plan44 vdcd external device API

Contents

About vdcd external device API

This document describes the socket based API included in the *p44vdc* framework (virtual device connector framework) and thus in the vdcd virtual device connector daemon

The external device API allows external scripts and programs to register themselves to the *vdcd* to implement custom digitalSTROM devices with very little effort.

To host external devices, *vdcd* must be started with the *--externaldevices* option, providing a port number or absolute unix socket path for device implementation to connect to. **For security reasons, it is recommended to run the scripts and programs implementing devices on the same device as vdcd itself**. However for development purposes the *externalnonlocal* command line option can be specified to allow device API connections from non-local clients.

The plan44.ch digitalSTROM products P44-DSB-DEH(2) and P44-DSB-E(2) support the *external device API* from version 1.5.0.8 onwards. Multiple devices sharing a single connection is supported from version 1.5.2.0 onwards. Single device support is supported from 1.6.0.2 onwards. However, at the time of writing, the *external device API* is active only for P44-DSB devices enabled for "testing"/beta (available upon request). In the free "P44-DSB-X" plan44.ch image for RaspberryPi, the *external device API* is always enabled. By default, vdcd uses port 8999 for the *external device API*

External Device API operation

Each external device implementation needs to

- open a connection to the TCP port or unix socket specified with the *--externaldevices* vdcd command line option (default is port 8999).
- optionally, send a *initvdc* message to adjust vdc-level data (like vdc model name and icon)
- send a *init* message declaring the properties of the device (specifying outputs, inputs, names, default group membership etc.). The *init* message uses JSON syntax. However, no JSON support is actually needed in a device implementation, because the *init* message can specify to use a extremely simple text protocol for any communication beyond the *init* message itself. And the *init* message is usually a constant string that can be sent easily using any programming language. (Note that for *single devices*, which have complex actions and events, only the JSON syntax is supported).
- enter a loop, waiting for messages from the *vdcd* indicating output channel changes, or sending messages to the *vdcd* indicating input changes.
- When connection closes (due to error or when vdcd explicitly closes it), the device implementation should restart, see first bullet point. This can be achieved within the device implementation itself, or by having the device implementation run as a daemon under control of a daemon supervisor like *runit*, which re-starts daemons when they terminate.
- from version 1.5.2.0 onwards, multiple devices can be created on the same TCP connection. To distinguish them, each one must be assigned a *tag* in the *init* message, which must/will then be used in all further messages to/from that device.

Message Format

Messages consist of strings, delimited by a single LF (0x0A) character. The *init* message must always be in JSON format. Further messages are either JSON or simple text messages, depending on the *protocol* option in the *init* message (see below).

Initvdc Message

The optional *initvdc* message can be sent by an external device implementation to set some vdc-global parameters.

Initvdc message structure

Field	Type	Description
message	string	identifies the message type, must be **initvdc** for the *initvdc* message
modelname	optional string	a string describing the vdc model. This will be displayed by the dSS as "HW Info". By default, the vdc has a model name like "P44-DSB-X external"
iconname	optional string	a string that is used to construct a file name of the icon to be displayed for the vdc. Default for *iconname* is “vdc_ext”.
configurl	optional string	a URL that will be shown in the context menu of the vdc in the dSS configurator (i.e. the contents of the "configURL" vdc property in the vDC API). If not specified, this will default to the vdchost's default URL (if any).
alwaysVisible	optional boolean	if set to true, the vdc will announce itself to the vDC API client even if it does not (yet) contain any devices.

Init Message

The *init* message is sent by a external device implementation as the first message after opening the socket connection. It needs to be formatted as a single line JSON object. It describes the device's outputs and inputs and other properties, such that vdcd can instantiate an appropriate digitalSTROM device with all standard behaviour required.

To initialize multiple devices sharing the same connection, multiple *init* messages can be put into a JSON array. This is required in particular to create multiple devices that are meant to communicate via the simple text API - because after the first message, the protocol will be switched to simple text and would not allow further JSON init messages.

A simple init message for a light dimmer might look like (on a single line):

```
{'message':'init','protocol':'simple','output':'light','name':'ext
dimmer','uniqueid':'myUniqueID1234'}
```

A init message for two light dimmers on the same connection might look like (on a single line):

```
[ {'message':'init', 'tag':'A', 'protocol':'simple', 'output':'light',
'name':'ext dimmer A', 'uniqueid':'myUniqueID1234_A'}, {'message':'init',
'tag':'B', 'protocol':'simple', 'output':'light', 'name':'ext dimmer B',
'uniqueid':'myUniqueID1234_B'} ]
```

The following tables describes all possible fields of the *init* message JSON object:

Init message structure

Field	Type	Description
message	string	identifies the message type, must be **init** for the *init* message
protocol	optional string	Can be set to **simple** to use the simple text protocol for all further communication beyond the *init* message. This allows implementing devices without need for any JSON parsing. If set to **json** (default), further API communication are JSON messages. Note: *protocol* can be specified only in the first *init* message. It is ignored if present in any subsequent *init* messages. Note that *single devices* (those that use *actions*, *events*, *properties* or *states*) *must* use the JSON protocol.
tag	optional string	When multiple devices are created on the same connection, each device needs to have a *tag* assigned. The *tag* must not contain '=' or ':'. The *tag* is used to identify the device in further communication.
uniqueid	string	This string must uniquely define the device at least among all other external devices connected to the same vdcd, or even globally. To identify the device globally, use a UUID string (XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX) or a valid 34 hex digit digitalSTROM dSUID. To identify the device uniquely among all other devices in the same vdcd, use any other string.
subdeviceindex	optional integer	This can be used to create multiple devices with the same base dSUID (i.e. same uniqueid passed in the init message), which is recommended for composite devices like fan coil units, which have a heater/cooler subdevice and a fan subdevice, or for multiple buttons that can be combined to form 2-way buttons.

colorclass	optional integer	defines the device's color class (if not specified, a default color is derived from *group* and/or *output*) 1: yellow/light, 2: grey/shadow 3: blue/climate 4: cyan/audio 5: magenta/video 6: red/security 7: green/access 8: black/joker 9: white/single device
group	optional integer	defines the group of the device's output (if not specified, the *output* type determines a default group) 1: yellow/light, 2: grey/shadow 3: blue/heating 4: cyan/audio 5: magenta/video 6: red/security 7: green/access 8: black/variable 9: blue/cooling 10: blue/ventilation 11: blue/windows 12: blue/air recirculation 48: room temperature control 49: room ventilation control
output	optional string	Defines the type of output: light: dimmer output with light behaviour colorlight: 6-channel digitalSTROM color light (brightness, hue, saturation, colortemp, cieX, cieY). ctlight: 2-channel tunable white light with only 2 channels (brightness, colortemp) exposed in vDC API. movinglight: color light with additional X and Y position channels. heatingvalve: 0..100% heating valve. ventilation: device with airflow intensity, airflow direction and louver position channels, according to ds-ventilation specification. Also see the *kind* field. fancoilunit: device with 0..100% fan output channel, receiving set point and current room temperature values via *control* message for regulation. shadow: jalousie type device with position and angle channel. Also see *move* and *kind* fields. action: For *single devices* only: this creates a scene table with scenes having a *command* field which can be used to assign device actions to scenes. basic: basic 0..100% output with no special behaviour. Can be used for relay outputs. Default is no output.

kind	optional string for *shadow* and *ventilation* output type	Defines the kind of device. For *shadow*: roller: simple roller blind, no angle sun: sunblind jalousie: jalousie with blade angle control For *ventilation*: ventilation: air supply/exhaust recirculation: air (re)circulation within rooms
endcontacts	optional boolean for *shadow* output type	If set to true, device implementation must report reaching top and bottom positions by updating channel value to 100 or 0, resp. (using the “channel” message). Otherwise, the shadow behaviour uses move time settings to derive actual positions from timing alone.
move	optional boolean	If set to true, the device must support the "move" or "MV" message (see below), which is issued by the device to start or stop a movement (increase, decrease) of the output value. The move semantic might be more useful for blind type devices than channel output values. Default is false (no move semantics)
sync	optional boolean	if set to true, the device must support the "sync" message and must respond to "sync" with the "synced" message. "sync" is issued by the vdcd when it needs to know current output values (e.g. for a saveScene operation). "synced" is sent by the device when updated output channel values have been sent to the vdcd. Default is false (no output value sync requests)
controlvalues	optional boolean	If set to true, control values (such as room temperature set points and actual values) are forwarded to the external device using the "control"/ CTRL message.
scenecommands	optional boolean	If set to true, the vDC will send the "scenecommand"/ SCMD message to devices for some special scene commands that may have additional semantics beyond chaning channel values.
groups	optional array of integers	can be used to specify output group membership different from the defaults of the specified *output* type.
hardwarename	optional string	a string describing the type of output hardware, such as "dimmer" or "relay" etc. Default is the string specified in *output*.
modelname	optional string	a string describing the device model. This will be displayed by the dSS as "HW Info".
vendorname	optional string	the vendor name
oemmodelguid	optional string	the GUID that is used to identify devices (in particular: single devices) in the digitalSTROM server side database to provide extended information. Usually this is a GS1 formatted GTIN like: 'gs1:(01)76543210123'

iconname	optional string	a string that is used as a base to construct a file name of the icon to be displayed for the device. vdcd will first try to load a icon named "*iconname_groupcolor"* (with *groupcolor* being the color of the device). If such a file does not exist, vdcd tries to load "*iconname*_other". Default for *iconname* is "ext".
configurl	optional string	a URL that will be shown in the context menu of the device in the dSS configurator (i.e. the contents of the "configURL" device property in the vDC API). If not specified, this will default to the vdchost's default URL (if any).
typeidentifier	optional string	an identifier specifying the (hardware implementation) type of this device. By default, it is just "external". This identifier is not used in digitalSTROM, but it is exposed as "x-p44-deviceType" for local Web-UI, and it is used to load type-specific scene tables/properties from .csv files.
deviceclass	optional string	a device class intended to group functionally equivalent single devices (washing machines, kettles, ovens, etc.). This must match a digitalSTROM-defined device class, and the device must provide the actions/ states/events/properties as specified for that class.
deviceclassversion	optional integer	a version number for the device class.
name	optional string	the default name the device will have in the digitalSTROM system. Note that this can be changed by the user via dSS Web interface.
buttons	optional array of objects	Defines the buttons of the device. See table below for fields in the button objects
inputs	optional array of objects	Defines the binary inputs of the device. See table below for fields in the input objects
sensors	optional array of objects	Defines the sensors of the device. See table below for fields in the sensor objects
configurations	optional object containing named configurations	Defines the device configurations (such as two-way button vs. two separate one-way buttons) possible for this device. Note that devices must use the JSON protocol to use configurations. The *configurations* object can contain 1..n *configuration* objects, see table below
currentConfigId	string	For devices with multiple *configurations* (see above), this field must contain the current configuration's id.
actions	optional object containing named actions	Defines the device actions of the device (for 'single' devices). Note that single devices must use the JSON protocol. The *actions* object can contain 1..n *action* objects, see table below

dynamicactions	optional object containing dynamic actions by their id	Defines a initial set of dynamic actions of the device (for 'single' devices). Note that single devices must use the JSON protocol. The *dynamicactions* object can contain 1..n *dynamicaction* objects, see table below. If no dynamic actions are known at the time of device instantiation, *dynamicactions* can be omitted. Note that dynamic actions (unlike normal actions) can als be added, changed and deleted during device operation using the *dynamicAction* message, see below.
noconfirmaction	optional boolean	Can be set to true when the device impelemnation does not want to confirm action completion
states	optional object containing states by their id	Defines the states of the device (for 'single' devices). Note that single devices must use the JSON protocol. The *states* object can contain 1..n *state* objects, see table below
events	optional object containing events by their id	Defines the events of the device (for 'single' devices). Note that single devices must use the JSON protocol. The *events* object can contain 1..n *event* objects, see table below
properties	optional object containing properties by their id	Defines the device properties of the device (for 'single' devices). Note that single devices must use the JSON protocol. The *properties* object can contain 1..n *property* objects, see table below

Button object in the buttons field of the init message

Field	Type	Description
id	optional string	string id, unique within the device to reference the button via vDC API.
buttonid	optional integer	**Note: this field can also be set as "id" for backward compatibility with earlier versions of this API.** Identifies the hardware button this button input belongs to. Two-way or multi-way buttons will have multiple button definitions with the same id. Defaults to 0
buttontype	optional integer	Defines the type of button: 0: kind of button not defined by device hardware 1: single pushbutton 2: two-way pushbutton or rocker (Note: if you use this, the first button must be the the down element and the second button must be the up element. 3: 4-way navigation button 4: 4-way navigation with center button 5: 8-way navigation with center button 6: On-Off switch Defaults to 1 (single pushbutton)
element	optional integer	Defines which element of a multi-element button is represented by this button input: 0: center element / single button 1: down, for 2,4,8-way 2: up, for 2,4,8-way 3: left, for 2,4,8-way 4: right, for 2,4,8-way 5: upper left, for 8-way 6: lower left, for 8-way 7: upper right, for 8-way 8: lower right, for 8-way Default is 0 (single button)
group	optional integer	defines the primary color (group) of the button: 1: yellow/light, 2: grey/shadow 3: blue/heating 4: cyan/audio 5: magenta/video 6: red/security 7: green/access 8: black/variable 9: blue/cooling 10: blue/ventilation 11: blue/windows 48: roomtemperature control Defaults to primary device group

combinables	optional integer	if set to a multiple of 2, this indicates that there are other devices with same *uniqueid* but different *subdeviceindex*, which can be combined to form two-way buttons. In this case, each of the combinable devices must have a single button only, and the subdeviceindex must be in the range *n..n+combinable-1*, with n=*subdeviceindex* MOD *combinable*
localbutton	optional boolean	If set to true, this button acts as a local button for the device (directly switches and dims the output)
hardwarename	optional string	a string describing the button element, such as "up" or "down" etc.

Input object in the inputs field of the init message

Field	Type	Description
id	optional string	string id, unique within the device to reference the binary input via vDC API.
inputtype	optional integer	Defines the type of input: 0: no system function 1: Presence 2: Light 3: Presence in darkness 4: twilight 5: motion 6: motion in darkness 7: smoke 8: wind 9: rain 10: solar radiation (sun light above threshold) 11: thermostat (temperature below user-adjusted threshold) 12: device has low battery 13: window is open 14: door is open 15: window handle 16: garage door is open 17: protect against too much sunlight 18: frost detector Defaults to 0 (no system function)
usage	optional integer	Defines usage: 0: undefined 1: room (indoors) 2: outdoors 3: user interaction Default is 0 (undefined)
group	optional integer	defines the primary color (group) of the button: 1: yellow/light, 2: grey/shadow 3: blue/heating 4: cyan/audio 5: magenta/video 6: red/security 7: green/access 8: black/joker Defaults to primary device group
updateinterval	optional double	defines the expected update interval of this input, i.e. how often the actual state is reported by the device. Defaults to 0, which means no fixed interval
hardwarename	optional string	a string describing the button element, such as "up" or "down" etc.

Sensor object in the sensors field of the init message

Field	Type	Description
id	optional string	string id, unique within the device to reference the sensor via vDC API.
sensortype	optional integer	Defines the type of sensor: 0: undefined 1: temperature in degrees celsius 2: relative humidity in % 3: illumination in lux 4: supply voltage level in Volts 5: CO (carbon monoxide) concentration in ppm 6: Radon activity in Bq/m3 7: gas type sensor 8: dust, particles <10μm in µg/m3 9: dust, particles <2.5μm in µg/m3 10: dust, particles <1μm in µg/m3 11: room operating panel set point, 0..1 12: fan speed, 0..1 (0=off, <0=auto) 13: wind speed in m/s 14: Power in W 15: Electric current in A 16: Energy in kWh 17: Electric Consumption in VA 18: Air pressure in hPa 19: Wind direction in degrees 20: Sound pressure level in dB 21: Precipitation in mm/m2 22: CO2 (carbon dioxide) concentration in ppm Defaults to 0 (undefined)
usage	optional integer	Defines usage: 0: undefined 1: room (indoors) 2: outdoors 3: user interaction Default is 0 (undefined)
group	optional integer	defines the primary color (group) of the button: 1: yellow/light, 2: grey/shadow 3: blue/heating 4: cyan/audio 5: magenta/video 6: red/security 7: green/access 8: black/joker Defaults to primary device group
updateinterval	optional double	defines the time precision of the sensor, i.e. how quickly it reports relevant changes. For sensors with a regular polling mechanism, this actually is the expected update interval. For sensors that update only when change is detected, this denotes the timing accuray of the update. Defaults to 5 seconds

alivesigninterval	optional double	defines after what time with no sensor update the sensor should be considered offline/invalid, in seconds. Defaults to 0 (meaning: no guaranteed alive reports)
changesonlyinterval	optional double	defines the minimum time interval between reporting the same sensor value again, in seconds. Defaults to 5 minutes (300 seconds) Note that this value is only a default used for new devices. The vDC API allows to change this value later.
hardwarename	optional string	a string describing the button element, such as "up" or "down" etc.
min	optional double	minimal value, defaults to 0
max	optional double	maximal value, defaults to 100
resolution	optional double	sensor resolution, defaults to 1

action object in actions field of the init message

Field	Type	Description
description	optional string	description text (for logs and debugging) of the action
params	optional object	defines the parameters of the device action

configuration object in configurations field of the init message

Field	Type	Description
id	string	configuration id
description	string	description text for the configuration

dynamicaction object in dynamicactions field of the init message and in the dynamicAction message

Field	Type	Description
title	string	The title (device-side user-assigned string in user language) for the dynamic action
description	optional string	description text (for logs and debugging) of the action
params	optional object	defines the parameters of the device action

state object in states field of the init message

Field	Type	Description
description	optional string	description text (for logs and debugging) of the state
type, siunit, min, max...	fields	definition of the state, see value description fields below

event object in events field of the init message

Field	Type	Description
description	optional string	description text (for logs and debugging) of the state
type, siunit, min, max...	fields	definition of the state, see value description fields below

property object in properties field of the init message

Field	Type	Description
readonly	optional boolean	can be set to make the property read-only (from the vDC API side - the device implementation can use updateProperty to update/push the value)
type, siunit, min, max...	fields	definition of the property, see value description fields below

value description fields (for action parameters, states, properties)

Field	Type	Description
type	string	must be one of "numeric", "integer", "boolean", "enumeration" or "string"
siunit	optional string	defines the SI-unit of numeric type
default	optional numeric	default value according to type (not for enumerations)
min	optional double	minimal value for numeric and integer types
max	optional double	minimal value for numeric and integer types
resolution	optional double	resolution for numeric types
values	array of strings	possible values for enumeration type, default value prefixed with an exclamation mark (!).

Messages from vdcd to device(s)

vdcd sends (depending on the features selected in the *init* message) the messages shown in the table below.

If devices were created with a *tag* (which is required when creating multiple devices on a single connection), all JSON protocol messages will include the *tag* field, and all simple text protocol messages will be prefixed by *tag* plus a colon.

JSON protocol	Simple protocol	Description
{ *'message':'status',* *'status':**s**,* *'errorcode':**e**,* *'errormessage':**m**,* *'errordomain':**d*** *'tag':**t*** *}*	OK or ERROR=**m**	Status for *init* message. If ok, **s** is the string "ok" in the JSON protocol. **m** is a textual error message **e** is the vdcd internal error code **d** is the vdcd internal error domain **t** is the tag (only present if device was created with a tag in the *init* message)
{ *'message':'channel',* *'index':**i**,* *'id':**id**,* *'type':**ty**,* *'value':**v**,* *'tag':**t*** *}*	C**i**=**v** or with tag: **t**:C**i**=**v**	Output channel index **i** has changed its value to **v**. v is a double value. The device implementation should forward the new channel value to the device's output. The JSON variant of this message additionally reports the channel's name/idstring as **id** and the channel type as **ty**: 0: undefined 1: brightness for lights 2: hue for color lights 3: saturation for color lights 4: color temperature for lights with variable white point 5: X in CIE Color Model for color lights 6: Y in CIE Color Model for color lights 7: shade position (blinds, outside) 8: shade position (curtains, inside) 9: shade angle (blinds, outside) 10: shade angle (curtains, inside) 11: permeability (smart glass) 12: airflow intensity 13: airflow direction (0=undefined, 1=supply/down, 2=exhaust/up) 14: airflow flap position 15: ventilation louver position **t** is the tag (only present if device was created with a tag in the *init* message)
{ *'message':'move',* *'index':**i**,* *'direction':**d**,* *'tag':**t*** *}*	MV**i**=**d** or with tag: **t**:MV**i**=**d**	When the init message has specified *move=true*, the vdcd can request starting or stopping movement of channel **i** as follows: 0: stop movement 1: start movement to increase channel value -1: start movement to decrease channel value **t** is the tag (only present if device was created with a tag in the *init* message)

{ *'message':'control',* *'name':**n,*** *'value':**v*** *}*	CTRL.**n**=**v**	When the init message has specified *controlvalues=true*, the vdcd will forward control values received for the device. **n** is the name of the control value (such as "heatingLevel", "TemperatureZone", "TemperatureSetPoint" etc.) **v** is a double value
{ *'message':'sync',* *'tag':**t*** *}*	SYNC or with tag: **t**:SYNC	When the init message has specified *sync=true*, the vdcd can request updating output channel values by sending *sync*. The device is expected to update channel values (using the "channel"/"C" message, see below) and then sending the *synced* message. **t** is the tag (only present if device was created with a tag in the *init* message)
{ *'message':* *'scenecommand',* *'cmd':**c*** *'tag':**t*** *}*	SCMD=c or with tag: **t**:SCMD=**c**	When the init message has specified scenecommands=true, the vdcd will send this message for some "special" scene calls, because not all scene call's semantics are fully represented by channel value changes alone. Currently, **c** can be one of the following values: OFF, SLOW_OFF, MIN, MAX, INC, DEC, STOP, CLIMATE_ENABLE, CLIMATE_DISABLE, CLIMATE_HEATING, CLIMATE_COOLING, CLIMATE_PASSIVE_COOLING. Note that in most cases, the scene commands are already translated into channel value changes at the vDC level (e.g. MIN, MAX), so there's usually no need to do anything at the device level. STOP is a notable exception for devices that need significant time to apply values (like blinds). These should respond to the STOP scene command by immediately stopping movement and then synchonizing the actual position back using "channel"/C messages.

Operations related to multiple device configurations are only available in the JSON protocol as follows:

JSON protocol	Description
{ *'message':'setConfiguration',* *'id':**configid**,* *'tag':**t*** *}*	The device is requested to the configuration identified by id **configid**. The device implementation MUST disconnect all devices affected by the configuration change (using the *bye* message), and then re-connect the device(s) with new *init* messages describing the new configuration (number of buttons, sensors, etc.) and containing **configid** in the *currentConfigId* field.

Operations related to 'single' devices are only available in the JSON protocol as follows:

JSON protocol	Description
{ *'message':'invokeAction',* *'action':**a**,* *'params':**p,*** *'tag':**t*** *}*	The action **a** has been invoked with parameters **p**. The device implementation must perform the action and then must return a *confirmAction* message (unless confirming actions is disabled by including the *noconfirmaction* field in the init message) **t** is the tag (only present if device was created with a tag in the *init* message)

{ *'message':'setProperty',* *'property':**p**,* *'value':**v,*** *'tag':**t*** *}*	The property **p** has been changed to value **v**. The device implementation should update its internal value for this property.

Messages from device(s) to vdcd

the device(s) can send (depending on the features selected in the *init* message) the messages shown in the table below.

If devices were created with a *tag* (which is required when creating multiple devices on a single connection), all JSON protocol messages must include the *tag* field, and all simple text protocol messages must be prefixed by *tag* plus a colon to identify the device.

JSON protocol	Simple protocol	Description
{ *'message':'bye',* *'tag':**t*** *}*	BYE or with tag: **t**:BYE	The device can send this message to disconnect from the vdcd, for example when it detects its hardware is no longer accessible. Just closing the socket connection has the same effect as sending *bye* (in case of multiple tagged devices, closing socket is like sending *bye* to each device) **t** is the tag (only needed when device was created with a tag in the *init* message)
{ *'message':'log',* *'level':**n,*** *'text':**logmessage,*** *'tag':**t*** *}*	L**n**=**logmessage** or with tag: **t**:L**n**=**logmessage**	The device can send a **logmessage** to the vdcd log. **n** is the log level (by default, levels above 5 are not shown in the log - 7=debug, 6=info, 5=notice, 4=warning, 3=error, 2=critical, 1=alert, 0=emergency) **t** is the tag (only needed when device was created with a tag in the *init* message)
{ *'message':'channel',* *'index':**i,*** *'type':**ty,*** *'id':**id,*** *'value':**v,*** *'tag':**t*** *}*	C**i**=**v** or with tag: **t**:C**i**=**v**	The device should send this message when its output channel with index **i** (or named **id** or typed **ty**, see below) has changed its value to **v** for another reason than having received a channel message (e.g. after initialisation, or for devices than can be controlled directly). Devices that cannot immediately detect output changes can specifiy *sync=true* in the *init* message, so the vdcd will request updating output channel values by sending *sync* only when these values are actually needed. The JSON variant of this message additionally allows selecting the channel by name/idstring **id** or type **ty** (instead of index **i**). **t** is the tag (only needed when device was created with a tag in the *init* message)
{ *'message':'button',* *'index':**i,*** *'id':**id,*** *'value':**v,*** *'tag':**t*** *}*	B**i**=**v** or with tag: **t**:B**i**=**v**	The device should send this message when the state of its button at index **i** (or named **id**, see below) has changed. If the button was pressed, **v** must be set to 1, if the button was released, **v** must be set to 0. To simulate a button press+release with a single message, set **v** to the press duration in milliseconds. **t** is the tag (only needed when device was created with a tag in the *init* message) The JSON variant of this message allows selecting the button by name/idstring **id** (instead of index **i**).

{ *'message':'input',* *'index':**i,*** *'id':**id,*** *'value':**v,*** *'tag':**t***	l**i**=**v** or with tag: **t**:l**i**=**v**	The device should send this message when the state of its input at index **i** (or named **id**, see below) has changed. If the input has changed to active **v** must be set to 1, if the input has changed to inactive, **v** must be set to 0. **t** is the tag (only needed when device was created with a tag in the *init* message) The JSON variant of this message allows selecting the button by name/idstring **id** (instead of index **i**).
{ *'message':'sensor',* *'index':**i,*** *'id':**id,*** *'value':**v,*** *'tag':**t*** *}*	S**i**=**v** or with tag: **t**:S**i**=**v**	The device should send this message when the value of its sensor at index **i** (or named **id**, see below) has changed. **v** is the new value (double) and should be within the range specified with *min* and *max* in the *init* message. **t** is the tag (only needed when device was created with a tag in the *init* message) The JSON variant of this message allows selecting the button by name/idstring **id** (instead of index **i**).
{ *'message':'synced',* *'tag':**t*** *}*	SYNCED or with tag: **t**:SYNCED	The device must send this message after receiving *sync* and having updated output channel values. **t** is the tag (only needed when device was created with a tag in the *init* message)

Operations related to 'single' devices are only available in the JSON protocol as follows:

JSON protocol	Description
{ *'message':'confirmAction',* *'action':**a,*** *'errorcode':**ec,*** *'errortext':**et,*** *'tag':**t*** *}*	Unless confirming actions is disabled by including the *noconfirmaction* field in the init message, the device must send this message to confirm the execution (or failure) of action **a** that has been invoked before. It can return a status code in **ec**, which must be 0 for successful execution or another value indicating an error. **et** can be used to supply an error text. **t** is the tag (only present when device was created with a tag in the *init* message)
{ *'message':'updateProperty',* *'property':**p,*** *'value':**v,*** ***'**push':**push*** *'tag':**t*** *}*	The device can send this message to update the value **v** of the property **p**, as seen from the vdc API. If the optional **push** boolean value is set, the property change will also be pushed upstream. Note it is possible to just push the current property value without changing its value by omitting value, only specifying push. **t** is the tag (only present when device was created with a tag in the *init* message)
{ *'message':'pushNotification',* *'statechange':{**s:v**}* *'events':**[e1,e2,...]*** *'tag':**t*** *}*	The device can use this message to push state changes and events. The *statechange* field is an optional object assinging a new value **v** to a state **s**, the *events* field is an optional array of event ids (**e1,e2,...**) to be pushed along the state change (or by themselves if the *statechange* field is missing). **t** is the tag (only present when device was created with a tag in the *init* message)

{ *'message':'dynamicAction',* *'changes':{**id:actiondesc, ...**}* *'tag':**t*** *}*	The device can use this message to add, change or remove dynamic actions while the device is already operating. The *changes* field is a JSON object, containing one or multiple actions identified by their **id**s and described by an **actiondesc** (same format as dynamicAction in the init message, see above). If a dynamic action is to be removed, pass null for **actiondesc**. **t** is the tag (only present when device was created with a tag in the *init* message)

Experimenting

The external device API can be experimented easily with by connecting via telnet, then pasting an *init* message and then simulating some I/O.
The following paragraphs show this for different device types. Please also refer to the sample code in different languages contained in the *external_devices_samples* folder of the vdcd project.

Light button simulation

Connect to the device API with telnet:

```
telnet localhost 8999
```

Now copy and paste (single line!!) a simple *init* message defining a light button:

```
{'message':'init','protocol':'simple','uniqueid':'experiment42','buttons':
[{'buttontype':1,'group':1,'element':0}]}
```

The vdcd responds with:

```
OK
```

Now, the vdcd has created a light button device. If this is the vdcd of a P44-DSB, you can see the device in the P44-DSB web interface, and if the vdcd is connected to a digitalSTROM system, a new button device will appear in the dSS. By default, it will be in the default room for the “external devices” vdc, but you can drag it to an existing room with digitalSTROM light devices.

Now you can switch lights by simulating a button click (250ms active) with

```
B0=250
```

For dimming, the button can be held down...

```
B0=1
```

...and later released

```
B0=0
```

Light dimmer simulation

Connect to the device API with telnet:

```
telnet localhost 8999
```

Now copy and paste (single line!!) a simple *init* message defining a dimmer output:

```
{'message':'init','protocol':'simple','uniqueid':'experiment42b','output':‘
light’}
```

The vdcd responds with:

```
OK
```

Now, the vdcd has created a light dimmer device. If this is the vdcd of a P44-DSB, you can see the device in the P44-DSB web interface, and if the vdcd is connected to a digitalSTROM system, a new light device will appear in the dSS

Now you can call scenes in the room that contains the light device (or use the sprocket button in the P44-DSB web interface to directly change the brightness). You will see channel value changes reported from the external device API:

```
C0=3.120000
C0=8.190000
C0=14.040000
C0=21.840000
C0=30.810000
C0=40.950000
C0=56.160000
C0=63.960000
C0=69.810000
C0=78.000000
C0=79.950000
C0=78.000000
C0=65.130000
C0=53.040000
C0=42.120000
C0=33.150000
C0=26.910000
C0=40.950000
C0=53.040000
C0=58.110000
C0=63.180000
```

Temperature sensor simulation

Connect to the device API with telnet:

```
telnet localhost 8999
```

Now copy and paste (single line!!) a simple *init* message defining a temperature sensor:

```
{'message':'init','protocol':'simple','group':
3,'uniqueid':'experiment42c','sensors':[{'sensortype':1,‘usage’:1,‘group’:
48,'min':0,'max':40,'resolution':0.1}]}
```

The vdcd responds with:

```
OK
```

Now, the vdcd has created a temperature sensor device.

Now you can simulate temperature changes with

```
S0=22.5
```

e.g. to report a room temperature of 22.5 degree celsius.

A light dimmer and a button sharing one API connection

Connect to the device API with telnet:

```
telnet localhost 8999
```

Now copy and paste (single line!!) two *init* messages packaged into a JSON array, defining the dimmer and the button device:

```
[ {'message':'init', 'tag':'DIMMER', 'protocol':'simple', 'group':3,
'uniqueid':'experiment42d', 'output':'light'}, {'message':'init',
'tag':'BUTTON', 'uniqueid':'experiment42e', 'buttons':[{'buttontype':1,
'group':1, 'element':0}]} ]
```

The vdcd responds with:

```
OK
```

Now simulate a button click (250ms active) with

```
BUTTON:B0=250
```

As both dimmer and button are initially in the same room, the button acting (by default) as a room button will switch on the light, so you will see:

```
DIMMER:C0=100.000000
```

A single device (simple kettle) with some actions, a state, some events and a property

Connect to the device API with telnet:

```
telnet localhost 8999
```

Now copy and paste (single line!!) two *init* messages packaged into a JSON array, defining the simple kettle single device:

```
{ 'message':'init', 'iconname':'kettle', 'modelname':'kettle',
'protocol':'json', 'uniqueid':'my-kettle', 'name':'virtual kettle',
'output':'action', 'noconfirmaction':true, 'actions': { 'std.stop':
{'description':'stop heating'},  'std.heat':{'description':'heat
water','params': {'temperature':
{'type':'numeric','siunit':'celsius','min':20,'max':100,'resolution':
1,'default':100} } } }, 'states': { 'operation':{ 'type':'enumeration',
'values':['!ready','heating','detached'] } }, 'events': { 'started':null,
'stopped':null, 'aborted':null, 'removed':null }, 'properties':
{ 'currentTemperature':{ 'readonly':true, 'type':'numeric',
'siunit':'celsius', 'min':0, 'max':120, 'resolution':1 }, 'mode':
{ 'type':'enumeration', 'values':['!normal','boost'] } } }
```

The vdcd responds with:

```
{ "message":"status","status":"ok"}
```

Now simulate state changes with

```
{ "message":"pushNotification","statechange":{ "operation":"heating" },
"events":["started"] }

{ "message":"pushNotification","statechange":{ "operation":"detached" },
"events":["aborted","removed"] }
```

Or a temperature change with

```
{ "message":"updateProperty","property":"currentTemperature", "value":42,
"push":true }
```

When an action is invoked on the device, you will see something like:

```
{"message":"invokeAction","action":"std.heat","params":{"temperature":
42.420000}}
```

When a property is changed in the device:

```
{"message":"setProperty","property":"mode","value":"boost"}
```

Button device with three configurations to switch between

This is for a device with two buttons that can be used as either a two-way (rocker) button or as two separate single buttons.

Connect to the device API with telnet:

```
telnet localhost 8999
```

Now copy and paste (single line!!) a *init* message defining the button in the two-way configuration:

```
{'message':'init','protocol':'json','group':
1,'uniqueid':'multiConfigSample','buttons':[{'buttontype':
2,'hardwarename':'down','element':1}, {'buttontype':
2,'hardwarename':'up','element':
2}],'currentConfigId':'twoWay','configurations':
[{'id':'twoWay','description':'two-way rocker button'},{'id':'oneWay',
'description':'separate buttons'},{'id':'twoWayInverse','description':'two-
way rocker button, direction inversed'}]}
```

The vdcd responds with:

```
{"message":"status","status":"ok"}
```

Now, the vdcd has created a two-way button, offering three configurations: "*twoWay*" (currently active), "*twoWayInverse*" and "*oneWay*".

Now, when the user changes the configuration to "*oneWay*", the device will receive the following message:

```
{"message":"setConfiguration","id":"oneWay"}
```

Now, the device implementation must disconnect the current device, and re-connect two devices with same *uniqueid*, but different *subdeviceindex* representing the new one-way configuration:

```
{'message':'bye'}

[{'message':'init','tag':'A','protocol':'json','group':
1,'uniqueid':'multiConfigSample', 'subdeviceindex':0,'buttons':
[{'buttontype':1,'hardwarename':'single','element':
0}],'currentConfigId':'oneWay','configurations':
[{'id':'twoWay','description':'two-way rocker button'},{'id':'oneWay',
'description':'separate buttons'},{'id':'twoWayInverse','description':'two-
way rocker button, direction inversed'}]},
{'message':'init','tag':'B','protocol':'json','group':
1,'uniqueid':'multiConfigSample', 'subdeviceindex':1, 'buttons':
[{'buttontype':1,'hardwarename':'single','element':
0}],'currentConfigId':'oneWay','configurations':
[{'id':'twoWay','description':'two-way rocker button'},{'id':'oneWay',
'description':'separate buttons'},{'id':'twoWayInverse','description':'two-
way rocker button, direction inversed'}]}]
```

Similarily, when the sure changes the configuration back to "*twoWay*" or "*twoWayInverse*", both single buttons must disconnect and a init message for a two-way button must be sent.

Note that this is even the case when switching between "*twoWay*" or "*twoWayInverse*" - although the number of virtual devices does not change, the configuration (button modes, ids, names) does, so the device must be disconnected and re-connected.

Two button devices that can be combined (at dSS/vdSM level) to act as a single two-way button

Connect to the device API with telnet:

```
telnet localhost 8999
```

Now copy and paste (single line!!) the following array containing two *init* messages, defining two buttons which have consecutive *subdeviceindex* and *combinables* set to 2:

```
[{"message":"init","protocol":"simple","tag":"up","uniqueid":"p44_dummy_but
ton_23465","subdeviceindex":0,"buttons":[{"buttontype":0,"combinables":
2,"hardwarename":"up","element":0}]},
{"message":"init","protocol":"simple","tag":"down","uniqueid":"p44_dummy_bu
tton_23465","subdeviceindex":1,"buttons":[{"buttontype":0,"combinables":
2,"hardwarename":"down","element":0}]}]
```

The vdcd responds with:

```
OK
```

Now, the vdcd has created two button devices with consecutive subdevice indices (last byte of dSUID) 0 and 1.

From the external device API side, these work like any other button, see other example.

For the dSS configurator however, the combinables==2 means that these two buttons can be paired to appear as a single two-way in the dSS.