Technical White Paper

Barco DRAMP API

Version: 2.0 | 2015/02/12
Authors: HUHE
Status: Draft

SUMMARY

This white paper describes the Video Wall Application Programming Interface (**DRAMP API**) for monitoring and controlling Barco video walls over TCP/IP based networks. DRAMP is specified according to **Representational State Transfer** (REST) principles using the standard application layer **HTTP** protocol for communication between clients and servers (devices). This protocol is a successor version to OLAPI with some changes in basic HTTP protocol handling and improvements and consolidation in the resource tree.

Clients send requests to walls or devices using a **Uniform Interface** instead of a large and arbitrary vocabulary of nouns and verbs to talk to the devices. Requests use the four standard HTTP methods GET, PUT, POST and DELETE only. Such requests refer to hierarchically structured **resources** provided by the wall and devices to retrieve and manipulate wall and device state. Individual resources are identified using Uniform Resource Identifiers (**URIs**), which are commonly used today for retrieving web pages on the World Wide Web. On request, clients and devices exchange representations of resources using the **JSON** (Java Script Object Notation) format to represent data.

Besides operations to directly retrieve and manipulate resources using GET and PUT requests, DRAMP provides “asynchronous” requests to interact with device hardware (**actions**). Such requests may complete immediately and send back a REQUEST_DONE response if processing time is very short, or the response tells the caller the command is IN_PROGRESS including a “job ID”, which can be used to access a corresponding resource to ask for completion.

.

TABLE OF CONTENTS

Document History

Date	Author	Reason for change
2015/02/12	HUHE	Initial version

References

[REST]	RESTful Web Services, Leonard Richardson/Sam Ruby, O'Reilly ISBN 13: 978-81-8404-332-7
[REST-2]	Representational State Transfer (REST) http://en.wikipedia.org/wiki/Representational_State_Transfer
[HTTP/1.1]	Hypertext Transfer Protocol - HTTP/1.1 http://www.w3.org/Protocols/rfc2616/rfc2616.html
[HTTP CODE]	HTTP/1.1 Status Code Definitions
[URI]	Uniform Resource Identifiers (URI): Generic Syntax http://www.ietf.org/rfc/rfc3986.txt
[JSON]	Introducing JSON (JavaScript Object Notation) http://www.json.org/
[JSON-MT]	The application/json Media Type for JavaScript Object Notation (JSON) http://www.ietf.org/rfc/rfc4627.txt
[ZEROCONF]	Zero Configuration Networking http://www.zeroconf.org/ http://en.wikipedia.org/wiki/Zero_configuration_networking
[CAJUN]	CAJUN library http://sourceforge.net/projects/cajun-jsonapi
[CURL]	cURL library http://curl.haxx.se
[RESTLET]	Restlet library http://www.restlet.org
[REST-JAX]	RESTful Web Services Support in JAX-WS http://java.sun.com/developer/technicalArticles/WebServices/restful/)
[HTTPCLT]	Apache HTTP Client http://hc.apache.org/httpclient-3.x
[GWT]	Google Web Toolkit GWT http://code.google.com/webtoolkit/doc/1.6/DevGuideServerCommunication.html#DevGuideHttpRequests
[WCF 3.5]	A Guide to Designing and Building RESTful Web Services with WCF 3.5 http://msdn.microsoft.com/en-us/library/dd203052.aspx
[WCF REST]	A Developer's Guide to the WCF REST Starter Kit

1 CONNECTING TO A WALL AND ITS DEVICES

Video wall devices are usually not directly accessible by clients. Instead, a BCMC box is mounted in the wall, which acts as a gateway and presents the devices via a wall resource to the clients. The BCMC box separates the wall internal network (labelled “Wall” on the box) from customer networks (labelled “Client”) thus making sure there is no interference of the wall internal network traffic with the client networks. Clients need only one TCPI/IP network connection to the gateway to talk to the wall device or any display in the wall.

Direct connection to display devices is possible by installing a client in the Wall network or in cases where a BCMC box is not available. In such a setup clients need to open one TCP/IP connection per display. The device protocol will be the same, only URLs of resources will change, i.e. the wall part needs to be inserted when talking to the gateway.

BCMC is running an embedded web server on standard port 80. Multiple services are published by the BCMC system, i.e. an HTTP service providing internal web pages (HTML format) and the DRAMP service providing JSON formatted representations. Both services share the same port.

BCMC services are announced in both networks via [ZEROCONF], for example:

API service: “BCMC._barco-dramp._tcp”

HTTP service: “BCMC._http._tcp”

DRAMP is based on HTTP/1.1 protocol. The usual “http” scheme is used to locate resources exposed via the API:
"http:" "//" host [":" port] [abs_path]

where *host* is the network DNS name or IP address of the device network interface and the Request URI of the resource is *abs_path*.

Use the IP address or DNS name of the BCMC box to create resource URIs, e.g.

http://<BCMC ip addr>/dramp/2/wall/1,1/data/device

to retrieve basic information on the device at top left wall position such as the operation state of the device. In case of a direct network connection to a device the URL would look like:

http://<display ip addr>/dramp/2/data/device

Device services are announced via [ZEROCONF] in the Wall network in a similar way than BCM services, for example:

“Barco OVD/KVD Series myWall A1._barco-dramp._tcp”

Multiple connections at the same time are allowed but the number of parallel connections will be limited by the implementation.
Please note that the web server running on the device usually closes a connection after sending the HTTP response. Clients are expected to handle this properly and reopen the connection with every call.

2 JSON REPRESENTATION

2.1 JSON Format Elements

Wall and device request and response data is transferred in HTTP request bodies using a JSON format with defined elements. The order of elements is not defined and clients should not rely on the order shown in the box below.

Depending on the request or response type these basic elements are mandatory or optional. Optional elements can be skipped, mandatory elements must be included in requests otherwise the device will return status code 400 “Bad Request” listing the mandatory resource with status “STATE_SET_ERROR”.

Each defined element has a reserved name and always the same structure. Depending on the request type some elements are not available in a request or response.

```
{ "resource" :  { "name"  : "<resource name value>"},
  "service"  :  { "name"  : "<service name value>"},
  "action"   :  { "name"  : "<action name>",
                  "state" : "<action state>",
                  "seq"   :  <action sequence value>,
                  "value" :  <action id>}
  "params"   : [{ "name"  : "<params[0].name value>",
                  "state" : "<params[0].state value>",
                  "seq"   : "<params[0].sequence value>",
                  "value" :  <params[0].value value>
                },
                ...
                { "name"  : "<params[n].name value>",
                  "state" : "<params[n].state value>",
                  "seq"   : "<params[n].sequence value>",
                  "value" :  <params[n].value value>
                }
   ]
}
```

- "**name**" is only used within the context of "resource", "params", "action" and "service". It gives the identifier of an element and is always mandatory.
- "**state**" is only used within the context of "params" and "action". It shows the status of "params" or "action" elements and is always mandatory. See 2.4 “Parameter and Action State Values” for a detailed description of possible state values.
- "**seq**" is only used within the context of "action" and "params". It returns the sequence number of action or "params" elements. Sequence numbers are a kind of time stamp, which can be used to determine if a value is more recent than another. “seq” is always mandatory when used with action representation or data resources.
- "**value**" is only used within the context of "params" and "action". It gives the value of a "params" entry and is optional within "params" depending on its "state", but mandatory with "action" in case of action responses sent from the device.
- "**resource**" is mandatory in answers on HTTP requests, in change events and action responses sent from the device.
- "**service**" is mandatory in change events and action responses sent from the device in order to simplify the identification of the originator.
- "**params**" is mandatory in PUT requests sent to the device, with POST requests sent to the device in case the corresponding action has mandatory parameters, with answers of the device on GET requests, if the response delivers values, and with change events sent from the device.
- "**action**" is mandatory in POST requests sent to the device, with answers on POST requests sent from the device, with action responses sent from the device and with answers on GET requests on action resources sent from the device.

2.2 JSON Media Type

The JSON media type as part of a HTTP request is "application/json". According to the HTTP/1.1 standard, the Content-Type header should be set in order to identify the media type of the message body.

The API checks incoming requests for the Content-Type header field to distinguish between requests containing JSON or other request types such as file uploads (e.g. firmware updates).[1]

Content type "application/json" is assumed, if the header is missing, empty or contains “application/x-www-form-urlencoded”.

The client may add media type parameters[2] to the content-type. The API currently ignores them nevertheless.

If the message does not match any of the known media types, the API returns HTTP status code 415 “Unsupported Media Type”.

The API sets content-type "application/json" in most responses containing a JSON body. Please note, that most responses with a HTTP error status code don't have a body and therefore no content-type.

2.3 Data Types

10.3.1 Basic Data Types

JSON values are not just strings; they belong to a certain data type like in most modern programming languages. DRAMP uses the following set of data types defined by the [JSON] standard:

- String, for text data and enumerated types
- Number, for floating point and integral data
- Boolean, for Boolean values
- Object, for enclosing elements within a joint namespace
- Array, for listing the contained elements within a resource, for listing the values of enumerated types and for listing arrays of values

In addition, sets of globally unique strings are defined as enumerated strings with some resources. The required data type for resource values is defined in the Resource Reference.

Note: Where Number is used with integral data or parameters, the data type is listed in the Resource Reference as "Number, int". In this case the API will always return integral values. On setting data or parameters in such a case, clients are allowed to send double values but decimal places will be cut off by the API.

Be aware: For data type “Number” (without “,int”) the API may still send integral values if the internal value can be represented as such. Some client side JSON parsers may have problems converting such values if you always convert “Number” into double values.

10.3.2 Enumerated types

Enumerated values are defined by a set of unambiguous strings giving the meaning of the enumeration elements. Unambiguousness is global throughout all enumerated values, not only within a single enumeration.
Strings passed through the API are checked against the set of strings defined for each enumeration. The attempt to set a string, which is not within the defined set, will be refused with HTTP status code 400 Bad Request. Illegal values passed out by the firmware will cause HTTP status code 500 Internal Server Error. Enumerations and all enumeration elements are accessible through the ***Error! Reference source not found.*** of the API.
Whenever referring to an enumerated type within the API specification, the affected enumeration is defined by referring to the path of the enumeration resource of the type (e.g. "/dramp/2/enums/logLevel"), but the

[1] The first API version provides JSON type requests only. Nevertheless, the client shall always set the Content-Type of a request to avoid future code changes.

[2] See http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.7

values to be used in requests are the string values defined by the enumeration resource (e.g. "LOGLEVEL_TRACE").

2.4 Parameter and Action State Values

Parameters and action values may sometimes be available, sometimes a value is “in error” or unavailable. The state of a value shall always be checked before accessing the value in a JSON structure. Sometimes the value itself might not be available in the structure, only the state. These are possible state strings:

- “STATE_VALID” – a value is considered valid, successfully read from hardware, etc.
- “STATE_OUT_OF_RANGE” – a value has been successfully read from hardware, but is considered as invalid, e.g. a faulty sensor was detected. The value can still be present but shall not be used for further considerations such as defining error conditions, etc.
- “STATE_INVALID_ARGUMENT” – an invalid value was sent in a PUT or POST request.
- “STATE_SET_ERROR” – general failure in a PUT or POST request, something went wrong. If a request contains invalid arguments this may be returned for arguments, which were not processed.
- “STATE_WRITE_ERROR” – failure in a PUT or POST request, a value could not be changed when accessing a hardware or software component.
- “STATE_NOT_SET” – a value is part of a response but has never been read or initialized, shall not be used for further processing.
- “STATE_TEMPORARILY_NOT_AVAILABLE” – a device is in a state where the value is temporarily not available, e.g. some other action is running, which blocks the requested action, some component is still heating up, etc.
- STATE_NOT_READY – a device is in a state where the value cannot be processed at all, e.g. some values show this state when the device is in IDLE state, where the formatter component is switched off. Another case is a device that cannot be connected where its connectionState indicates CONNECTIONSTATE_NOT_RESPONDING.
- “STATE_IN_PROGRESS” – an action has been accepted for processing, will start soon or has been started.
- “STATE_REQUEST_DONE” – a request has been successfully completed.
- “STATE_TIME_OUT” – an action has not been completed within specified time.
- “STATE_NOT_AVAILABLE” – the device doesn’t support the requested action or the resource value is not available for this device type or variant.
- “STATE_ERROR” – unspecific error returned by an action request.
- “STATE_LAST_VALID” – last known value of a device before connection to the device was lost.
- “STATE_UNKNOWN” – a cached device value was never updated from the real device. This may happen if a known device is not present when starting up the BCM server or current device firmware version does not support the value at all.

2.5 API Version

Software and Firmware are released with a certain version number, for example 2.0.0. The API is part of the software but has its own version number, which defines the set of requests and resources available to clients.

The BCM or device API version can be read using a GET request on resource "data/device/version/apiVersion".[3] The version information consists of three parts: e.g. "0.9.0":

- Part 1 and 2 give the document version of the relevant specification, in this case "0.9".
- Part 3 is incremented on bug-fixes of the API, as long as the changes don't invalidate the API specification. In this case, a bug-fix of version "0.9.0" would increment the API version to "0.9.1".

Updating the specification leads to a new API version, as changes of the specification need to be implemented in the API, and the API version will be updated to the new document version with 0 at third place: "1.0.0".

The version's first digit is incremented whenever the protocol is broken by an API change. All second digit changes are fully compatible, e.g. an API implementation 1.0.0 should be still valid fulfilling an API spec 1.1, Clients developed against 1.0 spec can be run without changes against API 1.1. On the other hand, API implementation 1.1.0 should be 100% valid and fulfilling spec 1.0 reg. scope of spec 1.0. So, clients using version 1.1 features can still be run against an API implementation 1.0 if the client checks for API version when using API extensions of version 1.1.

Now, if we only change KVD specific parts in a new spec version 1.2 this means all OVD existing implementations 1.0 and 1.1 are still valid and compatible with 1.2. If we change the generic part in 1.2, then want to add a change for OVD after that, the OVD API implementation has to implement the generic changes of 1.2. Whatever change has been implemented for KVD in spec version 1.2 (generic or specific part), the OVD change will lead to new spec version 1.3.

3 HTTP STATUS CODES

3.1.1 Standard Codes

All requests sent to OLAPI return a standard HTTP status code. Status codes are used in a generalized uniform way according to following table (see also [HTTP CODE]):

Code	Name	Reason
200	OK	A request has been successfully received and processed. The answer body contains valid data if applicable.
201	Created	The subscription request of a client has successfully been processed and the subscription has been added.
202	Accepted	A POST request has been successfully received, which is being processed asynchronously. The requested action has successfully been enqueued to firmware, but can still fail while being processed. The answer body contains valid data, if applicable.
400	Bad Request	see Status Code 400 "Bad Request" Details for further information
403	Forbidden	The request sent by the client can't be performed as currently all available subscriber or action IDs are in use. In the context of action requests, if the same action is already running or no more actions are accepted by the device The client may retry the request as soon as one of the mentioned possible blocking conditions is gone, e.g. an action is done, so the device can accept further actions.

[3] Note: in current version you can only GET data/device/version and extract the details from the structured resource, accessing the detailed resource is not yet available.

404	Not Found	The client used a URI which is not directly addressable or does not exist. Examples are: - wall doesn't exist (yet) - module doesn't exist, for example module 3,1 in a wall with 2 rows and 2 columns - no real device assigned to a module position, for example module 2,1 in a wall with 2 rows and 2 columns
405	Method Not Allowed	The client used a HTTP method, which the addressed resource does not support: e.g. DELETE used on an enumeration resource. See chapter"Resource Reference" for supported methods of each resource.
409	Conflict	The subscription request of a client could not be processed as the provided ID already exists.
411	Length Required	This code is returned when receiving a POST or PUT request without the mandatory Content-Length entity.
413	Request Entity Too Large	The request's body size in bytes exceeded the internal limit of max 1 MByte[4].
414	Request-URI Too Long	This code is returned if the request's URI begins with "/dramp"[5] and has a length exceeding the internal limit of max 10 kByte[4].
415	Unsupported Media Type	This code is returned when the request contained malformed or no JSON, e.g. a representation violating the JSON specification, or the request body doesn't meet the requirements of the "multipart/form-data" message type.
500	Internal Server Error	This code is returned by the HTTP server when OLAPI is not available. This occurs in case the device software is not (yet) running or not responding, e.g. during starting up of a device or due to internal errors.
503	Service Unavailable	This code is returned when the interfacing function call between dataIF and firmware failed, or when there's not enough free space within the internal file system to receive the binary file for updating software.

3.1.2 Status Code 400 "Bad Request" Details

3.1.2.1 Requests without known parameters

If a client's PUT request on a data resource contains an empty "params" array, or the "params" array contains resource names, which are unknown within the context of the addressed resource, the request will be denied completely. The response will list any of its writable resources with status "STATE_SET_ERROR".

Example response:

```
Status: 400 Bad Request
Content-Type: "application/json"

{ "resource" :  { "name"  : "http://10.2.1.12/dramp/2/wall/1,1/data/device"},
  "params"   : [{ "name"  : "/dramp/2/wall/1,1/data/device/startupState",
                  "state" : "STATE_SET_ERROR"
                }
```

[4] 1 MByte equals 1024 kBytes, 1 kByte equals 1024 Bytes

[5] Uri's not beginning with "/dramp" are handled by the http server and not by the API. The currently used lighttpd server returns HTTP status 500 Internal Server Error, if the URI length exceeds 254 bytes.

```
  ]
}
```

If a client's POST request on an actions resource contains an empty "params" array, or the "params" array contains parameter names, which are unknown within the context of the requested action, and the action has mandatory parameters, the request will be denied completely. The answer will list any of the possible parameters of the action with status "STATE_SET_ERROR".

Example response:

```
Status: 400 Bad Request
Content-Type: "application/json"

{ "resource" :  { "name"  : "http://10.2.1.12/dramp/2/wall/actions"},
  "action"   :  { "name"  : "updateOperationState",
                  "state"  : "STATE_SET_ERROR"
  }
  "params"   : [{ "name"  : "pOperationState",
                  "state" : "STATE_SET_ERROR"
                }
  ]
}
```

3.1.2.2 Requests with a parameter of wrong basic data type

Basic data types are listed in 2.3 “Data Types”. If a client's PUT request on a data resource or POST request on an actions resource contains a parameter of different basic data type than specified, the request will be denied completely. The response lists the resource or parameter with status "STATE_INVALID_ARGUMENT":

Example response: "startupState" is an enumerated value; its data type is expected to be String:

```
Status: 400 Bad Request
Content-Type: "application/json"

{ "resource" :  { "name"  : "http://10.2.1.12/dramp/2/wall/1,1/data/device"},
  "params"   : [{ "name"  : "/dramp/2/wall/1,1/data/device/startupState",
                  "state" : "STATE_INVALID_ARGUMENT"
                }
  ]
}
```

Example response (POST request on actions):
data type of " pBrightnessTarget" is expected to be Number:

```
Status: 400 Bad Request
Content-Type: "application/json"

{ "resource" :  { "name"  : "http://10.2.1.12/dramp/2/wall/actions/"},
  "action"   :  { "name"  : "updateColorBrightness",
                  "state" : "STATE_SET_ERROR"
                }
  "params"   : [{ "name"  : "pTargetBrightness",
                  "state" : "STATE_INVALID_ARGUMENT"
                }
  ]
}
```

3.1.2.3 Requests with a parameter value out of range

If a client's request contains a string value for an enumerated value, which does not match any of the allowed strings, the value is considered to be out of range.

Numerical values can also be out of range as the target data type, which is used internally can have a smaller value range than the JSON Number type (which is always a “double” data type).

Example: the pTargetBrightness component of the ControlBrightness resource is 16 bit integer.

Example response on a POST request:

```
Status: 400 Bad Request
Content-Type "application/json"

{ "resource" :  { "name"  : "http://10.2.1.12/dramp/2/wall/actions"},
  "action"   :  { "name"  : "updateColorBrightness",
                  "state" : "STATE_SET_ERROR"
                }
  "params"   : [{ "name"  : "pTargetBrightness",
                  "state" : "STATE_OUT_OF_RANGE"
                }
  ]
}
```

3.1.2.4 Requests with a parameter or mandatory value missing

If the “params” array of a client's PUT request is missing a mandatory value, or a POST request is missing a mandatory parameter, the request will be denied completely. The answer will list the affected non-optional resource or parameter with status "STATE_SET_ERROR".

Example response on a POST request:

```
Status: 400 Bad Request
Content-Type: "application/json"

{ "resource" :  { "name"  : "http://10.2.1.12/dramp/2/wall/actions"},
  "action"   :  { "name"  : "updateBrightnessMode",
                  "state" : "STATE_SET_ERROR"
                }
  "params"   : [{ "name"  : "pBrightnessMode",
                  "state" : "STATE_SET_ERROR"
                }
  ]
}
```

3.1.2.5 Actions request returning an error state

If a client's action request for an action could not be en-queued internally, because the device returns any error state or is in an error state, the request will be answered with HTTP status 400 “Bad request”. The action's state will give the error detail (see 2.4 “Parameter and Action State Values”).
A special situation where this happens is, where there is no connection to the device is detected, or the connection got lost for some reason. In this case “state” returns STATE_NOT_READY.

Example response:

```
Status: 400 Bad Request
Content-Type: "application/json"

{ "resource" :  { "name"  : "http://10.2.1.12/dramp/2/wall/actions/"},
  "action"   :  { "name"  : "updateTargetBrightness",
                  "state" : "STATE_TEMPORARILY_NOT_AVAILABLE"
                }
}
```

3.1.2.6 Actions request with unsupported action name

If a client request asks for an unknown action name the API returns HTTP status 400 “Bad request” and an action state of STATE_ERROR. The name of the unknown action is returned.

3.1.3 Resource root to access a wall in case of multi wall configurations

(Currently this feature is only supported by …).

There are two alternatives to specify the resource root for a wall:

In order to access the first wall in the configuration (or the one and only wall in a single wall configuration) use the “wall” resource name:

/dramp/2/wall/data/device

Example:


```
   "resource": {"name": "http://10.100.180.99:62000/dramp/2/wall/data/device/"},
   "params": [
      {
         "name": "/dramp/2/wall/data/device/wallName",
         "state": "STATE_VALID",
         "seq": 0,
         "value": "BCM WALL"
      },
      {
         "name": "/dramp/2/wall/data/device/wallColumns",
         "state": "STATE_VALID",
         "seq": 0,
         "value": 3
      },
      {
         "name": "/dramp/2/wall/data/config/wallRows",
         "state": "STATE_VALID",
         "seq": 0,
         "value": 2
      }
   ]
}
```


In order to specify a certain wall in a configuration with one or more walls the wall name has to be specified

/dramp/2/walls/{WallName}/data/device

/dramp/2/walls/{WallName}/1,1/data/device

Example:

```
   "resource": {"name": "http:// 10.100.180.99:62000/dramp/2/walls/kvd/data/device/"},
   "params": [
      {
         "name": "/dramp/2/walls/kvd/data/config/wallName",
         "state": "STATE_VALID",
         "seq": 0,
         "value": "LDX"
      },
      {
         "name": "/dramp/2/walls/kvd/data/device/wallColumns",
         "state": "STATE_VALID",
         "seq": 0,
         "value": 1
      },
      {
         "name": "/dramp/2/walls/kvd/data/device/wallRows",
         "state": "STATE_VALID",
         "seq": 0,
         "value": 2
      }
   ]
}
```

4 RESOURCE TYPES

Currently there are two resource types: data resources and the actions resource. Resources may be “general” resources or device specific (see Resource Reference in this document). General resources are available on all supported device types in the same generic way; device specific resources are only available if the device “behind” the API supports such a feature. For example, OVD and KVD support Sense-X, an entry level display maybe not. All Sense-X related resources are not available for the entry level device. Requests on not available resources return with HTTP status code 404 “Not available”.

4.1 Data Resources

Data resources represent the device model to the “outside world”. Data resource URI’s always begin with “/data”. GET requests are used to retrieve values, some resources can be modified using a PUT request.

Request bodies may contain these elements:

- "**resource**" is used to identify the response as answer from the addressed resource.

A single struct within the "params" array contains data name, state and value:

- "**name**" is the name of the resource as path beginning at the top level of the resource model hierarchy or relative to the resource entry. The JSON type of name is “String”.
- "**state**" is the status of the resource, see chapter “Parameter and Action State Values”. The JSON type of state is “String, enumeration”.
- "**seq**" is the sequence number of the resource as given by the firmware. The firmware updates this number internally when updating the resource value or state. A higher number means a newer resource value and state.

A change in sequence number does not necessarily show a change in value or state as well. It is explicitly allowed to send the same value or state with a higher sequence number, but the device will never send a changed state or value with a lower sequence number.[6]
Note that this number is reset on firmware start-up or overflow. It is the responsibility of the client to make sure that sequence numbers sent before and after the reset are never compared to each other. The JSON type of seq is “Number, int”.

- "**value**" is the current value of the resource. The JSON type can be one of the basic or enumerated types as defined above. The value element is missing, if state is not "STATE_VALID" or "STATE_OUT_OF_RANGE".

The device will answer on a GET request with one of these HTTP status codes[7]:

- 200 OK
- 404 Not Found
- 414 Request-URI Too Long
- 500 Internal Server Error
- 503 Service Unavailable.

Note: HTTP status code 200 OK is the only code where the response sends a body, with all other status codes the response body will be empty!

Some resources support reading only in some modes or states of the device. A GET request on these resources without being in the specified mode or state will still be answered with HTTP status 200 OK, but the state flags of the according values will indicate an error state (see chapter“Resource Reference”).

Some resources are directly writable. Clients must use HTTP method PUT on a data resource to change the current value of one or more sub-resources within a resource. PUT is a “synchronous” HTTP call, meaning the device's answer to the client won't be sent until the resource value has been changed.

The device will answer on a PUT request with one of these HTTP status code:

- 200 OK
- 400 Bad Request
- 404 Not Found
- 411 Length Required
- 413 Request Entity Too Large
- 414 Request-URI Too Long
- 415 Unsupported Media Type
- 500 Internal Server Error
- 503 Service Unavailable

Note: HTTP status code 200 OK is the only code where the request has been processed!

With HTTP status 400 the response body will contain further error details (see chapter“Status Code 400 Bad Request Details”).

Some resources support writing only in some modes or states of the device. A PUT request on these resources without being in the specified mode or state will be answered with HTTP status 400 Bad

[6] This is an implementation hint and especially useful in cases where tracking changes for every single value is too costly for the device. If a value or state changes, the device may send out a bundle of values and states where all have the same new sequence number, but only one value or state has really changed.

[7] See chapter “HTTP Status Codes” for detailed explanation.

Request, and the state flags of the according values will indicate an error state (see chapter“Resource Reference”).

When writing resources containing string values the value will be cut to the maximum allowed length if the length exceeds the limit. The request will send back HTTP status 200!

4.2 Actions Resource

The actions resource represents a collection of implicitly triggered state changes at the device. Actions will always be handled asynchronously, means the request to trigger an action is acknowledged immediately sending back an HTTP 202 response, but the action may be finished later after sending the acknowledge. An accepted action creates a temporary resource, which can be polled with GET requests for the result of the action.

4.2.1 Triggering Actions

To trigger an action, clients need to send a POST request to the actions resource, e.g. "http://10.2.1.12/dramp/2/wall/1,1/actions/". The request body must contain the action name specifying the action to be executed. If an action has additional parameters, they are passed in the request body.

Parameter names and data types are specific to the action and listed in chapter “Resource Reference, Actions Resource”. Although parameter names look very much like data resource names they are not directly related and should not be confused with data resources.

Some parameters are optional, whereas parameters not contained in the client's POST request body are skipped and not further handled.

Actions usually result in device state changes. The affected data resources are listed with each action in chapter “Resource Reference, Actions Resource”.

The device will answer on a POST request with one of these HTTP status codes:

- 200 OK, action has been completed immediately, no action response is sent to the client
- 202 Accepted, action has been accepted for further processing, action response is sent later
- 400 Bad Request
- 403 Forbidden
- 404 Not Found
- 411 Length Required
- 413 Request Entity Too Large
- 414 Request-URI Too Long
- 415 Unsupported Media Type
- 500 Internal Server Error
- 503 Service Unavailable

Note: HTTP status code 200 OK and 202 Accepted are the only codes where the request has been processed!

With HTTP status 400 the response body will contain further error details (see chapter “HTTP Status Codes, Status Code 400 “Bad Request” Details”).

If the requested action is processed, the POST request returns HTTP response with status code 200 or 202 and a JSON body containing name, state, sequence and value of the action:

- "**name**" is the action's name. The JSON type of name is “String”.
- "**state**" is the action's status as returned by firmware. The JSON type of state is “String, enumeration”.
- "**seq**": is the action's sequence number. A higher number means a newer resource state.
  Note that this number is reset on software start-up or overflow. It is the responsibility of the client to

make sure that sequence numbers sent before and after the reset are never compared to each other. The JSON type of seq is “Number, int”.

- "**value**" is the action's ID as assigned by the API. The JSON type of value is “Number”.

The action ID identifies the action as long as it is running, and for some time after it has been completed in order to give clients a chance to retrieve the processing state of the action. The action ID has no meaning for the order in which enqueued actions are being processed.
NOTE: The value (action ID) can be 0. In this case the API will not keep the processing state of the action, clients cannot retrieve the processing state after the action has been completed.

Example acknowledgement:

```
{ "resource" : { "name"  : "http://10.2.1.12/dramp/2/wall/actions/3"},
  "action"   : { "name"  : "updateOperationState",
                 "state" : "STATE_IN_PROGRESS",
                 "seq"   : 9001,
                 "value" : 3
  }
}
```

Accepting an action actually means that the firmware receives the action request and puts it into its queue. The request can still fail if e.g. preconditions are not met or parameters are out of range. The action itself may have not yet been started at the time the acknowledgement is sent out.

If en-queuing the requested action fails, because all available action IDs are in use, the acknowledgement will return HTTP status code 403 “Forbidden”. In this case the client should wait some seconds or check for completion of another action and retry.

After the requested action has been finished, the temporary action's state tells the client, if the action has been finished successfully or with failure.

4.2.2 Wall Actions

Actions can be triggered on the whole wall. In this case the action response will contain the status info for each individual module action in the following form.

The wall action is considered STATE_REQUEST_DONE when all corresponding module actions are STATE_REQUEST_DONE.

If a certain module returns a problem, params returns the error details where the problem occurred. If the problem is related to an action parameter the error detail will give the param name as error detail. If the action problem is related to a module resource the module's resource name is returned, the error details are mapped to this resource's name and added to the params array of the action's representation. Note, that the state of the actual resource in the data resource model is not affected by this.

```
"resource": {"name": "http://10.2.0.5/dramp/2/wall/actions/4711"},
   "action": {
      "name": "idle",
      "state": "STATE_ERROR",
      "seq": 0,
      "value": 4711
   },
   "params": [
      {
         "name": "/dramp/2/wall/1,1/data/device/operationState",
         "state": " STATE_TEMPORARILY_NOT_AVAILABLE",
```

```
         "seq": 0,
         "value": 4712
      },
      {
         "name": "/dramp/wall/1,3/data/device/operationState",
         "state": " STATE_TEMPORARILY_NOT_AVAILABLE ",
         "seq": 0,
         "value": 4713
      }
   ]
}
```

4.2.3 Action Identification

The number of actions which the device can queue up or even run simultaneously is limited. Therefore the device provides an amount of action IDs which are temporarily allocated when starting an action and freed again, after the action has been finished. Currently there are 8 action IDs available which can be allocated at the same time.

Action IDs are integral numbers, they are unique among all currently running actions.

Allocating an action ID is done when a client requests an action to be triggered.

Clients can poll for the action response by sending a GET request to the corresponding actions resource and the trailing action ID assigned to the action.

Example response on GET on "http://10.2.0.5/dramp/2/wall/1,1/actions/3" in case the action is still running:

```
{ "resource" : { "name"  : "http://10.2.0.5/dramp/2/wall/1,1/actions/3"},
  "action"   : { "name"  : "updateOperationState",
                 "state" : "STATE_IN_PROGRESS",
                 "seq"   : 12,
                 "value" : 3
  }
}
```

Example answer on GET "http://10.2.0.5/dramp/2/wall/1,1/actions/3" in case the action is finished:

```
{ "resource" : { "name"  : "http://10.2.0.5/dramp/2/wall/1,1/actions/3"},
  "action"   : { "name"  : "on",
                 "state" : "STATE_REQUEST_DONE",
                 "seq"   : 13,
                 "value" : 3
  }
}
```

In case the action could not be finished successfully, the action's state will be set to "STATE_ERROR". The error details will be returned in the "params" array with the parameter, which caused the error.

Example response on GET "http://10.2.0.5/dramp/2/wall/1,1/actions/3/" in case the action could not be processed successfully:

```
{ "resource" : { "name"  : "http://10.2.0.5/dramp/2/wall/1,1/actions/3"},
  "action"   : { "name"  : "switchInput",
                 "state" : "STATE_ERROR",
                 "seq"   : 12,
                 "value" : 3
  }
```

```
  "params"   : [{ "name"  : "pSelectedInput",
                  "state" : "STATE_OUT_OF_RANGE",
                  "seq"   : 1234567,
                  "value" : “INPUT_3”
                }
  ]
}
```

The device will answer on a GET request with one of these HTTP status codes:

- 200 OK
- 404 Not Found
- 414 Request-URI Too Long
- 500 Internal Server Error
- 503 Service Unavailable

Note: HTTP status code 200 OK is the only code where the response sends a body, with all other status codes the response body will be empty!

Accessing an action-ID which is already freed or has been deleted by a client will be answered with HTTP status code 404 “Not Found”.

4.2.4 Freeing an Action ID

Freeing an action ID means to remove the corresponding resource, the action ID itself may be reused for future action requests. The API will not reuse the freed action ID for the next 1000+ action requests.

Freeing an action ID is done:

- After a defined time period (currently 4 seconds) after the device has finished the action successfully or with failure.
- Immediately when a client does a DELETE request on the action's resource.
  Note: Deleting an existing action ID is possible for actions still in progress.

Example response on DELETE "http://10.2.0.5/dramp/2/wall/1,1/actions/3003":

```
HTTP/1.1 200 OK
Content-Length: 0
Date: Fri, 27 Nov 2009 07:21:17 GMT
```

The device will answer on a DELETE request with one of these HTTP status codes:

- 200 OK
- 404 Not Found
- 414 Request-URI Too Long
- 500 Internal Server Error
- 503 Service Unavailable

The response will never have a body.

5 RESOURCE REFERENCE – GENERAL

This functionality is available for all device types.

5.1 Data Resources

Video wall resources are organized along a hierarchical component structure, {…} is to be replaced by real strings:

- The top most resource is the wall service usually provided by BCMC at URL http://{BCMC ip or hostname}/dramp/2/data.
- The wall service can manage multiple walls at http://{BCMC ip or hostname}/dramp/2/walls/{WallName}[8]
- For the first wall of a wall service we provide a special resource name at http://{BCMC ip or hostname}/dramp/2/wall. The first wall is the first entry in the list returned by the /dramp/2/walls resource.
- A wall consists of several devices in a grid at coordinate “1,1”, “1,2”, 2,1”, etc.: http://{BCMC ip or hostname}/dramp/2/wall/1,2/data.
  Grid coordinates are given as <column>,<row> where numbering starts at top left position, counting to the right and bottom. For example 1,2 is the coordinate at column 1 (left most), row 2 (second row counting from top).
- Please note: resource names are given as relative URLs[9]. To turn a relative URL into a full URL you need to add protocol and machine information, as well as service and service context URL parts. The service part in case of DRAMP is always “/dramp/2”. The service context depends on which service is handling a request. For example, sending a GET request to the device service itself results in an empty service context. There is no need to provide more information than the service resource name. If you send the same device related request to the wall service you need to specify which device you want to address. In this case the service context is added as “wall/{col,row}/”
  A general URL follows this structure:
  [protocol]:[host]/[service]/[context]/[resource name]
  Example: http://10.20.30.40/dramp/2/wall/2,1/data/device
  The protocol and service parts are fix, “http://” and “/dramp/2”. Host part is either BCMC or the device (in case you are in the Wall network or BCMC is not available), for ex. 10.20.30.40. Context is “wall/{col,row}” or empty, and resource names are listed in the Resource Reference.
- How to deal with reserved characters?
  In case you use a character that is reserved for a special purpose you need to use percent-encoding to address the wall[10], e.g. %20 for the ASCII space character (SP), for example: /dramp/2/walls/Show%20Room/data/device.

[8] The current implementation is restricted to 1 wall per BCMC device but this may change in the future

[9] See http://www.w3.org/TR/WD-html40-970917/htmlweb.html#h-5.1.2

[10] See RFC 3986 „Uniform Resource Identifier (URI): Generic Syntax“, „2.1 Percent-Encoding“, http://tools.ietf.org/html/rfc3986#page-11

5.1.1 Wall Service Resources

5.1.1.1 Resource "data/isAlive"

isAlive is for clients to check the communication is working and the wall service being up by sending a GET request to this resource. The response of the device will never contain a params array.

Resource Name	Data type	supported methods
This resource has no sub resources		GET

5.1.2 Walls Resource

5.1.2.1 Resource "walls"

BCM versions supporting multiple walls in a single service expose a “walls” resource that clients can GET in the usual way to retrieve the list of names of existing walls. The name of a wall can then be used to address a certain wall.
Example: GET /dramp/2/walls returns:

```
{
  …
  "params" :  [
    { "name" : "/dramp/2/walls", "value" : ["wall1", "wall2"]
    }
  ]
}
```

For accessing wall1 this turns into such a URL:
/dramp/2/walls/wall1/data/device

5.1.3 Wall Resources

Currently none.

5.1.4 Device Resources

Currently none.

5.2 Actions Resource

There is only one actions resource for every component, but multiple actions will cause different state changes depending on parameters sent to the wall or device.

5.2.1 Wall Service Actions

Resource “actions”

Currently none.

5.2.2 Wall Actions

Resource “{wall}/actions”

5.2.2.1 Action "updateOperationState"

Switch all devices of the wall from “standby” or “idle” to “on” state.

Maximum runtime: 60 seconds, typically ?? seconds

Parameter Names	Data type	Optional
pOperationState	“enums/operationState”	no

5.2.3 Device Actions

Currently none.

5.3 Enumeration Resources

5.3.1 Resource “enums/operationState”

Value Name	Data type	
“OPERATIONSTATE_ON”	String	GET
“OPERATIONSTATE_IDLE”	String	GET

* The device may never return this state if it goes to deep standby where all components are switched off
** The device may return this state during shutdown after initiating a reboot as well as during startup.

5.3.2 Resource “enums/connectionState”

Value Name	Data type	
“CONNECTIONSTATE_OK”	String	GET
“CONNECTIONSTATE_NOT_RESPONDING”	String	GET

6 RESOURCE OVERVIEW

6.1 Wall Service Level

Resource Name	GET	PUT	POST	DELETE
isAlive	x	-	-	-

6.2 Wall Level

Resource Name	GET	PUT	POST	DELETE

6.3 Device Level

Resource Name	GET	PUT	POST	DELETE

7 REQUEST EXAMPLES – HTTP PROTOCOL LEVEL

Note: \x0d0a shows hexadecimal ASCII values, which have to be included in the string sent over to the device, all other line breaks, white space etc. shown below is for documentation only.

7.1 Switch Wall ON, action completes immediately

1) Action Request: HTTP Request Client -> Device

POST request to resource “http://10.20.30.40/dramp/2/wall/actions/”

```
POST\x20/dramp/2/wall/actions\x20HTTP/1.1\x0d0a
Content-Type:application/json\x0d0a
Content-Length:108\x0d0a
\x0d0a
{"action":{"name":"updateOperationState"},
  ”params”:[{“name”:”pOperationState“,”value”:”OPERATIONSTATE_ON”}]
}\x0d0a
```

2) Action Response: HTTP Response Device -> Client

```
HTTP/1.1\x20200\x20OK\x0d0a
Content-Type:application/json\x0d0a
Transfer-Encoding:\x20chunked\x0d0a
Date:\x20Sat,\x2001\x20Jan\x202000\x2022:21:04\x20GMT\x0d\0a
Server:\x20lighttpd/1.4.35\x0d0a
\x0d0a
{\x0d0a
\x09"resource"\x20:\x20{\x0d0a
\x09\x09"name"\x20:\x20"http://10.20.30.40/dramp/2/wall/actions/1"\x0d\0a
\x09},\x0d\0a
\x09"action"\x20:\x20{\x0d\0a
\x09\x09"name"\x20:\x20"updateOperationState",\x0d\0a
\x09\x09"state"\x20:\x20"STATE_REQUEST_DONE",\x0d\0a
\x09\x09"value"\x20:\x201\x0d\0a
\x09}\x0d\0a
}\x0d\0a
```

7.2 Switch Wall ON, action completion delayed

1a) Action Request: HTTP Request Client -> Device
POST request to resource http://10.20.30.40/dramp/2/wall/actions/
-> same as above 7.1 1)

1b) Action Request: HTTP Response Device -> Client

```
HTTP/1.1\x20202\x20Accepted\x0d0a
Content-Type:application/json\x0d0a
Transfer-Encoding:\x20chunked\x0d0a
Date:\x20Sat,\x2001\x20Jan\x202000\x2022:21:04\x20GMT\x0d\0a
Server:\x20lighttpd/1.4.35\x0d0a
\x0d0a
{\x0d0a
\x09"resource"\x20:\x20{\x0d0a
\x09\x09"name"\x20:\x20"http://10.20.30.40/dramp/2/wall/actions/1"\x0d\0a
\x09},\x0d\0a
\x09"action"\x20:\x20{\x0d\0a
\x09\x09"name"\x20:\x20"updateOperationState",\x0d\0a
\x09\x09"state"\x20:\x20"STATE_IN_PROGRESS",\x0d\0a
\x09\x09"value"\x20:\x208476\x0d\0a
\x09}\x0d\0a
}\x0d\0a
```

2a) Action Response: HTTP Request Client -> Device
GET request to http:// 10.20.30.40/dramp/2/wall/actions/8476”

```
GET\x20/dramp/2/wall/1,1/actions/8476\x20HTTP/1.1\x0d0a
Content-Type:application/json\x0d0a
Content-Length:0\x0d0a
\x0d0a
```

2b) Action Response: HTTP Response Device -> Client
-> same as above 7.1 2)

7.3 Set Wall into IDLE state

1) HTTP Request
POST request to resource “http://10.20.30.40/dramp/2/wall/actions/”

```
POST\x20/dramp/2/wall/actions\x20HTTP/1.1\x0d0a
Content-Type:application/json\x0d0a
Content-Length:108\x0d0a
\x0d0a
{"action":{"name":"updateOperationState"},
  ”params”:[{“name”:”pOperationState“,”value”:”OPERATIONSTATE_IDLE”}]
}\x0d0a
```

2b) Action Response: HTTP Response Device -> Client
-> same as above 7.1 2)

8 REQUEST EXAMPLES – CURL

[CURL] is a command line tool and library, which is free and open source and compiles and runs under a wide variety of operating systems, such as Linux, Win32/64, OS X, etc. This tool provides an easy way to test the API and study its behavior, or even use the library in production code.

Note: in our examples we assume the client is connected to the “Client” network on the BCM gateway device, gBCM IP address is 10.20.30.40.

8.1 GET isAlive

```
curl -v http://10.20.30.40/dramp/2/data/isalive
* About to connect() to 10.20.30.40 port 80 (#0)
*   Trying 10.20.30.40... connected
* Connected to 10.20.30.40 (10.20.30.40) port 80 (#0)
> GET /dramp/2/data/isalive HTTP/1.1
> User-Agent: curl/7.19.6 (i386-pc-win32) libcurl/7.19.6 zlib/1.2.3
> Host: 10.20.30.40
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json;
< Transfer-Encoding: chunked
< Date: Wed, 19 Jul 2014 08:51:24 GMT
< Server: lighttpd/1.4.31
<
{
        "resource": {
                "name": "http://10.20.30.40/dramp/2/data/isAlive/"
        }
}* Connection #0 to host 10.20.30.40 left intact
* Closing connection #0
```

8.2 Switch Wall ON

```
curl -v --data @post_on.json -H "Content-Type:application/json"
http://10.20.30.40/dramp/2/wall/actions/
* About to connect() to 10.20.30.40 port 80 (#0)
*   Trying 10.20.30.40... connected
* Connected to 10.20.30.40 (10.20.30.40) port 80 (#0)
> POST /dramp/2/wall/actions HTTP/1.1
> User-Agent: curl/7.19.6 (i386-pc-win32) libcurl/7.19.6 zlib/1.2.3
> Host: 10.20.30.40
> Accept: */*
>Content-Type: application/json;
>Content-Length: 108
>
<HTTP/1.1 200 OK
<Content-Type: application/json;
<Transfer-Encoding: chunked
< Date: Wed, 19 Jul 2014 08:51:24 GMT
< Server: lighttpd/1.4.31
<
{
```

```
        "resource": {
                "name": "http://10.20.30.40/dramp/2/wall/actions/1"
        },
        "action": {
      "name": "updateOperationState",
      "state": "STATE_REQUEST_DONE ",
      "value": 1
   }
}* Connection #0 to host 10.20.30.40 left intact
* Closing connection #0
```

Text file “post_on.json” contains the action body:

```
{"action":{"name":"updateOperationState"},"params":[{"name":"pOperationState",
"value":"OPERATIONSTATE_ON"}]}
```

8.3 GET last action state

After finishing an action the state can still be accessed on the actions resource with the action ID appended, for example we access the action state of last ON command returning ID 1.

```
curl -v http://10.20.30.40/dramp/2/wall/actions/1
* About to connect() to 10.20.30.40 port 80 (#0)
*   Trying 10.20.30.40... connected
* Connected to 10.20.30.40 (10.20.30.40) port 80 (#0)
> GET /dramp/2/wall/actions/1 HTTP/1.1
> User-Agent: curl/7.19.6 (i386-pc-win32) libcurl/7.19.6 zlib/1.2.3
> Host: 10.20.30.40
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json;
< Transfer-Encoding: chunked
< Date: Wed, 19 Jul 2014 08:51:24 GMT
< Server: lighttpd/1.4.31
<
{
        "resource": {
                "name": "http://10.20.30.40/dramp/2/wall/actions/1"
        },
        "action": {
        "name": "updateOperationState",
        "state": "STATE_REQUEST_DONE ",
        "seq” : 1,
        "value": 1
   }

}* Connection #0 to host 10.20.30.40 left intact
* Closing connection #0
```

8.4 Switch Wall IDLE

```
curl --data @post_idle.json -H "Content-Type:application/json"
http://10.20.30.40/dramp/2/wall/actions/
* About to connect() to 10.20.30.40 port 80 (#0)
*   Trying 10.20.30.40... connected
* Connected to 10.20.30.40 (10.20.30.40) port 80 (#0)
> POST /dramp/2/wall/actions HTTP/1.1
> User-Agent: curl/7.19.6 (i386-pc-win32) libcurl/7.19.6 zlib/1.2.3
> Host: 10.20.30.40
> Accept: */*
>Content-Type: application/json;
>Content-Length: 110
>
<HTTP/1.1 200 OK
<Content-Type: application/json;
<Transfer-Encoding: chunked
< Date: Wed, 19 Jul 2014 08:51:24 GMT
< Server: lighttpd/1.4.31
<
{
        "resource": {
                "name": "http://10.20.30.40/dramp/2/wall/actions/1001"
        },
        "action": {
      "name": "updateOperationState",
      "state": "STATE_REQUEST_DONE ",
      "seq" : 2,
      "value": 1001
   }
}* Connection #0 to host 10.20.30.40 left intact
* Closing connection #0
```

Text file “post_idle.json” contains the action body:

```
{"action":{"name":"updateOperationState"},"params":[{"name":"pOperationState",
"value":"OPERATIONSTATE_IDLE"}]}
```