Jellyfish Lighting API Documentation

Purpose:

The purpose of this document is to provide information regarding the Jellyfish Lighting API, and how to interact with the Jellyfish Lighting System.

Summary:

The Jellyfish Lighting API is a system in which JSON requests and responses can be sent and received over a Local Area Network via web sockets to manipulate the performance of the Jellyfish Lighting System. This includes commands regarding patterns, running patterns, minor adjustments to patterns, powering on and off the system and more.

Commands:

The Json Commands to operate the system consist of 2 main components

1) The Command direction is represented with the key: “cmd”, and a matching value of “toCtlrGet”, “toCtlrSet”, and “fromCtlr”. This distinguishes the direction in which this command is intended to go. “toCtlrGet”, and “toCtlrSet” imply that the command is being sent from the client to the controller, whereas “fromCtlr” implies that this a json response returning from the controller to the client.

2) The second component is the command itself. These commands are categorized into a few different categories whose keys include “get”, “runPattern”, and others. The values associated with these requests will vary greatly depending on what the desired behavior is. Most “get” commands will have the requested resource name nested inside two sets of brackets e.g: [[“patternFileList”]]. Other commands will contain more complicated key values and will be documented in greater detail in the table below.

| Command | Json Request | Json Response |
|---|---|---|
| Get Pattern List | {"cmd": "toCtlrGet", "get": [["patternFileList"]]} | {"cmd":"fromCtlr","patternFileList":[{"folders":"<folderName>","name":"<patternName>","readOnly":<readOnlyValue>}}]} |
| Get Zone List | {"cmd": "toCtlrGet", "get": [["zones"]]} | {"cmd":"fromCtlr","save":true,"zones":{"Zone":{"numPixels":<int>,"portMap":[{"ctlrName":"JellyFish-<number>.local","phyEndIdx":<int>,"phyPort":<portNum>,"phyStartIdx":<StartIndex>,"zoneRGBStartIdx":<int>}]}}} |

| | | |
|---|---|---|
| Run Given Pattern | {"cmd":"toCtlrSet","runPattern":{"file":"<folderName>/<patternName>","data":"","id":"","state":1,"zoneName":[]}} | {"cmd":"fromCtlr","runPattern":{"file":"<folderName>/<patternName>","data":"","id":"","state":1,"zoneName":[<zoneName>, <zoneName>]}} |
| Get Pattern File Data | {"cmd":"toCtlrGet", "get":[["patternFileData", "<folder>", "<patternName>"]] } | {"cmd":"toCtlrSet","runPattern":{"file":"","data":"{\"colors\":[255,0,0,0,255,0,0,0,255],\"spaceBetweenPixels\":14,\"effectBetweenPixels\":\"Progression\",\"ledOnPos\":{\"0\":[],\"1\":[0]},\"type\":\"Chase\",\"skip\":2,\"numOfLeds\":1,\"direction\":\"Center\",\"runData\":{\"speed\":75,\"brightness\":78,\"effect\":\"NoEffect\",\"effectValue\":0,\"rgbAdj\":[100,100,100]}}","id":"","state":1,"zoneName":["Zone","Zone1","Zone2","Zone3"]}} |

Examples

Provided below is a set of working examples to demonstrate the flow of commands between a client, and the controller.

Example 1: Get PatternList, get zone list, run specified pattern

Client: **Establishes Web Socket Connection with Controller**

Client:

{"cmd": "toCtlrGet", "get": [["patternFileList"]]}

Controller:

{"cmd":"fromCtlr","patternFileList":[{"folders":"Christmas","name":"","readOnly":false},{"folders":"Christmas","name":"Christmas Tree","readOnly":true},{"folders":"Christmas","name":"Red Yellow Green Blue","readOnly":true},{"folders":"Colors","name":"","readOnly":false},{"folders":"Colors","name":"Orange","readOnly":true},{"folders":"Colors","name":"Pink","readOnly":true}]}

- It is important to note that the first element of each folder, specifies only the folder name, and does not have a pattern associated with it. In the above response example you will see that the first element in the

patternFileList is: {"folders":"Christmas","name":"","readOnly":false}, notice that the “name” field is empty, which signifies that this is beginning of the Christmas patterns folder. You can see this again with the first element with the “folders” value of “Colors”.

- Client is responsible for storing the responses, and generating catalog of the Pattern List

Client:

{"cmd": "toCtlrGet", "get": [["zones"]]}

Controller:

{"cmd":"fromCtlr","save":true,"zones":{"Zone":{"numPixels":35,"portMap":[{"ctlrName":"JellyFish-3074.local","phyEndIdx":34,"phyPort":1,"phyStartIdx":0,"zoneRGBStartIdx":0}]},"Zone1":{"numPixels":40,"portMap":[{"ctlrName":"JellyFish-3074.local","phyEndIdx":74,"phyPort":1,"phyStartIdx":35,"zoneRGBStartIdx":0}]},"Zone2":{"numPixels":35,"portMap":[{"ctlrName":"JellyFish-3074.local","phyEndIdx":34,"phyPort":2,"phyStartIdx":0,"zoneRGBStartIdx":0}]},"Zone3":{"numPixels":40,"portMap":[{"ctlrName":"JellyFish-3074.local","phyEndIdx":74,"phyPort":2,"phyStartIdx":35,"zoneRGBStartIdx":0}]}}}

- This response returns all the available existing zone data. In the “zones” map, all of the keys represent the name of a given zone of lights and the values represent additional data regarding each zone, such as the number of pixels.
- In the context of this application, and running patterns on zones, the zone name is most applicable and important.

Client:

- The must select which pattern, as well as which zones to run this pattern on, and configure this data into the following request.

{"cmd":"toCtlrSet","runPattern":{"file":"Christmas/Christmas Tree","data":"","id":"","state":1,"zoneName":["Zone", "Zone1"]}}

- Using the “runPattern” command we must include in its values “file” which is the file path to the desired pattern, the “state” of the lights (1: on, or 0: off), and the “zoneName” list, which is the list of zones we wish to turn the lights on. In the accompanying application, when no zones are selected, the pattern is run on all zones.

Example 2: Making Adjustments to a Running Pattern

- Get pattern list (as documented above)
- Get zone list (as documented above)
- Note it may be ideal to cache a selected pattern and zones as doing so will make adjusting said pattern much easier.

Client:

{"cmd":"toCtlrGet", "get":[["patternFileData", "Legacy", "Red Yellow Green Blue"]]}

- This command uses the “get” command, followed by a nested array of parameters. The first “patternFileData”, tells the controller that we are requesting pattern file data, and the subsequent two arguments “Legacy”, and “63-Red White Blue", tell the controller the folder, and file whose data we desire.

Controller:

{"cmd":"fromCtlr","patternFileData":{"folders":"Legacy","jsonData":"{ \"colors\": 255,0,0,255,255,255,  0,0,255], \"spaceBetweenPixels\": 10, \"effectBetweenPixels\": \"No Color Transform\", \"type\": \"Multi-Paint\",  \"skip\": 1, \"numOfLeds\": 6, \"runData\": {\"speed\": 5,  \"brightness\": 100, \"effect\": \"No Effect\", \"effectValue\": 0, \"rgbAdj\": [ 100, 100, 100 }, \"direction\": \"Center\"}","name":"63-Red White Blue"}}

- The controller will respond with all the pattern data, including RGB color values, speed, brightness, and a host of other attributes. These are the values that will manipulated and changed when striving to adjust light patterns.

Client:

- Now to make adjustments to the running pattern we will use the patternFileData, and adjust specific values. When making adjustments ensure that no values are missing, even if they are not going to be adjusted, This will ensure that the controller can properly parse the request.

{"cmd":"toCtlrSet","runPattern":{"file":"","data":"{\"colors\":[255,0,0,255,255,255,0,0,255],\"spaceBetweenPixels\":10,\"effectBetweenPixels\":\"No Color Transform\",\"type\":\"Multi-Paint\",\"skip\":1,\"numOfLeds\":6,\"runData\":{\"speed\":15,\"brightness\":100,\"effect\":\"No Effect\",\"effectValue\":0,\"rgbAdj\":[100,100,100]},\"direction\":\"Center\"}","id":"","state":1,"zoneName":["Zone","Zone1"]}}

- In this command we have adjusted the brightness and speed of the pattern.
- When adjusting the speed it is important to understand that the speed value (x) will be parsed to a speed value as such 1/x, thus a speed value of 1 equates to 1/1, which is the fastest value for speed, where as a speed value of 100 equates to 1/100 which is incredibly slow.

Command Parameters and Data Fields

As seen above some commands contain many parameters. These parameters serve as instructions to the controller. This is most apparent when making adjustments to a given pattern by manipulating some of the existing parameters or data fields, but the same holds true for all commands. Below each command and its parameters will be explained in greater detail.

Get Pattern File List:

Command: {"cmd": "toCtlrGet", "get": [["patternFileList"]]}

- The get pattern file list command, along with other get commands is quite simple, requiring a single parameter, The resource which is being requested “patternFileList”. There are no additional Data Fields or parameters required or supported at this time for this command.

Get Zones:

Command: {"cmd": "toCtlrGet", "get": [["zones"]]}

- The get zones command, similar to the get pattern file list command, is quite simply just an extension of the “get” command. This command is simply requesting all the zone data available, and thus its only parameter

is the requested resources “zones”. There are no additional Data Fields or parameters required or supported at this time for this command.

Run Pattern (basic):

Command:

{"cmd":"toCtlrSet","runPattern":{"file":"<folder>/<fileName>","data":"<NOT USED>","id":"<NOT USED","state":<INT>,"zoneName":[<zone>, <zone>, <zone>]}}

- The run pattern command as displayed above is one of the first commands we see with data fields that require input. Based on the inputs provided the run pattern command is quite powerful in manipulating the performance of the Jellyfish Lighting System. Above is the run pattern basic command, later the more robust run pattern command will be additionally documented.

Fields:

- file: the file field is used to tell the controller which preset pattern to run based on its existing file data. In the command above you see it is split into a <folder> and <file> section separated by a “/”, just as one might assume this format is essential for the controller to navigate its internal file structure to find the requested pattern file.
  Example: … “runPattern”: {“file”: “Christmas/Christmas Tree”, …
  Example: … “runPattern”: {“file”: “Halloween/Spoooky”, …

- data: this field will be more applicable in the more verbose run pattern command.

- id: this field is not used.

- state: This field simply tells the controller what the state of the lights should be using an integer. The integer 1 for on, 0 for off. Note that this state field will only manipulate the state of lights in selected zones which will be documented later.
  Example: …”state”: 0, …
  Example: …”state”: 1, ...

- zoneName: The zone field consists of a list of the zones on which the pattern is to be run. To run lights on all zones, each zone name must be inside the zoneName list, this is applicable to any combination of zones.
  Example: … “zoneName”: [“zone1”, “zone2”, “zone3”, “zone4”] …
  Example: … “zoneName”: [“zone1”, “zone3”] …

Run Pattern (advanced):

Command:

{"cmd":"toCtlrSet","runPattern":{"file":"","data":"{\"colors\":[<int>,<int>,<int>],\"spaceBetweenPixels\":<int>,\"effectBetweenPixels\":<effect>,\"type\":<type>,\"skip\":<int>,\"numOfLeds\":<int>,\"runData\":{\"speed\":<int>,\"brightness\":<int>,\"effect\":<effect>,\"effectValue\":<int>,\"rgbAdj\":[<int>,<int>,<int>]},\"direction\":<direction>}","id":"","state":<int>,"zoneName":[<zone>,<zone>]}}

- The advanced run pattern command contains many data fields that can be changed to customize the behavior of the lights. When wanting to change the behavior of a specific pre-existing pattern, it will be quite useful to run the get pattern file data command (documented below), and store its response values, as these values will be useful when making adjustments. It is important that the request is formatted in the correct order and follows the conventions above. If said convention is broken, the controller will not be able to parse the request.

Fields:

- file: Note that in the advanced run pattern request no file is given, this is because all the instructions required to run the pattern are included, and thus a file is not required to be referenced.
- data: the “data” field is a map with several important fields inside. Data refers to the data the controller needs to operate the lights.
    - colors: this field found inside the data map is a list of RGB color values (integer between 0 and 255) telling the controller which colors it is to use for the pattern. Create a color there must be 3 values, thus the length of the colors list must be a multiple of 3, or else the colors will not be parsed correctly. Limit of 30 colors (90 values).
      Example: … \”colors\”: [0,255,114,255,255,255], … (2 colors)
      Example: … \”colors\”: [105,255,114], … (1 color)

    - spaceBetweenPixels: this field is specified as an integer and is dependent on the “effectBetweenPixels” field. This essentially will set the length of each patternEffect when applicable.
      Example: … \”spaceBetweenPixels\”: 4, …
      Example: … \”spaceBetweenPixels\”: 2, ...

    - effectBetweenPixels: this field sets the effect that the pattern will display on the lights. The effectBetweenPixel values are as follows: “No Color Transformation”, “Repeat”, “Progression”, “Fade”, “Fill with Black”.

Example: …\”effectBetweenPixels\”: \”Repeat\”, …
Example: …\”effectBetweenPixels\”: \”Progression\”, ...

- type: type is the field which describes how the differing colors interact with each other. The values include: “Color”, “Chase”, “Paint”, “Stacker”, “Sequence”, and “Multi-Paint”.
  Example: … \”type\”: \”Paint\”, …
  Example: … \”type\”: \”Chase\”, ...

- skip: the skip value is an integer that depending on the type, either acts as a speed enhancement, or changes the number of lights involved in each iteration of the pattern type.
  Example: ...\”skip\”: 1, …
  Example: ...\”skip\”: 20, ...

- numOfLeds: numOfLeds is similar to skip, but is only applicable when using the Multi-Paint type.
  Example: ...\”numOfLeds\”: 1, …
  Example: ...\”numOfLeds\”: 20, ...

- runData: the runData field is a map with values, similar to the data field. Its values are documented below.

    - speed: the speed field is an int which is used to calculate the speed of the lights. The speed is calculated with 1 being the fastest speed, and increased integers running the lights slower.
      Example: … \”speed\”: 15, …
      Example: … \”speed\”: 1, ...

    - brightness: the brightness field is an integer which tells the controller how bright to display the lights. This integer is interpreted as a percentage with a limit of 100. Thus a brightness value of 25 is 25% brightness.
      Example: … \”brightness”\: 100, ...

    - effect: Not used, and value should be set to \”No Effect\”

    - effectValue: Not used and should be set to 0

- rgbAdj: Not used and should be set to the value: [100, 100, 100]

- direction: the direction value is used to tell the controller which direction the pattern is to run. Its possible values are \"Right\", and \"Left\"
  Example: … \"direction\": \"Left\", …
  Example: … \"direction\": \"Right\", ...

- id: Not used, and should be left empty.

- state: This field is documented above in the run pattern (basic) documentation.

- zoneName: This field is documented above in the run pattern (basic) documentation.

Get Pattern File Data:

Command: {"cmd":"toCtlrGet", "get":[["patternFileData", "<folderName>", "<fileName>"]] }

- This command is quite simple, but very important. This command is what allows the client to access the pattern file data stored on the controller. This pattern file data will give you the data needed to modify specific fields in pattern. It is beneficial to cache this data as it will make it much easier to configure any pattern adjustment requests.

Parameters:

- The first parameter is the requested resource, the “patternFileData”.

- The second parameter is the folderName which is the folder in which the file is stored

- The last parameter is the the file name.

Example: {"cmd":"toCtlrGet", "get":[["patternFileData", "Christmas", "Mint"]] }

Conclusion:

The Jellyfish Lighting System, and its robust API allows great functionality and customization when used properly. The above documentation will provide you with a good baseline knowledge of how the system works, how to configure requests, and how to effectively interact with the system. The accompanying API Explorer application can be a great resource to help you see how requests should be structured, as well as the workflow of such requests. In addition to the provided resources, using a web socket tester can be very helpful.