Redwood API v3.0 – Functional Specification

Overview

This document describes the functional details of the features within the API sphere that are delivery targeted for the v3.0 Software Release by the PD team. The specific features described in this document are:

- Redwood API v3.0 Features:
    - REST-based Write via HTTP PUT
    - Unified API Subscriptions
    - Basic Lighting Control

The reader is assumed to be familiar with the following:

- Redwood Systems Platform / Product landscape.
- Redwood API Product User Guide 091112 (v2.1) - The features described below are extensions on the foundational feature set described in this document.

This document is **not** a complete description of the Redwood API including prior functionality combined with the extensions supported in the v3.0 release. This document **only** describes the incremental changes to support the v3.0 feature set. Readers are expected to refer to the original Redwood API Product User Guide 091112 (v2.1)

Redwood API v3.0 Features

REST-based Write via HTTP PUT

In this release, the **RESTful** API form is extended to support the ability to write / update (set) data.

Service Entry Point

The **RESTful** URL query form to specify the resource to write / update is similar to the form used to specify a read.

http[s]://<DirectorName>/rApi/<ResourceURI>

The only difference is that JSONp callback functionality is not supported for writes and thus there is no additional query string allowed on the URL.

While the HTTP GET method is used to retrieve (read) data, the HTTP PUT method must be used to set (write) data. The HTTP PUT protocol is composed of a URL and payload. The URL specifies the resource to write (as described above) and the payload specifies the data that the resource is to be updated with.

The payload must be a JSON-framed object that represent the Data Model attribute referenced by the <ResourceURI>. This is identical to the data that would be returned if a **RESTful** read request is made. IE, the payload in an HTTP PUT write request will be similar in JSON structure to the data returned when an HTTP GET read request is made to the same <ResourceURI>. See the Redwood API Product User Guide 091112 (v2.1) for further details on the JSON framing structure of the Data Model.

A successful write will result in an HTTP 200 response code.

A write can fail for various reasons. The failure reasons and the resulting HTTP error codes are as follows:

- The <ResourceURI> is invalid. This is either because it names an invalid schema attribute or a non-existent object. This will result in an HTTP 404 (Not Found) error.
- The payload is not properly formatted. This is either because it does not represent proper JSON or it represent a JSON type that does not match the type of the resource being referenced. IE a syntax error. This will result in an HTTP 400 (Bad Request) error.
- The payload data is not semantically valid. Individual elements of the Data Model may have further constraints on what valid values are supported. See the Data Model sections of the Redwood API Product User Guide 091112 (v2.1).
    - If the failure is due to a static constraint, an HTTP 400 (Bad Request) error is returned. A static constraint failure is one where the payload data is something that will never be valid in any situation the cluster may be in. For example, specifying a number value that is out-of-range for a given attribute.
    - If the failure is due to a dynamic constraint, an HTTP 409 (Conflict) error is returned. A dynamic constraint failure is one where the payload data is not valid for the current state of the cluster, but could be if the cluster state changes. For example, trying to set a /location/<id>/sceneControl/activeSceneName to a non-existent scene name is considered a dynamic constraint failure. If a scene is created and given the non-existent name, then setting the activeSceneName to that name will work successfully.

This is an extension to the API **mechanisms** available to support write / update. It does not impact which elements of the <Data Model> can be set. See the Data Model sections of the Redwood API Product User Guide 091112 (v2.1) for further details on elements which support being set.

Examples
Below are examples showing how **RESTful** write requests are structured and the resulting responses. For the purposes of these examples, the following snapshot of a <DataModel> instance is used. It is not a full representation of all data that is included per the <DataModel> specification. It describes only the portions of the <DataModel> necessary to highlight the write functionality.

```
{
        “location” : [
                {
                        “id” : 100,
                        “sceneControl” : {
                                “activeSceneName” : “”,
                                “scene” : [
                                        {
                                                “name” : “Bright”,
                                                “order” : 1
                                        },
                                        {
                                                “name” : “Dim”,
                                                “order” : 2
                                        }
                                ]
                        }
                }
        ]
}
```

Example #1: Set activeSceneName to Valid Scene Name
This example shows how a user would set the activeSceneName to a valid scene via the **RESTful** API form.

The HTTP PUT request is composed of the following components:

URL:
http[s]://<DirectorName>/rApi/location/100/sceneControl/activeSceneName

Payload:
“Bright”

If this request is applied to the <Data Model> instance from above, the response will be:

200 OK

Upon successful completion of this request, the <Data Model> instance will be updated to the following:

```
{
	“location” : [
		{
			“id” : 100,
			“sceneControl” : {
				“activeSceneName” : “Bright”,
				“scene” : [
					{
						“name” : “Bright”,
						“order” : 1
					},
					{
						“name” : “Dim”,
						“order” : 2
					}
				]
			}
		}
	]
}
```

In this example, the “activeSceneName” has been changed from “” to “Bright”, resulting in the “Bright” scene now being active.

The following Linux commands can be used to specify the HTTP PUT transaction described in this example:

```
echo ‘ “Bright” ’ | curl -T - -k -u <username:password>
	https://<DirectorName>/rApi/location/100/sceneControl/activeSceneName
```

Note that the echo string is composed of a singe quotes wrapping a double quoted value.

Example #2: Set activeSceneName to Empty (Inactive) Scene Name
This example shows how a user would set the activeSceneName to the empty string (resulting in scene control being inactive) via the **RESTful** API form.

The HTTP PUT request is composed of the following components:

URL:

http[s]://<DirectorName>/rApi/location/100/sceneControl/activeSceneName

Payload:

“”

If this request is applied to the <Data Model> instance from Example #1 above (after completion), the response will be:

200 OK

Upon successful completion of this request, the <Data Model> instance will be updated to the following:

```
{
	“location” : [
		{
			“id” : 100,
			“sceneControl” : {
				“activeSceneName” : “”,
				“scene” : [
					{
						“name” : “Bright”,
						“order” : 1
					},
					{
						“name” : “Dim”,
						“order” : 2
					}
				]
			}
		}
	]
}
```

In this example, the “activeSceneName” has been changed from “Bright” to “”, resulting in the scene control becoming inactive.

The following Linux commands can be used to specify the HTTP PUT transaction described in this example:

echo ‘ “” ’ | curl -T - -k -u <username:password>
https://<DirectorName>/rApi/location/100/sceneControl/activeSceneName

Example #3: Set activeSceneName to Invalid Scene Name
This example shows how a user would set the activeSceneName to an invalid string via the **RESTful** API form.

The HTTP PUT request is composed of the following components:

URL:
http[s]://<DirectorName>/rApi/location/100/sceneControl/activeSceneName

Payload:
“Invalid Scene Name”

If this request is applied to the original <Data Model> instance from above, the response will be:

409 Conflict

This will result in the <Data Model> instance not being updated.

The following Linux commands can be used to specify the HTTP PUT transaction described in this example:

echo ‘ “Invalid Scene Name” ’ | curl -T - -k -u <username:password>
https://<DirectorName>/rApi/location/100/sceneControl/activeSceneName

Unified API Subscriptions

In this release the **Unified** API form is extended to support subscriptions. Subscriptions through the **RESTful** API are not supported.

Subscription requests can be made on any element of the Data Model that supports **get** requests.

Service Entry Point
The **Unified** API, single entry, access URL remains the same. IE:

http[s]://<DirectorName>/uApi/

If the JSON-framed Request object specifies a subscription request, the response will be an ongoing sequence of JSON-framed Response objects. See Request / Response Structure section below for further details.

The sequence of Response objects will continue to be generated while the client keeps the bi-directional HTTP POST connection open.

API Model / Framing

A subscription request results in an ongoing sequence of responses. The initial response is similar to the response to a get request. It returns the current state of the data model components the request is being made on. All further responses specify incremental / ongoing changes to the data model components and are cumulative (IE: each further response reflects a change from the overall data model state resulting from the application of all earlier responses). See the Data Model Updates Structure section below for details on how changes are specified in a response.

When a subscription request is submitted, the first response [containing initial state] is delivered immediately. Subsequent responses [containing updates] are only delivered when a change to the data model components occur. If the components under observation have not changed, the HTTP POST connection will remain open with no data flowing.

When no data is flowing, a periodic newline (\n) heartbeat is sent. This is designed to verify that the client is still connected. This heartbeat is designed to be non-intrusive to client side stream parsing.

Each response is a JSON-framed Response object. After each Response, a sentinel marker is inserted in the data stream. The <Sentinel> is as follows:

\r\n\r\n\r\n

This is a triple sequence of carriage-return ('\r' or ascii 0x0D) and line-feed ('\n' or ascii 0x0A). This sentinel is designed to be a 'silent' marker. It is a sequence of whitespace characters. If a context-sensitive, streaming, JSON parser is being used to process the response stream, these characters can be safely ignored (most publicly available JSON parsers do so). If a buffered parser is preferred, the detection of the sentinel can be used to signal a complete response is available in the buffer for further processing.

Request / Response Structure

The support for subscriptions is provided via extensions to the Request / Response structures.

Request Structure

The changes to the Request Structure to support subscriptions is as follows:

{
“requestType” : <String>
}

- requestType: This is a required parameter that specifies the type of request. In this release, the following, additional, value is supported:
    - “subscribe”: Specifies a request to subscribe to the portions of the requestData specified.

Response Structure
The changes to the Response Structure to support subscriptions is as follows:

{
“responseType” : <String>
“time” : <Number>
}

- responseType: This is always included and specifies the type of response. In this release, the following, additional, value is supported:
    - “event”: Specifies that this is a valid response to a subscribe request. A stream of Response Structures, each with this responseType, will be generated for a valid subscribe request.
- time: This is the current time on the cluster when the response was generated. It is only included when the responseType is “event”. It is an unsigned real number. The dimension is Seconds since Epoch with three digits of precision (millisecond). Examples are 1, 2.34, and 56.789.

Data Model Structure / Change Flag Attributes
The main Data Model schema is unaffected by subscription support. However, additional, internal attributes are introduced as a means to communicate ongoing changes that have occurred to a cluster’s Data Model instance.

These internal attributes are called Change Flag Attributes. The basic structure of any Change Flag Attribute is as follows:

“_c_<context>” : <String>

- The attribute name will always start with _c_ which specifies this is a Change Flag Attribute. An additional suffix, <context>, may also be included and is used to specify which component in the main Data Model instance [in the surrounding context] is being updated. The details and semantics of each attribute name is

context sensitive and is described further below. The value of this attribute is one of the following:

- "ADD": The element being referenced by the name of the Change Flag Attribute is being added.
- "DEL": The element being referenced by the name of the Change Flag Attribute is being deleted.
- "MOD": The element being referenced by the name of the Change Flag Attribute is being modified.

The Data Model schema supports the following types:

- Primitive: These are any of the following types: <String>, <Number>, <Boolean>. Attributes of this type are encapsulated within an Object type. /name and /currentTime are examples of primitive attributes.
- Object: This is a composite type that aggregates a static collection of attributes of any type. The JSON framing for objects is '{}'. /location/100/sensorStats and /location/100/sceneControl are examples of object attributes.
- Array: This is a composite type that aggregates a dynamic collection. The JSON framing for arrays is '[]'. Arrays can contain Objects or Primitives. /location, /fixture, /location/100/childLocation, and /location/100/childFixture are examples of array attributes.

For each of the above mentioned types, a cluster's Data Model instance may be changed because an attribute of that type may be either added, deleted, or updated. Further, in the case of array attributes, individual elements within the array may be added or deleted.

Change Flag Attributes for Named Attributes

For Primitive type named attributes in the Data Model schema, a change to that attribute is signaled via a Change Flag Attribute with a name of _c_<AttributeName>. This Change Flag Attribute will be in the same enclosing context as the original attribute.

The following examples illustrate this:

- /name: If this primitive type attribute is changed, there will be a corresponding attribute with the name _c_name. Although it cannot be accessed directly, it's logical URI is /_c_name

Change Flag Attributes are **not** generated for named attributes of Object or Array type. The change to such attributes are implicit based on the changes signaled to components attributes as follows:

- Object and Array attributes are implicitly deleted when all sub-component attributes have been deleted.

- Object and Array attributes are implicitly added when some sub-component attribute is added.
- Object and Array attributes are not directly modified. Only Primitive sub-components attributes can be modified.

Change Flag Attributes for Array Elements
An Array is an ordered sequence of either Objects or Primitives. The elements of an Array can be either added or deleted. They cannot be modified (if the element is an Object, its sub-components can be modified).

To signal an add / delete of an Object element in an Array, a Change Flag Attribute for the **key-attribute** of the Object will signal the change. Objects in an Array always have a **key-attribute**.

To signal an add / delete of a Primitive element in an Array, the element being added / deleted is listed in the sequence and immediately followed by an extra object-encapsulated Change Flag Attribute. This is called a <ChangeFlagObject>:

<ChangeFlagObject>:
{
"_c_" : <String>
}

- _c_: Same as Change Flag Attribute description above. <context> is unnecessary. The semantics are that this Change Flag Attribute is in reference to the immediately preceding Object Primitive in the Array sequence.

Service Level Agreement
A maximum of 50 open **Unified** API requests is supported. Any requests beyond the maximum return a response with the following:

- responseType: This will be set to "errorResponse".
- responseErrorType: This will be set to "unsupportedService".
- responseErrorDetail: This will include a description that briefly states that the service is unavailable due to too many open connections. Please contact your customer service representative for further assistance.

Examples
Below are a set of examples showing how subscriptions are structured and the resulting responses. The subscription interface is comprised of a single request that results in an ongoing stream of responses as changes occur. The examples below are structured to show this sequence of messages as a subscription is initiated and changes are ongoing. For the purposes of the examples, the following snapshot of a <DataModel> instance is used as the

starting state of a cluster. It is not a full representation of all the dta that is included per the <DataModel> specification. It describes only the portions of the <DataModel> necessary to hightlight the subscription functionality.

```
{
        “location” : [
                {
                        “id” : 0,
                        “name” : “Floor”,
                        “childLocation” : [
                                “/location/1”
                        ],
                        “sensorStats” : {
                                “motion” : {
                                        “instant” : 5
                                }
                        }
                },
                {
                        “id” : 1,
                        “name” : “Cube”
                }
        ]
}
```

Example #1: Subscribe to All Locations

This example shows how a user would subscribe to updates to any location. Using the Unified API form, the request would be composed as follows:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “requestType” : “subscribe”,
        “requestData” : {
                “location” : []
        }
}
```

The **first** response will be a follows:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “responseType” : “event”,
```

```
        “time” : 123
        “responseData” : <EntireDataModelExampleFromAbove>
}
<Sentinel>
```

Note the following details:

- After the response is sent, a sentinel marker is also sent.
- The connection will remain open. Subscription connections remain open until they are closed by the client.

Example #2: Primitive Update

This example shows what the next response will be if a Primitive attribute in the <DataModel> has changed. If /location/1/name had been updated in the cluster from “Cube” to “Office”, the following response will arrive next on the open connection:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “responseType” : “event”,
        “time” : 125
        “responseData” : {
                “location” : [
                        {
                                “id” : 1,
                                “name” : “Office”,
                                “_c_name” : “MOD”
                        }
                ]
        }
}
<Sentinel>
```

Note the following details:

- _c_name signals that the “name” attribute is being updated. In this example, being modified from a previous value to a new value
- On a MOD or ADD update, the content of the “name” attribute included in the event is the new value. On a DEL, it is the old value.

Example #3: Object Update

This example shows what the next response will be if an Object attribute in the <DataModel> has changed. If /location/0/sensorStats has been updated to add the “power” attribute of <PowerStats> type, the following response will arrive next on the open connection:

```
{
```

```
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “responseType” : “event”,
        “time” : 127
        “responseData” : {
                “location” : [
                        {
                                “id” : 0,
                                “sensorStats” : {
                                        “power” : {
                                                “instant” : 1.0,
                                                “_c_instant” : “ADD”
                                        }
                                }
                        }
                ]
        }
}
<Sentinel>
```

Note the following details:

- The addition of /location/0/sensorStats/power is implicit as a result of the “instant” attribute being added.
- Since “instant” is the only attribute within the <PowerStats> object, if the _c_instant attribute had been “DEL” instead, it would implicitly mean that /location/0/sensorStats/power has also been deleted.

Example #4: Array of Objects Update

This example shows what the next response will be if an Array [of Objects] attribute in the <DataModel> has changed. If /location has been updated to add another location with an id of 2 and name of “Room”, the following response will arrive next on the open connection:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “responseType” : “event”,
        “time” : 129
        “responseData” : {
                “location” : [
                        {
                                “id” : 2,
                                “_c_id” : “ADD”,
                                “name” : “Room”,
```

```
                    "_c_name" : "ADD"
                }
            ]
        }
}
<Sentinel>
```

Note the following details:

- An Object being added added or deleted in an array is signaled by the **key-attribute** update. In the example above, the "_c_id" Change Flag Attribute signals this.
- Any additional attributes in the Object that are being added or deleted also have corresponding Change Flag Attributes

Example #5: Array of Primitives Update

This example shows what the next response will be if an Array [of Primitives] attribute in the <DataModel> has changed. If /location/0/childLocation has been updated to add /location/2, the following response will arrive next on the open connection:

```
{
    "protocolVersion" : "1",
    "schemaVersion" : "1.3.0",
    "responseType" : "event",
    "time" : 131
    "responseData" : {
        "location" : [
            {
                "id" : 0,
                "childLocation" : [
                    "/location/2",
                    {
                        "_c_" : "ADD"
                    }
                ]
            }
        ]
    }
}
<Sentinel>
```

Note the following details:

- A <ChangeFlagObject> immediately following the value being added or deleted signals the update. In this case, a <ChangeFlagObject> signaling an add of the "/location/2" value.

Example #6: Subscribe to Subset of Data in All Locations

This example shows how a user would subscribe to a subset of updates to any location. Consider that a user only want to be notified of updates to the “name” attribute of any locations. Using the Unified API form, the request would be composed as follows:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “requestType” : “subscribe”,
        “requestData” : {
                “location” : [
                        {
                                “name” : null
                        }
                ]
        }
}
```

The responses on this will be limited to only changes to the “name” attribute. This will include modifications to the “name” attribute of existing locations, the addition of the “name” attribute to an existing or new location, and the deletion of the “name” attribute to an existing or deleted location.

As shown in Example #2, the responses will always include the **key-attribute** for the location to identify which location the “name” attribute is being updated in.

Example #7: Subscribe to Data in Specific Location

This example shows how a user would subscribe to updates to a specific location. Consider that a user only want to be notified of updates to /location/0. Using the Unified API form, the request would be composed as follows:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “requestType” : “subscribe”,
        “requestData” : {
                “location” : [
                        {
                                “id” : 0
                        }
                ]
        }
}
```

The responses on this will be limited to only changes to /location/0. It will include updates to any sub-component within this location as well as the creation or deletion of this location.

Basic Lighting Control

In this release, the Data Model Structure is extended to enable API clients to interact with the lighting control features of a Redwood Systems Cluster.

The Location and Fixture sensor statistics are extended to include real-time data regarding light output levels being driven by the Cluster.

Lighting for a Location may be directly controlled by an API client through a set of Wall Switch extensions. The Wall Switch extensions are divided into a high level and low level set of controls.

The high level Wall Switch controls provide convenient and centralized functionality similar to physical wall switches available in the Redwood Systems Platform. This set of interfaces effectively allow API clients to implement virtualized Wall Switches that are functionally consistent with physical wall switches.

The low level Wall Switch controls provide a more basic set of primitives to control a Location's lighting however a client prefers. If the semantics provided by the high level controls are not suitable for an API client, the low level controls can be used to implement whatever lighting semantics a client wants. It is expected that this set of interfaces would commonly be used in conjunction with the lighting-related sensor statistics described above for fine-tuned, client-controlled, real-time lighting management.

The further details and semantics of each of the above extension is described in the Data Model Structure section below.

Service Entry Point
There are no changes to the service entry points for the Basic Lighting Control feature. All existing mechanisms (**Unified** and **RESTful** API forms) for accessing the Data Model Structure can be used to access the Basic Light Control extensions.

Get / Set Support
In the data model extensions described below, unless otherwise specified, only **get** requests are supported on all extension elements.

Access Control

In this release, no access controls are available for an administrator to limit the set of Locations which an API client can control via the Basic Lighting Control extensions.

Data Model Structure
The changes to the Data Model to support Basic Lighting Control is as follows:

The <LocationSensorStats> type is modified as follows:

<LocationSensorStats>:
{
“brightness” : <BrightnessStats>
}

- brightness: An object that aggregates all brightness related statistics. Brightness refers to the average level of light being output across all luminaire fixtures within the location.

The <FixtureSensorStats> type is modified as follows:

<FixtureSensorStats>:
{
“brightness” : <BrightnessStats>
}

- brightness: An object that aggregates all brightness related statistics. Brigthness refers to the level of light being output by the fixture. This attribute is only reported for luminaire-based fixtures.

The following additional type is introduced:

<BrightnessStats>
{
“instant” : <Number>
}

- instant: The value represents the real-time brightness being output. It is an unsigned real value with up to two digits of precision. The dimensions is Percentage with a range from 0.00 to 100.00. Examples are 5, 33.3, and 75.59.

The <Location> type is extended as follows:

<Location>:
{

"wallSwitch" : <WallSwitch>
}

- wallSwitch: An object that aggregates all location wall switch controls.

The following additional types are introduced:

<WallSwitch>:
{
"lowLevelControl" : <WallSwitchLowLevelControl>,
"highLevelControl" : <WallSwitchHighLevelControl>
}

- lowLevelControl: An object that aggregates the low level wall switch controls
- highLevelControl: An object that aggregates the high level wall switch controls

<WallSwitchLowLevelControl>:
{
"brightness" : <Number>,
"activated" : <Number>
}

- brightness: This specifies the brightness that all fixtures within the location are set to when the wall switch is currently active (See "activated" description below). It is an unsigned real value with up to two digits of precision. The dimensions is Percentage with a range from 0.00 to 100.00. 0.00 means that no light should be output while 100.00 means that the maximum supported amount of light should be output. Examples are 5, 33.3, and 75.59. **This value can be set.** When a wall switch is first activated and/or the brightness configuration has been changed, the current brightness output is typically something else and the brightness output must be adjusted by the Redwood Systems Cluster. The manner in which the adjustment from the current brightness to the new brightness value is made is called the ramp. The ramp semantics implemented on a brightness change is based on a 3 seconds rate from a brightness value of 0 to 100 (or 33% per second). The amount of time it will take for the new brightness valuee to be fully realized will be some fraction of the 3 seconds depending on the difference between the current and new brightness value. For example, if the current brightness value is 25 and the new brightness value is 75, it will take 1.5 seconds for the adjustment to occur.
- activated: This is a time when the wall switch was most recently activated. It is an unsigned real number. The dimensions is Seconds since Epoch with three digits of precision (milliseconds). **This value can be set.** The semantics of a wall switch activation are as follows:

- If the value is set to a time that is greater than the current system time, the value is normalized to the current system time. Setting this value to a sufficiently large number can be used as a convenient way to set this to the current system time without having to read the current system time prior to each update.
- A wall switch is considered “active” if the difference between this value and the current system time is less than the occupancy timeout value of the currently active policy.
- When a wall switch is “active”, the brightness described above is set for all fixtures within the location.
- When a wall switch is not active, the active policy for the location determines the brightness for the fixtures.
- Locations can be constructed in a hierarchical manner. IE a Location may have descendant Locations. When the activated value of a Location is changed, the activated value of all descendant Locations is updated likewise. Further, the brightness of all descendant Locations is also updated to match the configured brightness of this Location. In effect, activating a wall switch in a parent location causes all descendant locations to also be fully updated to the settings of the parent’s <WallSwitchLowLevelControl>.

<WallSwitchHighLevelControl>:

{

“upPress” : <Number>,
“upRelease” : <Number>,
“downPress” : <Number>,
“downRelease” : <Number>,
“togglePress” : <Number>

}

- upPress: This is a time when the wall switch “up” button was most recently pressed. It is an unsigned real number. The dimensions is Seconds since Epoch with three digits of precision (milliseconds). **This value can be set.** The semantics of an upPress update are as follows:
  - If the value is set to a time that is greater than the current system time, the value is normalized to the current system time. Setting this value to a sufficiently large number can be used as a convenient way to set this to the current system time without having to read the current system time prior to each update. If this value is set to a time that is less than the current system time, the value is normalized to 0. Effectively, this attribute can only be updated to the current time to make it active or reset to 0 to make it inactive. When updated to be active, the further semantics below apply.
  - The brightness attribute of the <WallSwitchLowLevelControl> is updated to 100%

    - The activated attribute of the <WallSwitchLowLevelControl> is updated to the current system time.
    - See the <WallSwitchLowLevelControl> description above for details on the Cluster lighting impact from these changes.
- upRelease: This is a time when the wall switch “up” button was most recently released. It is an unsigned real number. The dimensions is Seconds since Epoch with three digits of precision (milliseconds). **This value can be set.** The semantics of an upRelease update are as follows:
    - If the value is set to a time that is greater than the current system time, the value is normalized to the current system time. Setting this value to a sufficiently large number can be used as a convenient way to set this to the current system time without having to read the current system time prior to each update. If this value is set to a time that is less than the current system time, the value is normalized to 0. Effectively, this attribute can only be updated to the current time to make it active or reset to 0 to make it inactive. When updated to be active, the further semantics below apply.
    - The activated attribute of the <WallSwitchLowLevelControl> is updated to the current system time.
    - The brightness attribute of the <WallSwitchLowLevelControl> is updated based on the delta between upRelease and upPress. If the delta is less than 250ms, there is no change to brightness. If the delta is greater than 250ms, then the brightness is [logically] updated to the current value of the Location’s sensorStats/brightness/instant value. Effectively, the semantics are that when an upPress is done, the actual brightness of the fixtures in the Location begins to ramp to 100%. If a quick upRelease is seen, the ramp continues until the brightness reaches 100%. However, if a slower upRelease is seen, it stops the ramp at whatever the current brightness is. This simulates holding a physical “up” button down for an extended period of time causing lights to brighten until the button is released at which point the lights stay at whatever level they have reached up to that point. While a quick press and release causes the lights to brighten to the full level capable.
- downPress: This is a time when the wall switch “down” button was most recently pressed. It is an unsigned real number. The dimensions is Seconds since Epoch with three digits of precision (milliseconds). **This value can be set.** The semantics of a downPress update are as follows:
    - If the value is set to a time that is greater than the current system time, the value is normalized to the current system time. Setting this value to a sufficiently large number can be used as a convenient way to set this to the current system time without having to read the current system time prior to each update. If this value is set to a time that is less than the current system time, the value is normalized to 0. Effectively, this attribute can only be updated to the current time to make it

active or reset to 0 to make it inactive. When updated to be active, the further semantics below apply.
    - The brightness attribute of the <WallSwitchLowLevelControl> is updated to 0%
    - The activated attribute of the <WallSwitchLowLevelControl> is updated to the current system time.
    - See the <WallSwitchLowLevelControl> description above for details on the Cluster lighting impact from these changes.
- downRelease: This is a time when the wall switch “down” button was most recently released. It is an unsigned real number. The dimensions is Seconds since Epoch with three digits of precision (milliseconds). **This value can be set.** The semantics of a downRelease update are almost identical to those described above for upRelease. The only differences are as follows:
    - The brightness attribute of the <WallSwitchLowLevelControl> is updated based on the delta between downRelease and downPress. IE, while upRelease stops the ramp as the actual brightness of fixtures goes to 100%, downRelease stops the ramp as the actual brightness goes to 0% (completely off).
- togglePress: This is a time when the wall switch “toggle” button was most recently pressed. It is an unsigned real number. The dimensions is Seconds since Epoch with three digits of precision (milliseconds). **This value can be set.** The semantics of a togglePress update are as follows:
    - If the value is set to a time that is greater than the current system time, the value is normalized to the current system time. Setting this value to a sufficiently large number can be used as a convenient way to set this to the current system time without having to read the current system time prior to each update. If this value is set to a time that is less than the current system time, the value is normalized to 0. Effectively, this attribute can only be updated to the current time to make it active or reset to 0 to make it inactive. When updated to be active, the further semantics below apply.
    - The activated attribute of the <WallSwitchLowLevelControl> is updated to the current system time.
    - The brightness attribute of the <WallSwitchLowLevelControl> is updated based on the Location’s current, actual, brightness value (ie sensorStats/brightness/instant). If the actual brightness value (xxxlxxx) is currently 0%, wallSwitch/lowLevelControl/brightness is updated to 100%. Otherwise, it is updated to 0%.
    - See the <WallSwitchLowLevelControl> description above for details on the Cluster lighting impact from these changes.

Examples

Below are a few example <DataModel> instances showing different aspects of the Direct Lighting Control feature and brief descriptions of what the state represents. While a typical <DataModel> instance contains further data, all unrelated data has been removed from the

examples below. Examples outlining the mechanics of writing <WallSwitch> data and system impacts of such are also included.

Example #1: Real-Time Brightness Stats
This example shows a <DataModel> with a location and fixture in which the luminaire is currently on and outputting light at an 85% brightness level.

```
{
	“location” : [
		{
			“id” : 100,
			“childFixture” : [
				"\/fixture\/sn1"
			],
			“sensorStats” : {
				“brightness” : {
					“instant” : 85.0
				}
			}
		}
	],
	“fixture” : [
		{
			“serialNum” : “sn1”,
			“sensorStats” : {
				“brightness” : {
					“instant” : 85.0
				}
			}
		}
	]
}
```

Example #2: Active Wall Switch
This example shows a <DataModel> with a location in which the wall switch is currently active.

```
{
	“currentTime” : 123
	“location” : [
		{
			“id” : 100,
			“wallSwitch” : {
```

```
                        “lowLevelControl” : {
                                “brightness” : 100.0,
                                “activated” : 121.1
                        },
                        “highLevelControl” : {
                                “upPress” : 121.0,
                                “upRelease” : 121.1,
                                “downPress” : 0,
                                “downRelease” : 0,
                                “togglePress” : 0
                        }
                },
                “sensorStats” : {
                        “brightness” : {
                                “instant” : 100.0
                        }
                }
        }
    ]
}
```

This example shows a wall switch in which the upPress and upRelease were pressed within 0.1s of each other, resulting in the brightness going all the way to 100. The button presses happened ~2 seconds ago. As long as the occupancy timeout associated with this location is more than two seconds, then the wall switch is currently activate. A look at the instant brightness in the sensor stats for the location verifies this.

Example #3: Inactive Wall Switch

This example shows a <DataModel> with a location in which the wall switch is currently inactive.

```
{
        “currentTime” : 12345
        “location” : [
                {
                        “id” : 100,
                        “wallSwitch” : {
                                “lowLevelControl” : {
                                        “brightness” : 100.0,
                                        “activated” : 121.1
                                },
                                “highLevelControl” : {
                                        “upPress” : 121.0,
```

```
                                        “upRelease” : 121.1,
                                        “downPress” : 0,
                                        “downRelease” : 0,
                                        “togglePress” : 0
                                }
                        },
                        “sensorStats” : {
                                “brightness” : {
                                        “instant” : 0.0
                                }
                        }
                }
        ]
}
```

This example is almost identical to Example #2, except that the time elapsed since the button presses happened and the current time is far more than the occupancy timeout of the location. In this case, the instant brightness sensor stat shows that the lights are not currently outputting anything.

Example #4: Set Wall Switch Low Level Control

This example shows how a user can set the “brightness” and “activated” attributes of the <WallSwitchLowLevelControl>. This example shows a request that would transition the state of the system from Example #3 to one where the wall switch has been activated and the brightness has been set to 50%.

Using the **Unified** API form, the request is:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “requestType” : “set”,
        “requestData” : {
                “location” : [
                        {
                                “id” : 100,
                                “wallSwitch” : {
                                        “lowLevelControl” : {
                                                “brightness” : 50.0,
                                                “activated” : 1234567
                                        }
                                }
                        }
```

```
                ]
        }
}
```

The response is:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “responseType” : “setResponse”
}
```

Following this request, the <DataModel> for the system would be as follows:

```
{
        “currentTime” : 12350
        “location” : [
                {
                        “id” : 100,
                        “wallSwitch” : {
                                “lowLevelControl” : {
                                        “brightness” : 50.0,
                                        “activated” : 12349.0
                                },
                                “highLevelControl” : {
                                        “upPress” : 121.0,
                                        “upRelease” : 121.1,
                                        “downPress” : 0,
                                        “downRelease” : 0,
                                        “togglePress” : 0
                                }
                        },
                        “sensorStats” : {
                                “brightness” : {
                                        “instant” : 50.0
                                }
                        }
                }
        ]
}
```

Note the following details:

- The “activated” has been renormalized to the current system time. The example shows it being 1 second behind the currentTime to reflect the difference between when the set request was made and when a subsequent get request might have been issued.
- The “highLevelControl” attributes are unchanged. Changes to “lowLevelControl” do not impact the “highLevelControl”.
- The sensorStats confirm that the wall switch is now active and the actual brightness being generated is aligned to the wall switch configured brightness.

A single, equivalent set request cannot be made via the RESTful API form. However, two separate such requests can be made and will result in an equivalent system state. The following two Linux commands can be used to initiate the RESTful API form sets:

```
echo ‘50.0’ | curl -T - -k -u <username:password>
        https://<DirectorName>/rApi/location/100/wallSwitch/lowLevelControl/brightness

echo ‘1234567’ | curl -T - -k -u <username:password>
        https://<DirectorName>/rApi/location/100/wallSwitch/lowLevelControl/activated
```

Example #5: Set Wall Switch High Level Control

This example shows how a user can set the “togglePress” attributes of the <WallSwitchHighLevelControl>. This example shows a request that would transition the state of the system from Example #4 to one where the wall switch toggle button has been pressed, resulting in the lights being turned off.

Using the **Unified** API form, the request is:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “requestType” : “set”,
        “requestData” : {
                “location” : [
                        {
                                “id” : 100,
                                “wallSwitch” : {
                                        “highLevelControl” : {
                                                “togglePress” : 1234567
                                        }
                                }
                        }
                ]
        }
```

```
}
```

The response is:

```
{
        “protocolVersion” : “1”,
        “schemaVersion” : “1.3.0”,
        “responseType” : “setResponse”
}
```

Following this request, the <DataModel> for the system would be as follows:

```
{
        “currentTime” : 12360
        “location” : [
                {
                        “id” : 100,
                        “wallSwitch” : {
                                “lowLevelControl” : {
                                        “brightness” : 0.0,
                                        “activated” : 12359.0
                                },
                                “highLevelControl” : {
                                        “upPress” : 121.0,
                                        “upRelease” : 121.1,
                                        “downPress” : 0,
                                        “downRelease” : 0,
                                        “togglePress” : 12359.0
                                }
                        },
                        “sensorStats” : {
                                “brightness” : {
                                        “instant” : 0.0
                                }
                        }
                }
        ]
}
```

Note the following details:

- The “togglePress” has been renormalized to the current system time. The example shows it being 1 second behind the currentTime to reflect the difference between when the set request was made and when a subsequent get request might have been issued.

- The “lowLevelControl” attributes have been changed. Changes to “highLevelControl” cause automatic changes to the “lowLevelControl”.
- The sensorStats confirm that the wall switch is now active and the actual brightness being generated is aligned to the wall switch configured brightness - ie off.

An equivalent set request can be made via the RESTful API form. The following Linux command can be used to initiate the RESTful API form set:

```
echo ‘1234567’ | curl -T - -k -u <username:password>
    https://<DirectorName>/rApi/location/100/wallSwitch/highLevelControl/togglePress
```