PANDORA PRODUCTS

Pandora Products Manual
Network Firing Box

Jim Schimpf

Document Number: PAN-200908001
Revision Number: 1.5
20 January 2011

Pandora Products.
215 Uschak Road
Derry, PA 15627

©2009-2010 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: 412.829.8145

Fax: 724.539.1276

Web: pandora.dyn-o-saur.com:8080/~jim/

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
0.1	js	Initial Version	6-Aug-2009
0.2	js	Add application layer commands	8-Aug-2009
0.3	js	Add sync clocks in nodes and sync firing	12-Aug-2009
0.4	js	Add in information about scripting support	4-Sep-2009
0.5	js	Add in information about firenet_sup.lua	13-Sep-2009
0.6	js	Scripting operations and play_file_stop() routine0	21-Sep-2009
0.7	js	Rewrite packet protocol [d26a3ab0e9]	11-May-2010
0.8	js	Add set address command [4faba11f43]	15-May-2010
0.9	js	Update document for NEWNET [de1bb57002]	19-May-2010
1.0	js	Update document with RESTful interface in Lua	31-Oct-2010
1.1	js	Show RESTful command structure	14-Nov-2010
1.2	js	Add Login commands	18-Dec-2020
1.3	js	Add set fire time command	9-Jan-2011
1.4	js	[e4208b8f56] Add ACK message	16-Jan-2011
1.5	js	[dcb5f4c816] Add VERSION message	20-Jan-2011

Contents

1 System Design **2**

1.1 Introduction 2

1.2 Software Design 2

1.3 User Interface 3

1.4 RESTful Interface 3

1.4.1 REST use 3

1.4.2 REST API 4

1.4.3 ADMIN 4

1.4.3.1 ADMIN/TIME 4

1.4.3.2 ADMIN/POWER 5

1.4.3.3 ADMIN/LOGIN 5

1.4.4 FIRENET 6

1.4.4.1 FIRENET/STATUS 6

1.4.4.2 FIRENET/ARM 7

1.4.4.3 FIRENET/FIRE 8

1.4.4.4 FIRENET/SYNC 8

1.4.4.5 FIRENET/DELAY 8

2 Lua Extension Classes **8**

2.1 RESTful Interface CLASS 8

2.1.1 Introduction 8

2.1.2 Class Operation 9

2.1.3 Class Methods - Server Control 9

2.1.3.1 http.start() 9

2.1.3.2 http.stop() 10

2.1.3.3 http.lock() 10

2.1.3.4 http.open() 10

2.1.3.5 http.close() 10

2.1.3.6 http.url(h) 11

2.1.3.7 http.data() 11

2.1.3.8 parsers.json() 12
2.2 Timer CLASS 12
2.2.1 Timer Class Methods 12
2.2.1.1 timer.sleep() 12
2.2.2 Timer Constructors/Destructors 13
2.2.2.1 timer.new() 13
2.2.2.2 timer.delete() 13
2.2.3 Timer Methods 13
2.2.3.1 timer.start() 13
2.2.3.2 timer.done() 13
2.2.3.3 timer.read() 14
2.3 Firenet Class 14
2.3.1 Data Format 14
2.3.2 Creators/Destructors 14
2.3.2.1 firenet.new() 14
2.3.2.2 firenet.delete() 14
2.3.3 Methods 15
2.3.3.1 firenet.read() 15
2.3.3.2 firenet.write() 15
2.4 Keyboard Class 15
2.4.1 Class Methods 15
2.4.1.1 kbd.prep() 15
2.4.1.2 kbd.close() 16
2.4.1.3 kbd.getc() 16
2.4.2 Use 16

3 Lua Support Code (firenet_sup.lua) 17
3.1 Introduction 17
3.2 Lua Globals 17
3.2.1 BCAST_ADDR 17
3.2.2 FNET_MAP 17
3.2.3 OS_MUSIC_PLAYER 17

3.3 Network Support Routines 18
3.3.1 build_fnet_status() 18
3.4 Misc Support Routines 18
3.4.1 play_file() 18
3.4.2 play_file_stop() 18

4 FIRENET Node Design **19**

5 Firenet Network Design **19**
5.1 Physical Layer 19
5.2 Data Link Layer 20
5.2.1 Packet Format 20
5.2.2 Addressing 20
5.2.3 Transmission 20
5.3 Application Layer 21
5.3.1 Commands 21
5.3.1.1 ARM Command 21
5.3.1.2 FIRE Command 22
5.3.1.3 DELAY FIRE Command 22
5.3.1.4 STATUS Command 22
5.3.1.5 WHO (are you) Command 22
5.3.1.6 TIME Command 23
5.3.1.7 SYNCHRONIZE Node Clocks 23
5.3.1.8 REPLY to messsage 23
5.3.1.9 VERSION of Node code 23

6 Firenet Board **23**
6.1 Introduction 23
6.2 Node Design 24
6.2.1 Power Supply 25
6.2.2 Network 26
6.2.3 Firing Circuit 26
6.2.4 Processor 26

List of Figures

1 FIRENET System 2
2 REST command structure 3
3 RS-485 Network 19
4 Packet Format 20
5 Firenet System 24
6 Node Schematic 25

1 System Design

1.1 Introduction

The FIRENET system is shown in 1. There are two control computers in addition to the FIRENET firing boards. The Control Computer is directly connected to the RS-485 FIRENET network and interacts with the FIRENET boards. It is also running the FIRENET Web Server/REST software. Representational State Transfer or REST is a simple protocol on top of HTTP that uses GET/PUT/POST and DELETE HTTP messages to allow control of a system. [7]. For FIRENET a User Interface for the system is running in a browser on the Interface computer. It uses REST style interaction to send commands which the Control Computer sends out on the FIRENET network to the nodes which then run the fireworks show. Each FIRENET node controls up to 6 pieces and can read out their status and and fire these pieces on command.

The connection between the two computers is over TCP/IP and in practice would use a WPA encrypted WiFi connection allowing the Interface Computer to be at a safe distance from the pieces and morters firing the show.

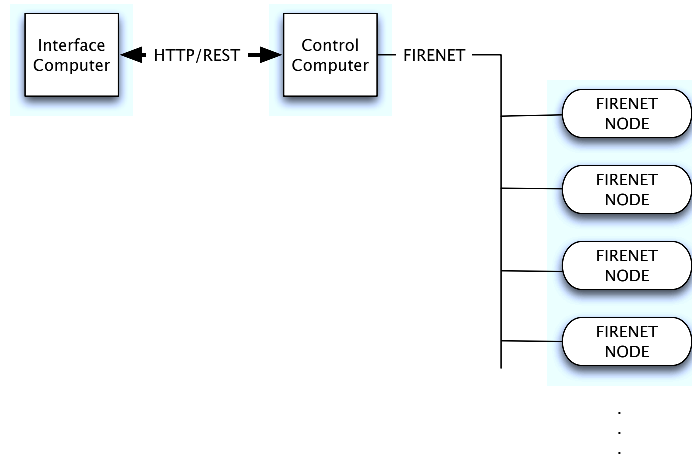

Figure 1: FIRENET System

1.2 Software Design

The software consists of a number of parts. At the lowest level Arduino[1] code is written for each FIRENET node. This allows it to communicate over the FIRENET RS-485 network and to have commands to read status and fire pieces. Above that we have the program running on the control computer which is a modified Lua interperter[6]with extensions to support FIRENET and HTTP webserving [3]. This latter acts as a Web Server and transports the REST protocol commands. Finally a Cappuccino[5], a Web Framework tool is used to build a Javascript application that runs on the browser in the Interface Computer, this is used supply an OS agnostic user interface to the system.

1.3 User Interface

<TBD>

1.4 RESTful Interface

The web server running on the Control Computer actally has two functions, it can just serve up web pages and second under control of Lua scripting it can interact in a RESTful manner. Serving up web pages allows the system when contacted by the interface computer to serve up Login pages or the Javascript application. On the application is running the interface computer browser it then interacts with the web browser in a RESTful manner.

1.4.1 REST use

In the system used here we restrict ourselves to just two commands GET and POST. The HTTP get command is used to query the system for information and the POST command is used cause the system to do some action. There is a wide latitude in how RESTful commands are done, you can either put command parameters (in a POST) in a JSON[2]encoded data block sent with the POST or encode it into the URL of the command.

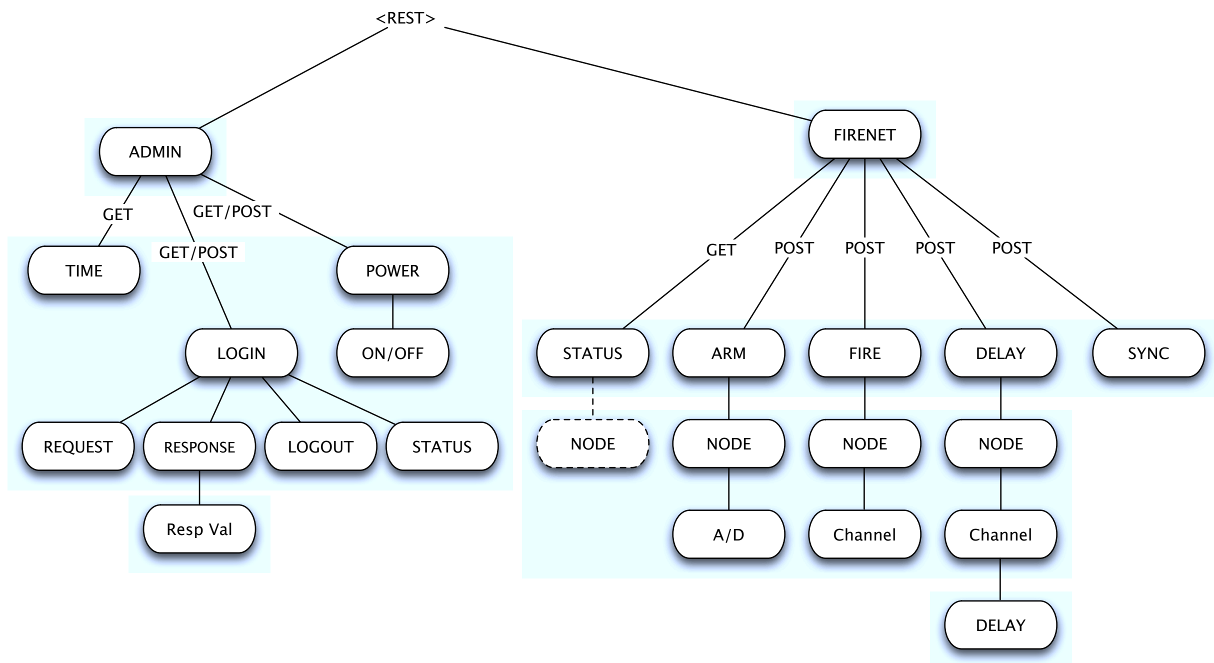

Figure 2: REST command structure

In the FIRENET system we have encoded most of the command syntax into the URI. As you can see from the base node (which is programmable see: 2.1.3.1 on page 9) we then have nodes to get

the administrative or FIRENET functions. Doing GETs will return data from a node and doing a POST will cause some action.

For the following examples we will assume the web server (Control computer) is at http://192.168.1.100:8080 and that the base of the RESTful tree is at REST.

So a full FIRENET system status is done with an HTTP GET to:

```
http://192.168.1.100:8080/rest/firenet/status
```

The response from this query is a JSON data block.

Status of a single node is found via

```
http://192.168.1.100:8080/rest/firenet/status/5
```

where 5 is the selected node number.

Firing a node is done by a POST to:

```
http://192.168.1.100:8080/rest/firenet/fire/5/3
```

would fire channel 3 on node 5

1.4.2 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 2.1.3.1 on page 9).

1.4.3 ADMIN

These are overall control functions for the server and are not concerned with the FIRENET nodes or network.

1.4.3.1 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" }
```

1.4.3.2 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 to do the shutdown you get:

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

1.4.3.3 ADMIN/LOGIN This is a path with multiple sub commands and it used for logging in. When a client is logged in no other IP address can access the web server (see the http.lock() command 2.1.3.3 on page 10). The system is a simple challenge/response system. When a request is made to log in the system supplies a challenge value, a 16 bit HEX value. The user then responds with another 16 bit (i.e. 4 hex digit) value. If this matches the criteria that client is logged in and no other IP address is allowed by the server.

The challenge is one of these 4 byte hex 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).

ADMIN/LOGIN/REQUEST

This is used to request a challenge value.

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

ADMIN/LOGIN/RESPONSE

This is used to return the response to the challenge.

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

ADMIN/LOGIN/STATUS

This is used to query the server on the login status

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

ADMIN/LOGIN/LOGOUT

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

1.4.4 FIRENET

This URI us connected to the FIRENET network and will control and return status of the operating FIRENET boards.

1.4.4.1 FIRENET/STATUS This URI is used to read the status of the entire network or a single node status. The entire network status is useful when starting the system or to recover from a restart.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/status	GET	Entire network status
http://<server>/<rest>/firenet/status/<node#>	GET	Single node status

The entire network status is:

```
{ "STATUS" : "SUCCESS"
    {
       "1" : {
          "ARMED" : "0"
          "UNFIRED" : {
            "1" : "1"
            "2" : "1"
            "3" : "1"
            "4" : "1"
            "5" : "1"
            "6" : "1"
           }
          "FIRED" : {
            "1" : "0"
            "2" : "0"
            "3" : "0"
            "4" : "0"
            "5" : "0"
            "6" : "0"
           }
    }
    {
```

```
        "2" : {
        :
        :

    }
```


Each node is grouped separately and "1" node is shown fully here. As you can see each node has its ARMED status (see below) and the status of each firing channel listed. The UNFIRED values with a 1 means there is ignitor connected to that channel and it has not been fired. A 0 means there is nothing connected to that channel. The FIRED value, 0 means it has not been fired yet and 1 means it has.

When a single node status is read you get the following JSON. This is almost the same as the full status but the node number is not given in this case since you requested a particular node in the URI that value is not returned with the data.


```
{ "STATUS" : "SUCCESS"
    {
       "ARMED" : "0"
       "UNFIRED" : {
          "1" : "1"
          "2" : "1"
          "3" : "1"
          "4" : "1"
          "5" : "1"
          "6" : "1"
        }
       "FIRED" : {
          "1" : "0"
          "2" : "0"
          "3" : "0"
          "4" : "0"
          "5" : "0"
          "6" : "0"
        }
     }

}
```


1.4.4.2 FIRENET/ARM This command is used to ARM, i.e. turn on firing power in a FIRENET node. The unit will not fire an ignitor unless it is armed.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/arm/<node #>/A	POST	Node # is the particular node to be armed
http://<server>/<rest>/firenet/arm/<node #>/D	POST	Node # is the particular node to be disarmed

The ARMED or DISARMED state will be reflected in the node status.

1.4.4.3 FIRENET/FIRE This command is used to fire a ignitor on a node. The data includes the node # and channel (1-6) on the node.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/fire/<node #>/<Channel #>	POST	Node # and Channel # must be supplied

After a ignitor is fired the status will be changed for that channel. The FIRED value will go to 1 and the UNFIRED will go to 0 if the firing was successful.

1.4.4.4 FIRENET/SYNC This message is sent to all FIRENET nodes and synchronizes their millisecond clocks. This will be needed to support delayed firing commands.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/sync	POST	All node millisecond clocks are set to 0

1.4.4.5 FIRENET/DELAY This is similar to the FIRE command but in this case the firing command will be executed at a set time. All FIRENET nodes have a 32 bit millisecond timer. The SYNC command zeros the timers on all nodes. Sending a DELAY command will queue up a FIRE command to occur at a specific 32 bit millisecond time. Thus a number of nodes can fire pieces as precisely the same time.

Request URI	TYPE	DATA
...firenet/delay/<node #>/<Channel #>/<MS time>	POST	Fire channel on specified node at 32 bit ms time

2 Lua Extension Classes

Firenet the firing system is controlled by Lua scripts run on the **FIRENET** program. There are a number of extension classes written to support these scripts. First there are extensions allowing FIRENET communication with the nodes. There is a class layered over the libmicrohttpd Web server library allow Lua access to web data for the RESTful operation. In addition there are support extension classes added for timers, JSON parsing and keyboard control.

Following this will be a discussion of the scripts written with these extensions to support both the RESTful interface and interfacing with the FIRENET network.

2.1 RESTful Interface CLASS

2.1.1 Introduction

The **FIRENET** 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/)[4]

is very well suited for this task. It is small, selfcontained 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.

2.1.2 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.

2.1.3 Class Methods - Server Control

2.1.3.1 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.

2.1.3.2 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.

2.1.3.3 http.lock()

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
	h	HTTP RESTful request handle
	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.

2.1.3.4 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.

```
h = http.open()
```

INPUT	NAME	USE
OUTPUT	h	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.

2.1.3.5 http.close()

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

INPUT	NAME	USE
	h	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.

2.1.3.6 http.url(h)

This method is used to return the addressed URL of the request.

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

INPUT	NAME	USE
	h	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-1/fire/1 would be:

INDEX	VALUE
1	REST
2	action
3	node-1
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.

2.1.3.7 http.data()

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

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

INPUT	NAME	USE
	h	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.

2.1.3.8 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.JOBJ,print)
key value
key1 value1
```

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

2.2 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.

2.2.1 Timer Class Methods

2.2.1.1 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	

2.2.2 Timer Constructors/Destructors

2.2.2.1 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.

2.2.2.2 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

2.2.3 Timer Methods

2.2.3.1 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

2.2.3.2 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

2.2.3.3 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

2.3 Firenet Class

This is the interface to the Firenet network. This class will be able to sent and receive messages with the firing boxes on the net. This will send raw packets and receive raw packets.

2.3.1 Data Format

Packets transmitted or received on the Firenet interface are in the form of Lua lists. There are three elements, the TO field where the destination address is specified, the FROM field where the source is specified and the DATA field for the packet data. All of these fields are present for a received packet, while the transmit packet does not need a FROM field.

2.3.2 Creators/Destructors

2.3.2.1 firenet.new()

This creates a new network interface object. You must call this before starting a run.

```
handle = firenet.new()
```

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

The object is created and the map is initialized for nodes present at this time.

2.3.2.2 firenet.delete()

This will dispose of a network interface and map when no longer needed.

```
handle:delete()
```

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

2.3.3 Methods

2.3.3.1 firenet.read()

This returns the next message from the network.

```
msg = handle:read()
```

INPUT	NAME	USE
	handle	Open firenet handle
OUTPUT	result	nil if no message or message list if found

When read the retuned message is a list with the following fields:

Field	Use	Example
TO	Destination node	Usually your node but can be broadcast address
FROM	Source Node	msg.from = Returns physical address
DATA	Message Dependent Data fields	Message dependent data (see Section 5.3.1 on page 21)

2.3.3.2 firenet.write()

This writes a message to the network

```
handle:write(msg)
```

INPUT	NAME	USE
	handle	Open firenet handle
OUTPUT	result	1 if sent, nil if problem

The send message is a Lua list with the following format

Field	Use	Example
TO	Message destination	msg.TO = 2 Send to node 3
DATA	Message Dependent Data fields	Message dependent data (see Section 5.3.1 on page 21)

2.4 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.

2.4.1 Class Methods

2.4.1.1 kbd.prep()

Prepares the keyboard for async input

```
kbd.prep()
```

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

2.4.1.2 kbd.close()

Called to shut down the async processing of input. (See 2.4.2)

```
kbd.close()
```

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

2.4.1.3 kbd.getc()

Reads the keyboard without stopping

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

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

2.4.2 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
-- Remember to close on exit
kdb.close()
```

3 Lua Support Code (firenet_sup.lua)

3.1 Introduction

It was a design decision to put the minimum number and most basic functions in the system as C extensions to Lua. Then build on those using Lua to make a useful API for system operation. Thus functions like timers and Firenet I/O were added as Lua extensions. But other functions like playing a music file or managing the network status were built on these extensions and written in Lua. In consequence a set of support functions were written and would be included into user written scripts.

3.2 Lua Globals

3.2.1 BCAST_ADDR

This variable contains the Firenet broadcast address. This can be used to set the destination address of a packet as follows:

```
packet = {}
packet.DATA = "S"          -- Send a STATUS command
packet.TO = BCAST_ADDR    -- Send to everyone
h:write(packet)           -- Send to the network
```

3.2.2 FNET_MAP

This variable holds the current table of nodes on the network and their status. See 3.3.1 on the following page) on how these variables are filled but when they are set the have the following fields:

```
FNET_MAP[<node number>]   -- Information on <node number>

    FNET_MAP[<node number>].ARMED    -- 1 if armed 0 if not
    FNET_MAP[<node number>].FIRED    -- 1-6 Array with 1 == Fired channels
    FIRE_MAP[<node number>].UNFIRED  -- 1-6 Array with 1 == Unfired ch
```

3.2.3 OS_MUSIC_PLAYER

This is the string name of the OS dependent command line music player program. In the case of OS X it is "afplay".

3.3 Network Support Routines

3.3.1 build_fnet_status()

This will fill the FNET_MAP variable with the current network node status. This will update the Arm/Disarm values plus the fired and unfired channels.

```
h = firenet.new()
build_fnet_status(h)
```

INPUT	NAME	USE
	addr	If present query only one node, if nil do broadcast
	h	Open firenet handle
OUTPUT	result	1 if status OK, nil if not

3.4 Misc Support Routines

3.4.1 play_file()

This will use an operating system dependent program that can play a sound file. It will be used to play the music file associated with a fireworks display

```
result = play_file(file)
```

INPUT	NAME	USE
	file	File name string (OS dependent)
OUTPUT	result	pid of music player

3.4.2 play_file_stop()

This will use an operating system dependent pid value from the play_file() routine to stop the music player.

```
play_file_stop(pid)
```

INPUT	NAME	USE
	pid	Pid return from play_file()
OUTPUT	NONE	Music player stops

4 FIRENET Node Design

This is a networked Fireworks controller system that uses a RS-485 Carrier Sense Muliple Access two wire network. Each firing box or node in the network has 6 firing circuits and up to 30 of these boxes can be networked together. Each box has fixed network address.

The nodes are controlled by a laptop also on the network running software that allows either manual or scripting firing sequences. The laptop through the controlling software can also monitor the health and connectivity of the nodes and the network.

5 Firenet Network Design

5.1 Physical Layer

The physical layer of the network uses the RS-485 standard interface with a two wire circuit. RS-485 uses a balanced circuit where the signal is one wire and the inverse is on the other wire. This gives greater noise immunity and a range of about 300 meters for the wiring. Since only two wires are used we both transmit and receive on the same pair. This means that a sender must enable the transmitter outputs then send the data, and when done disable the transmitter outputs. This is very similar (but much slower) that the original ethernet where all the signals were on a single coaxial cable.

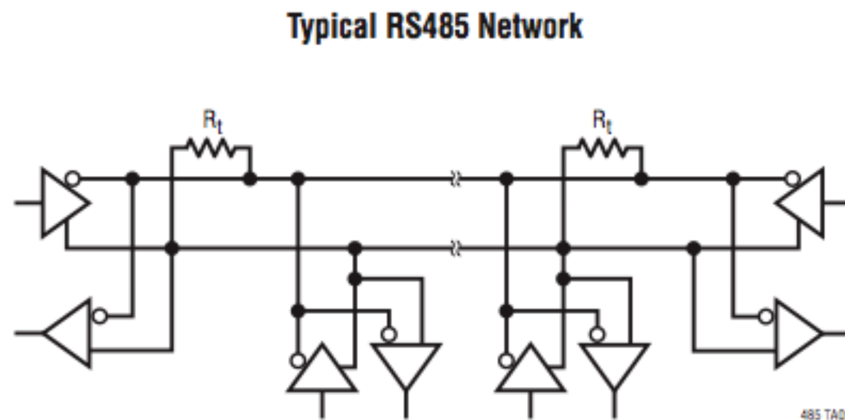

Figure 3: RS-485 Network

In this design we will be using the Linear Technology chip LTC485[?] chip that is RS-485 compliant and is low power.

5.2 Data Link Layer

5.2.1 Packet Format

This layer handles the carrier sense multiple access and addressing of the data packets exchanged. The data packet on the network looks like this:

Figure 4: Packet Format

The data in a packet is as follows:

Name	Data	Example
HDR	:	The ':' character
TO	ASCII two digit number	01 - 32 with 00 -> Broadcast
FROM	ASCII two digit number	01 - 32 with 00 -> Broadcast
DATA	Up to 32 bytes of commands with ,'s between cmds	F1,F2,F3
EOP	;	The ';' character
CKSUM	ASCII two digit number cksum	Mod 100 sum of all fields except : and ;

The data is sent as ASCII strings and multiple commands can be sent in the same data packet by using , separators between commands. The entire packet is a printable ASCII string.

5.2.2 Addressing

There are up to 32 devices in the network which have addresses ranging from 1 to 32. The addresses are fixed in the devices and there should be no duplicate addresses in the network. The address 32 is the command node. The address 0 is the broadcast address and all devices will receive messages sent to this address.

5.2.3 Transmission

When sending the data the RS-485 network has the receiver and transmitter on the same wire. In this way the packet sender will receive the packet when sent. If two units send packets simultaneously the data will be mixed in the receivers and the checksums will be wrong. If the transmission is wrong then both sender will stop, then wait a different random amount of time and then try up to three more times before giving up. This is similar the Aloha network.[**?**]

5.3 Application Layer

In this very simple network we will skip the Transport and Network layers of the OSI model. [**?**]The messages will be restricted to one packet and will be stand alone. Also the model here is with a controller (i.e. a laptop) controlling a number of firing boxes. This simplifies things as the nodes will not be talking to each other and will be driven by messages from the controller.

The unit of data to/from the network is a packet of the form:

```
#define PHY_DATA_SIZE 32
typedef struct {

    unsigned char type;
    unsigned char from;
    int len;
    unsigned char data[PHY_DATA_SIZE]; // Packet data

} PACKET;
```

5.3.1 Commands

Command	NAME	**Data**	**Information**
A	ARM	[D/A]	Disarm (D), Arm (A) firing circuits
F	FIRE	<Ckt #>	Fire this circuit
D	DELAY_FIRE	<Ckt #><Ms to fire>	Delay fire, fire at or at Time ms
S	STATUS	NONE	Status (see below for resp msg)
W	WHO	# new address	Changes address of node
T	TIME	# New firing time (ms)	Change the firing time
Z	SYNC	NONE	Sync all node clocks
R	REPLY	R+ Last cmd letter	Used to ACK no-reply messages
V	VERSION	RV__DATE__ __TIME__	Creation time of Node code

5.3.1.1 ARM Command This command is issued to enable/disable the firing circuits. If node is not armed then all firing commands are ignored. The node light will blink when box is armed.

AD Disarm node

AA Arm node

This command is ignored if sent to the broadcast address. There is no response.

5.3.1.2 FIRE Command This command will fire a selected circuit in the node immediatly. The node will be out of communication till the firing cycle is over. The available firing channels are numbered from 0-5.

F3 Fire circuit # 3

This command is ignored if sent to the broadcast address. There is no response.

5.3.1.3 DELAY FIRE Command This is similar to the firing command but will fire the selected ciruit at the specified time. All nodes keep a 32 bit millisecond counter. The Z command (see 5.3.1.7 on the next page) will sync the time for all the nodes. Thus you can send in a number of firing commands for different nodes and they will all fire at the same time. The available firing channels are numbered from 0-5. The command takes a single byte firing circuit value and 32 bit time value.

D316384 Fire circuit 3 after at 16,384 ms into the run.

This command is ignored if sent to the broadcast address. There is no response.

5.3.1.4 STATUS Command This command generates a response message showing the node status

S Return status of node

The return message looks like this (it is sent to the address of the unit that sent the status command

RS[A/D]<Fired circuits><Unfired circuits>

Field	Meaning
RS	Status response designator
[A/D]	Armed (A) or Disarmed (D) status of node
Fired circuits	Number 0-31 with it being the binary representation of fired ckts 03 - ckt, 0& 1
Unfired circuits	Number 0-32 with it being the binary representation of unfired ckts 03 -ckt 0 & 1

The status message can be sent as a broadcast.

5.3.1.5 WHO (are you) Command Is used to set the address of a node. After running the the node will now have the new address if valid. This will not be accepted if sent to the broadcast address. It will also not be accepted is sent to the control node address (32).

W04 Set address of node to 4

The address value must be in the range 1-31 to be accepted. After it is read the node now has the sent address value.

5.3.1.6 TIME Command This command is used to set the firing time, by default it is 100 ms. This is the firing pulse time for any channel. This can be changed with this command and will be set in the device till changed again.

T1000 Set firing time to 1000 ms (1 second)

Any value can be used and will depend on the device being fired.

5.3.1.7 SYNCHRONIZE Node Clocks

Z Synchronize node clocks

This command is used to set all node (and controller clocks to zero. All nodes in the system have a 32 bit millisecond counter. This command when received will set that counter to zero. Then this counter can be used to schedule synchronized events in the various firing boxes. (See Delay fire command 5.3.1.3 on the preceding page).

5.3.1.8 REPLY to messsage

R Reply to message

Only **Status** messsage returns a response so this message is the response returned by all the other message. The data consists of the Command letter of the responded to command. It was found (see bug [e4208b8f56]) that the Firenet hardware does not echo the transmitted characters so the sender does not know if the message was sent correctly. This way he will wait for a response from the receiver.

5.3.1.9 VERSION of Node code

V Query node code creation date

This message is used to get the creation date of the code running in a node. The message returned is a string of the compiler constants __DATE__ and __TIME__ which shows the creation date of the code. This is ment to be human interperted and not machine used.

6 Firenet Board

6.1 Introduction

The Firenet system has two main parts, the controller a standard computer running the Firenet program and a number of Firenet nodes which actually fire the pieces. The Firenet node is a small computer system with 6 firing circuits and a network connection. The system will look like this:

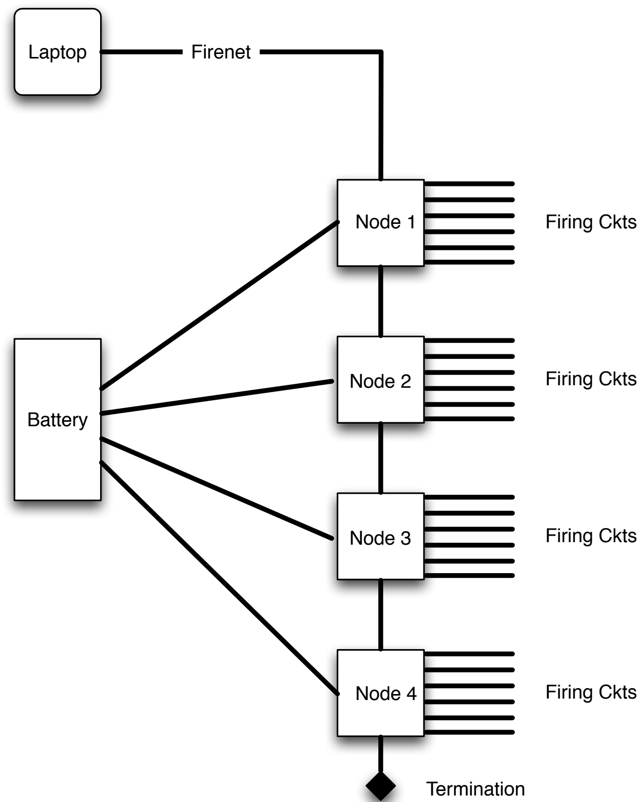

Figure 5: Firenet System

This shows the nodes connected in a daisy-chained network with the laptop. Each node is connected to up to 6 firework pieces and can fire them. They are also connected to a large 12V battery that supplies the firing current and power for their processor.

6.2 Node Design

Each node will have a processor running the Arduino kernel and then running a custom program that will execute the Firenet commands. The network as shown in Section 5.1 on page 19 is an RS-485 two wire system. The interconnect between each node is a stereo audio cable. The in and out connections on each node are a standard audio jack. The schematic for the node is:

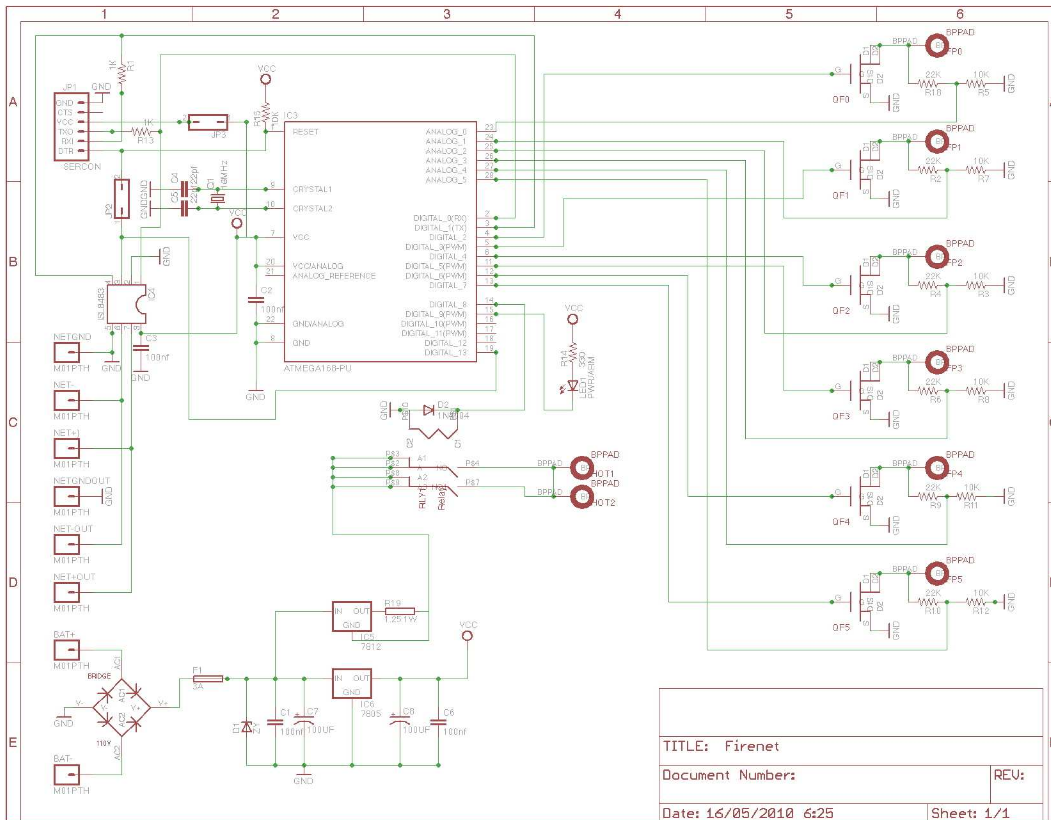

Figure 6: Node Schematic

6.2.1 Power Supply

The power supply uses two regulators. One supplies 5V for the logic and processor circuits and the second is a 1 Amp constant current supply (IC5) to supply the firing current. In front of the regulators is a full wave bridge, a fuse and a 15 V Zener diode. These are a protection circuit, to make the wiring of the system simple and inexpensive we are using standard 120 outlet strips and 120 V plugs to distribute the 12 V battery power. These connectors are very inexpensive and readily available. Also we can use standard 16 Gauge lamp cord for the power leads to the nodes. But with standard plugs on the ends of these wires there are two problems:

1. The plugs are not polarized.

2. They could be plugged into a 120 V circuit.

The full wave bridge on the front end solves problem (1) since it will always supply the correct polarity no matter which way the plug is inserted. The second problem is handled by the Zener diode and fuse. The full wave bridge is rated for 120 V so it could handle the high voltage. But the Zener conducts at 15 Volts and would short the high voltage to ground which would blow the fuse and protect the rest of the circuit.

6.2.2 Network

The network hardware is ISL8483 RS-485 chip. The transmit and receive lines are brought out to pads where they will be connected to two stereo phone jacks (one in and the other out). The network will be like that show in Section 5.1 on page 19 . The user will be responsible for putting terminating resistors in the first and last node.

The transmit output of the ISL8483 is under control of the processor which will assert the transmit before sending data and remove it when the transmission is done. Since the transmit and receive are wired together the sending unit will be able to see what it sends and monitor this for correctness.

6.2.3 Firing Circuit

The firing circuit includes the 1 Amp constant current regulator, a relay and then 6 FET drivers. IC5 (7812) is wired to produce 1 Amp output for loads down to 0 ohms. This current is switched by a relay (RLY1) so the firing circuits be rendered OFF except when needed. The 6 FETs (QF0-QF5) are wired as simple switches so when the processor makes the gate high the FET goes to low resistance. The Fireworks ignitor is wired between the CC supply and the FET drain so up to 1 A flows firing the piece. The resistor divider off the drain is used to feed the voltage value present on the drain to the processor A/D. If the transistor is OFF this will be the full voltage of the battery. The approximately 3->1 divider will give about 4 volts out which the A/D can handle. This will show which circuits have unfired ignitors and can be used to control firing time when the circuit fires an ignitor.

6.2.4 Processor

The processor circuit is largely copied from the Arduino board. The serial, reset and power leads are brough to a header which fits the SparcFun RS-232 <-> USB board. This can be used to talk to the board for testing. Also two jumpers are added to allow the USB board to power the board and control the network chip JP2,JP3). A board with only the network chip and these jumpers connected will be the board used by the laptop to control the system.

Index

ARM, 7

BCAST_ADDR, 17
build_fnet_status(), 18

DELAY, 8

FIRE, 8
firenet.delete(), 14
firenet.new(), 14
firenet.read(), 15
firenet.write(), 15
FNET_MAP, 17

http.close(), 10
http.data(), 11
http.lock(), 10
http.open(), 10
http.start(), 9
http.stop(), 10
http.url(), 11

kbd.close(), 16
kbd.getc(), 16
kbd.prep(), 15

LOGOUT, 6

parsers.json(), 12
play_file(), 18
play_file_stop(), 18
POWER, 5

REQUEST, 5
RESPONSE, 5

STATUS, 6
STATUS, 5
SYNC, 8

TIME, 4
timer.delete(), 13
timer.done(), 13
timer.new(), 13
timer.read(), 14
timer.sleep(), 12
timer.start(), 13

VERSION, 23

References

[1] Arduino. Arduino home page.

[2] D. Crockford. The application/json media type for javascript object notation. Request for Comments 4627, Network Working Group, July 2006.

[3] Christian Grothoff. Gnu libmicrohttpd.

[4] Christian Grothoff. Gnu libmicrohttpd.

[5] 280 North. Cappuccino web framework.

[6] PUC-Rio Roberto Ierusalimschy, Departamento de Informática. Lua home page.

[7] Wikipedia. Representational state transfer.