HCA Tech Note

HCA Support for JSON decoding

JSON is increasingly used as the format of data sent to and returned from many services. These functions are available in HCA 14 and later versions to facilitate decoding and extracting data.

If you are not familiar with the JSON format here is a helpful introduction.

https://en.wikipedia.org/wiki/JSON

When working with JSON in HCA you first pass the JSON text into a HCA function to parse it and you receive back a "handle". A handle is the term for a value that you shouldn't look into or care what the actual value is – just capture it and pass it into subsequent functions.

After you have the JSON open then there are several functions to retrieve the value of a key based upon a path to the key or to move from node to node.

When complete you must remember to close the JSON using the handle or the memory allocated to it will not be released.

Open and Closing JSON

You can have up to 16 JSON objects open at one time shared between all programs. You are responsible for opening and closing. If you forget to close, then you will run out of available JSON objects.

To open a JSON object, use this function:

Handle = _JSONOpen(text)

The JSON you are passing in may not be of a form that HCA can process as HCA doesn't implement the full JSON specification only the more common parts. You should check to see if HCA accepted or rejected it by checking for the result equal to zero. Any other value except zero is a valid handle value.

When complete with the JSON object then you must close it to free the memory and make its slot available.

Void = _JSONClose(handle)

The handle variable should be a local variable if at all possible.

String parse


```
{
   "1":{
      "name":"office",
      "lights":[
         "4",
         "5"
      ],
      "type":"Room",
      "state":{
         "all_on":false,
         "any_on":true
      },
      "class":"Office",
      "action":{
         "on":false,
         "bri":254,
         "hue":8418,
         "sat":140,
         "effect":"none",
         "xy":[
            0.4573,
            0.4100
         ],
         "ct":366,
         "alert":"none",
         "colormode":"ct"
      }
   },
   "2":{
      "name":"Test Area",
      "lights":[
         "6"
      ],
      "type":"Room",
      "state":{
         "all_on":false,
         "any_on":false
      },
      "class":"Bedroom",
      "action":{
         "on":false,
         "bri":254,
         "hue":8418,
         "sat":140,
         "effect":"none",
         "xy":[
            0.4573,
            0.4100
         ],
         "ct":366,
         "alert":"none",
         "colormode":"ct"
      }
   }
}
```


Basic value retrieval

JSON is parsed by HCA into a structure like this:

```
[node] – [node] – [node]
  |                 |
  |                 |--- [node] – [node] – [node]
  |
  | --- [node] – [node]
```

The basic JSON function takes the handle and then one or more arguments that construct a path to the value to be retrieved

Text = _JSON(handle, text, [text], [text], …) up to 10 arguments

This is very useful if you know what the JSON looks like in advance.

Using the example to the left:

Text = _JSON(handle, "1", "type") evaluates to “Room”.

Text = _JSON(handle, "2", "state", "all_on") evaluates to “false”

Each argument in effect goes “down” a level to the node’s “children” to start a new list and then that list is searched for a key with the supplied name.

In addition to using names when constructing a path to the data to be retrieved, at any point in the path you can use numbers instead of names to refer to where you want to go. The number is an index into the list of nodes at that level. This is necessary if there are unnamed sections like arrays. Again, using the JSON to the left, for example:

Text = _JSON(handle, "1", "action", "xy", 0) evaluates to 0.4573

Text = _JSON(handle, "1", "action", "xy", 1) evaluates to 0.4100

The 5th argument in these _JSON functions refers to the array index. Note how in the JSON to the left that there are two values in the “XY” key but unlike other parts of the JSON they aren’t of the form “key” : “value”.

Note: In programmer world things always start at zero. So the 1st array element is zero, the second is 1, etc. Just something to get used to.

Referencing nodes by number is not limited to arrays. Instead of a name you can use a node index – a number - rather than a name for any argument to achieve a result. Here is an example.

Text = _JSON(0, 3, 0) evaluates to false

The first argument - 0 - means on the outermost list look at the 1st node – that would be “1”. Then go down into the children of that node and go to the 4th node – that would be “state”. Then go down into the children of that node and go to the 1st element – that would be “all_on”.

Navigation

HCA also maintains a current location within the parsed result and you can use these operations to move the position

_JSONNext, _JSONPrev, _JSONUp, _JSONDown

These four functions take a single argument which is the JSON handle and returns true if the move worked and false if it didn’t. That is, there is no next from the current node, or no previous from the current node, etc. If the move failed then the position is unchanged.

The JSON function starts looking for items on the path given by its argument list starting at the current location. Right after you open the JSON, the current position is always at the first node of the outermost list and remains there unless you change the current position using one of these four functions.

These functions that change the current location can be useful if you are getting back data like the example where there are many sections of data returned and you don’t know in advance how many there will be. For example, here are three statements and what they evaluate to using the above JSON example:

handle = _JSONOpen (text)

text = _JSON (handle, 0, “class”) evaluates to “Office”.

void = _JSONNext (handle)

text = JSON (handle, 0, “class”) evaluates to “Bedroom”

Since the _JSONNext function moved the current node, the second JSON function returns different data than the previous retrieval.

##end##