DCP-555

Digital Conferencing Processor with VoIP

Control API Guide

Table of Contents

Introduction

This guide describes how to control and monitor the DCP-555 using third-party control. This guide should be used in conjunction with a device specific document that details the objects and parameters of that specific device.

Basic Concepts

- TheDCP-555APIisacommandandresponseprotocol.Eachcommandrequiresaresponse,andanother command cannot be issued until the response has been received.
- TheDCP-555APIisdesignedforTCP/IPconnectionsandusestheregisteredport4197.
- TheDCP-555APIiscasesensitiveandallcommandsshouldbesentinlowercase.
- TheDCP-555APIdevicesaremodeledusingobjects.Objectscancontainotherobjectsandparameters.
- TheDCP-555APIobjectsandparametersareaddressedusingapath.Thepathstoobjectsand parameters will be defined in a separate document that is specific to the device you are wanting to control.

Protocol Definition

The protocol is command/response from client to server and from server to client. A client can send one command to the server, then it must wait for a response from the server before sending another command. In the meantime, the server can send one command to the client, but must wait for a response from the client before sending another command. In a way, this is peer-to-peer, but the difference is, the client is simply the endpoint that initiated the TCP connection.

Syntax

The general syntax of a DCP-555 API message is:

<command/reply> <JSON> [\r]\n

Spaces are used as delimiters between the items in a message, and the message is terminated by an optional carriage return (\r) followed by a mandatory newline (\n).

Leading and trailing spaces on all arguments and commands are stripped and not preserved. Embedded spaces are preserved, so keys and names and so on can have embedded spaces.

The JSON portion contains all the arguments for the command. The JSON portion must be a JSON Object {}.

Case Sensitivity

Case-sensitive: commands, paths, names, replies, message-forwarding info, parameter names.

The DCP-555 API is case sensitive. All commands and keywords are lower case.

Case-insensitivity only applies to ascii, since UTF-8 may contain encodings that are not to-lower’able.

Path Syntax

A crucial concept in the DCP-555 API is that of a path name. The path name is the name given to an element. The path name includes the path and the name of the element. A forward slash “/” is used to distinguish hierarchy in the path name. Paths can be surrounded by double quotes if required due to white space in the path. An example of a path name is:

“/Audio/P1/Channel 1/Gain”

The forward slash at the start indicates that the path is absolute. There is no need to include the device name in the path.

Command / Reply

Each command has a specific reply. The reply prepends @ to the original command to form the reply. This prepending makes it easy for a parser to quickly distinguish a reply command.

command

@command

Arguments

Arguments are specified by JSON. It is possible for a command to have no arguments. If a command/ reply only ever has one argument, it may be a more simple JSON string (always quoted) or JSON number (never quoted).

Creating the Connection

DCP-555 devices listen on port 4197 and can support multiple connections. To connect to a DCP-555 device, you must open a TCP socket to the IP address of the device. Once the connection is open, the device will not close the connection until either the client (your program) closes the connection or issues a goodbye command.

Working with Parameters

A DCP-555 device is made up of objects and parameters. All parameters have a parent object and some objects may also have other objects as children. Each object and parameter is named and you can think of the structure like s file system where objects are like folders and parameters are like files. You can access a parameter or object using its full path name. An example of a path name could be:

“/Audio/Output Level/Gain”

In this example, Audio and Output Level are both objects, and Gain is the parameter.

Understanding Control Laws

All Parameters in the model have an underlying ‘control law’. This allows the parameter to convert between various representations of the parameter value. The main representations (also called formats) are:

- Default:astringrepresentationofthevaluewithasensibleprecisionandunits(e.g“1.1kHz”).
- Number:astringrepresentationofthevalueasafloatingpointnumberwithnoroundingorunits (e.g. “1100.0”).
- Normalized(Norm):astringrepresentationofthevalueasafloatingpointnumberbetween0.0and 1.0, where 0.0 indicates the parameter is at its minimum value and 1.0 indicates the parameter is at its maximum value. Note that the value is not always mapped linearly to the range of the parameter. For example, frequency parameters typically use a logarithmic mapping.

The mapping from normalized to value is part of the control law and the DCP-555 API is designed in such a way that the GUI designer can mostly simply control the parameter using values from 0.0 to 1.0 without having to understand what is going on inside the model.

When controlling a parameter using a slider you would generally simply map the value of the slider to a range from 0.0 to 1.0. Then you can control the parameter using the normalized format of the value. To use the normalized value you must specify “format”:”Norm” in the set, get, or subscribe messages:

set {“path”:”/Config/Channel4PEQ/Band_1_Frequency”, “value”:”0.5”, “format”:”Norm”}

And the model responds with:

@set {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Frequency”,”value”:”0.49993”}

As can be seen, the model has responded with a very slightly different value to the one requested (0.49993 instead of 0.5). Since the normalized value may not map precisely to an exact value in Hz it is rounded by the model and then the actual normalized value returned. In the case of this frequency parameter the difference is small but for other parameters it may be much more significant.

For example, the filter type parameter is in integer from 0 to 2 and represents the following values:

- 0: Bell
- 1: LowShelf
- 2: HighShelf

Although you would typically use a list box for this type of parameter it could be controlled by a slider too. In this case setting a normalized value will return a rounded value:

set {“path”:”/Config/Channel4PEQ/Band_1_Type”, “value”:”0.2”, “format”:”Norm”} @set {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Type”,”value”:”0”}

set {“path”:”/Config/Channel4PEQ/Band_1_Type”, “value”:”0.4”, “format”:”Norm”} @set {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Type”,”value”:”0.5”}

This is because the parameter can only have 3 values and these correspond to the following normalized values:

- 0.0: Bell
- 0.5: LowShelf
- 1.0: HighShelf

Default Format

The ‘Default’ format for a parameter varies depending on the type of the parameter. However, the ‘Default’ format is intended to be a ‘nice’ human readable string. For the frequency parameter we’ve used above this means rounding to either Hz or kHz depending on the value and adding the correct units after the number.

For other parameters this may mean returning a string from an enum. For example, the filter type actually has internal values of 0, 1, and 2 but these are represented by strings “Bell”, “Low Shelf” and High Shelf”. Therefore, the ‘Default’ format would be one of the 3 enum strings. Interestingly, the ‘Number’ format would return 0, 1, or 2, and the ‘Normalized’ format would be 0.0, 0.5, or 1.0.

Parameter Formats and Control Types

The different control types will be used by different types of control in your application. Examples are:

- **Slider:** Generally uses normalized value from 0.0 to1.0.
- **Text display or edit box:** Uses the default format (text).
- **List box:** Uses the default format (text).
- **Button:** Usesnormalizedvalue0.0and1.0.

Commands

Set Value

Sets one or more parameter values using paths to each element being set. The path points to a parameter. Setting attributes from the client is not supported.

<value> is a JSON Value, so it can be true, false, null, JSON String, or JSON Number.

Set one parameter with no format string

set { “path” : “<parameter path>”, “value” : <value> }

@set { “path” : “<parameter path>”, “value” : <value> }

or

@set { “path” : “<parameter path>”, “error” : “<error message>” }

The value is echoed back in the set reply because it may have changed, rounded, or been bounded by min/max.

Example:

set { “path” : “/channel 1/gain”, “value” : 0.5 }

@set { “path” : “/channel 1/gain”, “value” : 0.5 }

Set one parameter with a format string

The format specifier is always a string.

set { “path” : “<parameter path>”, “value” : <value>, “format” : “<format string>” }

@set { “path” : “<parameter path>”, “value” : <value>, “format” : “<format string>” }

or

@set { “path” : “<parameter path>”, “error” : “<error message>” }

Example:

set { “path” : “/channel 1/gain”, “value” : 0.5, “format” : “n” }

@set { “path” : “/channel 1/gain”, “value” : 0.5, “format” : “n” }

Set multiple parameters with no format string

In the case of multiple parameters, a JSON array is used. If an error happened with one of the parameters, an error message is returned.

set [{ “path” : “<parameter path>”, “value” : <value> },

{ “path” : “<parameter path>”, “value” : <value> }]

@set [{ “path” : “<parameter path>”, “value” : <value> },

{ “path” : “<parameter path>”, “value” : <value> }]

or

@set [{ “path” : “<parameter path>”, “error” : “<message>” },

{ “path” : “<parameter path>”, “value” : <value> }]

The “error” keyword tag is used to identify which parameters could not be set.

Example:

set [{ “path” : “/channel 1/gain”, “value” : 0.5 },

{ “path” : “/channel 1/mute”, “value” : 0 }]

@set [{ “path” : “/channel 1/gain”, “value” : 0.5 },

{ “path” : “/channel 1/mute”, “value” : 0 }]

Set multiple parameters with format strings

set [{ “path” : “<parameter path>”, “value” : <value>, “format” : “<format string>” },

{ “path” : “<parameter path>”, “value” : <value>, “format” : “<format string>” }]

@set [{ “path” : “<parameter path>”, “value” : <value>, “format” : “<format string>” },

{ “path” : “<parameter path>”, “value” : <value>, “format” : “<format string>” }]

or

@set [{ “path” : “<parameter path>”, “error” : “<message>” },

{ “path” : “<parameter path>”, “value” : <value>, “format” : “<format string>” }]

Set multiple parameters within same object (optimized)

Setting multiple parameters in the same object is an optimized version of ‘set multiple parameters’. It does not support format strings and is set using the data type of the parameter.

In the case of multiple parameters, the <path> points to the parent object instead of all the way to the parameter name. Then the parameter names appear inside the “value” object as keyword tags.

set { “path” : “<object path>”, “value” : { “<param1>” : <value>, “<param2>” : <value> } }

@set { “path” : “<object path>”, “value” : { “<param1>” : <value>, “<param2>” : <value> } }

or

@set { “path” : “<path>”, “value” : { “<param1>” : <value>, “<param2>” : { “error” : “<message>” } } }

The “error” keyword tag is used to identify which parameters could not be set.

Example:

set { “path” : “/channel 1”, “value” : { “gain” : 0.5, “mute” : 0 } }

@set { “path” : “/channel 1”, “value” : { “gain” : 0.5, “mute” : 0 } }

Error-handling

The Set message wants to return the values that the parameters were set to, in case they were bounded by min/max or rounded off. In case of errors, the Set message wants to return an error message associated with each parameter that had an error (could not be set).

Attempt to set a parameter but it did not exist:

set { “path” : “/channel 1/gain”, “value” : <value> }

@set { “path” : “/channel 1/gain”, “value” : { “error” : “Parameter not found” } }

Attempt to set multiple parameters on multiple objects using format strings but one of the parameters did not exist.

```
set [ { “path” : “/channel 1/gain”, “value” : 0.5, “format” : “n” },
    { “path” : “/channel 2/mute”, “value” : false, “format” : “bool” } ]
@set [ { “path” : “/channel 1/gain”, “value” : 0.5, “format” : “n” },
    { “path” : “/channel 2/mute”, “error” : “Parameter not found” } ]
```

Errors can be returned for just the parameters that failed, and values for the parameters that worked. The <error message> is a JSON String.

Get Value

Get is exactly like Set, except the Get Command won't have values, but the Get Reply will. Therefore, the Get Reply will look exactly like the Set Reply.

Get one parameter

get { "path" : "<parameter path>" }

@get { "path" : "<parameter path>", "value" : <value> }

or

@get { "path" : "<parameter path>", "error" : "<error message>" }

Example:

get { "path" : "/channel 1/gain" }

@get { "path" : "/channel 1/gain", "value" : 0.5 }

Get one parameter with a format string

get { "path" : "<path>", "format" : "<format string>" }

@get { "path" : "<parameter path>", "format" : "<format string>", "value" : <value> }

or

@get { "path" : "<parameter path>", "error" : "<error message>" }

Example:

get { "path" : "/channel 1/gain", "format" : "n" }

@get { "path" : "/channel 1/gain", "format" : "n", "value" : 0.5 }

Get multiple parameters

In the case of multiple parameters, a JSON array is used. If an error happened with one of the parameters, an error message is returned.

get [{ "path" : "<parameter path>" }, { "path" : "<parameter path>" }]

```
@get [ { "path" : "<parameter path>", "value" : <value> },
       { "path" : "<parameter path>", "value" : <value> } ]
```

or

@get [{ “path” : “<parameter path>”, “error” : “<message>” },

      { “path” : “<parameter path>”, “value” : <value> }]

The “error” keyword tag is used to identify which parameters could not be set.

Example:

get [{ “path” : “/channel 1/gain” }, { “path” : “/channel 1/mute” }]

@get [{ “path” : “/channel 1/gain”, “value” : 0.5 },

      { “path” : “/channel 1/mute”, “value” : 0 }]

Get multiple parameters with format strings

This example would pick off two parameters from possibly different objects:

get [{ “path” : “<path>”, “format” : “<format string>” },

    { “path” : “<path>”, “format” : “<format string>” }]

@get [{ “path” : “<path>”, “format” : “<format string>”, “value” : <value> },

     { “path” : “<path>”, “format” : “<format string>”, “value” : <value> }]

Example:

get [{ “path” : “/channel 1/gain”, “format” : “n” },

    { “path” : “/channel master/mute”, “format” : “bool” }]

@get [{ “path” : “/channel 1/gain”, “format” : “n”, “value” : 0.5 },

    { “path” : “/channel master/mute”, “format” : “n”, “value” : false }]

Get multiple parameters for an object; get an attribute

If the path points to an object name instead of a parameter, then all the parameters that live directly under that object will be returned in the Get Reply. The “format” string is not used in an “object get.” The parameters will come back in their default formats. In the following example, an object is queried, and an attribute is also queried.

get [{ “path” : “<object path>”, },

    { “path” : “<attribute path>” }]

@get [{ “path” : “<parameter path>”, “value” : <value> },

    { “path” : “<parameter path>”, “value” : <value> },

    { “path” : “<attribute path>”, “value” : <value> }]

Example:

get [{ “path” : “/channel 1” },

 { “path” : “/channel 1/gain/min” },

 { “path” : “/channel 1/gain/max” }]

@get [{ “path” : “/channel 1/gain”, “value” : 0.5 },

 { “path” : “/channel 1/mute”, “value” : 0 },

 { “path” : “/channel 1/gain/min”, “value” : 0.0 },

 { “path” : “/channel 1/gain/max”, “value” : 1.0 }]

So in the above example, the “/channel 1” object contained two parameters, both of which were returned. The two attributes of the gain parameter were specifically asked for and returned.

An attribute may be an array of values. An example of this is the “flags” attribute on an SV. This would be represented as a JSON Array.

Example:

get [{ “path” : “/channel 1/gain/flags” },

 { “path” : “/channel 1/meter/flags” }]

@get [{“path”:”/channel 1/gain”,”value”:{“flags”:[“nvo”]}},{“path”:”/channel 1/
meter”,”value”:{“flags”:[“ro”,”ns

ub”]}}]

Error-handling

Since the Get Reply is identical to Set Reply, the error-handling is identical as well. So for the above Get Command, if there were errors, the error reply might look like this:

@get [{ “path” : “<parameter path>”, “error” : “<error message>” },

 { “path” : “<parameter path>”, “error” : “<error message>” },

 { “path” : “<attribute path>”, “error” : “<error message>” }]

Subscriptions

As well as using the set messages to control the parameter, you will also want to subscribe to listen for changes that come from other sources (i.e. from other controllers or internally from the device itself). All subscription messages response with an @subscription that contain the current values. Once subscribed, parameter changes will be sent to the client using ‘publish’ messages.

NOTE: Publish, @set, @get and @subscribe messages all use the same JSON format and for the purposes of the GUI can be treated in a very similar fashion. The main difference is that publish messages MUST be acknowledged with an @publish response from the GUI before another publish will be sent. Failure to respond to a publish message will mean that publish messages would stop.

To read more about the JSON format for publish, @set, @get and @subscribe messages, see “Understanding the Parameter Value Format” on page 15.

Subscribe to a Parameter Directly

Subscribe to a parameter directly with different formats:

Default

subscribe {“path”:”/Config/Channel4PEQ/Band_1_Frequency”}

or

subscribe {“path”:”/Config/Channel4PEQ/Band_1_Frequency”, “format”:”Default”}

@subscribe {“path”:”/Config/Channel4PEQ”,”value”:{“Band_1_Frequency”:”100Hz”}}

Norm

subscribe {“path”:”/Config/Channel4PEQ/Band_1_Frequency”, “format”:”Norm”}

@subscribe {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Frequency”,”value”:”0.229819”}

Number

subscribe {“path”:”/Config/Channel4PEQ/Band_1_Frequency”, “format”:”Number”}

@subscribe {“format”:”Number”,”path”:”/Config/Channel4PEQ/Band_1_Frequency”,”value”:”100”}

If you are subscribed to the same parameter using multiple formats then each parameter change will result in a single publish message containing all subscribed formats. For example, if you subscribe to the frequency parameter like so:

subscribe {“path”:”/Config/Channel4PEQ/Band_1_Frequency”, “format”:”Default”}

@subscribe {“path”:”/Config/Channel4PEQ”,”value”:{“Band_1_Frequency”:”100Hz”}}

subscribe {“path”:”/Config/Channel4PEQ/Band_1_Frequency”, “format”:”Norm”}

@subscribe {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Frequency”,”value”:”0.229819”}

You are now subscibed to the /Config/Channel4PEQ/Band_1_Frequency parameter for Default and Norm formats. A parameter change would result in the following publish message:

publish [{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Frequency”,”value”:”0.558616”}, {“path”:”/Config/Channel4PEQ”,”value”:{“Band_1_Frequency”:”1kHz”}}]

Subscribe to All Parameters in an Object

It is possible to subscribe to all parameters in an object. This subscription is recursive. Therefore, all parameters in all child objects will also be subscribed.

subscribe {“path”:”/Config/Channel4PEQ”, “format”:”Norm”}

And the model responds with the current values of all parameter in the Channel4PEQ object:

@subscribe [{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Frequency”,”value”:”0.558616”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Gain”,”value”:”0.5”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Q”,”value”:”0.007037”}, {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Slope”,”value”:”0”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Type”,”value”:”0”}, {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_2_Frequency”,”value”:”0.558616”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_2_Gain”,”value”:”0.5”}, {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_2_Q”,”value”:”0.007037”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_2_Slope”,”value”:”0”}, {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_2_Type”,”value”:”0”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_3_Frequency”,”value”:”0.558616”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_3_Gain”,”value”:”0.5”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_3_Q”,”value”:”0.007037”}, {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_3_Slope”,”value”:”0”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_3_Type”,”value”:”0”}, {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_4_Frequency”,”value”:”0.558616”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_4_Gain”,”value”:”0.5”}, {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_4_Q”,”value”:”0.007037”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_4_Slope”,”value”:”0”}, {“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_4_Type”,”value”:”0”},

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Flatten”,”value”:”0”},{“format”:”Norm”,”path”:”/Config/Channel4PEQ/ParametricEQ”,”value”:”0”}]

Or subscribing to the Default format:

subscribe {“path”:”/Config/Channel4PEQ”, “format”:”Default”}

And the model responds with the current values of all parameter in the Channel4PEQ object using the Default format:

@subscribe {“path”:”/Config/Channel4PEQ”,”value”:{“Band_1_Frequency”:”1kHz”,”Band_1_Gain”:”0dB”,”Band_1_Q”:”1”,”Band_1_Slope”:”3”,”Band_1_Type”:”Bell”,”Band_2_Frequency”:”1kHz”,”Band_2_Gain”:”0dB”,”Band_2_Q”:”1”,”Band_2_Slope”:”3”,”Band_2_Type”:”Bell”,”Band_3_Frequency”:”1kHz”,”Band_3_Gain”:”0dB”,”Band_3_Q”:”1”,”Band_3_Slope”:”3”,”Band_3_Type”:”Bell”,”Band_4_Frequency”:”1kHz”,”Band_4_Gain”:”0dB”,”Band_4_Q“:”1”,”Band_4_Slope”:”3”,”Band_4_Type”:”Bell”,”Flatten”:”Restore”,”ParametricEQ”:”Off”}}

It is even possible to subscribe to all parameters in all objects under Config. Since the subscribe is recursive this means that ALL parameters in all child objects are subscribed to. Note that when subscribing to all parameters under Config you will get a very large @subscribe message back since it contains values for ALL parameters in all objects.

subscribe {“path”:”/Config”, “format”:”Norm”}

Or

subscribe {“path”:”/Config”, “format”:”Default”}

The responses to these messages are far too large to show here.

Understanding the Parameter Value Format

It is important to understand the JSON format used to return parameter values in publish, @set, @get and @subscribe messages. There are various ways in which the DCP-555 server may decide to format the JSON.

Individual values can be formatted in a JSON object like so:

{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Frequency”,”value”:”0.558616”}

Or if the format is ‘Default’ then the format is left off and one or more values can be represented like so:

{“path”:”/Config/Channel4PEQ”,”value”:{“Band_1_Frequency”:”100Hz”}}

In the above format, the path points to the owning object and then the value contains pairs of parameter:value.

When decoding this you should look at the value type and, if it is a JSON object, then you know every key/value pair inside is a parameter/value pair. In this case you must concatenate the path and the parameter name to get the full path to the parameter.

There may be more than one parameter inside the value object:

{“path”:”/Config/Channel4PEQ”,”value”:{“Band_1_Frequency”:”1kHz”,”Band_1_Gain”:”0.1dB”,” Band_1_Q”:”1”,”Band_1_Slope”:”3”,”Band_1_Type”:”Bell”,”Band_2_Frequency”:”1kHz”,”Band_2_ Gain”:”0dB”,”Band_2_Q”:”1”,”Band_2_Slope”:”3”,”Band_2_Type”:”Bell”,”Band_3_ Frequency”:”1kHz”,”Band_3_Gain”:”0dB”,”Band_3_Q”:”1”,”Band_3_Slope”:”3”,”Band_3_ Type”:”Bell”,”Band_4_Frequency”:”1kHz”,”Band_4_Gain”:”0dB”,”Band_4_Q”:”1”,”Band_4_ Slope”:”3”,”Band_4_Type”:”Bell”,”Flatten”:”Restore”,”ParametricEQ”:”Off”}}

Finally, multiple objects (described above) may be sent in a JSON array:

[{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_ Frequency”,”value”:”0.558616”},{“format”:”Norm”,”path”:”/Config/Channel4PEQ/ Band_1_Gain”,”value”:”0.5025”},{“format”:”Norm”,”path”:”/Config/Channel4PEQ/ Band_1_Q”,”value”:”0.007037”},...,{“format”:”Norm”,”path”:”/Config/Channel4PEQ/ ParametricEQ”,”value”:”0”}]

Including a mixture of formats:

[{“format”:”Norm”,”path”:”/Config/Channel4PEQ/Band_1_Frequency”,”value”:”0.558616”}, {“path”:”/ Config/Channel4PEQ”,”value”:{“Band_1_Frequency”:”1kHz”}}]

Metering

All meters are read using either a get message or a subscribepoll message. Subscribepoll is an optimization to avoid sending long “get” and “@get” messages but for simplicity get is recommended. For example:

get [{“path”:”/Config/PeakMeters/In1InputLevel”, “format”:”Number”}, {“path”:”/Config/PeakMeters/

In2InputLevel”, “format”:”Number”}, {“path”:”/Config/PeakMeters/In3InputLevel”,
“format”:”Number”}]

And the model responds with the current values:

@get [{“format”:”Number”,”path”:”/Config/PeakMeters/In1InputLevel”,”value”:”-120”},{“format”:”Number”,”path”:”/Config/PeakMeters/In2InputLevel”,”value”:”-120”},{“format”:”Number”,”path”:”/Config/PeakMeters/In3InputLevel”,”value”:”-120”}]

Although get messages and their @get responses can be quite long, they have the advantage of using the same JSON format as the @set, @subscribe, and publish messages. Therefore, a single piece of code can decode all those messages. Another advantage is that different formats can be requested in the same message. For example:

get [{“path”:”/Config/PeakMeters/In1InputLevel”, “format”:”Number”}, {“path”:”/Config/PeakMeters/

In1InputLevel”, “format”:”Default”}, {“path”:”/Config/PeakMeters/In2InputLevel”,
“format”:”Number”},

{“path”:”/Config/PeakMeters/In2InputLevel”, “format”:”Default”}, {“path”:”/Config/PeakMeters/In3InputLevel”,“format”:”Number”}, {“path”:”/Config/PeakMeters/In3InputLevel”, “format”:”Default”}]

And the model responds with both Number and Default formats:

@get [{“format”:”Number”,”path”:”/Config/PeakMeters/In1InputLevel”,”value”:”-120”},{“format”:”Number”,”path”:”/Config/PeakMeters/In2InputLevel”,”value”:”-120”},{“format”:”Number”,”path”:”/Config/PeakMeters/In3InputLevel”,”value”:”-120”},{“path”:”/Config/PeakMeters”,”value”:{“In1InputLevel”:”-120dB”,”In2InputLevel”:”-120dB”,”In3InputLevel”:”-120dB”}}]

Subscribepoll allows the GUI to specify a list of meter parameters that will be read together. This list is given a name and then polled collectively using the ‘poll’ message. The advantage with the subscribepoll is that it avoids having to specify the paths of all the parameters you wish to read in every meter read.

For example, a subscribepoll is initially set up like so:

subscribepoll { “name”:”PeakMetersDefault”, “targets”:[“/Config/PeakMeters/In1InputLevel”, “/Config/PeakMeters/In2InputLevel”, “/Config/PeakMeters/In3InputLevel”], “format” : “Default” }

@subscribepoll

Now all 3 meters can be read together:

poll { “name” : “PeakMetersDefault” }

@poll {“name”:”PeakMetersDefault”,”targets”:[“-120dB”,”-120dB”,”-120dB”]}

Or to specify a format that should be used for the values:

subscribepoll { “name”:”PeakMetersNumber”, “targets”:[“/Config/PeakMeters/In1InputLevel”, “/Config/PeakMeters/In2InputLevel”, “/Config/PeakMeters/In3InputLevel”], “format” : “Number” }

@subscribepoll

poll { “name” : “PeakMetersNumber” }

@poll {“name”:”PeakMetersNumber”,”targets”:[“-120”,”-120”,”-120”]}

The advantage of the subscribepoll messages are the small size and simplicity of the poll and @poll response. This could make parsing much faster on small devices. However, they do require the GUI to maintain information about the collections of parameters that it has subscribed to so that it can decode the responses.

Phone: (801) 566-8800

Website: bssaudio.com

Support: bssaudio.com/en-US/support

LAST REVISED: 10/22/2018