >
  > Cannot be executed when output is managed by alarm central (output function is not 24, 25 or 64-79). You can check if your device contains `managed_by_alarm_central` label.

  Argument:

  PIN code used to turn on output *(string)*

- `turn_off`

  Turns off output.

  > **Note**
  >
  > Cannot be executed when output is managed by alarm central (output function is not 24, 25 or 64-79). You can check if your device contains `managed_by_alarm_central` label.

  Argument:

  PIN code used to turn off output *(string)*

- `toggle`

  Changes output to opposite state.

  > **Note**
  >
  > Cannot be executed when output is managed by alarm central (output function is not 24, 25 or 64-79). You can check if your device contains `managed_by_alarm_central` label.

  Argument:

  PIN code used to toggle output state *(string)*

Examples

Set output state based on alarm input sensor violated state

```lua
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 devices

LoRa wireless devices.

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.

> **Note**
>
> LoRa devices are available only in Sinum Pro.

Property modification is possible via REST API, web app or directly from scripts using `lora` container e.g. `lora[6]` gives you access to device with **ID 6**. LoRa devices have global scope and they are visible in all executions contexts.

Common LoRa device properties

- `signal` *(number, read-only)*

  Signal value.

  Unit: %.

- `software_version` *(string, read-only)*

  Software name and version description.

- `eui` *(string, read-only)*

  Lora 64-bit device EUI (Extended Unique Identifier).

- `sub_id` *(integer, read-only)*

  Unique (per physical device) identifier that help to distinguish same device types in one physical device.

- `battery` *(number, read-only)*

  Battery status.

  Unit: %.

  > **Note**
  >
  > Parameter is optional. Available when device is battery powered – check whether `battery_powered` label is present.

Flood sensor

Battery powered, flood sensor. Detects water leak on flat surfaces.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `flood_detected` *(boolean, read-only)*

  A flag representing the detection of flood / water leak by the sensor.

Device properties (full spec)

- `class` *(string, read-only)* = `"lora"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"flood_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

LoRa device properties (full spec)

- `signal` *(number, read-only)*

- `visible` *(boolean, read-only)*

- `software_version` *(string, read-only)*

- `eui` *(string, read-only)*

- `sub_id` *(integer, read-only)*

- `battery` *(number, optional, read-only)*

Examples

Catching alarms

```lua
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

```lua
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
```

Humidity sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `humidity` *(number, read-only)*

  Measured humidity value.

  Unit: rH% with one decimal number, multiplied by 10.

Device properties (full spec)

- `class` *(string, read-only)* = `"lora"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"humidity_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

LoRa device properties (full spec)

- `signal` *(number, read-only)*

- `visible` *(boolean, read-only)*

- `software_version` *(string, read-only)*

- `eui` *(string, read-only)*

- `sub_id` *(integer, read-only)*

- `battery` *(number, optional, read-only)*

Opening sensor

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `open` *(boolean, read-only)*

  Opening sensor state. Open/Closed.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"lora"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"opening_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

LoRa device properties (**full spec**)

- `signal` *(number, read-only)*

- `visible` *(boolean, read-only)*

- `software_version` *(string, read-only)*

- `eui` *(string, read-only)*

- `sub_id` *(integer, read-only)*

- `battery` *(number, optional, read-only)*

Examples

Catch open and close events

```lua
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
```

Relay

Execution module that changes state depending on the control signal.

Relay can be assigned to virtual thermostat in web application.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` and `setValue` methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.

- `state` *(boolean)*

  State of the output. On/Off.

  Cannot be changed if device is assigned to thermostat, thermostat output group (virtual contact), wicket or gate. (has label `managed_by_thermostat`, `managed_by_tog`, `managed_by_wicket` or `managed_by_gate`).

- `inverted` *(boolean)*

  Indicates if should invert physical state of relay compared to represented state in application.

- `work_mode` *(string)*

  Relay work mode. One of: `standard`, `alarm_siren`.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"lora"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"relay"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

LoRa device properties (full spec)

- `signal` *(number, read-only)*

- `visible` *(boolean, read-only)*

- `software_version` *(string, read-only)*

- `eui` *(string, read-only)*

- `sub_id` *(integer, read-only)*

- `battery` *(number, optional, read-only)*

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

```lua
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
```

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.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `temperature` *(number, read-only)*

  Measured 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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"lora"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"temperature_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

LoRa device properties (full spec)

- `signal` *(number, read-only)*

- `visible` *(boolean, read-only)*

- `software_version` *(string, read-only)*

- `eui` *(string, read-only)*

- `sub_id` *(integer, read-only)*

- `battery` *(number, optional, read-only)*

Two state input sensor

Boolean input sensor checks input state and send it to central unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"lora"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"two_state_input_sensor"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

LoRa device properties (full spec)

- `signal` *(number, read-only)*

- `visible` *(boolean, read-only)*

- `software_version` *(string, read-only)*

- `eui` *(string, read-only)*

- `sub_id` *(integer, read-only)*

- `battery` *(number, optional, read-only)*

System modules

System modules include built-in transceivers and additional extenders, used to connect other devices to the central.

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 e.g. `system_module[2]` gives you access to module with **ID 2**. System modules have global scope and they are visible in all executions contexts.

Common system module properties

- `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.

WTP or SBus extenders

Device extends signal range of wireless WTP devices or extends SBus communication line. It passes all communication with WTP/SBus devices to Sinum Central Unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 Wi-Fi network extender is connected to.

- `network_signal` *(integer, read-only)*

  Value from 0 to 100 indicating how strong Wi-Fi signal is.

- `network_channel` *(integer, read-only)*

  Wi-Fi network channel.

Required label: `"has_ethernet"`

- `ethernet_connected` *(boolean, read-only)*

  Indicates if extender has ethernet cable connected.

- `ip` *(string, read-only)*

  IP address of device.

Required label: `"diagnostic_support"`

- `diagnostic` *(object-like table, read-only)*

  Internal diagnostic data of extender. Has data if enabled by command `enable_diagnostic`.

Device properties (full spec)

- `class` *(string, read-only)* = `"system_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"wtp_extender, sbus_extender"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

System module properties (full spec)

- `uuid` *(string, read-only)*

- `enabled` *(boolean)*

- `software_version` *(string, read-only)*

Commands

> **Required label:** `"diagnostic_support"`
>
> - `enable_diagnostic`
>
>   Turn on gathering diagnostic informations.
> - `disable_diagnostic`
>
>   Turn off gathering diagnostic informations.

IR remote

Device which can send and learn IR codes. Currently, only Broadlink IR devices are supported.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `broadlink_device_type` *(string, read-only)*

  Broadlink device type identifier.

- `learn_status` *(string, read-only)*

  Indicates if device is in learning mode. Available values: `"inactive"`, `"in_progress"`, `"done"`, `"failed"`

- `last_learned_data` *(string, read-only)*

  Last learned IR code in Hex-String Broadlink format.

- `ip` *(string, read-only)*

  IP address of device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"system_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"ir_remote"`

- `variant` *(string, read-only)* = `"broadlink"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

System module properties ([full spec](#))

- `uuid` *(string, read-only)*

- `enabled` *(boolean)*

- `software_version` *(string, read-only)*

Commands

- `send`

  Send IR code.

  Arguments:

  ○ *string* - IR code in Broadlink Hex-String format.

- `learn`

  Starts learning mode.

RF remote

Device which can send and learn RF codes. Currently only Broadlink RF devices are supported.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `broadlink_device_type` *(string, read-only)*

  Broadlink device type identifier.

- `learn_status` *(string, read-only)*

  Indicates if device is in learning mode. Available values: `"inactive"`, `"in_progress"`, `"done"`, `"failed"`

- `last_learned_data` *(string, read-only)*

  Last learned RF code in Broadlink Hex-String format.

- `last_learned_frequency` *(number, read-only)*

  Frequency of last learned RF code in MHz.

- `ip` *(string, read-only)*

  IP address of device.

Device properties (**full spec**)

- `class` *(string, read-only)* = `"system_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"rf_remote"`

- `variant` *(string, read-only)* = `"broadlink"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

System module properties ([full spec](#))

- `uuid` *(string, read-only)*

- `enabled` *(boolean)*

- `software_version` *(string, read-only)*

Commands

- `send`

  Send RF code.

  Arguments:

  ◦ *string* — RF code in Hex-String Broadlink format.

- `learn`

  Starts learning mode.

  Arguments:

  ◦ *number?* — Frequency of RF code in MHz (e.g. 433.95). If not given, device will try to detect it.

LoRa gateway

Responsible for communication with LoRa devices connected to Sinum Central Unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

Device properties (**full spec**)

- `class` *(string, read-only)* = `"system_module"`
- `color` *(string)*
- `icon` *(string)*
- `id` *(integer, read-only)*
- `labels` *(string[], read-only)*
- `messages` *(sequence, read-only)*
- `name` *(string)*
- `room_id` *(integer, read-only, optional)*
- `status` *(string, read-only)*
- `tags` *(string[])*
- `type` *(string, read-only)* = `"lora_gateway"`
- `variant` *(string, read-only)* = `"generic"`
- `visible` *(boolean, read-only)*
- `voice_assistant_device_type` *(string, read-only)*
- `purpose` *(string)*

System module properties (**full spec**)

- `uuid` *(string, read-only)*
- `enabled` *(boolean)*
- `software_version` *(string, read-only)*

WTP or SBus Modbus transceiver

Representation of built-in module which is responsible for communication with certain type (WTP or SBus) of devices connected to Sinum Central Unit.

This system module cannot be added or removed by user, as it is built-in.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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.

- `ip` *(string, read-only)*

  IP address of device.

Required label: `"diagnostic_support"`

- `diagnostic` *(object-like table, read-only)*

  Internal diagnostic data of transceiver. Has data if enabled by command `enable_diagnostic`.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"system_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- room_id *(integer, read-only, optional)*

- status *(string, read-only)*

- tags *(string[])*

- type *(string, read-only)* = `"wtp_transceiver, sbus_transceiver"`

- variant *(string, read-only)* = `"generic"`

- visible *(boolean, read-only)*

- voice_assistant_device_type *(string, read-only)*

- purpose *(string)*

System module properties (**full spec**)

- uuid *(string, read-only)*

- enabled *(boolean)*

- software_version *(string, read-only)*

Commands

Required label: `"diagnostic_support"`

- enable_diagnostic

  Turn on gathering diagnostic informations.

- disable_diagnostic

  Turn off gathering diagnostic informations.

Modbus transceiver

Representation of built-in module which is responsible for communication with Modbus devices connected to Sinum Central Unit.

This system module cannot be added or removed by user, as it is built-in.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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.

- `ip` *(string, read-only)*

  IP address of device.

Required label: `"diagnostic_support"`

- `diagnostic` *(object-like table, read-only)*

  Internal diagnostic data of transceiver. Has data if enabled by command `enable_diagnostic`.

- `slave_mode` *(boolean, read-only)*

  Indicates if module is in slave mode. Can be changed only using commands `enable_slave` or `disable_slave`.

Required label: `"modbus_slave_support"`

- `slave_config.baud_rate` *(integer)*

  Baud rate on which the slave device works. One of: 2400, 4800, 9600, 19200, 38400, 57600, 115200.

- `slave_config.parity` *(string)*

> Parity UART setting which slave device uses. One of: `"none"` `"odd"` `"even"`
>
> - `slave_config.stop_bits` *(string)*
>
> Stop bits UART setting which slave device uses. One of: `"one"` `"two"`.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"system_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"modbus_transceiver"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

System module properties ([full spec](#))

- `uuid` *(string, read-only)*

- `enabled` *(boolean)*

- `software_version` *(string, read-only)*

Commands

> **Required label:** `"modbus_slave_support"`

- `enable_slave`

  Turn on Modbus slave mode for extender using Modbus RTU configuration provided in argument.

  Arguments:

  packed arguments *(table)*:

  - `baud_rate` *(number)*:

    Baud rate on which Modbus slave operates.

  - `parity` *(string)*:

    UART parity configuration for slave mode.

  - `stop_bits` *(string)*:

    UART stop bits configuration for slave mode.

- `disable_slave`

  Turn off Modbus slave mode for extender. Returns to standard Modbus master mode.

Required label: `"diagnostic_support"`

- `enable_diagnostic`

  Turn on gathering diagnostic informations.

- `disable_diagnostic`

  Turn off gathering diagnostic informations.

Examples

Turn on slave mode for transceiver

```lua
local moduleId = 3
system_module[moduleId]:call("enable_slave", {baud_rate=115200, parity='even',
    stop_bits='one'})
```

Turn off slave mode for transceiver

```lua
local moduleId = 3
system_module[moduleId]:call("disable_slave")
```

Modbus extender

Device extends Modbus RTU communication line. It passes all communication with Modbus devices to Sinum Central Unit.

Properties

Direct access to properties is not allowed. You can read or change values using `getValue` *and* `setValue` *methods. An attempt at retrieving a nonexistent object property, or setting wrong value type will cause a script error.*

- `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 Wi-Fi network extender is connected to.

- `network_signal` *(integer, read-only)*

  Value from 0 to 100 indicating how strong Wi-Fi signal is.

- `network_channel` *(integer, read-only)*

  Wi-Fi network channel.

Required label: `"has_ethernet"`

- `ethernet_connected` *(boolean, read-only)*

  Indicates if extender has ethernet cable connected.

- `ip` *(string, read-only)*

  IP address of device.

Required label: `"diagnostic_support"`

- `diagnostic` *(object-like table, read-only)*

  Internal diagnostic data of extender. Has data if enabled by command `enable_diagnostic`.

- `slave_mode` *(boolean, read-only)*

  Indicates if module is in slave mode. Can be changed only using commands `enable_slave` or `disable_slave`.

Required label: `"modbus_slave_support"`

- `slave_config.baud_rate` *(integer)*

  Baud rate on which the slave device works. One of: 2400, 4800, 9600, 19200, 38400, 57600, 115200.

- `slave_config.parity` *(string)*

  Parity UART setting which slave device uses. One of: `"none"` `"odd"` `"even"`

- `slave_config.stop_bits` *(string)*

  Stop bits UART setting which slave device uses. One of: `"one"` `"two"`.

Device properties ([full spec](#))

- `class` *(string, read-only)* = `"system_module"`

- `color` *(string)*

- `icon` *(string)*

- `id` *(integer, read-only)*

- `labels` *(string[], read-only)*

- `messages` *(sequence, read-only)*

- `name` *(string)*

- `room_id` *(integer, read-only, optional)*

- `status` *(string, read-only)*

- `tags` *(string[])*

- `type` *(string, read-only)* = `"modbus_extender"`

- `variant` *(string, read-only)* = `"generic"`

- `visible` *(boolean, read-only)*

- `voice_assistant_device_type` *(string, read-only)*

- `purpose` *(string)*

System module properties (full spec)

- `uuid` *(string, read-only)*

- `enabled` *(boolean)*

- `software_version` *(string, read-only)*

Commands

Required label: `"modbus_slave_support"`

- `enable_slave`

  Turn on Modbus slave mode for extender using Modbus RTU configuration provided in argument.

  Arguments:

  packed arguments *(table)*:

  - `baud_rate` *(number)*:

    Baud rate on which Modbus slave operates.

  - `parity` *(string)*:

    UART parity configuration for slave mode.

  - `stop_bits` *(string)*:

    UART stop bits configuration for slave mode.

- `disable_slave`

  Turn off Modbus slave mode for extender. Returns to standard Modbus master mode.

Required label: `"diagnostic_support"`

- `enable_diagnostic`

  Turn on gathering diagnostic informations.

- `disable_diagnostic`

  Turn off gathering diagnostic informations.

Examples

Turn on slave mode for extender

```lua
local moduleId = 3
system_module[moduleId]:call("enable_slave", {
    baud_rate = 115200,
    parity = 'even',
    stop_bits= 'one'
})
```

Turn off slave mode for extender

```lua
local moduleId = 3
system_module[moduleId]:call("disable_slave")
```