PANDORA PRODUCTS

Pandora Products Manual
Network Firing Box

Jim Schimpf

Document Number: PAN-200908001
Revision Number: 2.9
26 December 2011

Pandora Products.
215 Uschak Road
Derry, PA 15627

Pandora Products.

215 Uschak Road

Derry, PA 15627

Phone: 724.539.1276

Web: http://pandora.dyn-o-saur.com:8080/cgi-bin/Firenet.cgi

Email: vze35xda@verizon.net

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
1.5	js	[4fd91ac4c2] Add Channel message	24-Jan-2011
1.6	js	Web interface	16-Apr-2011
1.7	js	Update REST interface information	22-Apr-2011
1.8	js	Add Backoff & Reset commands	1-May-2011
1.9	js	Add docs on commands returning values	1-May-2011
2.0	js	Allow status command to clear map	8-May-2011
2.1	js	Apply Chip Maguire corrections	31-May-2011
2.2	js	Add Get Data command	16-Jul-2011
2.3	js	[9e81c1a523] Adaptive firing operation	16-Oct-2011
2.4	js	[156fc0f116d] Node design information	6-Nov-2011
2.5	js	[fd8800dae4] Add firing queue mode	12-Nov-2011
2.6	js	[9eae8bcde5] Add PROGRAM to rest commands	20-Nov-2011
2.7	js	[349529d41d] Add AUTO run to web interface	10-Dec-2011
2.8	js	Updated information about SYNC	26-Dec-2011
2.9	js	Add AUTO show images	26-Dec-2011

Contents

1 System Design **2**
1.1 Introduction . . . 2
1.2 Software Design . . . 2
1.3 User Interface . . . 3
1.3.1 Interface Operation . . . 3
1.4 RESTful Interface . . . 9
1.4.1 REST use . . . 9
1.4.2 Startup . . . 10
1.4.2.1 Example . . . 11
1.4.3 REST API . . . 11
1.4.4 ADMIN . . . 11
1.4.4.1 ADMIN/TIME . . . 11
1.4.4.2 ADMIN/POWER . . . 12
1.4.4.3 ADMIN/LOGIN . . . 12
1.4.5 FIRENET . . . 13
1.4.5.1 FIRENET/STATUS . . . 13
1.4.5.2 FIRENET/ARM . . . 15
1.4.5.3 FIRENET/FIRE . . . 15
1.4.5.4 FIRENET/SYNC . . . 15
1.4.5.5 FIRENET/PGM . . . 15
1.4.5.6 FIRENET/DELAY . . . 15
1.4.5.7 FIRENET/VERSION . . . 16
1.4.5.8 FIRENET/CHANNEL . . . 16
1.4.5.9 FIRENET/LITERAL . . . 16
1.4.6 PROGRAM . . . 17
1.4.6.1 Introduction . . . 17
1.4.6.2 Show Directory . . . 17
1.4.6.3 FIRENET/PROGRAM/LIST . . . 18
1.4.6.4 FIRENET/PROGRAM/SET . . . 18
1.4.6.5 FIRENET/PROGRAM/PGM . . . 19
1.4.6.6 FIRENET/PROGRAM/START . . . 19
1.4.6.7 FIRENET/PROGRAM/STATUS . . . 19
1.4.6.8 FIRENET/PROGRAM/ABORT . . . 20

2 Lua Extension Classes **20**

2.1 RESTful Interface CLASS 20

2.1.1 Introduction 20

2.1.2 Class Operation 20

2.1.3 Class Methods - Server Control 21

2.1.3.1 http.start() 21

2.1.3.2 http.stop() 21

2.1.3.3 http.lock() 22

2.1.3.4 http.open() 22

2.1.3.5 http.close() 22

2.1.3.6 http.url(h) 23

2.1.3.7 http.data() 23

2.1.3.8 parsers.json() 24

2.2 Timer CLASS 24

2.2.1 Timer Class Methods 24

2.2.1.1 timer.sleep() 24

2.2.2 Timer Constructors/Destructors 25

2.2.2.1 timer.new() 25

2.2.2.2 timer.delete() 25

2.2.3 Timer Methods 25

2.2.3.1 timer.start() 25

2.2.3.2 timer.done() 25

2.2.3.3 timer.read() 26

2.3 Firenet Class 26

2.3.1 Data Format 26

2.3.2 Creators/Destructors 26

2.3.2.1 firenet.new() 26

2.3.2.2 firenet.delete() 26

2.3.3 Methods 27

2.3.3.1 firenet.read() 27

2.3.3.2 firenet.write() 27

2.4 Keyboard Class 27
2.4.1 Class Methods 27
2.4.1.1 kbd.prep() 27
2.4.1.2 kbd.close() 28
2.4.1.3 kbd.getc() 28
2.4.2 Use 28

3 Lua Support Code (firenet_sup.lua) 29
3.1 Introduction 29
3.2 Lua Globals 29
3.2.1 BCAST_ADDR 29
3.2.2 FNET_MAP 29
3.2.3 OS_MUSIC_PLAYER 29
3.3 Network Support Routines 30
3.3.1 build_fnet_status() 30
3.4 Misc Support Routines 30
3.4.1 play_file() 30
3.4.2 play_file_stop() 30

4 FIRENET Node Design 31

5 Firenet Network Design 31
5.1 Physical Layer 31
5.2 Data Link Layer 32
5.2.1 Packet Format 32
5.2.2 Addressing 32
5.3 Application Layer 32
5.3.1 Commands 33
5.3.1.1 ARM Command 33
5.3.1.2 FIRE Command 33
5.3.1.3 DELAY FIRE Command 34
5.3.1.4 PGM EVENT Command 34
5.3.1.5 STATUS Command 34

5.3.1.6 WHO (are you) Command 34
5.3.1.7 TIME Command 35
5.3.1.8 SYNCHRONIZE Node Clocks 35
5.3.1.9 VERSION of Node code 36
5.3.1.10 CHANNEL information 36
5.3.1.11 TEST MODE Command 37
5.3.1.12 BACKOFF Command 37
5.3.1.13 RESTART Command 38
5.3.1.14 REQUEST Data Command 38

6 Firenet Hardware **38**

6.1 Introduction 38
6.2 Node Electronics 39
6.2.1 Power Supply 40
6.2.2 Network 41
6.2.3 Firing Circuit 41
6.2.3.1 Adpative firing 41
6.2.4 Processor 42
6.3 Node Hardware 42
6.3.1 Enclosure 42
6.3.2 Printed Circuit board 45

List of Figures

1 FIRENET System 2
2 Index page 3
3 Login page 3
4 Main Screen 4
5 Debug Screen 4
6 Manual Screen 5
7 Arm Screen 5
8 Ready Screen 6
9 Fire Screen 6
10 Inventory display 7
11 Automated Show 7
12 Automated Show Node Unarmed 8
13 Armed Show ready to run 8
14 Running Show 9
15 REST command structure 10
16 RS-485 Network 31
17 Packet Format 32
18 Node Command Table 33
19 Synchronized Firing 36
20 Firenet System 39
21 Node Schematic 40
22 Node Layout 42
23 Node Exterior 43
24 Node Interior 44
25 Firenet PC board 45

1 System Design

1.1 Introduction

The FIRENET system is shown in 1. There are two 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. The second computer is running a web browser where it has the FIRENET user interface.

The control computer is 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. [6]. 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 mortars firing the show.

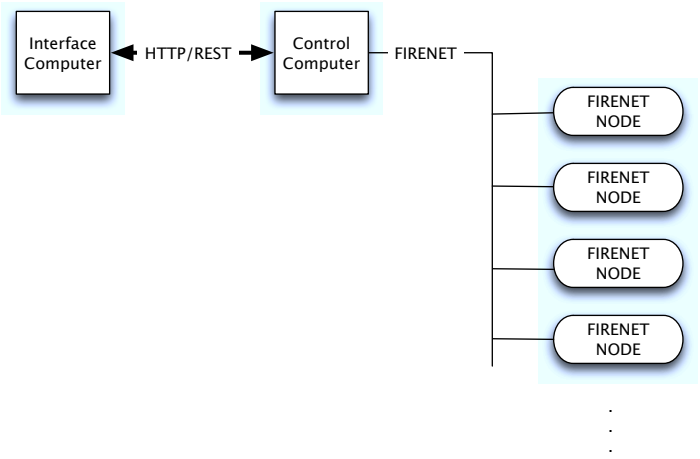

Figure 1: FIRENET System

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[5]with extensions to support FIRENET and HTTP webserving [3]. This latter acts as a Web Server and transports the REST protocol commands. Finally an HTML and 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

The user interface is an AJAX web application consisting of HTML pages and Javascript application code. In this case only a single HTML page is used and the Javascript is used to replace parts of the page with different parts of the user interface. In this case the HTML/Javascript combination is acting like an application but is running in the browser. the advantage of this method is that the interface is independent of the operating system and to a large extent the browser used. As long as the browser is current (i.e. Firefox, Safari, Chrome or IE 7) then it will support this interface on Windows, OS X or Linux.

1.3.1 Interface Operation

When the user opens the index.html page on the server he gets:

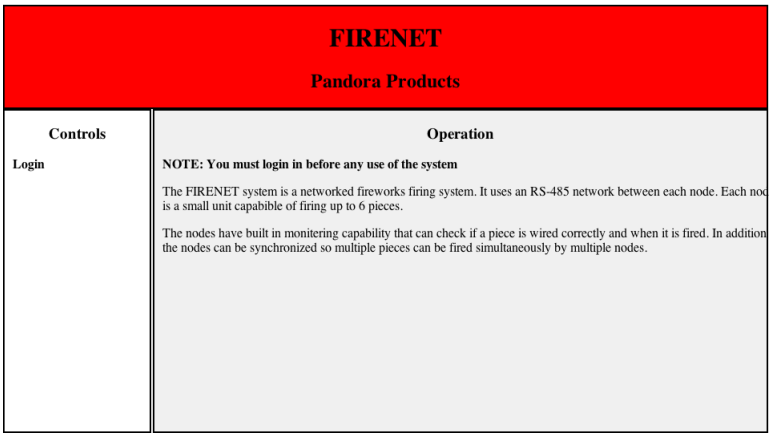

Figure 2: Index page

This instructs the user to log into the system, picking the Login control item you get:

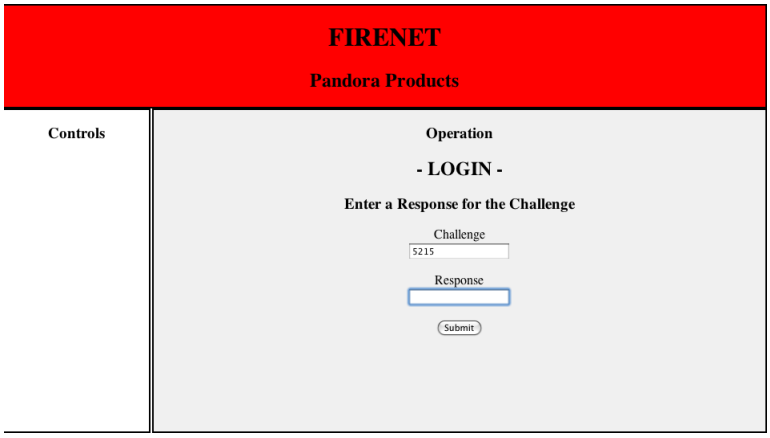

Figure 3: Login page

The user then has to enter a response to the challenge value. These challenge/response pairs are random 16 bit Hex values and the user must have this list to log in. The challenge value is found in the table and corresponding response 16 bit value is typed into the response field. If the choice is correct you then get:

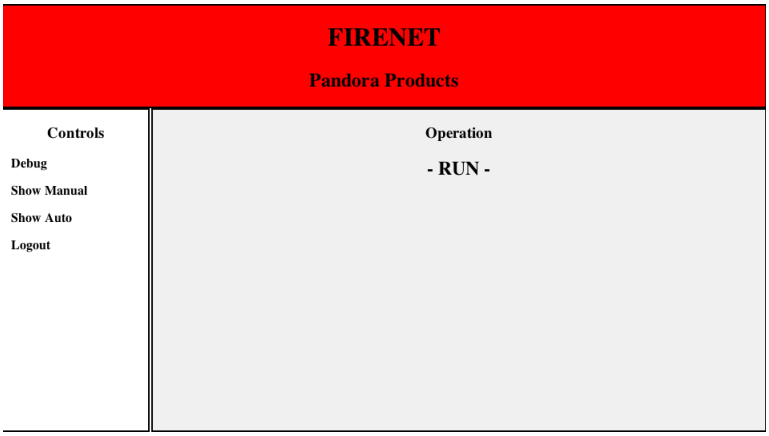

Figure 4: Main Screen

From here the user can pick any of the Control values on the left. Picking Debug gets this:

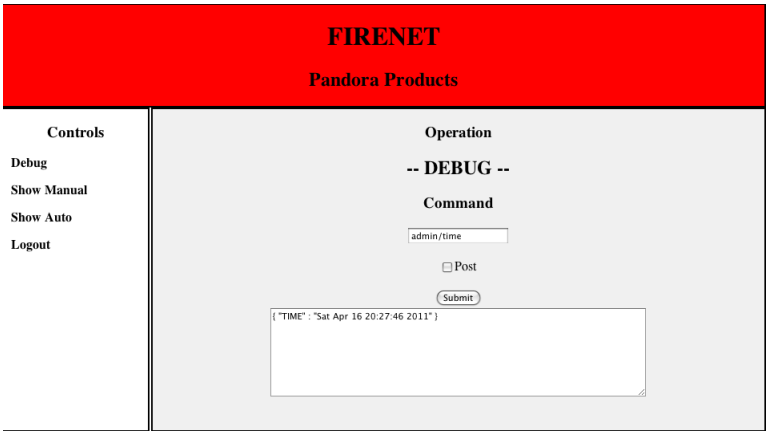

Figure 5: Debug Screen

The user can type any RESTful command (see 1.4 on page 9), press Submit and the results are returned. This is useful in debugging system problems. Picking Show Manual allows manual operation of the firing boxes as:

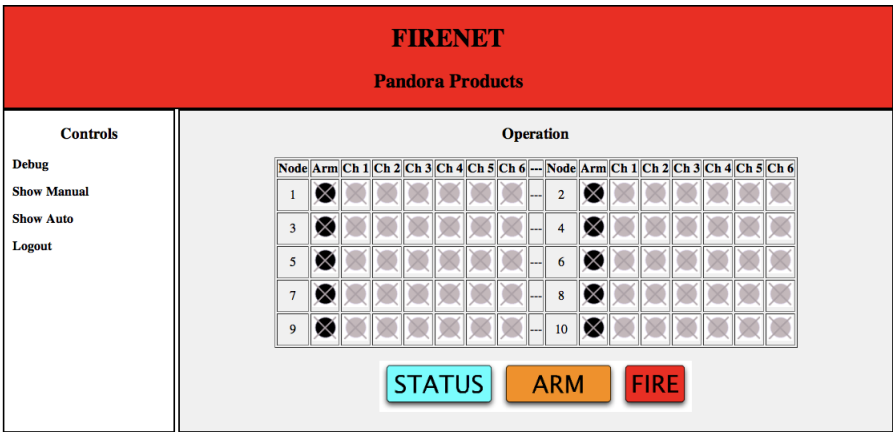

Figure 6: Manual Screen

Here the user is shown all the available firing nodes and their state. In this case all are not ARMed so the status of the channels are unknown. Picking the ARM button will turn all the boxes to ARMed and then show the channel status (i.e. is there something connected).

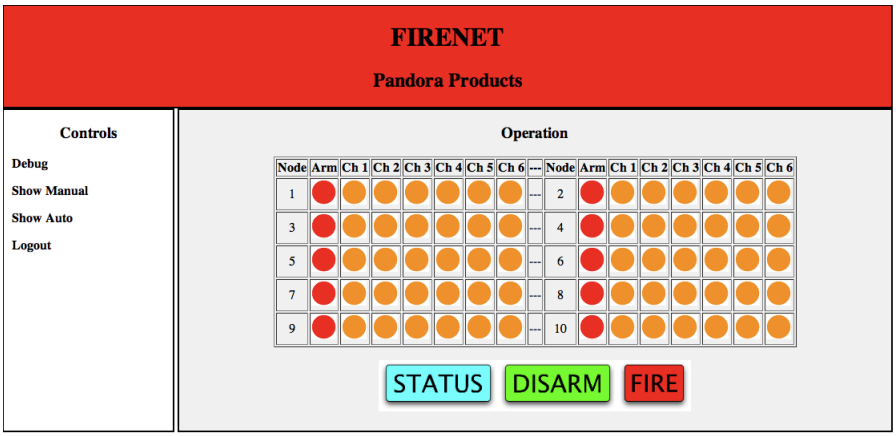

Figure 7: Arm Screen

This now shows pieces connected on all channels and all are ready to fire. You then can pick the pices to fire (note only 1/node) and they then show ready to fire.

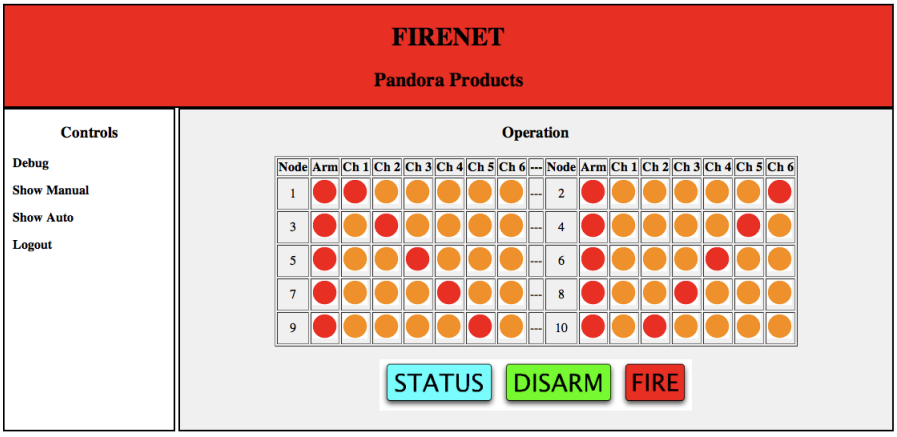

Figure 8: Ready Screen

If you press the FIRE button these pieces will be fired and their status will change:

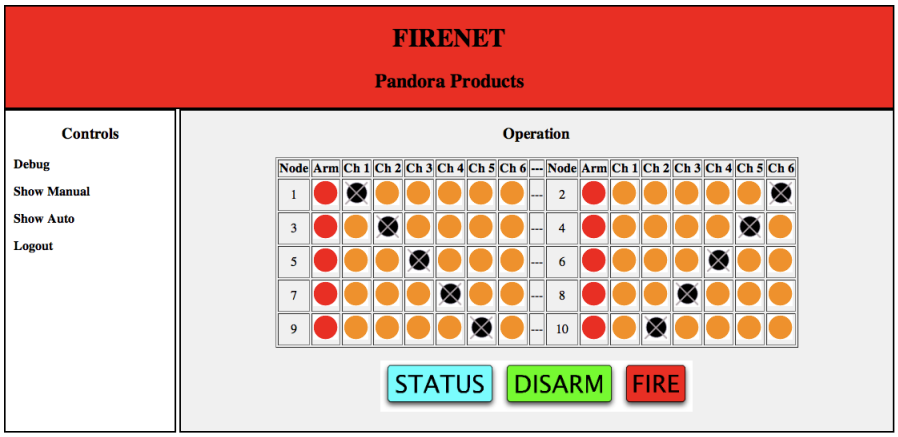

Figure 9: Fire Screen

An inventory screen is avaliable to show the firing boxes in the network, their software version and status of EEPROM values. This is useful to see make sure all boxes have the same settings for a show.

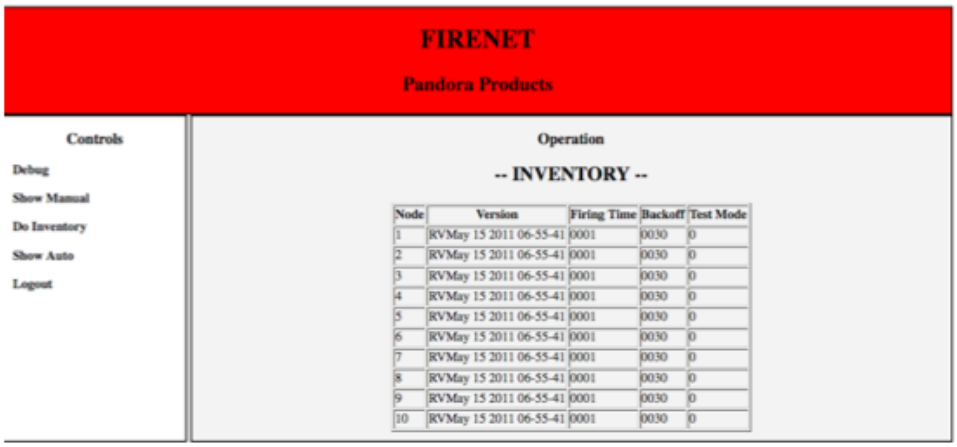

Figure 10: Inventory display

When the Automated show choice is entered the user can pick the store show.

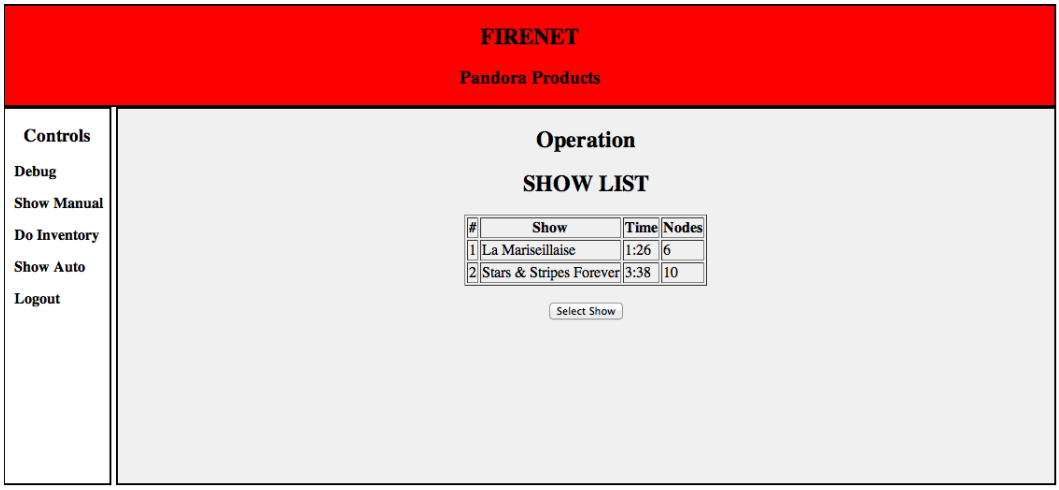

Figure 11: Automated Show

Once the show is picked then the nodes are shown like the Manual show view.

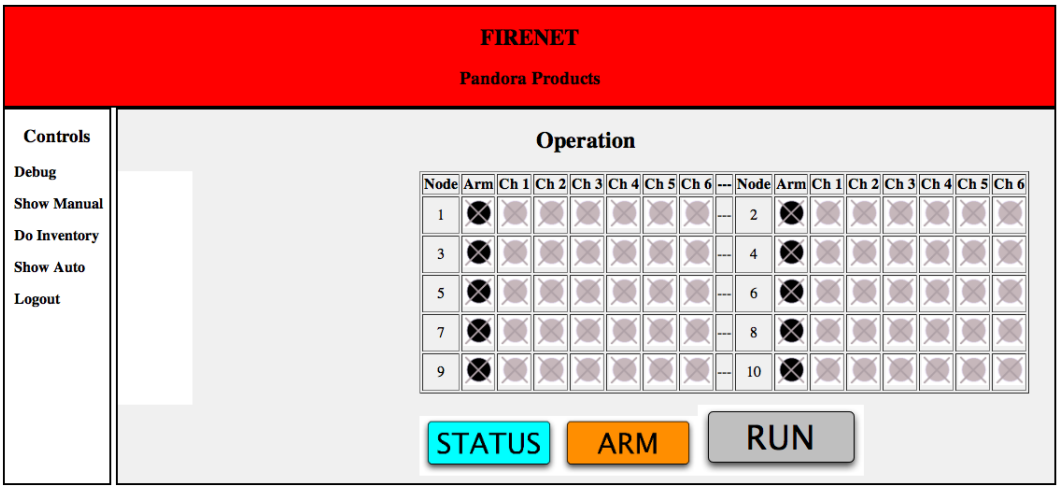

Figure 12: Automated Show Node Unarmed

After arming the show is ready to start

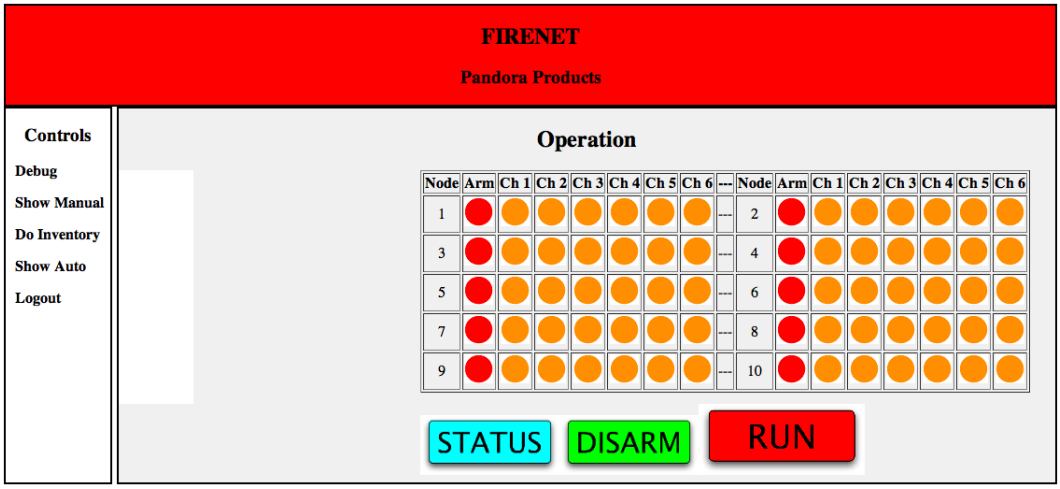

Figure 13: Armed Show ready to run

once the show is running then an indicator shows how much has been run

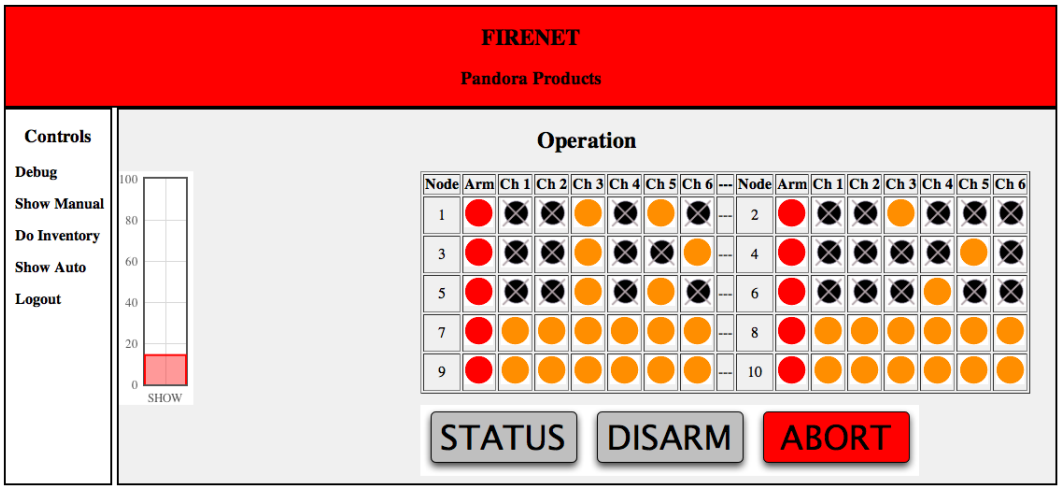

Figure 14: Running Show

Note the large ABORT button that will stop the show instantly.

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.

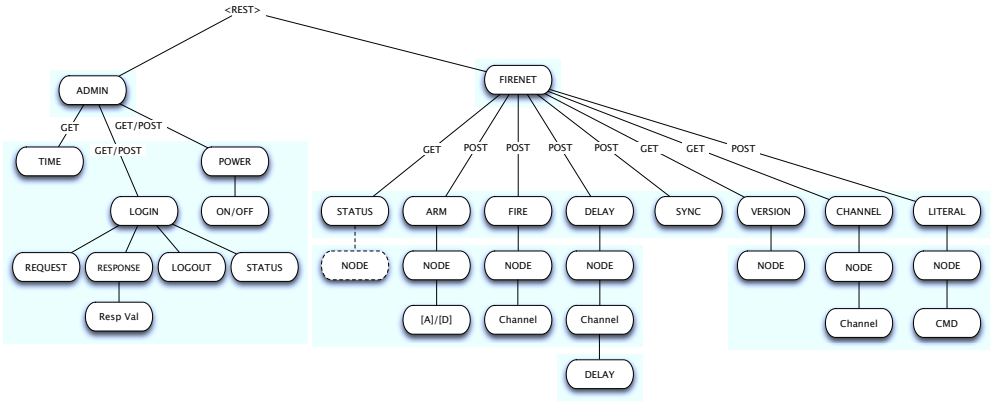

Figure 15: REST command structure

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 21) 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 Startup

The REST operation is controlled by both HTTP pages (html & javascript) and a set of Lua scripts. When started the Firenet main program reads both of these sets of files to produce the desired actions. The command line options are:

CommandLine Options

-p # Port # of serial port connected to Firenet

-c <REST Lua main code>

-rest <Directory of Lua REST code>

-http <Directory of HTML & Javascript code, location of index.html>

-show <Directory of Automated show files>

-log <File> [Optional] Log file of Firenet operations during the run

1.4.2.1 Example

```
./Firenet -p 1 -c Site/Rest/main.lua -http Site/Site_J -show Shows -log log.txt
```

Here the Firenet is started using device 1 as the Firenet network connection. The Lua file main.lua is run as the initial script, the HTML is stored in Site/Site_J and the show files are stored in the directory Shows. The initial connection to the server would be:

```
http://<server IP>:8081/Site_J/index.html
```

1.4.3 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 21).

1.4.4 ADMIN

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

1.4.4.1 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" }
```

1.4.4.2 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" }
```

1.4.4.3 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 2.1.3.3 on page 22). 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).

ADMIN/LOGIN/REQUEST

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” }

1.4.5 FIRENET

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

1.4.5.1 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/C	GET	Entire network status & clear network map
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.

The "C" command will do the same network status for all the nodes but in additon will clear the map of any previous entries and build a new map. If a node had changed address it will still be in the old map. This "C" will ensure that none of the phantom entries will be present.

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"
        }
     }
}
```

1.4.5.2 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.

1.4.5.3 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.

1.4.5.4 FIRENET/SYNC This message is sent to all FIRENET nodes and synchronizes their millisecond clocks. This will be needed to support delayed firing commands. The optional P node on the end will start the program

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/sync/[P]	POST	All node millisecond clocks are set to 0, iP added program is started

1.4.5.5 FIRENET/PGM This is similar to the DELAY command but in this case the delay command will be stored. All FIRENET nodes have a 32 bit millisecond timer. The SYNC command zeros the timers on all nodes. Sending a PROGRAM command will store 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.When the Sync is set with a P node then the sequence of stored commands will be started

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

1.4.5.6 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

1.4.5.7 FIRENET/VERSION This message is used to get the version information on a Firenet node. When set to a specific node it returns the date time of the software version in the node.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/version/<node #>	GET	Node # must be supplied

This then returns the JSON:

```
{ "STATUS" : "XMIT", "VERSION" : "RVMar  6 2011 16-23-19"  }
```

The software was loaded on the node 6 March 2011 at 16:23:19 local time.

1.4.5.8 FIRENET/CHANNEL This command is used to read out the A/D value on a particular channle of a node.

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

This returns the JSON:

```
{ "STATUS" : "XMIT", "AD" : "0039", "DATA" : "RC00039", "FIRED" : "0" }
```

This says the AD value is 0039 (with 1023 MAX) and the data shows the message received back (see message section) and FIRED shows the status of the fired bit (this channel is not fired).

1.4.5.9 FIRENET/LITERAL This command allows any FIRENET network message to be sent and the response received. The message is sent only to the node (0 address not allowed). The Firenet response is returned.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/literal/<node #>/<MSG>	POST	Node # must be supplied

For example sending the command V (version) you send:

...firenet/literal/1/v Sends the command V to node 1 you get the JSON:

```
{ "STATUS" : "XMIT", "FROM" : "1", "DATA" : "RVMar 6 2011 16-23-19" }
```

1.4.6 PROGRAM

1.4.6.1 Introduction These are a set of command used to start and run an automated show. In this case the nodes are programmed with the firing sequences see (5.3.1.4 on page 34) and then run. These commands allow selection of a show (music and firing sequence files), loading of the nodes and running the show.

ProgramCommands

list Return a list of available shows

set Select an available show

pgm Program nodes with selected show

start Start selected show with both firing actions and music

status Readout of fired pieces during show

abort Stop show as an emergency action

1.4.6.2 Show Directory The FIRENET program has been given the directory of the available shows with the -shows <directory> command line flag. In this directory are music files and corresponding firing sequence files. Also there is a file called **shows.json** which lists the available shows in the following form:

```
{
    "shows" : [
{ "show" :  "La Mariseillaise",
  "file" :  "mariseillaise.json",
  "nodes" :  "6",
  "time" :  "1:26"
  },

  {
   "show" :  "Stars & Stripes Forever",
   "file" :  "ss_forever.json",
   "nodes" :  "10",
   "time" :  "3:38"
  }
]
}
```

Where each set of KVP's corresponds to a particular show detailing the name, show description file, # nodes required and the time of the show.

The show (e.g ss_forever.json) file for any particular show looks as follows:

```
{
    "show"       :  "Stars & Stripes Forever",
    "music"      :  "stars_stripes_forever.mp3",
    "time"       :  "3:38",
    "nodes"      :  "10",
    "program"    : [
{ "node"  :  " 1",  "ch"  :  "1",  "delay"  :  "1000"},
{ "node"  :  " 1",  "ch"  :  "2",  "delay"  :  "2000"},
{ "node"  :  " 1",  "ch"  :  "3",  "delay"  :  "3000"},
                        :
                        :
    ]
}
```

Where the music file (in the same directory) and the programming commands for the show are stored.

1.4.6.3 FIRENET/PROGRAM/LIST This command is used to return a list of available shows to the user. The FIRENET program has been given the directory containing the show files and it reads a file "shows.json" and returns that data to the requestor. It returns the JSON from the shows.json file (see above)

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/program/list	POST	No data, only one referant

The returned JSON data will exactly match the **shows.json** file from the designated shows directory.

1.4.6.4 FIRENET/PROGRAM/SET This command is used to select the show to be run. After receiving the list the user sends back a name from the list of available shows and that show's data is read. This will also program all the nodes with the show's data and queue up the music file.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/program/set/#	POST	Show position in the list

The response is a JSON status response:

```
{
    "STATUS" : "OK"   <-- If successful, other responses if the number is not correct
}
```

1.4.6.5 FIRENET/PROGRAM/PGM This command will program all the connected firenet nodes with the show times as given in the particular show file. This will also check that all nodes have been programmed correctly.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/program/set/#	POST	Show position in the list

The response is a JSON status response:

```
{

    "STATUS" : "OK"   <-- If successful, other responses nodes were not programmed

}
```

1.4.6.6 FIRENET/PROGRAM/START This is the command to start the show. If the show has been SET then this will start the sync the nodes, start the music file and the programmed sequence in the nodes. The user has to have armed the nodes before sending this command.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/program/PGM	POST	Will return status message

```
{

    "STATUS" : "OK"   <-- If successful, other responses if the number is not correct

}
```

1.4.6.7 FIRENET/PROGRAM/STATUS This command is used to return status during the show. It will tell if the show is still running and what nodes have fired since the last status message.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/program/status	POST	Will return status message

```
{
   "STATUS"  : "OK",     <-- Other values "eos" or "abort"
     "FIRED" : [

        { "FROM" : "<Node #>", "CH" : "<Fired Channel>" },<-- There will be 1 or more
           :
           :
    ]

}
```

1.4.6.8 FIRENET/PROGRAM/ABORT This command is used to immediately stop a show. All nodes are reset and the music stopped. It returns a status message.

Request URI	TYPE	DATA
http://<server>/<rest>/firenet/program/status	POST	Will return status message

```
{

    "STATUS" : "OK"   <-- If successful, other responses if problem

}
```

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. 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.

2.1.3.6 http.url(h) This method is used to get the URI 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 #/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.

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 33)

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 33)

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 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 node address is programmed into the EEPROM of the processor and each node box has the address marked on the outside in **large** numbers.

The nodes are controlled by a laptop on the network running software that allows either manual or scripting firing sequences. 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.

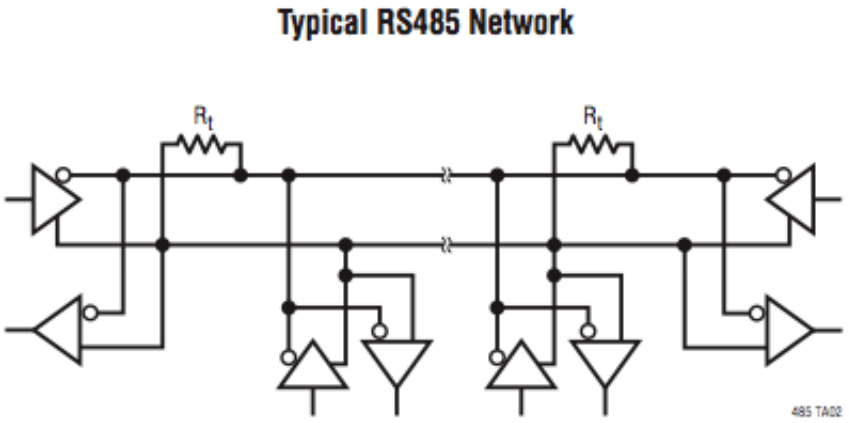

Figure 16: RS-485 Network

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 17: 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 command data	F1
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 30. The addresses are fixed in the devices and there should be no duplicate addresses in the network. The address 30 is the command node. The address 0 is the broadcast address and all devices will receive messages sent to this address.

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	[P]	Sync all node clocks, if P start firing queue
R	REPLY	R+ Last cmd letter	Used to ACK no-reply messages
V	VERSION	RV__DATE__ __TIME__	Creation time of Node code
C	CHANNEL	<Ckt #>	Return firing status & A/D of channel
M	TEST	0/1	Turn test mode ON (1)/OFF (0)
B	BACKOFF	# Backoff time (0-255)	Modify backoff time
X	RESTART	NONE	Restart noad software
R	GET_DATA	[<Data request>]	Return requested data
P	PGM	<Ckt #><Ms to fire>	Put event into firing queue

Figure 18: Node Command Table

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

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 5.3.1.8 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 PGM EVENT Command This is similar to the DELAY FIRE but stores the command into a 6 element queue. All nodes keep a 32 bit millisecond counter. The ZP command (see 5.3.1.8 on the following page) will sync the time for all the nodes, also it will start the node cycling through the queue firing the events in time order. The available firing channels are numbered from 0-5. The command takes a single byte firing circuit value and 32 bit time value.

P316543 Store firing command for circuit 3 at 16,543 miliseconds.

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

When the node fires it returns RP<ch #> allowing tracking of the autofiring.

5.3.1.5 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.

5.3.1.6 WHO (are you) Command Is used to set or read the address of a node. If the command is sent with a parameter the node's address will be changed. If no value is sent, the node's address is not changed. In either case the response will contain the current node address. A new address 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

W No address change

Message	Data	Response Data	Meaning
Change node address	W04	RW01	Returns current node address
Get node address	W	RW01	Returns current node address

The address value must be in the range 1-31 to be accepted. NOTE: That after the command is sent the node's address will NOT change till it has been restarted. (See 5.3.1.13 on page 38)

5.3.1.7 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.

Message	Data	Response Data	Meaning
Set firing time to 1000 ms	T1000	RT100	Returns current firing time
Get firing time	T	RT100	Returns current firing time

NOTE: That after the command is sent the node's firing time will NOT change till it has been restarted. (See 5.3.1.13 on page 38)

5.3.1.8 SYNCHRONIZE Node Clocks

Z[P] 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 previous page). If P is present then the node goes into auto mode and beings firing the queue

Testing of the sync command with a script that syncs the nodes then has them fire at the same time shows the following spread in 10 runs with 3 nodes, (4,5 and 6). Then monitering the firing time of channel 0 and using that to trigger a scope, it captured each of the nodes firing. Measuring between the first firing node and the last we get the following spreads in 10 trials.

Spread (ms)

```
0.4
1.384
1.542
1.742
0.7336
0.6536
1.038
1.334
1.798
-------
```

Mean 1.181 ms
SD 0.5000

A capture of the 10th firing run is shown In Figure 19 Trace 1 = Node 4 channel 0 firing, Trace 2 Node 5 and Trace 3 Node 6

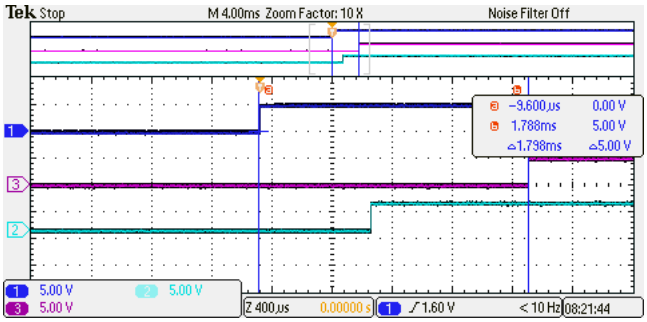

Figure 19: Synchronized Firing

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.

5.3.1.10 CHANNEL information

C Query for channel information

This message is used to inquire about a particular firing channel. The return message contains the fired status (0/1) and the A/D reading of that channel.

RC<Fire Status 0-1><A/D Value 0-1024>

5.3.1.11 TEST MODE Command

M Turn test mode ON (1) OFF (0)

This message is designed to enable and disable test mode. When the Node is placed in test mode can be used to simulate firing a show without having any ignitors attached to the firing channels and will return status to the command node as if all channels had ignitors connected. It will also show the ignitor going open on a Fire command. This allows a system test without using any of the one time use ignitors.

Message	Data	Response Data	Meaning
Enable Test mode	M1	RM0	Returns current test mode status
Get Test mode status	M	RM1	Returns current test mode status

NOTE: That after the command is sent the node's test mode will NOT change till it has been restarted. (See 5.3.1.13 on the next page)

In addition in this mode the Version message will be returned with an X in the data.

Instead of:

```
RVMar 6 2011 16-23-19
```

you get

```
RVXMar 6 2011 16-23-19
```

5.3.1.12 BACKOFF Command

B Modify backoff time

This message is used to change the timing of response messages to a broadcast message. Each node calculates when it will send its response with the forumula:

```
wait = (node addr - 1) * backoff time
```

Thus node 1 would wait 0 ms, node 2 would wait one backoff time and so on. This command is used to set the backoff time in ms. The number following the command (0-255) is stored and used as the new back off time. This allows tuning for differing networks.

```
B50 Set backoff time to 50 ms
```

Message	Data	Response Data	Meaning
Set backoff to 50 ms	B50	RB30	Returns current backoff time
Get backoff time	B	RB30	Returns current backoff time

NOTE: That after the command is sent the node's backoff time will NOT change till it has been restarted. (See 5.3.1.13 on the following page)

5.3.1.13 RESTART Command

X Restart node software

A number of commands like BACKOFF, TEST and TIME reset parameters in EEPROM but those don't immediately take effect. They only work after the node has been restarted. This command is used to do that without cycling power.

5.3.1.14 REQUEST Data Command

RF Request firing data

This command is used to extract data from a node. At present there is only one request, return the firing times after a run but further commands will be added.

RF The response will be the firing times of all 6 channels as decimal numbers separated by a space.

6 Firenet Hardware

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.

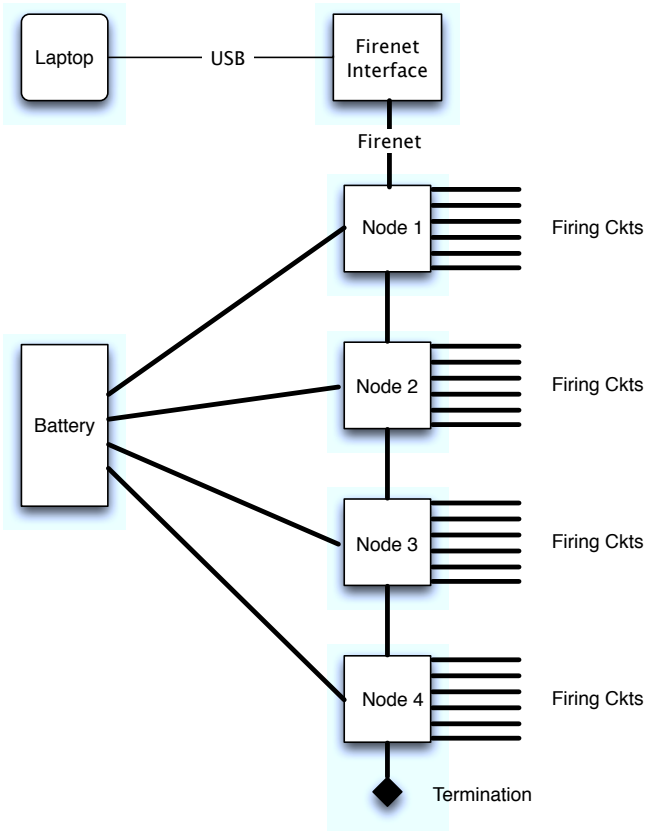

Figure 20: Firenet System

The Firenet Interface is a USB <-> RS485 interface that allows the Notebook onto the Firenet network.

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 the nodes.

6.2 Node Electronics

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 31 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:

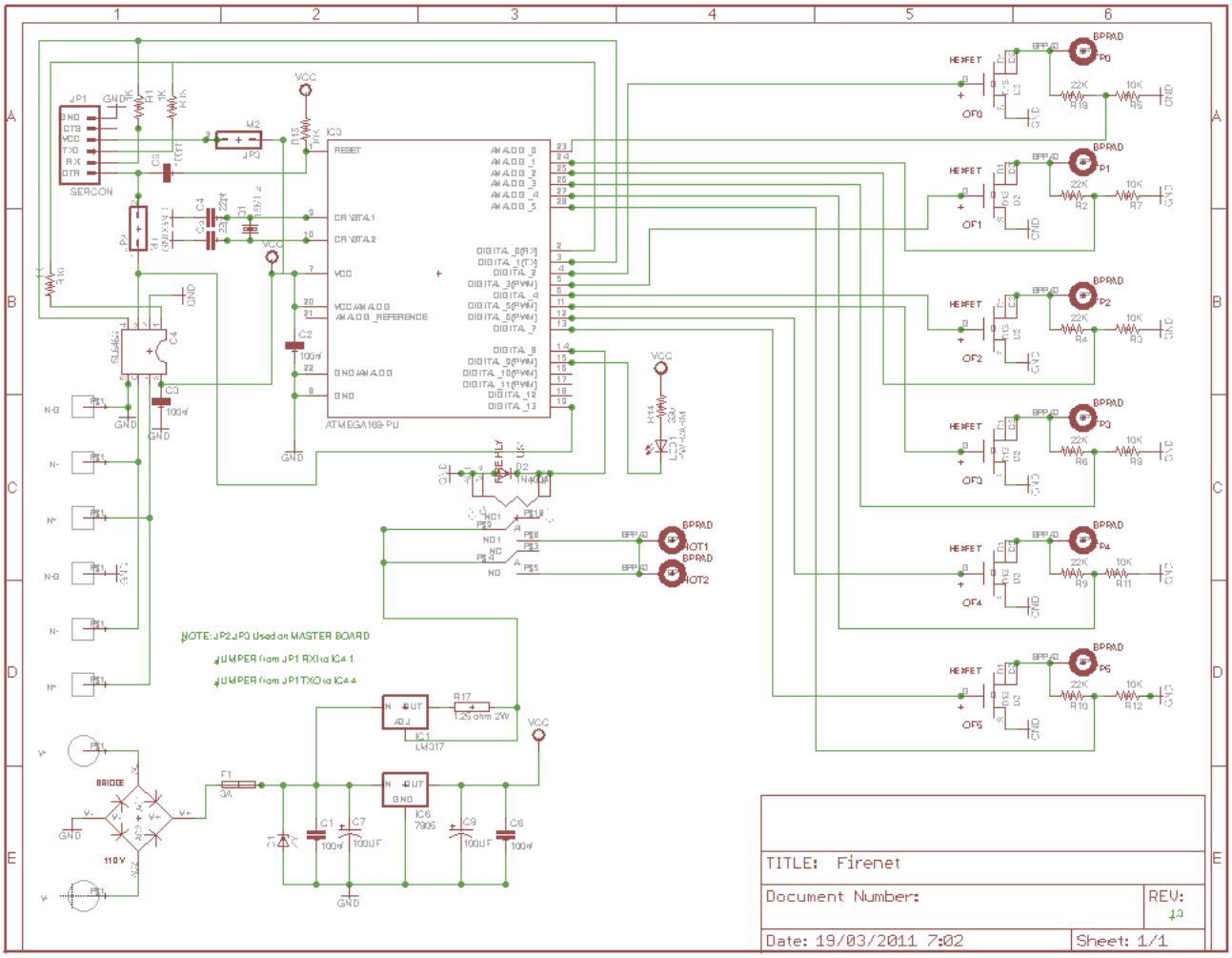

Figure 21: Node Schematic

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 31 . RS-485 networks require 120 ohm termination resistors at each end of the network. The control computer interface has one built in and the user will be responsible for putting a 120 ohm resistor plug in the last box on the daisy chain.

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 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 are 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.

6.2.3.1 Adpative firing Since the circuit can both fire the ignitor and check it's continunity we can set up the software to do adaptive firing. Ignitors can vary from lot to lot and from different manufacturers. It's simpler to have the software adapt than have the user have to change firing times. (See 5.3.1.7 on page 35). The firing time can now be set to a maximum that will guarantee that ANYTHING will be fired. We them monitor the continuity during the firing and turn off the firing channel when the ignitor blows.

We cannot measure the continuity while the FET firing transistor is ON so while the ignitor is being fired the transitor is switched off for a milisecond. The software reads the A/D for that channel and if the ignitor is blown it will show a low voltage since the ignitor binding post is no longer connected to the firing voltage (HOT1/HOT2). If the ignitor is not blow there will be a connection and the A/D will read a high voltage. Thus as soon as the ignitor is blown we can cut off the FET firing transitor and stand down from the firing state.

There are some considerations for this testing, it should be often enough to stop the firing reasonable early. That is if the max firing time is 3000 ms testing at 2000 ms intervals wouldn't be too useful in ending early. On the otherhand testing too often at say 1 ms intervals would not let the ignitor heat up as we would be turning off the firing FET at very frequent intervals. So testing was done to try to determine some reasonable values, testing at 100 ms intervals seems to be a good balance between ending early and not heating the ignitor enough.

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 Serial <-> 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). This modified board is used as the control computer interface to the network.

6.3 Node Hardware

6.3.1 Enclosure

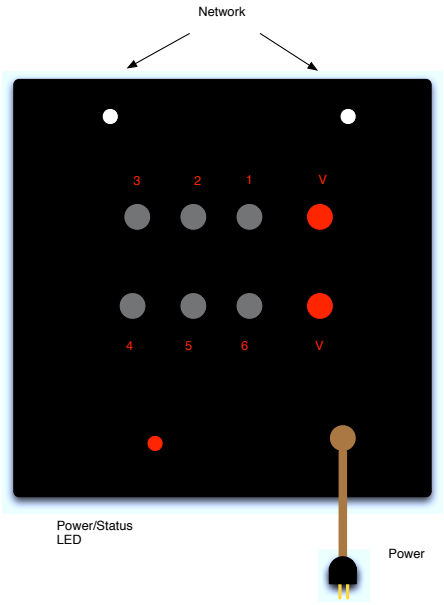

Figure 22: Node Layout

The node hardware was built into a 4.6" square box 2.65" high (Polycase DC44C). The circuit board was designed so the binding posts used for the ignitor connections also bolt the board to the top of the box.

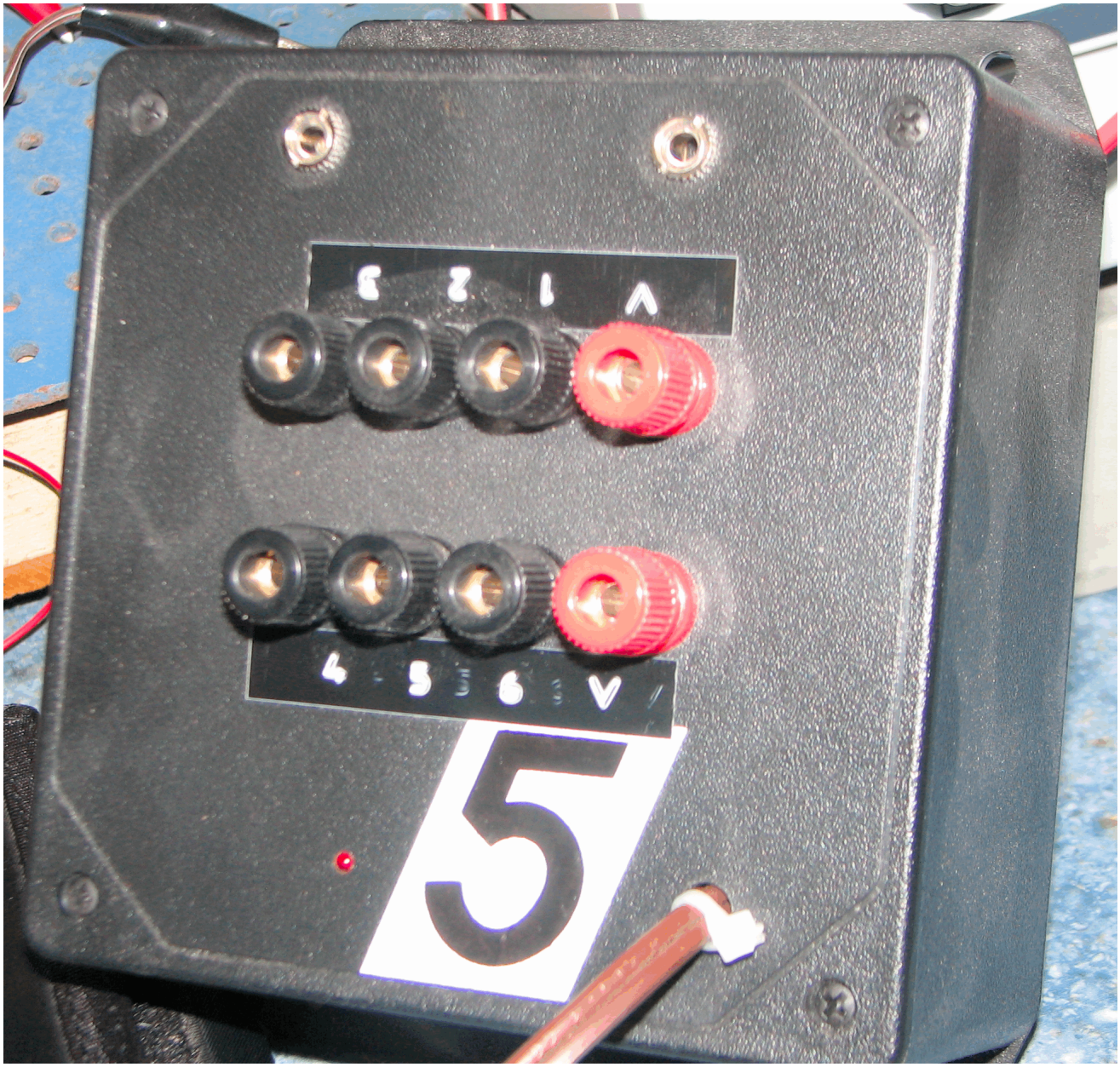

Figure 23: Node Exterior

The network connections are stereo phone jacks, one in and one out. The are electrically identical so either can be used as in or out. The Power/Status LED is a single color RED led that comes on when the processor is ready and will blink when the node is armed and ready to fire. The 6 black binding posts are the 6 firing channels, you would connect an ignitor to one of these and the other end to the RED V (or power post). Finally the power lead is a standard 16 gauge 2 wire lamp cord leading to a regular 120V two wire plug. As explained above this is a cost saving measure to allow

the use of regular 120 power cords and extension strips to distribute the 12 battery power.

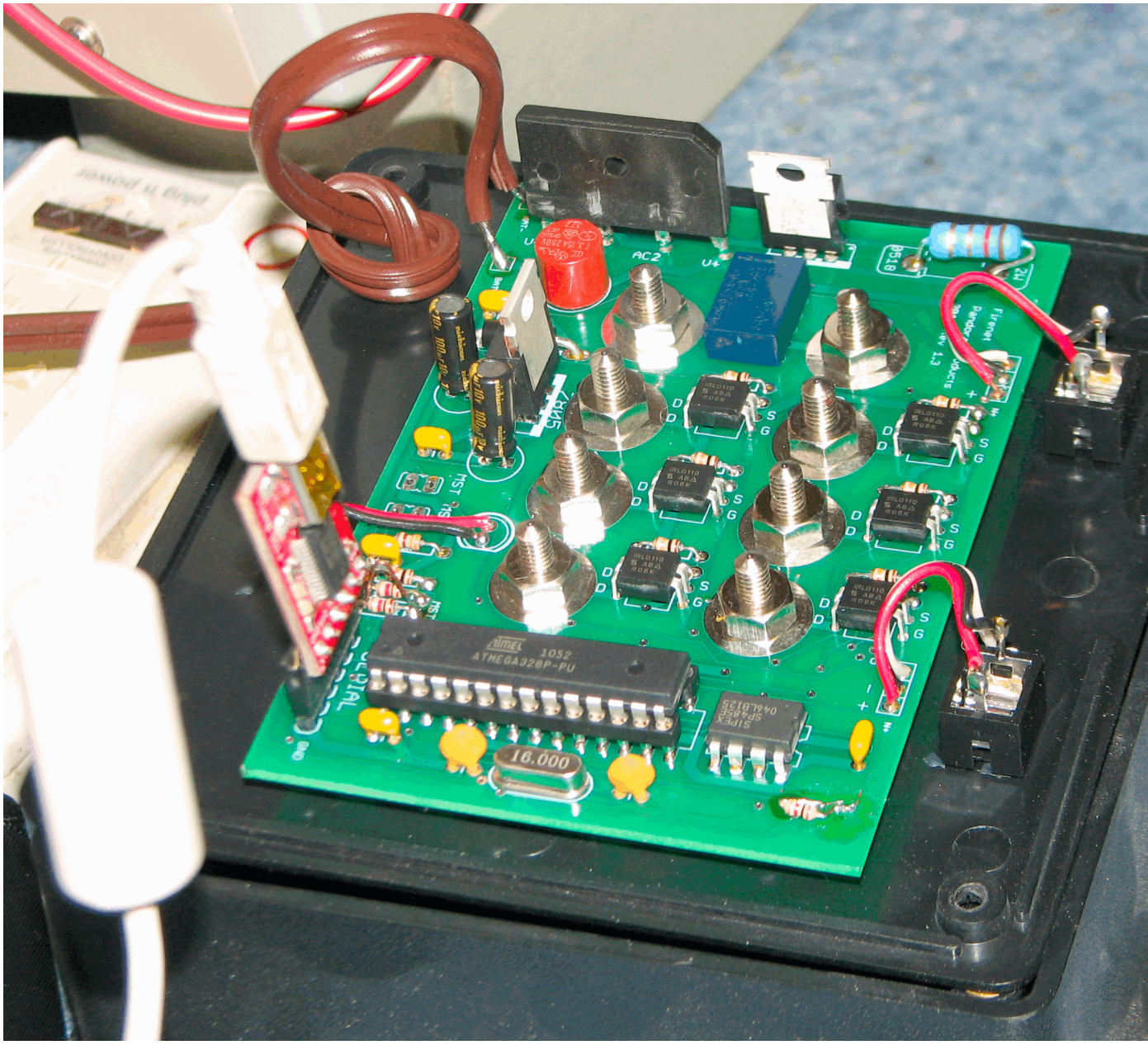

Figure 24: Node Interior

When placed in the box the back of the board is next to the lid of the box (see above) and thus all the parts are facing down into the box. This is handy taking the lid off the box exposes all the circuit and parts for easy debugging. As shown a Sparkfun 5V FTDI BASIC USB <-> Serial is connected to the board and this allows it to be programmed via the Arduino IDE.

6.3.2 Printed Circuit board

The printed circuit board was designed using Eagle CAD from the schematic shown above. Large pads were used as the bolting points for the binding posts used as the ignitor connectors.

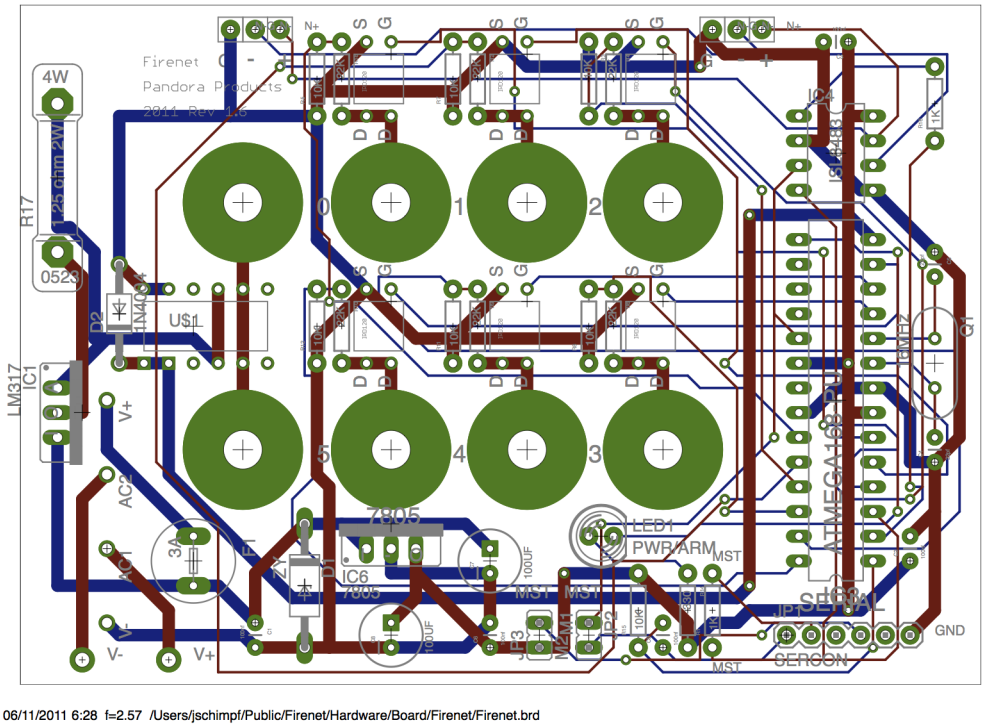

Figure 25: Firenet PC board

Index

ABORT, 20
ARM, 15, 33

BACKOFF, 37
BCAST_ADDR, 29
build_fnet_status(), 30

CHANNEL, 36

DELAY, 15
DELAY FIRE, 34

EVENTCommand, 34

FIRE, 15, 33
firenet.delete(), 26
firenet.new(), 26
firenet.read(), 27
firenet.write(), 27
FNET_MAP, 29

http.close(), 22
http.data(), 23
http.lock(), 22
http.open(), 22
http.start(), 21
http.stop(), 21
http.url(), 23

kbd.close(), 28
kbd.getc(), 28
kbd.prep(), 27

LIST, 18
LOGOUT, 13

parsers.json(), 24
PGM, 15, 19
play_file(), 30
play_file_stop(), 30
POWER, 12

REQUEST, 38
REQUEST, 12
RESPONSE, 12
RESTART, 38

SET, 18
START, 19
STATUS, 13, 19, 34
STATUS, 12
SYNC, 15
SYNCHRONIZE, 35

TEST MODE, 37
TIME, 11, 35
timer.delete(), 25
timer.done(), 25
timer.new(), 25
timer.read(), 26
timer.sleep(), 24
timer.start(), 25

VERSION, 36

WHO, 34

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

[6] Wikipedia. Representational state transfer.