

PANDORA PRODUCTS



Pandora Products Manual Minnow Restful Web server

Jim Schimpf

Document Number: PAN-201201001

Revision Number: 1.1

11 February 2012

Pandora Products.
215 Uschak Road
Derry, PA 15627

©2012 Pandora Products. All rights reserved. Pandora product and company names, as well as their respective logos are trademarks or registered trademarks of Pandora Products. All other product names mentioned herein are trademarks or registered trademarks of their respective owners.

Pandora Products.

215 Uschak Road

Derry, PA 15627

Phone: 724.539.1276

Web: <http://pandora.dyn-o-saur.com:8080/cgi-bin/Minnow.cgi>

Email: vze35xda@verizon.net

Pandora Products. has carefully checked the information in this document and believes it to be accurate. However, Pandora Products assumes no responsibility for any inaccuracies that this document may contain. In no event will Pandora Products. be liable for direct, indirect, special, exemplary, incidental, or consequential damages resulting from any defect or omission in this document, even if advised of the possibility of such damages.

In the interest of product development, Pandora Products reserves the right to make improvements to the information in this document and the products that it describes at any time, without notice or obligation.

Document Revision History

Use the following table to track a history of this document's revisions. An entry should be made into this table for each version of the document.

Version	Author	Description	Date
1.0	js	Initial Version	28-Jan-2012
1.1	js	Editorial work	11-Feb-2012

Contents

1	Minnow Web server	2
1.1	Introduction	2
1.2	Demonstrations	2
1.2.1	Building Minnow from source	2
1.2.2	Web server Example	2
1.2.3	Lua Example	3
1.2.4	Web server with REST	4
1.2.5	Debug Operation	5
2	Demonstration Site Operation	7
2.1	Server RESTful operation	7
2.1.1	Server Code	8
2.1.2	Server Summary	11
2.2	SAMPLE REST API	12
2.2.1	ADMIN/TIME	12
2.2.2	ADMIN/POWER	12
2.2.3	ADMIN/LOGIN	13
3	Lua Extension Classes	14
3.1	RESTful Interface CLASS	14
3.1.1	Introduction	14
3.1.2	Class Operation	14
3.1.3	Class Methods - Server Control	14
3.1.3.1	http.start()	14
3.1.3.2	http.stop()	15
3.1.3.3	http.lock()	15
3.1.3.4	http.open()	16
3.1.3.5	http.close()	16
3.1.3.6	http.url(h)	16
3.1.3.7	http.data()	17
3.1.4	JSON Support	17

3.1.4.1	parsers.json()	17
3.1.4.2	json_kv()	18
3.2	Timer CLASS	18
3.2.1	Timer Class Methods	18
3.2.1.1	timer.sleep()	18
3.2.2	Timer Constructors/Destructors	19
3.2.2.1	timer.new()	19
3.2.2.2	timer.delete()	19
3.2.3	Timer Methods	19
3.2.3.1	timer.start()	19
3.2.3.2	timer.done()	19
3.2.3.3	timer.read()	20
3.3	Keyboard Class	20
3.3.1	Class Methods	20
3.3.1.1	kbd.prep()	20
3.3.1.2	kbd.close()	20
3.3.1.3	kbd.getc()	20
3.3.2	Use	21

List of Figures

1	Simple Site	2
2	Initial Screen	4
3	Challenge/Response Screen	5
4	Run Mode	5
5	Debug Page	6
6	Time Command Test	6
7	REST command structure	8

1 Minnow Web server

1.1 Introduction

This software is a Lua[5]interpreter (5.1 version) that has extensions [2]to support serving web pages it is also designed to easily support RESTful [7]interaction. The RESTful interaction is done by calling Lua scripts to do server actions. Included in the package is a sample web site supporting AJAX [6]. The Lua interpreter also has extensions added to support JSON[1]and some other useful server side actions.

The package has source code and an Mac Xcode project for building and a make file for Linux systems.

1.2 Demonstrations

1.2.1 Building Minnow from source

On download of the Minnow source you have to build it (XCode on OS X or makefile in Linux) .The download include demonstration websites to show the capabilities of Minnow. If you are on OS X then double click the X Code project and and built it. You will then find the Minnow executable in build\Debug Intel and can copy it to the Minnow directory for further work.

On Linux cd to the Minnow directory after expanding the ZIP download and then type make. The executable should build and leave it in the Minnow directory.

1.2.2 Web server Example

Start Minnow with the command line (I'm assuming you are in the Minnow directory and the executable is there):

```
./Minnow -c Site_Simple/main.lua -http Site_Simple -port 8081
```

Now you have a simple web server running on your machine on port 8081 so if you type `http://localhost:8081` in your browser you will get:

Minnow Simple Test Page

This is a simple test page showing that Minnow can server up simple pages

[Other Page](#)

[More Pages](#)

Figure 1: Simple Site

You can click on the links and move to other pages and while doing this you will see that Minnow will print a log of page operations on it's stdout. You can then use it to support a regular website with pages, images and links.

1.2.3 Lua Example

Minnow is also built around a Lua interpreter. If you start Minnow with no command line options you will be in the Lua interpreter. The only non-standard thing is the **quit** keyword which will exit the interpreter.

```
552 > ./Minnow
Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio
> a = 10
> b = 20
> print( a,b,a*b)
10 20 200
> quit
-- END RUN --
553 >
```

It will also run any standard Lua script (and use the extensions described below) by starting it with `Minnow -c <script name>`

```
-- Test script
print("-- Minnow Script --")
io.write("Input Count: ")
count = io.read("*l")
for i=0,count
do
print(i,i*i,i*i*i)
end
```

and saving that into a file called test.lua.

Running the script we get:

```
555 > ./Minnow -c test.lua
-- Minnow Script --
Input Count: 5
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
-- End Run --
```


1.2.4 Web server with REST

Included in the demo code for Minnow is a RESTful example where Minnow not only serves up webpages but also accepts and responds to RESTful commands from the browser client.

The simple site is built with a challenge/response login system and a debugger that allows you to test RESTful commands and responses.

Start Minnow with the command line (I'm assuming you are in the Minnow directory and the executable is there):

```
./Minnow -c Site/rest/main.lua -http Site/Site -rest Site/rest -port 8081
```

At this point you have a web server running on your machine (i.e. type **http://localhost:8081** in your browser) . You should get this:



Figure 2: Initial Screen

You can now Login to the system by clicking on the Login control on the Left. This gets you the challenge response screen:

Minnow
Pandora Products

Controls

Operation
- LOGIN -
Enter a Response for the Challenge

Challenge
4FE1

Response

Submit

Figure 3: Challenge/Response Screen

The challenge has been made very simple, just type in the challenge value (**4FE1**) into the Response field and hit Submit and you will get:

Minnow
Pandora Products

Controls
Debug
Logout

Operation
- RUN -

Figure 4: Run Mode

At this point you can click on Debug or Logout. Logout will take you back to the initial screen while debug will let you issue RESTful commands and see the responses.

1.2.5 Debug Operation

If you click the Debug operation you get:



Figure 5: Debug Page

This page allows you to input a RESTful and observe the result. Not only can you use the commands we have supplied in Minnow this page can be useful when you build new commands and want to test them.



Figure 6: Time Command Test

In the example above I used the RESTful command **admin/time** (see 2.2.1 on page 12) and got back the JSON key/value pair with the current time. The POST checkbox allows you to send GET commands (not checked) or POST commands (checked). At the same time the STDOUT of Minnow shows what is happening:

```
Searching for RESPONSE
-- In Login Response handler --
*** DATA RECEIVED ***
HTTP TYPE: 3
1 REST
2 ADMIN
```

```
3 TIME
n 3
2
Searching for ADMIN
-- Admin Actions --
3
Searching for TIME
```

When you are working on developing server side scripts to support your commands any **print()** command you put into the Lua script will appear here and will help with your debugging.

2 Demonstration Site Operation

The operation of the Minnow as a Lua interpreter and simple web server are easily seen from the demonstration. Operation as a RESTful server is much more complicated and involves interaction between client code (HTML/CSS/Javascript/JQuery) and server code (Lua). This will be set out in detail here and will give you enough information to develop your own sites. NOTE: You will need knowledge of Javascript/JQuery and Lua to use Minnow in this way. If you need a tutorial then these websites might help:

- HTML <http://www.w3schools.com/html/default.asp>
- CSS <http://www.w3schools.com/css/default.asp>
- Javascript <http://www.w3schools.com/js/default.asp>
- JQuery http://www.w3schools.com/jquery/jquery_intro.asp
- Lua <http://lua-users.org/wiki/LuaTutorial>

2.1 Server RESTful operation

How did all of the above work and what do you have to write to make Minnow do what you want on your web site ? In this case it is done with a combination of server and client side scripting. The client side code used Javascript and the server scripting is in Lua supported by Minnow. In addition the client side scripting uses JQuery[4] as a support library.

The interaction that occurs is RESTful, in that the client sends in GET or POST requests and the server via the Lua scripts builds responses in JSON. The script in the client then parses the JSON and changes the screen in the appropriate fashion. The full set of RESTful commands is show in the figure.

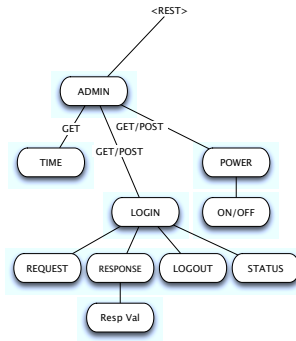


Figure 7: REST command structure

The client would issue a command in a RESTful fashion by building a URI string like

HTTP://localhost:8081/rest/admin/login/request

and doing a GET to the server. The server returns the challenge value and the client puts that value on the page and waits for the user to input the response. When the user presses submit it sends another RESTful command with the response value and if that checks out, the response allows the client script to show the RUN page.

2.1.1 Server Code

If you go to the Site directory in Minnow and then go to the rest folder in there, that is the server scripting code. In the command line that started the server:

```
./Minnow -c Site/rest/main.lua -http Site/Site -rest Site/rest -port 8081
```

The highlighted portion is the initial server script this **main.lua** reads the command line for the subsequent parameters and uses them to start the web server. The RESTful operation is supported by code that turns each incoming request URI into a Lua list. It's important to note that this script does not get all the requests only those that start with **http://localhost:8081/rest**. The non-RESTful requests, i.e. regular page requests are handled automatically by the libmicrohttp code. On receipt of a RESTful request the Lua list in form of the URI is passed to a routine that looks at the element following rest, which in **HTTP://localhost:8081/rest/admin/login/request** is **admin**. A handler has been assigned to this node:

```
*** DATA RECEIVED ***
HTTP TYPE: 3
1 REST
2 ADMIN
```

```
3 LOGIN
4 REQUEST          <-- LOGIN GET request
n 4
2
Searching for ADMIN <-- Find ADMIN level commands (LOGIN)
-- Admin Actions --
3
Searching for LOGIN <-- Find LOGIN level commands REQUEST
-- Login actions --
4
Searching for REQUEST <-- Handling request
-- In Login Request handler --
*** DATA RECEIVED ***
HTTP TYPE: 6
1 REST
2 ADMIN
3 LOGIN
4 RESPONSE
5 6F15           <-- POST with challenge-resp from browser
n 5
2
Searching for ADMIN
-- Admin Actions --
3
Searching for LOGIN
-- Login actions --
4
Searching for RESPONSE
-- Build the site structure list
site = {}
site.ADMIN = admin_handler
```

So that any request with **admin** will be passed in this case to `admin_handler`. If you look at the file **admin.lua** in `Site/rest` you will see at the bottom it then assigns handlers to subsequent parts of the `admin` command.

```
-----
-- STRUCTURE
-----
-- Build action list for admin actions
admin_actions = {}
admin_actions.POWER = stop_handler
admin_actions.TIME = time_handler
```

```
admin_actions.LOGIN = login_handler
```

In **rest_sup.lua** there is a handler routine that is used to call the proper handler given the URI list and position. This is called at the entrance to each section of the command structure and automatically calls the proper handler or if not found returns an error.

With the above list the admin handling routine knows that if the list element following admin is **login** the login handler is called. (i.e. in **HTTP://localhost:8081/rest/admin/login/request**). In the login handler code (in login.lua) we have another list of possible commands as:

```
-----  
-- STRUCTURE  
-----  
-- Build action list for login actions  
login_actions = {}  
login_actions.REQUEST = login_request_handler  
login_actions.RESPONSE = login_response_handler  
login_actions.STATUS = login_status_handler  
login_actions.LOGOUT = login_logout_handler
```

So that if the keyword following **login** in the URI is **request** then the `login_request_handler()` function is called.

Inside the `login_request_handler()` the actions are rather simple, at this point the handler has the URI list and knows what data it needs from that list. The handler builds a response as a Lua list which has elements which will be the JSON key's in a key value JSON response. Filling out those fields a support routine (in Lua) is called to convert a Lua list to a JSON string. This is then returned to the server and we are done.

```
function login_request_handler(rqst, htype, url, n, al)  
local out = {}  
local data  
local i  
-- Is this a GET, if so look pick random value from random list  
-- Also fail if already logged in  
if( htype == HTTP_GET and LOGIN_OK == nil)  
then  
-- Add some randomness to our BAD random # gen  
data = os.time() % 1000  
math.randomseed(data)  
for i=1,data  
do  
math.random()  
end  
data = math.random(1,16)
```

```
-- Get index into random # list
LOGIN_CHAL_PSN = data
-- NOTE: This makes it simple as just return the
-- same random number not the NEXT one
-----
-- Build response
out.CHAL = LOGIN_CHAL_LIST[LOGIN_CHAL_PSN]
else
out.CHAL = "UNKNOWN RQST"
end
-- Return response
out = json_kvps(out) -- At this point the result is -> JSON
h:close(out) -- Send the result to the client
end
```

2.1.2 Server Summary

On the server side you will need main script like main.lua to start the process and define the top level commands (following rest/...) . In the demonstration we only have one (**admin**) but for more complicated situations you will have many more. Also in main you will have require lines bringing in these support scripts

First you will read these values in from the command line

```
-- Pull up items from command line
-- NOTE THESE ARE GLOBALS FOR THE REST
-- OF THE SYSTEM
REST_LCN = flag_srch( arg, "-rest", 1 )
HTTP_LCN = flag_srch( arg, "-http", 1 )
PORT_NUM = flag_srch( arg, "-port", 1 )
if( REST_LCN == nil )
then
print("SYNTAX ERROR: -rest <REST file lcn> NOT GIVEN")
print("Shutting down")
os.exit(-1)
end
if( HTTP_LCN == nil )
then
print("SYNTAX ERROR: -http <HTTP file lcn> NOT GIVEN")
print("Shutting down")
os.exit(-1)
end
if( PORT_NUM == nil )
then
```



```
print("WARNING: Server Port # not given")
print("Server Port = 8080")
PORT_NUM = 8080
end
```

Now you can use these input values to set up the search path for the support lua script

```
-- Set that into the package search path
_G.package.path = REST_LCN .. "/?.lua;" .. _G.package.path -- This sets up t
-- Now all modules will load with just a simple require + name
require "rest_sup"      -- Support for REST
require "login"         -- LOGIN support script
require "admin"        -- ADMIN support script
```

The rest of the code in main.lua is simply a loop reading handling REST request using the rest_handler() routine to call the correct supporting function.

2.2 SAMPLE REST API

The REST API consists of a number of URI's and specifications for the data sent or received with the requests. We will specify the URI as http://<server>/<rest>/... Where <server> stands for the TCP/IP address and port of the server and <rest> stands for the path link set up for REST interactions (see 3.1.3.1 on page 14). The overall design on the API is shown in Figure 7 on page 8.

2.2.1 ADMIN/TIME

This is used to read out the server system clock.

Request URI	TYPE	DATA
http://<server>/<rest>/admin/time	GET	NONE

The returned data is the current server system time.

```
{ "TIME" : "Sun Nov 14 16:46:21 2010" }
```

2.2.2 ADMIN/POWER

This URI is used to check for system running and to shutdown the system

Request URI	TYPE	DATA	ACTION
http://<server>/<rest>/admin/power	GET	NONE	Return status
http://<server>/<rest>/admin/power/on	POST	NONE	Return status
http://<server>/<rest>/admin/power/off	POST	NONE	Return status and shutdown

The returns status for the GET and first POST is:

```
{ "SYSTEM" : "ON" }
```

For the second POST shutdown the response is:

```
{ "SYSTEM" : "OFF" }
```

2.2.3 ADMIN/LOGIN

Multiple sub commands used for logging in. A client is logged in no other IP address can access the web server (see the `http.lock()` command 3.1.3.3 on page 15). It is a simple challenge/response system. When a request is made to log in the system supplies a challenge value, a 16 bit value. The user responds with 16 bit (i.e. 4 digit) value. If this matches that client is logged in and no other IP address is allowed by the server.

The challenge is one of these 4 byte values:

```
"50A0", "F748", "95B1", "8D8B", "4F34", "52C7", "85B6", "EA03",  
"E6B8", "D37F", "4FE1", "5215", "A868", "9336", "2885", "6F15"
```

The correct response is the value following the challenge value (with the last value 6F15, looping back to the first 50A0).

NOTE: The demonstration version has been modified to accept the same number not the NEXT see in (2.1.1 on page 11)

ADMIN/LOGIN/REQUEST

Used to request a challenge value.

Request URI	TYPE	DATA
<code>http://<server>/<rest>/admin/login/request</code>	GET	{ "CHAL" : "<Chal value>" }

ADMIN/LOGIN/RESPONSE

This is used to return the response to the challenge.

Request URI	TYPE	DATA
<code>http://<server>/<rest>/admin/login/response/<resp hex></code>	POST	{ "LOGIN" : "OK" }

ADMIN/LOGIN/STATUS

This is used to query the server on the login status

Request URI	TYPE	DATA
<code>http://<server>/<rest>/admin/login/status</code>	GET	{ "LOGIN" : "OK" } or { "LOGIN" : "NO" }

ADMIN/LOGIN/LOGOUT

Request URI	TYPE	DATA
<code>http://<server>/<rest>/admin/login/logout</code>	POST	{ "LOGOUT" : "OK" }

3 Lua Extension Classes

There are a number of extension classes written to support Minnow. There is a class layered over the libmicrohttpd Web server library allow Lua access to web data for the RESTful operation, there are support extension classes added for timers, JSON parsing and keyboard control.

3.1 RESTful Interface CLASS

3.1.1 Introduction

The **Minnow** Lua system has a built in Web server so it can be used in conjunction with a browser based client application. This allows a rather full GUI application on any OS using the same JavaScript/Browser application. In order for this to work an HTTP support library has been added. The library GNU libmicrohttpd (<http://www.gnu.org/software/libmicrohttpd/>)[3] is very well suited for this task. It is small, self-contained and is designed to be added into applications.

In addition a JSON parser has been added to support data sent in this form. JSON messages out are easily generated using formatted write statements in Lua.

3.1.2 Class Operation

The Web server/RESTful class has three major parts. The first is used to stop and start the server. The start command requires parameters to tell it the TCP/IP port to use for the HTTP transfer, data where standard web pages are located and a location base for the RESTful request URIs. The system can be used as a normal web server serving up pages, in fact this is used to load up the Javascript application to the client's browser.

The second part is a simple open/close set of calls to accept a request and return a response. Each request is an opaque Lua user object that is generated on receipt of a request and is automatically destroyed when the response is returned.

The third section of the class is used to extract information from the request, this includes the request URL and any data included in the request.

3.1.3 Class Methods - Server Control

3.1.3.1 http.start() This will start the HTTP server running.

```
status = http.start(port, http_base, rest_base)
```

INPUT	NAME	USE
	port	TCP/IP used for HTTP messages
	http_base	Absolute path to standard html files served
	rest_base	Base path for RESTful actions of server
OUTPUT	status	1 if server started OK, nil if failure

If the port was set to 8080 and the server's address was say 192.168.1.100 then the following URI would access index.html stored at the http_base path.

```
http://192.168.1.100:8080/index.html
```

If rest_base was set to REST then following URI would access some RESTful action on the path action/node-1/fire/1

```
http://192.168.1.100:8080/REST/action/node-1/fire/1
```

More details on the data and path information will be found in the http.url() method.

3.1.3.2 http.stop() This will stop the HTTP server.

```
http.stop()
```

INPUT	NAME	USE
OUTPUT	NONE	

This code can be used at any time after the http.start() command is given and you can then restart the server without shutting down the program.

3.1.3.3 http.lock()

```
http.lock(rh, flag) or rh:lock(flag)
```

This command is used to lock or unlock the server to only speak to one IP address. It is run in the context of handling a request and if locked, the request client's IP address will be saved and only requests from that IP address will be accepted till the server is unlocked. All other requesters will get a 404.

INPUT	NAME	USE
	rh	HTTP RESTful request handle(see 3.1.3.3)
	flag	If <> nil then lock is done, if nil then unlock
OUTPUT	NONE	

In the context of a request from IP=192.168.1.103 then:

```
h:lock(1)
```

Would lock the server to 192.168.1.103 any other client would get a 404 on any request.

3.1.3.4 http.open()

```
rh = http.open()
```

This code is a **non-blocking** call to receive a request. If a request is present you will be returned a handle to the request, if none you will get nil.

INPUT	NAME	USE
OUTPUT	rh	HTTP RESTful request handle. nil if no request pending

You can use this call to poll for requests, note the requests are queued so any pending requests are held till you retrieve them. You can have multiple outstanding requests.

3.1.3.5 http.close()

```
http.close(rh, data) or rh:close(data)
```

INPUT	NAME	USE
	rh	HTTP RESTful request handle
	data	Data to be returned to client
OUTPUT	NONE	

As you can see this is an instance class and you may use the request handle as the selector to pick the method (close()). Either form is acceptable. This is used to return the result of a RESTful request to the client. The data is usually encoded as JSON and will available to the client.

3.1.3.6 http.url(h)

```
http.url(rh) or rh:url()
```

This method is used to get the URI of the request.

INPUT	NAME	USE
	rh	HTTP RESTful request handle
OUTPUT	url	URL returned as a numeric index list of the path parts [1] = rest_base
OUTPUT	htty_type	Numeric value of HTTP request type, see below for list

This class like close may be used either as a class method or an instance method. In addition it returns two parameters the url of the request and the HTTP request type.

The List returned for the example REST request `http://192.168.1.100:8080/REST/action/node#/fire/1` would be:

INDEX	VALUE
1	REST
2	action
3	node #
4	fire
5	1

The `http_type` value returned is a number with the following meanings.

Value	Meaning
0	HTTP_NONE
1	HTTP_CONNECT
2	HTTP_DELETE
3	HTTP_GET
4	HTTP_HEAD
5	HTTP_OPTIONS
6	HTTP_POST
7	HTTP_PUT
8	HTTP_TRACE

The highlighted items are the only ones normally used in the RESTful protocol.

3.1.3.7 `http.data()`

```
post_data = http.data(rh) or rh:data()
```

This is used to get the data on a POST request.

INPUT	NAME	USE
	rh	HTTP RESTful request handle
OUTPUT	data	POST data from request, all other types return nil

This method is used to get the POST data on a request. This is returned as a string. In most cases in FIRENET it will be a JSON string which can be processed using the JSON parser.

3.1.4 JSON Support

This class is used to turn JSON into a LUA list.

3.1.4.1 `parsers.json()` This is used to turn a JSON string into a Lua list structure.

```
json_list = parsers.json(json_string)
```

INPUT	NAME	USE
	json_string	Valid JSON string
OUTPUT	json_list	List version of JSON structure or nil if invalid JSON

This will turn a JSON string structure for example:

```

json_string = "{ \"key\" : \"value\" \"key1\" : \"value1\" }"
list = parsers.json(json_string)
print(list)
table: 0x100108bd0
table.foreach(list,print)
JOBJ table: 0x100108c10
table.foreach(list.JOJB,print)
key value
key1 value1

```

See the JSON references for an idea of what sort of structures you can expect.

3.1.4.2 json_kvp() This is a support function in the rest_sup.lua file. It takes a lua list and turns it into JSON. (i.e. the opposite of parsers.json())

```

rtnval = json_kvp(myresult)

```

INPUT	NAME	USE
	myresult	A Lua list indexed by key strings
OUTPUT	rtnval	A JSON string of key/value pairs and lists

For example if myresult was a lua list of the form

myresult.KEY= "VALUE"

The JSON string would be "{ \"KEY\" : \"VALUE\" }

3.2 Timer CLASS

This class is used for millisecond timing of events. It can be used to delay actions or time actions to millisecond accuracy. The general sequence is one creates a timer object with new() (which also starts the timer) and then can query to determine if a fixed time has passed with the done() method. The time period can be re-started with the start() method.

There is one class method (sleep()) that can be used without creating a timer object. The other methods work from a specific timer object.

3.2.1 Timer Class Methods

3.2.1.1 timer.sleep() Will cause Lua to idle for a set number of milliseconds.

```

handle:sleep(ms)

```

INPUT	NAME	USE
	ms	Time to sleep in milliseconds
OUTPUT	NONE	

3.2.2 Timer Constructors/Destructors

3.2.2.1 timer.new() This creates a timer object that Lua can use for periodic operations and to check for elapsed time. Each timer has an individual handle and there is no limit to the number a script can have. Note all timers should be closed when they are no longer needed.

```
handle = timer.new()
```

INPUT	NAME	USE
	NONE	No input needed
OUTPUT	handle	Handle to open timer, nil if failure

The handle is opened and timer is started at 0.

3.2.2.2 timer.delete() This will dispose of a timer when it is no longer needed.

```
handle:delete()
```

INPUT	NAME	USE
	handle	Open timer handle
OUTPUT	result	1 if deleted, nil if failure

3.2.3 Timer Methods

3.2.3.1 timer.start() This is used to reset an active timer to 0. Useful to reset elapsed time to 0.

```
result = timer.start(handle) or handle:start()
```

INPUT	NAME	USE
	handle	Open timer handle
OUTPUT	result	1 if reset, nil if failure

3.2.3.2 timer.done() This is used to check for timer done, returns true when interval passed or nil if not

```
handle:done(delay)
```

INPUT	NAME	USE
	handle	Open timer handle
	delay	Delay time in ms
OUTPUT	result	1 if => delay time, nil if not

3.2.3.3 timer.read() This is used to return the current elapsed time for a particular timer

```
handle:read()
```

INPUT	NAME	USE
	handle	Open timer handle
OUTPUT	result	Current elapsed time in ms or nil if error

3.3 Keyboard Class

This class which has no methods is used to query the keyboard while a script is running. This can be used to allow the user to hit keys to modify the script actions without halting script loops.

3.3.1 Class Methods

3.3.1.1 kbd.prep() Prepares the keyboard for async input

```
kbd.prep()
```

INPUT	NAME	USE
OUTPUT	result	1 if prep OK, nil if not

3.3.1.2 kbd.close() Called to shut down the async processing of input. (See 3.3.2 on the following page)

```
kbd.close()
```

INPUT	NAME	USE
OUTPUT	result	1 if close OK, nil if not

3.3.1.3 kbd.getc() Reads the keyboard without stopping

```
ch = kbd.getc()
```

INPUT	NAME	USE
OUTPUT	result	Keyboard character as a string or nil if none

3.3.2 Use

This code is used so the script can stay in a loop and query the keyboard in passing. This query does not halt the loop and allows characters to be input while the loop proceeds. Code example

```
kbd.prep() -- Get read for user input
while( flag )
do

    -- Do real work here....
    work_routine()
    -- User input ?
    ch = kbd.getc()
    if( ch ~= nil and ch == "Q")
    then
        -- User asked to quit
        print("** QUIT ENTERED **")
        break
    end

end

end
-- Remember to close on exit
kdb.close()
```

References

- [1] D. Crockford. The application/json media type for javascript object notation. Request for Comments 4627, Network Working Group, July 2006.
- [2] Christian Grothoff. Gnu libmicrohttpd.
- [3] Christian Grothoff. Gnu libmicrohttpd.
- [4] JQuery. JQuery.
- [5] PUC-Rio Roberto Ierusalimschy, Departamento de Informática. Lua home page.
- [6] Wikipedia. Ajax (programming).
- [7] Wikipedia. Representational state transfer.