digitalSTROM System Interfaces

digitalSTROM

Version: v1.3-branch*

August 19, 2015

*Revision: 3c451f5c0c98db7edb9555940b5215106499d5d1

digitalSTROM Alliance
Building Technology Park Zurich
Brandstrasse 33
CH-8952 Schlieren
Switzerland

Contents

1 Introduction

This document contains instructions how to use and access the digitalSTROM Server, both for internal addon modules that enhance the system functionality and for external applications.

The application interface itself is defined in two additional documents:

- Webservice via JSON
- Server Scripting

The documents can be downloaded from http://developer.digitalstrom.org/Architecture.

> **Notice** This documents refers to digitalSTROM system release R1.6 and dSS release 1.17.3. Any later release may have additional or removed functionality.

2 Webservices

2.1 JSON

External applications communicate with the digitalSTROM Server through the JSON API. The full documentation and description of all the available JSON functions and parameters can be found on the digitalSTROM developer website in the dss download directory. Please refer to the document version corresponding to your dSS software release:

http://developer.digitalstrom.org/download/dss/

Access is granted with a token based login described in 6.

JSON requests to the API are built up like this:
https://<server ip>:<port>/json/<class>/<function>?<paramter>&<parameter>

For example this function calls scene 5 on light devices in zone 1307:
https://10.0.0.2:8080/json/zone/callScene?id=1307&groupID=1&sceneNumber=5&force=true&token=xxxxxxxx

Not all JSON functions take parameters, for example *json/apartment/getConsumption*.

The JSON formatted reply returns true for "ok" if the digitalSTROM Server successfully processed the query, and JSON objects and arrays if data was requested.

Example reply (json/apartment/getConsumption):

```
{
     "ok": true,
     "result": {
         "consumption": 74
     }
}
```

2.2 SOAP

The SOAP interface has been removed.

3 Property Tree

The property tree allows the dSS to expose data to the outside world, it also provides means of communication with internal scripts, serves as a data storage for internal scripts and remote applications (such as smartphone apps or HTML based user interfaces) and also allows limited control over the scheduled events queue.

Each node of the property tree can be either a container, holding further child nodes, or a leaf node, carrying actual data. When you look at the root level, you will see a logical structure of five nodes, each holding specific system properties:

- **/system** - general information about the dSS, information and control of the event queue
- **/config** - configuration of the dSS and it's subsystems
- **/apartment** - information about your digitalSTROM installation, here you will see the configured zones, connected meters, devices and their configurations.
- **/usr** - shared dynamic configuration for all Apps (like User-Defined-Actions, system states, triggers ...)
- **/scripts** - internal scripts running on the dSS will show up here by their configured script_id's and can use this area to store data.

3.1 Supported Data Types

The property tree supports three data types:

- **string**: - sequence of characters
- **boolean**: - "true" or "false"
- **integer**: numeric integer value

3.2 /system

The system block covers general information about the dSS and the host it is running on, shows a list of scheduled events in the event queue and allows to cancel them, it also presents security settings such as available users and application tokens.

3.2.1 /system/uptime

This is a leaf node which contains the uptime of the dSS in seconds.

3.2.2 /system/version

This is a container node that holds information about the dSS version, version of the Linux distribution that the dSS is running on, name of the build host and the git revision of the dSS source code.

3.2.3 /system/EventInterpreter

This is a container node that holds information about the event queue of the dSS, providing statistics on processed events and also listing the scheduled events.

Note: removing a scheduled event node will remove the event from the queue and thus prevent it's execution.

3.2.4 /system/security

This is a container node that hols information about system users and application tokens.

/system/security/users Provides information on users available on the system, along with their salt and encrypted password.

/system/security/applicationTokens This is a container node that provides information about enabled application tokens.

/system/security/roles Currently unsupported.

3.2.5 /system/host

This is a container node that provides information about the network interfaces and network configuration of the host on which the dSS is running.

3.2.6 /system/js

This is a container node that provides information about some internal script settings

/system/js/timings No clue.

/system/js/features Not sure.

/system/js/logfiles This node contains a list of internal script log files and their locations on disk. These log files can be downloaded via the JSON API.

3.3 /config

This node holds information about the system configuration. On the top level you will find settings for various directories used by the dSS, like directory where to search for data, webroot of the http server, etc.

Below is a list of directory configuration nodes:

- node name: ***datadirectory***
  value type: *string*
  description: *the data directory of the dSS, most other directories and files will be stored relative to it unless configured otherwise. For example, apartment.xml will be stored in the data directory.*

- node name: ***configdirectory***
  value type: *string*
  description: *directory where dSS will search for configuration files like config.xml, subscriptions, etc.*

- node name: ***webrootdirectory***
  value type: *string*
  description: *web root directory of the builtin web server*

- node name: ***jslogdirectory***
  value type: *string*
  description: *directory where to store log files that are produced by internal JS scripts that are running on the dSS*

- node name: ***savedpropsdirectory***
  value type: *string*
  description: *directory where the dSS will save persitent property tree entries or saved properties, configurations of internal scripts will usually be stored here.*

Further, configuration of each subsystem is stored under the **/config/subsystems** node. One leaf node, that is present in each subsystem is the **loglevel** node. The log level is stored as a numeric value, the higher the value the less log output is produced, the values have the following meaning:

- loglevel 0: debug
- loglevel 1: information
- loglevel 2: warning
- loglevel 3: error
- loglevel 4: fatal

Apart from the **loglevel**, another common subsystems node is **enabled**. It's a leaf node containing a boolean value that specifies if the subsystem is currently enabled or disabled.

Note: while each subsystem will show an **enabled** node, not all subsystems can be actually disabled.

3.3.1 /config/subsystems/Metering

This node contains information about the metering subsystem.
Following leaf nodes are available:

- node name: ***storageLocation***
  value type: *string*
  description: *location of the rrd metering databases on disk*

- node name: ***rrdDaemonAddress***
  value type: *string*
  description: *URI of the rrd cache daemon, if rrdcached is being used*

3.3.2 /config/subsystems/Apartment

This node contains information about the apartment subsystem.

- node name: ***configfile***
  value type: *string*
  description: *location of the apartment.xml configuration file on disk*

3.3.3 /config/subsystems/DSSim

Although this subsystem is showed as enabled, it is currently inactive and deprecated, simulation will be implemented outside of the dSS.

3.3.4 /config/subsystems/DSBusInterface

This node contains information about the dS485 bus interface subsystem.

- node name: ***connectionURI***
  value type: *string*
  description: *URI pointing to the dS485 daemon, all communication between the dSS and the connected hardware goes via this interface*

Note: if the dS485 connection URI is invalid or if the dS485 daemon is not running, dSS will still start up and try to connect to the given URI repeatedly. However, as long as the connection is not established, you will not see any active devices and you will not be able to control and configure your digitalSTROM installation.

3.3.5 /config/subsystems/WebServer

This node contains information about the built in web server configuration.

- node name: ***listen***
  value type: *string*
  description: *interfaces and ports on which the web server is listening for incoming connections. By default SSL is used, however an "h" behind the port means, that SSL was disabled for the given port and that it is available without encryption.*

- node name: ***trustedPort***
  value type: *integer*
  description: *when accessing the dSS via a trustedPort it is enough to provide a valid username in the HTTP Authorization header, the dSS will login this user automatically.*

- node name: ***webroot***
  value type: *string*
  description: *location web root directory on disk*

- node name: ***bindip***
  value type: *string*
  description: *ip of the interface to which the web server is bound*

- node name: ***announcedport***
  value type: *integer*
  description: *port that is announced via avahi/bonjour*

- node name: ***sslcert***
  value type: *string*
  description: *location of the web servers SSL certificate on disk*

- node name: ***sessionTimeoutMinutes***
  value type: *integer*
  description: *timeout value of the session*

/config/subsystems/WebServer/files This node contains a list of files that can be downloaded from the dSS. In order to download the file issue an HTTP GET request to the following URL (depending on your setup):
http(s)://hostip:port/download/filename.extension

3.3.6 /config/subsystems/EventInterpreter

- node name: ***subscriptionfile***
  value type: *string*
  description: *location of the main subscription file on disk*

- node name: ***subscriptiondir***
  value type: *string*
  description: *directory where dSS will search for further subscription configurations*

3.3.7 /config/geodata

This node contains information on the dSS location as well as sunset and sunrises times, the data is updated automatically when the dSS is running. Following information is available:

- node name: ***latitude***
  value type: *string*
  description: *geographic coordinate of the dSS*
- node name: ***longitude***
  value type: *string*
  description: *geographic coordinate of the dSS*
- node name: ***sunrise***
  value type: *string*
  description: *time when the sun rises at the given location (see **latitude** and **longitude**)*
- node name: ***sunset***
  value type: *string*
  description: *time when the sun sets at the given location (see **latitude** and **longitude**)*
- node name: ***civil_dawn***
  value type: *string*
  description: *time of civil dawn at the given location (see **latitude** and **longitude**)*
- node name: ***civil_dusk***
  value type: *string*
  description: *time of civil dusk at the given location (see **latitude** and **longitude**)*
- node name: ***nautical_dawn***
  value type: *string*
  description: *time of nautical dawn at the given location (see **latitude** and **longitude**)*
- node name: ***nautical_dusk***
  value type: *string*
  description: *time of nautical dusk at the given location (see **latitude** and **longitude**)*

- node name: ***astronomical_dawn***
  value type: *string*
  description: *time of astronomical dawn at the given location (see **latitude** and **longitude**)*

- node name: ***astronomical_dusk***
  value type: *string*
  description: *time of astronomical dusk at the given location (see **latitude** and **longitude**)*

3.4 /apartment

This section provides information about your digitalSTROM installation, it lists all available meters and devices, configured zones and more.

3.4.1 /apartment/zones

This node contains a list of zones that are configured in the apartment.

Note: the zone with id zero is a special virtual zone that contains all available devices and all devices that were known to the dSS.

Each **zone** node contains the same set of sub nodes:

- node name: ***ZoneID***
  value type: *integer*
  description: *numeric id of the zone*

- node name: ***name***
  value type: *string*
  description: *name of the zone as set by the user*

Further, each **zone** node provides the following container nodes: **devices**, **SensorHistory** and **groups**.

/apartment/zones/zoneX/devices The devices node contains a list of individual device nodes, each device node has the following properties:

- node name: ***dSID***
  value type: *string*
  description: *unique digitalSTROM ID, the device dSID*

- node name: ***present***
  value type: *boolean*
  description: *flag specifying if the device is currently present in the installation, or if this is a device that is known to the dSS but that is not currently available*

- node name: ***name***
  value type: *string*
  description: *name of the device as set by the user*

- node name: ***dSMeterDSID***
  value type: *string*
  description: *digitalSTROM id of the meter to which this device is connected*

- node name: ***ZoneID***
  value type: *integer*
  description: *numeric id of the zone in which the device resides*

- node name: ***functionID***
  value type: *integer*
  description: *function id of the device, for example the class of the device (i.e. yellow, grey, etc.) is encoded in the function id*

- node name: ***revisionID***
  value type: *integer*
  description: *revision id of the device which is the encoded firmware version*

- node name: ***productID***
  value type: *integer*
  description: *numeric id of the product which identifies the device type and can be decoded to map the human readable product types like KM, TKM, KL and so on.*

- node name: ***lastKnownZoneID***
  value type: *integer*
  description: *numeric id of the last known zone, for present devices this will be the same as the* ***ZoneID****.*

- node name: ***lastKnownMeterDSID***
  value type: *string*
  description: *digitalSTROM ID of the meter to which the device was last connected*

- node name: ***firstSeen***
  value type: *string*
  description: *time stamp when the device was seen by the dSS for the very first time*

- node name: ***lastDiscovered***
  value type: *string*
  description: *time stamp when the device wat last discovered by the dSS*

- node name: ***inactiveSince***
  value type: *string*
  description: *time stamp since when the device became inactive, this field only makes sense for devices where the **present** flag equals to false. For devices that are present this field should be ignored as it will show the unix epoch time.*
- node name: ***locked***
  description: *deprecated*
- node name: ***outputMode***
  value type: *integer*
  description: *numeric value representing the output mode of the device*
- node name: ***button***
  description: *container for button information nodes*
  - node name: ***id***
    value type: *integer*
    description: *numeric value representing the id of the button, i.e. zone/area/app*
  - node name: ***inputMode***
    value type: *integer*
    description: *button input mode configuration value, i.e. 2way-up, 1-way, etc.*
  - node name: ***inputIndex***
    value type: *integer*
    description: *index of the input buttons for this device*
  - node name: ***inputCount***
    value type: *integer*
    description: *total number of input buttons of the physical device*
  - node name: ***activeGroup***
    value type: *integer*
    description: *group in which events from this device are processed*
  - node name: ***setsLocalPriority***
    value type: *boolean*
    description: *this flag indicates the automatic setting of calls in area scenes*
- node name: ***SensorEvents***
  description: *list of sensor events if configured (for example ZWS "verbrauchsmeldung")*
- node name: ***tags***
  description: *deprecated*

- node name: ***groups***
  description: *container for a list of group nodes, providing the group membership information of the device*

- node name: ***sensorTable***
  description: *this node contains a list of nodes that provide information about sensors that are available for this device*

/apartment/zones/zoneX/SensorHistory List of sensor events that happened in the zone.

/apartment/zones/zoneX/groups/groupX This node contains information on the group, such as the group id and name, last called scene value and a list of devices that are part of the group.

- node name: ***group***
  value type: *integer*
  description: *numeric id of the group*

- node name: ***name***
  value type: *string*
  description: *human readable name of the group*

- node name: ***scenes***
  description: *list of nodes with custom scene names, if scenes have been renamed by the user*

- node name: ***devices***
  description: *container for a list of device nodes that carry information about devices which are part of the group*

3.4.2 /apartment/dSMeters

This node contains a list of dSMs that are available in the digitalSTROM installation. Each dSM node has the following properties:

- node name: ***dSID***
  value type: *string*
  description: *unique digitalSTROM ID, dSM dSID*

- node name: ***powerConsumption***
  value type: *integer*
  description: *current power consumption of the dSM*

- node name: ***powerConsumptionAge***
  value type: *string*
  description: *time stamp when the power consumption value was recorded*

- node name: ***energyMeterValue***
  value type: *integer*
  description: *current energy meter value in Wh*
- node name: ***energyMeterValueWs***
  value type: *integer*
  description: *current energy meter value in Ws*
- node name: ***energyMeterValueAge***
  value type: *string*
  description: *time stamp when the energy meter values were recorded*
- node name: ***isValid***
  value type: *boolean*
  description: *flag indicating if the dSM has been read out by the dSS*
- node name: ***present***
  value type: *boolean*
  description: *flag indicating if the dSM is present in the installation and was found by the dSS*
- node name: ***energyLevelRed***
  description: *deprecated*
- node name: ***energyLevelOrange***
  description: deprecated
- node name: ***hardwareVersion***
  value type: *integer*
  description: *version of the dSM hardware*
- node name: ***armSoftwareVersion***
  value type: *integer*
  description: *version of the ARM firmware*
- node name: ***dspSoftwareVersion***
  value type: *integer*
  description: *version of the DSP firmware*
- node name: **apiVersion**
  value type: *integer*
  description: *version of the dSM API*
- node name: ***hardwareName***
  description: *deprecated*
- node name: ***name***
  value type: *string*
  description: *name of the dSM as set but the user*

- node name: ***zones***
  description: *container node, holding a list of **zone** nodes that represent the zones that are configured on this dSM. The structure of the **zone** nodes is the same as previously described.*

- node name: ***devices***
  description: *container node, holding a list of **device** nodes that represent the devices that are connected to this dSM. The structure of the **device** nodes is the same as previously described.*

3.5 /usr

This section holds some dynamic configured values, which are configured and used for all Apps globally.

3.6 /usr/states

The system-states are stored in this location. This states are managed by the digitalSTROM server itself in configuration and changing the values. They will be used as a filter for triggering a trigger and when a action-node should be executed.

3.7 /usr/addon-states

Each addon can register own states, these are stored here in a own subnode. This states are controlled by the proper addon by using scription-calls. This states canl be used as a filter for triggering a trigger and when a action-node should be executed.

3.8 /usr/triggers[0..x]

In this location all registered triggers are stored. That triggers can be registered on standardized events; when one of the events are raised, the dSS evaluates the definition found in *triggerPath* and when all parameters matches and conditions (such as timeframes and system-states) are met, it will raise a Event named *relayedEventName* with the original path of the trigger, the original parameters of the incomming event and the *additionalRelayingParameter*.

- node name: ***id***
  value type: *integer*
  description: *internal ID of that trigger-registration. Please don't touch it*

- node name: ***triggerPath***
  value type: *string*

description: *path of the trigger defination, where the matching parameters for the trigger is located. For description of the format please refer to scripting documentation*

- node name: ***relayedEventName***
  value type: *string*
  description: *the result event, which will be raised, when the trigger matched the conditions*

- node name: ***additionalRelayingParameter***
  value type: *string*
  description: *adding some extra parameters for the relayed event*

3.9 /usr/events[0..x]

In this location all User-Defined-Actions are stored. They will be accessible for all other apps and UIs for execute them oder register a trigger on them. The App *User Defined Action* configure that values and restore of them on startup, so modifying that events should be only done though the UDA-App.

- node name: ***id***
  value type: *integer*
  description: *internal ID of that UDA. Please don't touch it*

- node name: ***name***
  value type: *string*
  description: *name of the UDA*

- node name: ***lastSaved***
  value type: *int*
  description: *timestamp when the entry is been last saved*

- node name: ***lastExecuted***
  value type: *int*
  description: *timestamp when the entry is been last executed via UI*

- node name: ***actions***
  value type: *subnode - type action (please refer to the scripting documentation*
  description: *actions that will be executed*

- node name: ***conditions***
  value type: *subnode - type condition (please refer to the scripting documentation)*
  description: *condition which should be met when executed*

3.10 /scripts

This is the place where internal JS scripts that are running on the dSS will store their data, the node name of each sub node uses the script id that was configured for the particular script.

4 Events

The digitalSTROM Server is the central engine to process system events. Internally the server uses an event interpreter to process events and to execute event handlers from extension scripting modules and server addons.

Events originate from different sources:

- digitalSTROM System-Level-Events, originating from the dS485 bus
- digitalSTROM High-Level-Events, raised by dSS Addons
- Externally generated events, received through web service interface
- Server internal and data model related events
- Addon generated events

Events can be connected to an event handler using a subscription mechanism. Events carry context and parameters that allows context evaluation and further processing by the event handler.

The JSON API allows remote applications to register and wait for particular events. The remote call is blocking and will return when the event occur. The returned values contain the same parameters that would be passed to internal event handler.

The JSON API function "/json/event/raise" allows to inject events into the event queue. Required parameter is the event name, optionally additional parameters can be passed.

4.1 Subscriptions

4.1.1 Static Subscriptions

The dSS internal subscriptions to events are configured in the *data/subscriptions.xml*. Custom subscriptions for dSS Addons are placed in separate files in the the directory *data/subscriptions.d/*. These binding of scripts to specific events is a static configuration option which is evaluated once at startup of the dSS.

Subscriptions connect a handler to an event source and adds additional parameters that are required for the event handler execution.

The following excerpt shows how to run a script (data/initialize.js) on startup:

Listing 1: Subscription Example 1

```
<subscription event-name="running" handler-name="javascript">
  <parameter>
    <parameter name="filename1">data/initialize.js</parameter>
  </parameter>
</subscription>
```

4.1.2 Dynamic Subscriptions

The digitalSTROM Server JSON and Scripting API allow to dynamically add and remove subscriptions. Those subscriptions are not persistent.

4.1.3 Filter

The event interpreter is able to evaluate filter expressions for subscription. This allows to have an efficient preprocessing of events rather than running all events through a custom JavaScript handler.

The following example checks two conditions and only executes the event handler if both conditions match. The first filter expressions checks for the existence of the event property *phonenumber*, the second filter compares the event parameter *source* to a given string.

Listing 2: Filter Example 1

```
<subscription event-name="phonecall" handler-name="raise_event">
  <parameter>
    <parameter name="event_name">bell</parameter>
  </parameter>
  <filter match="all">
    <property-filter type="exists" property="phonenumber" />
    <property-filter type="matches" value="0123456789" property="source" /
        >
  </filter>
</subscription>
```

The second example has a condition to check for a particular scene command and raise a custom event *MyAlarm* if the scene command value matches "74" (which is a digitalSTROM Alarm System-Level-Event).

Listing 3: Filter Example 2

```
<subscription event-name="callScene" handler-name="raise_event">
  <parameter>
    <parameter name="event_name">MyAlarm</parameter>
  </parameter>
  <filter match="all">
    <property-filter type="matches" value="74" property="sceneID" />
  </filter>
</subscription>
```

4.2 Event Reference

The following sections list the digitalSTROM Server event classes and their parameters. digitalSTROM system or device level events are detailed in the dS-Basics document.

The common event parameter *originDeviceId* is either the dSID of the digitalSTROM Device from where the System-Level-Event has been initiated or one of the following values:

Pseudo originDeviceId	Description
0	Unknown Origin
1	Scripting
2	JSON
3	SOAP
4	Subscription
5	Simulation
6	Test

Table 1: Event Sources

digitalSTROM events and their parameter details are explained in the dS-Basics document. Please refer to the corresponding chapters.

4.2.1 callScene and undoScene

The *callScene* and *undoScene* events are raised if the digitalSTROM Server receives a call scene or undo scene action request. The source of the event may be either the digitalSTROM system, internally generated by dSS Addons or externally injected via remote JSON calls.

Listing 4: Example callScene Event

```
Parameter: 'groupID' = '1'
Parameter: 'sceneID' = '32'
Parameter: 'zoneID' = '4011'
Parameter: 'originDeviceID' = '3504175fe000000000183f2'
```

When originating from the digitalSTROM system this event is delayed by 2 seconds to ensure that only a single appropriate event is raised for consecutive pushbutton tips. To reduce latency effects the digitalSTROM Meter issues scene calls faster and before the last pushbutton tip takes place.

For special applications the corresponding *callSceneBus* event is raised as soon as the digitalSTROM Server receives the system level event from a digitalSTROM Meter.

4.2.2 buttonClick

The *buttonClick* event is raised if the digitalSTROM Server receives a push-button tip event from a digitalSTROM Device configured in the color "Joker" and working in "App Button" mode.

Listing 5: Example buttonClick Event

```
Parameter: 'clickType' = '1'
Parameter: 'buttonIndex' = '0'
```

When originating from the digitalSTROM system this event is delayed by 2 seconds to ensure that only a single appropriate event is raised for consecutive pushbutton tips.

For special applications the corresponding *buttonClickBus* event is raised as soon as the digitalSTROM Server receives the system level event from a digitalSTROM Meter.

4.2.3 deviceSensorEvent

The *deviceSensorEvent* event is raised if the digitalSTROM Server receives a sensor table event form a digitalSTROM Device. The event parameters refer to the devices property tree entry "sensorEvents/" branch where details and specific names of the event source are stored.

Listing 6: Example deviceSensor Event

```
Parameter: 'sensorIndex' = 'event0'
Parameter: 'sensorEvent' = 'event0'
```

4.2.4 running

The *running* event is raised by the event interpreter to indicate system startup. Scripts that need early initialization can make use of this event.

Notice At this time the data model is not synchronized with the digitalSTROM Meters. If any data model or property tree access is performed it has to be considered that the status of the devices is not up to date.

4.2.5 model_ready

After the initial readout of the connected digitalSTROM Meters and synchronization of the data model the event *model_ready* is raised.

4.2.6 dsMeter_ready

The *dsMeter_ready* event is raised each time a digitalSTROM Meter is newly connected and the data of its connected devices has been synchronized with the data mode.

Listing 7: Example dsMeter_ready Event

```
Parameter: 'dsMeter' = '3504175fe0000010000012e9'
```

4.3 Private Addon Events

The server scripting addons may provide individual script handler for their privately used events. Those events have to comply to the namespace convention that the event name is prefixed with the unique script_id name.

Rule 1 digitalSTROM Server Addons that implement private events have to prefix all event names with their own unique addon name.

In the following example the timed-events addon subscribes to the solar computer time updates. The subscription arranges for a new event to be raised with the name **timed-events.config**.

Listing 8: Addon Namespace

```
<subscription event-name="solar_computer.update" handler-name="raise_event
    ">
  <parameter>
    <parameter name="event_name">timed-events.config</parameter>
    <parameter name="actions_default">suntime-reschedule</parameter>
    <parameter name="script_id">timed-events</parameter>
    <parameter name="time">+60</parameter>
  </parameter>
</subscription>
```

4.4 Event Handler

4.4.1 JavaScript Handler

The digitalSTROM Server has the ability to run scripts using a JavaScript interpreter. The dSS Scripting API includes access to the digitalSTROM data model, the property tree, metering time series, and provides methods to execute digitalSTROM action requests and to raise new events using JSON or Scripting calls.

It is possible to execute several script files in the same context, the order of script execution is defined by the index number that is appended to the filename parameter, it allows to have 1-255 scripts in the same context, "holes" in the enumeration are not allowed. The following example show how to run two scripts in the same context for a given event:

Listing 9: Subscription Example 2

```
<subscription event-name="model_ready" handler-name="javascript">
  <parameter>
    <parameter name="filename1">data/funclibrary.js</parameter>
    <parameter name="filename2">data/initialize.js</parameter>
  </parameter>
</subscription>
```

The following parameters can be passed to a "javascript" event handler. Additional private parameters can be passed by appending "_default" to the

Parameter	Description
script_id	unique identifier for the script handler
filename1	path to javascript source file
actions_default	passed to the script as additional parameter default
eventpropertyxyz_override	override default value of eventpropertyxyz

Table 2: Parameter for handler-name="javascript"

parameter name. Existing parameters can be overridden using the "_override" postfix.

The embedding JavaScript interpreter context for a subscription contains additional meta data about the event source and the subscription in the global variable **raisedEvent**:

Listing 10: Variable raisedEvent

```
raisedEvent.name = callScene
raisedEvent.source = [object Object]
  raisedEvent.source.set = .zone(4011).group(1)
  raisedEvent.source.groupID = 1
  raisedEvent.source.zoneID = 4011
  raisedEvent.source.isApartment = false
  raisedEvent.source.isGroup = true
  raisedEvent.source.isDevice = false
raisedEvent.parameter = [object Object]
  raisedEvent.parameter.groupID = 1
  raisedEvent.parameter.sceneID = 32
  raisedEvent.parameter.zoneID = 4011
  raisedEvent.parameter.originDeviceID = 3504175fe000000000183f2
raisedEvent.subscription = [object Object]
  raisedEvent.subscription.name = callScene
```

The *source* field is provided as reference to the source of the device. Scripts can evaluate the isApartment, isGroup and isDevice fields to distinguish between the different kinds of digitalSTROM system events.

4.4.2 Raise Event Handler

The *raise_event* handler allows to propagate an event and forward it to another handler.

```
<subscription event-name="phonecall" handler-name="raise_event">
  <parameter>
    <parameter name="event_name">Mother-in-law-calls</parameter>
  </parameter>
  <filter match="all">
    <property-filter type="exists" property="phonenumber" />
    <property-filter type="matches" value="0123456789" property="source" /
        >
  </filter>
</subscription>
```

4.4.3 High Level Event Handler

The *highlevel* handler executes user defined actions. The corresponding actions are stored in a defined format in the */usr/events/* branch of the property tree.

Subscription highlevelevent

Parameter The parameter "id" is used to find the corresponding user defined action in the */usr/events/* subtree, see 5.7.

4.4.4 Action Execute Handler

The *action_execute* handler executes a sequence of user defined actions. The actions are stored in the property tree path given in the event data.

Subscription action_execute

Parameter The parameter *path* is used to find the corresponding user defined action in the */usr/events/* subtree, see 5.4. The optional parameter *delay* has a value in seconds and can used to defer the event execution and schedule it for a later time.

4.4.5 Trigger Handler

The *system_triffer* handler is a common evaluator of conditions. The handler has subscriptions to certain system events and then checks registered system triggers in the */usr/triggers/* branch of the property tree. A new event is raised if the conditions within the trigger path do match. Event name and additional parameters are stored in the property tree trigger node.

4.4.6 Sendmail Handler

The *sendmail* handler formats the raw e-mail text that is then delivered to the host systems mail transfer agent. Depending on the digitalSTROM Server compile-time configuration the raw text is only written out to a file and further processed by external agents.

Subscription sendmail

Parameter

Parameter	Description
to	recipients
from	sender
cc	carbon copy recipients
bcc	blind carbon copy recipients
subject	e-mail subject
body	e-mail body text
header	additional mail header lines, seperated by a new line character

Table 3: Parameter for handler-name="sendmail"

5 System Scripts

5.1 States - Systemstates

Systemstates are specialized values in the properties. Primary they represent a specific state of the installation like *Panic* or *Holiday*. These states can be used in the apps to change the behaviour of the system. They are already in use for the presence-simulator system-addon, where the presence-simulator is only in control of the holiday-state and which of the entries in the timed-events system-addon have a condition based on the holiday-state.

The systemstates are controlled by the digitalSTROM system itself and generated based on the actual configuration:

- states with apartment-scope:
  - presence
  - hibernation
  - daynight
  - twilight
  - daylight
  - holiday
  - alarm
  - alarm2
  - alarm3
  - alarm4
  - panic
  - fire
  - wind
  - rain

  - hail

- states with zone-scope:
  - zone.<zoneID>.light : set when light is on or off (called a specific on or off scene)
  - zone.<zoneID>.motion : set when a motion is detected be any sensoric device configured as motion-detector. State is only avaible when a proper sensoric device is present in zone.
  - zone.<zoneID>.presence : set when presence is detected be any sensoric device configured as presence-detector. State is only avaible when a proper sensoric device is present in zone.

- states with group-scope:
  - wind.group<group-id> : state is only avaible for usergroups configured as shadow groups. State is set by any sensoric devices configured as wind sensor.

- states with device-scope:
  - dev.<dsid>.<index> : state is avaible for each sensoric device (like a AKM). The <index> is for each input of the device

The system state values will be stored in */usr/states*.

5.2 States - Addonstates

A Addon can also register own states, which are controlled by the app. This states can queried by other apps, can be used as condition or trigger can be set on but only the registering app can set the specific value.

The Addonstates ares are stored in */usr/addon-states/<addon-id>*

5.3 Trigger

In the dSS all add-ons are event-driven, so a app can register some js-code when a specific events happens. There are a couple of events, which are generated through the system, mainly per taster-events. It is possible to make a subscription for all of these events, and every time when such a events happened, the scripts runs and make a check against some conditions. But there is a easier way: using a trigger. A trigger acts like a dynamic subscription, which can filter the system-events by it's parameter's, take some other conditions like states or timeframes in account and finally, if all requirements are met, raise a custom event for the app. This prefiltering is done in the server outside the scripting with greater performance and lesser resource impact. The parameter of a trigger must be

stored in the propertytree and registered using a script, which is provided by the dss (/usr/shared/dss/data/scripts/system_register_trigger.js):

- *registerTrigger(tPath, tEventName, tParamObj)*: This function register a trigger definition found in tPath. When a Event comes and matches to the trigger, a Event with with the name *tEventName* is raised. This relayed event carries all parameter of the original event plus the triggerpath (Parametername *path*) and parameter which are provided in *tParamObj*.
- *unregisterTrigger(tPath)*: This function unregister a trigger, which was defined in *tPath*.

Typical the registering of a trigger is done when the app is initializing or if a new behavior is configured in the app.

The layout of the property-nodes of a trigger is:

- <basenode>/triggers/0/<triggerdefinition #1>
- <basenode>/triggers/1/<triggerdefinition #2>
- ...

Each triggerdefinition must be places in a subpath *triggers/x* where *triggers* is required and *x* is a arbitary term. For the triggerdefinition there are many possibilities:

- *zone-scene*: This trigger will react on a scene-call in a zone. This will be typical happen through a taster click on a digitalSTROM Device. Parameters:
  - *type*: string, must be *zone-scene*
  - *zone*: integer, id of the zone. The zone with the id *0* will be used for apartmentwide scene-calls like bell or panic.
  - *group*: integer, id of the group. The group with the id *0* will be used for apartmentwide scene-calls like bell or panic.
  - *scene*: integer, id of the scene. please refer to the scene-table
  - *dsid*(optional): string, id of a device, which has caused the scene-call. this parameter can be omitted or *-1* to skip the source device filtering.
- *device-scene*: This trigger will react on a scene-call for a device. This will happen, if a local switch on a device has been used. Parameters:
  - *type*: string, must be *device-scene*
  - *dsid*: string, id of a device, which has caused the scene-call.

  - *scene*: integer, id of the scene. please refer to the scene-table. use *-1* as scene-id for triggering on all scene-calls from the device

- *device-sensor*: This trigger will react on a sensor-message of a device. On the 1.5 dS-System a sensor message will come from a consumption message, later there will be more sensoric events. Parameters:
  - *type*: string, must be *device-sensor*
  - *dsid*: string, id of a device, which has caused the scene-call.
  - *eventid*: integer, id of the sensor-event.
- *device-msg*: This trigger will react on a message from a taster, which is not interpreted by a statemaschine of the dsm. typical that will be black taster configured not to a specific color. Parameters:
  - *type*: string, must be *device-msg*
  - *dsid*: string, id of a device, which has caused the device-message.
  - *msg*: integer, id of the message.
  - *buttonIndex*(optional): integer, id of the button-index of the taster. if this parameter is omitted or *-1*, it will be ignored
- *custom-event*: This trigger will react on a custom event, which is raised by a other app or UI. Parameter:
  - *type*: string, must be *custom-event*
  - *event*: string, id of the custom-event
- *event*: This trigger will react on the event given in the name parameter. All parameters in "/parameter" have to match equally named properties in the event. Extra properties in the event are ignored. Parameter:
  - *type*: string, must be *event*
  - *name*: string, name of the matched event
  - *parameter/<name>*: string, optional, arbitrary named parameter to match event property
  - *parameter/<name1>*: string, optional, arbitrary named parameter to match event property

Additional to the trigger-parameter the *conditions* will first be evaluated, if a trigger is generally enabled. Please refer to conditions section.

5.4 Actions

Beside direct system-calls in the apps or using the JSON, there is a more convenient method to define some actions, which should be executed by digitalSTROM. The definition of the actions is stored in the propertytree and they can be started by raising the *action_execute* event with the parameter *path* where the definition of the actions are stored. The dSS-core will execute the actions step by step. When two actions causes dSM-Api calls there might be a small delay to ensure that all calls will be executed. Each step can also have a intentional defined delay to get some delayed action-sequences. Every time when a action should be executed, regardless it is started just now be a event or delayed step from a sequence, which has started earlier, a condition-definition will be evaluated (please refer to *Conditions* for details on conditions), which finally decides if the particular action should be executed. A condition can disable the execution of the stept in a action-sequence, but the whole action-sequence is not been stoped, so later steps of the sequence, which has been delayed might been executed anyhow, if later the conditions has been checked then successfully.

The layout of the property-nodes of a actiondefinition is:

- <basenode>/actions/0/<actionsdefinition #1>
- <basenode>/actions/1/<actionsdefinition #2>

Each action-step must be places in a subpath *actions/x* where *actions* is required and *x* is a arbitary term. For the actions-step definition there are many possibilities:

- *zone-scene*: This action will cause a scene-call to a specific zone and group Parameters:
  - *type*: string, must be *zone-scene*
  - *zone*: integer, id of the zone. The zone with the id *0* will be used for apartmentwide scene-calls like bell or panic.
  - *group*: integer, id of the group. The group with the id *0* will be used for apartmentwide scene-calls like bell or panic.
  - *scene*: integer, id of the scene. please refer to the scene-table
  - *force*(optional): bool, causes a force-call-scene instead of a call-scene. this parameter can be omitted
- *device-scene*: This action will make a scene-call for a device. Parameters:
  - *type*: string, must be *device-scene*
  - *dsid*: string, id of the target device

  - *scene*: integer, id of the scene. please refer to the scene-table.
  - *force*(optional): bool, causes a force-call-scene instead of a call-scene. this parameter can be omitted

- *device-value*: This action causes a *setOutputValue*-action to a device. It is not adviseable to use this action, please prefer scene-calls (like MIN-Scene and MAX-Scene for turning on/off), because this can be executed faster and the system can better keep track of the current room state. At last, the parameter *value* is bound to a specific behavior of the devices, and it is not guaranteed, that all devices might act on the same manner on that value. Parameters:
  - *type*: string, must be *device-value*
  - *dsid*: string, id of the target device
  - *value*: integer, 8-Bit value, send directly to the device.

- *zone-blink*: This action causes a *blink*-action to a group in a zone. Light devices will blink, shutters will shortly twitch etc. Parameters:
  - *type*: string, must be *zone-blink*
  - *zone*: integer, id of the zone. The zone with the id *0* will be used for apartmentwide calls.
  - *group*: integer, id of the group. The group with the id *0* will be used for apartmentwide calls.

- *device-blink*: This action causes a *blink*-action to device. Light devices will blink, shutters will shortly twitch etc. Parameters:
  - *type*: string, must be *device-blink*
  - *dsid*: string, id of the target device

- *custom-event*: This action will cause a execution of a custom event. This will not be the direct execution of the action-nodes of a custom event, rather this will raise a new event *highlevelevent* for requesting and queueing execution (and before evaluation the conditions) of the custom event. So if the conditions of a custom-event prohibits it execution, it will not be executed, regardless if the request comes from a external command or a relaying throught this action-step. Parameter:
  - *type*: string, must be *custom-event*
  - *event*: string, id of the custom-event

- *url*: This action will cause a URL-Request. Both HTTP-Request and HTTPS-Request are possible, but actual only GET-Requests with Parameters in the Query-String are possible. Be aware, when a URL is not accessable, it will slow down the execution of the action seriously. Parameter:
  - *type*: string, must be *url*
  - *url*: string, uri of the request, following the structure *http://www.digitalstrom.org*

Each action-step can have a additional parameter regardless of the type:

- *delay*: integer, execution delay in seconds from the initial event raising.

5.5 Conditions

The conditions provides a mechanism to manipulate the execution of a action or the evaluation of a trigger. They can be used to define conditions based on system-states or timeframes. Before a trigger is evaluated or a action is executed, all conditions must be checked successfully. If a condition is not defined, it is ignored. The layout of the property-nodes of a condition definition is:

- <basenode>/conditions/<type-of-condition>/...
- <basenode>/conditions/<type-of-condition>/...

This conditions are currently avaible:

- *enabled*: bool, must be true. If false, the check fails.
- *states*: defines conditions based of system-states which are located in /usr/state. Multiple states can de specified, which all must be equal to the current states. For each state two parameter must be provided:
  - conditions/states/<X>/name : name of the requested state
  - conditions/states/<X>/value : value of the requested state

  There can be more than one definition, the naming of <X> is arbitrary.
- *zone-states*: defines which last scene in one or more zones/groups must has been called. For each zone-state three parameter must be provided:
  - conditions/zone-states/<X>/zone : id of the zone
  - conditions/zone-states/<X>/group : id of the group
  - conditions/zone-states/<X>/scene : id of the last called scene

There can be more than one definition, the naming of <X> is arbitrary. For successful checking this condition, only one of this zone-states must be equal to the current last called scenes.

- *weekdays*:string, format comma-seperated with a number for each weekday (0: sunday, 1: monday ...6: saturday)
- *time-start*:string, format HH:MM:SS. The check fails, if the actual time is before the defined time-string.
- *time-end*:string, format HH:MM:SS. The check fails, if the actual time is before the required time-string.
- *date*: define time periods that are checked against the execution time of the action. For each date three parameter must be provided:
  - conditions/date/<X>/start : interval start; ISO date-time string (e.g. "20150707T170000")
  - conditions/date/<X>/end : interval end; ISO date-time string (e.g. "20150707T180000")
  - conditions/date/<X>/rrule : ical RRULE specifying the reoccurrence (e.g. "FREQ=DAILY;BYMONTH=7;UNTIL=20160801T000000Z")

  (In the above example the action is only executed between 5 pm and 6 pm in July 2015.)

  There can be more than one definition, the naming of <X> is arbitrary. For successful checking this condition, the current date has to be in only one of this date ranges.

5.6 Chaining Trigger and Actions

All trigger, conditions and actions are stored in the property-tree. When defining a action, it must be in a childnode named action resp. trigger. When a trigger matches and raises a relayed event, it provides the original path in the parameter *path*. To execute a action, you must raise a event with name *action_execute* and the parameter *path*. This convention is defined on purpose: by putting a trigger-definition and a action-definition in the same propertytree path and register the trigger with the path and event-name *action_execute*, a chaining of a trigger with a action is defined and that chain will be executed without using any app-specific script. This chain can be controlled outside by setting parameters like *enabled* in the conditions. If a delay between triggering event and reaction should be defined, just provide as additional Parameter in the *registerTrigger* call *time=+10* (for a 10 seconds delay). If a more complex logic is needed, the chain can be broken by register the trigger with a other eventname, but the action-definition can also be stored in the same location.

5.7 Included scripts - UDA

*U*ser*D*efined*A*ctions are named action-sequences (with conditions) which are stored globally in the dss in the */user/event* path in the propertytree. The system-addon user-defined-actions is in charge of administration of these actions, so modifing these events with a own app should be not be done per direct property tree manipulation, but by inter-app communication (please refer to *App to App communication*). The main advantage is, that all addons can utilize this actions as systemwide actions, for example the timed events app list all UDAs for scheduling, and if a own app specify some UDA, they will be accessible through the allready provides timed-events app.

5.8 Included scripts - Solar computer

The solar-computer scripts are some scripts how calculate each day the sunrise, sunset, dawn and dusk time, based on the stored geografic position of the dss and astronomical calculations. The calculated values are stored in */config/geodata/sunrise*, */config/geodata/sunset*, */config/geodata/civil_dusk* and */config/geodata/civil_dawn* in a stringpattern pattern of HH:mm:ss. That will done at 3 o'clock and the solar-computer raises a *solar_computer.update* - event when new calculation should start. If a other app using that values, it is adviceable to schedule a timedevent right 1 second after that update to get fresh values.

6 Authentication

Notice The digitalSTROM Server uses a self signed certificate, so in order to connect the user should accept that the certificate is not signed by a known authority. This can also be solved by simply accepting any certificate in your network client.

6.1 Configurator and Addons

The dSS11 configurator and dSS Addons are accessible with the https protocol on TCP port 443. This access method uses the HTTP Digest Authentication.

The configurator itself and addons access the dSS JSON interface over port 443 where certain requests are redirected using a proxy server. For example all URLs starting with */json/* are passed on to the dSS.

6.2 Applications

The dSS is also accessible for external applications through a HTTP based JSON interface on TCP port 8080, likewise using encrypted https. This is the preferred interface for external applications and automation systems interacting with the digitalSTROM Server.

External applications accessing the dSS should not store passwords at any time. Instead they should request an application-token which has to be activated by the user.

6.2.1 Getting a Token

First an application needs to get and store an application token from the dSS:

```
https://yourdss:8080/json/system/requestApplicationToken?applicationName=readableNameOfApplication
```

Notice When requesting an application token, the application must not be logged in with username/password or access the dSS through the default HTTPS port.

6.2.2 Approving the token

Once the token is retrieved and stored, it can be activated from the dSS11 Web Interface. Alternatively, the token can be approved from the application by asking the user for dSS username/password and use this to login:

`https://yourdss:8080/json/system/login?user=dssadmin&password=mysupersecretpassword`

This returns a temporary session token, which can be used to enable the application token using this command:

`https://yourdss:8080/json/system/enableToken?applicationToken=theApplicationToken&token=theTemporarySessionToken`

6.2.3 Logging in

After the token has been approved the application may obtain a session-token by providing the application token:

`https://yourdss:8080/json/system/loginApplication?loginToken=yourtokenhere`

6.2.4 Using the session token

Add the token to the http header, or add "token=yoursessiontokenhere" to each request:

`https://yourdss:8080/json/apartment/getStructure?token=yourtokenhere`

The session token has a timeout of 60 seconds, but will be prolonged each time it is used/touched. If the session token is invalid, a new session token should be acquired.

7 Metering

7.1 Available Data

The digitalSTROM Meters provide the digitalSTROM Server with power measurements per circuit in one second resolution. The measurements are averaged over different time periods and stored as time-value-pairs (time series) in a Round Robin Database (RRD) per digitalSTROM Meter. Internally, the digitalSTROM Server uses the RRDTool library to store this data.

Table 4 shows the available time series.

Resolution	Number of values	Storage duration
1 second	600	10 minutes
1 minute	720	12 hours
15 minutes	2976	31 days
1 day	370	~1 year
7 days	260	~5 years
30 days	60	~5 years

Table 4: Metering Time Series

There are three different types of time series:

consumption The data points represent the average power used during the previous time slot. The data is represented in floating point numbers and has the unit of Watt [W].

energy The data points represent an energy counter with always increasing values. This type functions like a traditional power/energy meter. The data is represented in floating point numbers and the unit is selectable either Watt × Seconds [Ws] or Watt × Hours [Wh].

energyDelta The data points represent the energy used during the previous time slot. This is equivalent to the "consumption" values multiplied by the time slot duration (resolution). The data is represented in floating point numbers and the unit is selectable either Watt × Seconds [Ws] or Watt × Hours [Wh].

7.2 Accessing the Data

Currently, there is no data aggregation done over multiple digitalSTROM Meters, so data can only be queried per single digitalSTROM Meter.

7.2.1 Current Metering Data

The current metering data can be accessed either via the Property Tree (as described in subsubsection 3.4.2) or the JSON interface or scripting functions.

7.2.2 Time-series Data

The time series data can only be accessed via the JSON interface or scripting functions. By default all available data for the selected type and resolution is returned. The APIs have options to limit the time window that is returned.

Time is represented with UNIX timestamps (seconds since 1970-01-01).

8 Communication using custom events

AddOns on the dSS can be understand as small programms on the dSS with their own capatilities and features. The events, which are consumed by each Addon, are raised globally, so each Addon can raise a event, which can consumed by each other Addon.

8.1 Conventions

Currently there is no namespace-handling for event-names implemented in the dSS. To avoid naming conficts, there is the convention to add the script-id of the app before the event name like *system-addon-timed-events.config* where *system-addon-timed-events* is the script-id and *config* the specific event.

8.2 App to App communication

There are two ways to get data from one app to an other:

- by writing in the propertytree-part of the foreign app
- by raising a well known event, which is consumed by the foreign app

8.2.1 Directly accessing the propertytree

Writing directly in the propertytree of a other app might be the easier way, but there are two major drawback: there is no direct way to store that changes in the saved properties of the foreign app and the foreign app is not aware of the change in his properties. so this changes are completly dynamical and non-persistent. To make the foreign app to store that changes or just be aware of the changes, the only possibility is to raise a event for the foreign app. Also there must be some special issues regarding access-rights. Currently there is a access-right to properties in the /usr/state subtree using the *setStatusValue* scripting call.

8.2.2 raising a event

The better way of communication between apps is to raise some events, which are known to be consumed by the foreign app. The advantage of that approach is, that the consuming app will get the desired parameters and it is in the responsibility of the foreign app to store the parameters and react on the changes.

8.3 UI to App communication

A UI of a app is not fixed to a specific app apart from the fact, that a UI is installed with its app. The UI is a rather independly operating part of the app

when you look at the interaction between foreground-UI and background-script. So they are asynchron to each other and the UI must utilize the same methods to communicate with his background app as two apps communicate with each other:

- by writing in the propertytree via json-api
- by raising a event, which is consumed by the background app

8.3.1 Directly accessing the propertytree

Writing in the propertytree is possible, but it has two major drawbacks:

- one value can be set using the json-calls, so parsing multiple parameter results in multiple json-calls and that may have some major performance impact.
- the background app is not aware of any change in the propertytree, so the background-app must poll changes in the propertytree or just wait for a event from the UI. Finally only the background-app can serialize the changes in the propertytree.

Direct writing to the propertytree is not advisable.

8.3.2 raising a event

The UI can also raise with the json-interface a Event with multiple parameters, which can contain all needed values in one call. That event will be processed by the dSS-EventQueue, so using this technique to change data might not result in instant changes if it is compared to directly writing to the propertytree, but if more than a few values must be changed or the background-script must react on that changes (serialize or reinitialize itself), it worth the minor lag in response. The other advantage is, that the background-script is naturally aware of the request and can do appending actions when some data has been changed by serialize them or make a reinitialisation etc. The final advantage when using this method, that a general configuration interface for the app will be specified, which can be used by other apps.

8.4 App to UI communication

There are two methods to get some data from the background-app to the UI and they are both equal preferable depending on the situation.

- The UI uses the property-querying JSON-calls do get data from the dSS. As with all property-tree querying the background app will not be aware, if the UI is querying data from the tree, so that can be used

to load some static data oder settings. On the upside, the background-App can store the data asynchronly and the UI can query them at any time. There is also no need for extra coding in the background app. This method is good for getting structure information of the dss or the app and data, which requires no interaction.

- The UI can subscripte to a event, which can be raised by a background app. The UI will start a http-request, which will return data via event-parameters when a event is raised or on time out. This method is usefull when the UI had issued some kind of command to the background app (like save a value) and now it should get a confirmation or answer. When using this technique take in account, that the ressources on the dSS is limited, so it is not advisable to open to much http-requests simultaneously and only when needed.

9 Certification Rules

Rule 1 digitalSTROM Server Addons that implement private events have to prefix all event names with their own unique addon name.