EE 403W

John Darling, Thadoe Swan-Yi, Michael Ross

Spring 2016

Executive Summary

This report contains a detailed summary of the development of the Raspberry Jam home monitoring system, designed for our Capstone Project at the Pennsylvania State University. Included in this document are our thought process during development, segments of relevant code, and the implementation of specific features. All images are labeled with Figure footnotes for your reference. The final design of the home monitoring system functions as intended with the exception of a couple features.

Problem Statement

Home monitoring systems on the market today are both expensive to purchase as well as costly to upkeep requiring monthly payments for the service. Because of the high costs of a home monitoring system, this leaves many individuals without the ability to keep watch on their residence while they are away, making them prime targets for burglary or vandalism. In modern times, it is unacceptable that there are not more affordable alternatives in the home monitoring market.

Proposed Solution

Our proposal to solve this problem utilizes the Raspberry Pi Model B to manage a hardware system that is less expensive to create than the current alternatives and free from any monthly payments while still maintaining much of the core functionality of the most popular systems including: live video feed, infrared alarm triggering, and notifications sent to the user when the alarm is set off. The Raspberry Pi Model B is incredibly affordable and compact, yet it is remarkably versatile. To avoid any monthly payments there will be an individual, independent server for each unit hosted directly on each Pi. The video feed will be output to a website hosted on the Raspberry Pi itself. To set up an alarm, we will connect the Raspberry Pi to an infrared sensor that when triggered will send a signal to the Pi, which will in turn output a PWM via one of its GPIO pins to a speaker when the alarm is armed. When the alarm is triggered the user will be sent an E-Mail notification stating that there may be an intruder. The entirety of the system will be set up by the user so there will be no need to pay for someone to come to the owners home to install it.

Desired Specifications

In the table below are the specifications we hoped to achieve from the onset of the development of the Raspberry Jam home monitoring system. The majority of which we were successfully able to reach.

Feature	Specifications
Size and weight	120 cubic inches; 3 lbs
Video Output	1080p continuous video stream
Power Consumption	0.08 kWh/month if active at all times
Motion Sensor Range (Maximum room size)	20 feet
Connectivity	Any device with an internet connection and HTML5 capable web browser
Hardware Add-ons	Temperature Sensor, Light sensor

Figure 1 - Desired Specifications for our System

We wanted it to be small and light enough to be put just about anywhere including mounting it on a wall if the user desires. We initially aimed for the resolution of the video feed to reach 1080p, widely considered the modern standard for various forms of media. The system will be plugged into a typical 120V wall outlet, so it needs to be relatively inexpensive to power at all times. The infrared sensor should be able to detect movement from at least 20 feet away with a wide range of detection, up to 180 degrees. The user will be able to access their video feed and system via any device with an internet connection as well as a browser capable of utilizing HTML5. The system should include sensors to monitor temperature and light levels. With all of these the user will be able to completely monitor the status of their home from anywhere in the world that they may be by accessing a unique URL address of their website.

Implementation Details

Hardware Architecture: The biggest portion of hardware we had to implement came from the alarm system. This included a relay that would switch when the alarm is set to armed from our website. It switches between the alarm which will sound when the motion sensor is tripped, and silent mode where an LED is turned on when the motion sensor is tripped. By using a BJT in tandem with the relay, we could control the base voltage of the BJT via a GPIO pin. When the base voltage exceeds a threshold, current can flow between the emitter and collector. When this happens, the relay sees 5V across its terminals and this causes the relay to mechanically switch. You can see from our circuit that in the off position, the pwm signal travels to the LED. When on, the signal travels to the speaker.

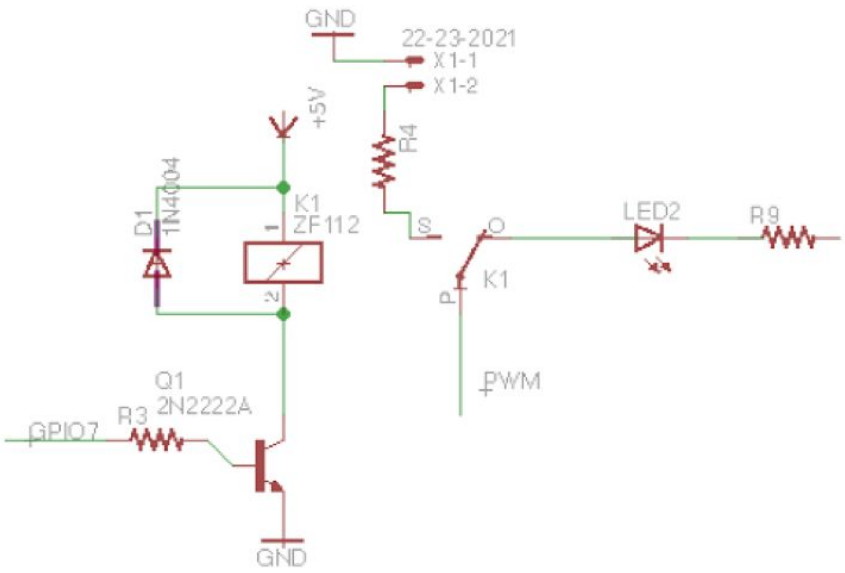

Figure 2 - Alarm Circuit

The Raspberry Pi does not come with analog inputs so the analog readings we get from multiple components must be converted to digital signals before going to the Pi. We chose a MAX11645 ADC and connected it to the 3.3 V line from the Pi to power it. The MAX11645 is compatible with both SPI and I^2C BUS protocols, however, we chose to use I^2C protocol so that we were not using as many GPIO pins. The serial clock and serial data lines are directly connected to that of the Pi which serves as the master of the ADC. The light and temperature sensors will be read by the Pi via the ADC. We also set our reference voltage to be 2V. Using a smaller reference voltage meant that we could not detect as small of temperature changes, but we only needed 1 degree celsius of resolution, so this did not matter. In the actual implementation of this ADC, we did not include pull-up resistors between the SDA and SCL lines and the 3.3V rail. Because we forgot to include these resistors, we were not able to read accurate data from the ADC, and therefore could not include temperature sensing as a feature of our monitoring system.

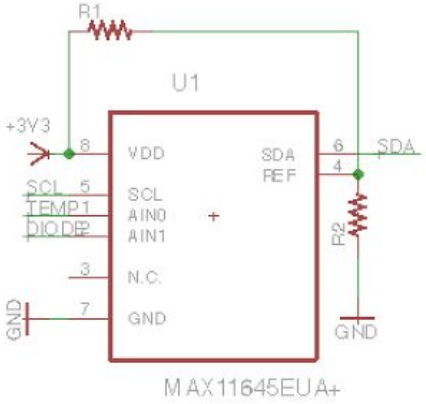

Figure 3 - ADC Circuit

Software Architecture: Our software design turned out to be fairly streamlined. We have a total of only 2 Python scripts running on the Pi and one shell script that initializes our videostream. We decided on having 2 separate python scripts so that we could debug each piece of software separately. For future implementation, we would have ultimately made these files into a single executable python script and added this to our Pi's boot sequence. First, we will take a look at the python script that creates a websocket between the Pi (our server), and the web page (the client). Websockets are an HTML5 capability that allow for bidirectional communication between the server and the client.

```
41
42  if __name__ == "__main__":
43    http_server = tornado.httpserver.HTTPServer(application)
44    http_server.listen(8888)
45    tornado.ioloop.IOLoop.instance().start()
```

Figure 4 - Websocket Creation Code

This part of our code establishes the websocket and then 'listens' for any strings that are received over port 8888. In order for this to work, we had to open up port 8888 on our local home network. This also means that this websocket is in no way connected to our website...YET. In order to have this websocket work in conjunction with our website, we used javascript embedded in our HTML page that establishes a connection with the external ip address of our Pi over port 8888. For a more detailed explanation of the javascript code, please see our website design explanation on page 6.

```
13  class WSHandler(tornado.websocket.WebSocketHandler):
14
15    def open(self):
16      print 'user is connected.\n'
17
18    def on_message(self, message):
19          currentState = GPIO.input(relay)
20          print "Message received: {}".format(self.request.remote_ip)
21          if message == "on":
22                  if currentState == True:
23                          self.write_message("Alarm is already armed")
24                  else:
25                          GPIO.output(relay, True)
26                          self.write_message("ON")
27          elif message == "off":
28                  if currentState == False:
29                          self.write_message("Alarm is already disarmed")
30                  else:
31                          GPIO.output(relay, False)
32                          self.write_message("OFF")
33
34    def on_close(self):
35      print 'connection closed\n'
```

Figure 5 - Websocket Event Handler code

We defined a class called WSHandler, this class has several functions that are triggered by different events that occur on the client side. When the web page is opened, a message will be printed in the terminal of the server saying 'user is connected'. In order to control GPIO pins on the Pi, we used the on_message function that handles messages received from the web page via

the websocket. By using a nested if-else statement, we are able to control the GPIO pin state. The GPIO pin that we controlled from the website was connected to a BJT that controlled a relay on the board. This relay ultimately controlled whether the PWM signal was sent to an LED or to the alarm. Next, we will take a look at the python script that handled the motion sensor events and controlled the PWM output of the Pi.

```
12  # Setup PWM output
13  GPIO.setup(18, GPIO.OUT)
14  GPIO.output(18, False)
15  pwm = GPIO.PWM(18, 2400)
16
17
18  while True:
19      state = GPIO.input(sensor)
20      if state == 0:
21          GPIO.output(18, False)
22          pwm.stop()
23          time.sleep(0.1)
24      elif state == 1:
25          print "Intruder!", state
26          pwm.start(50)
27          GPIO.output(18, True)
28          time.sleep(30)
```

Figure 7 - Motion Sensor Code

Our motion sensor was connected to a GPIO pin and defined as an input. We call this pin 'sensor' here in the code. We have used a while loop to continuously sample the state of the pin to determine if the motion sensor has been tripped. When the sensor is tripped, a message is printed server side (this was useful for debugging purposes) and the pwm pin is initialized. When the sensor is tripped and is in 'state == 1', our while loop stops sampling the input to the sensor for 30 seconds. This means that the alarm will be on for 30 seconds before resetting. For our pwm settings, we used a frequency of 2400 Hz. This frequency provided a loud enough alarm since we are limited to only 3.3V on our output pin, we did not have much signal level to work with on our speaker output.

Also, embedded in our elif branch of our while loop, we have included a function that sends an email to the user in order to notify them when the alarm has been triggered. Please see Figure 8 below.

```
37          server = smtplib.SMTP("smtp.mail.yahoo.com",587)
38          server.ehlo()
39          server.starttls()
40          server.login(username,password)
41          server.sendmail(fromaddr, toaddrs, msg)
42          server.quit()
```

Figure 8 - Email Code

The camera that we used was connected to the Pi via a MIDI bus interface. To begin running the camera, all that is needed is a one line shell command. To stream the images that the camera is taking, we created a tmp folder that stores all the images that have been taken. This folder is deleted whenever the system is restarted. MJPG streamer is an application that looks for a tmp folder and continuously streams the most recent image (from its timestamp) over a specified port. For our purposes, we decided on port 8080 for MJPG streamer. Since the entire process of starting the camera, creating a temporary folder, and starting MJPG streamer requires multiple lines of shell commands, we created an executable shell script called start_stream.sh that does all of these things.

```
11  else
12      raspistill -w 900 -h 675 -q 50 -o /tmp/stream/pic.jpg -tl 100 -t 9999999 -th 0:0:0 -n >
13      echo "raspistill started"
14  fi
15
16  if pgrep mjpg_streamer > /dev/null
17  then
18      echo "mjpg_streamer already running"
19  else
20      LD_LIBRARY_PATH=/usr/local/lib mjpg_streamer -i "input_file.so -f /tmp/stream -n pic.jpg
21      echo "mjpg_streamer started"
22  fi
```

Figure 9 - Camera Startup Code

The 'raspistill' command has a number of options that we used. For one, we specified the resolution of the camera to capture at 900p. **NOTE:** *You can see here that we actually defined the capture backwards so that the capture resolution is 900x675 and not 1600x900 which is what we actually wanted.* We also defined the quality to be 50, which means that the captured images are only captured at half of the resolution but they still maintain their same aspect ratio of 16 by 9. This definition is important for our website, where we have a responsive iframe that changes dimensions based on the size of the embedded object. The framerate of the streamed video is capped at 15 fps. This limit was set based on our networks data upload rate of our network which was about 5 Mbps. Given the fact that each JPG image is a couple hundred KB, this really means that we are only able to stream about 2-3 images per second. However, if you are on a local network with the Pi, this rate improves tremendously. So that is why we chose to use 15 fps.

Raspberry Jam Website: Rather than host our website on an external server, which would add unnecessary cost to the design, we opted to host our website on the Pi using Apache. We also used a website called No-ip, which allows us to have a static URL address despite the fact that our external ip address is dynamically changed every couple of weeks by our ISP. When accessing a website, you can also type in the ip address of where the website is hosted. Our ip address is essentially masked as a URL address instead. For our purposes, this address was www.homemonitoringee403w.ddns.net. When a client types this address into the URL bar, they are actually accessing the Pi itself via Port 80. Apache handles incoming requests from port 80 and redirects the client to an index.html page if there is one. Thus, we saved our web page as index.html on our pi.

Our index.html page is our Raspberry Jam Website that functions as a GUI for the entire system. This website can be accessed by any HTML5 capable web browser. Users are able to watch live video feed, send commands to Pi and receive the data from our sensors. We designed our website so that it is easy to use on any device without affecting the quality of features available on our page. We used HTML5 as the basic framework and used CSS elements to make it aesthetically pleasing. One of main reason we chose HTML5 is that it provides a new feature called Websockets which are a bi directional communication protocol that allows data to be sent between servers and clients. **NOTE:** *Refer to Software Architecture section for more details on the websocket implementation in python*. Javascript is responsible for most of main features of website. We also included a JQuery library that gives us easier control of event handling. When

users open or refresh the page, Javascript will try to establish connection with the websocket server by looking for the external IP address of the server and port 8888 where the websocket is listening. You can see this implementation on line 24.

```
22  $(document).ready(function () {
23      //change example.com with your IP or your host
24      var ws = new WebSocket("ws://homemonitoringee403w.ddns.net:8888/ws");
25      ws.onopen = function(evt) {
26        var conn_status = document.getElementById('conn_text');
27        conn_status.innerHTML = "Connection status: Connected!"
28      };
29      ws.onmessage = function(evt) {
30        var newMessage = document.createElement('p');
31        newMessage.textContent = "Server: " + evt.data;
32        document.getElementById('messages_txt').appendChild(newMessage);
33      };
34      ws.onclose = function(evt) {
35        alert ("Connection closed");
36      };
```

Figure 10 - Javascript that Connects to Websocket Server

The alert, " The connection is closed", will pop up whenever this connection fails. All the buttons on our website are also handled by Javascript.

```
38      $('#alarm').click(function() {
39
40          if ($(this).val() == "off") {
41              $(this).val("on");
42              var message = "off";
43          }
44          else if($(this).val() == "on") {
45              $(this).val("off")
46              var message = "on";
47          }
48          ws.send(message);
49      });
50  });
```

Figure 11 - Javascript Button Handler Code

Once websocket connection is established, commands can be sent to the Pi via push buttons that we have included on the webpage. These buttons have 2 states. Each button has an ID associated with it. The code above shows a javascript event handler for when the button with the '#alarm' ID is clicked. When clicked, the value of the button is changed. This is turn changes the value of the message variable. Ultimately, this variable is sent over the websocket to the Pi and controls the state of the alarm.

```
81      <div style="background-color: #333333; padding:20px" class="col-sm-6 left">
82
83          <div class="embed-responsive embed-responsive-4by3">
84              <iframe id="highlight" class="embed-responsive-item"  src="http://homemonitoringee403w.ddns.net:8080/stream_simple.html"><
                /iframe>
85          </div>
```

Figure 12 - HTML Embedding MJPG URL

Bootstrap is the HTML and CSS framework for developing mobile projects on web. We implement some bootstrap elements which allowed us to design our website so that the web page can scale to any screen size. Lastly, the live video streaming on page is possible by using an embedded iframe redirect which allows us to embed the video stream inside of our website. This was necessary because the video stream is actually at a different URL address, namely the one in the code above on line 84 in Figure 12.

Security is one of the biggest shortcomings of our design. Since we have to open several ports on our home network, this leaves open many vulnerabilities. A malicious person might want to access the Pi's command line. To combat this threat, we installed UFW or uncomplicated firewall on the Pi, and only allowed access to the Pi via port 80 and port 8888, our web server and websocket respectively. So someone trying to access the pi over port 22 using SSH would be unable to do so.

Conclusion and Results

Overall, we successfully developed a system that met our primary goals in the 4 months of the development period. Our system has a video feed linked to a website that is viewable from any device connected to the internet on an HTML5 capable browser. The alarm sounds when the IR sensor is tripped, and E-Mail notifications are automatically sent to the user when the sensor detects movement. Some areas which we fell short of our goals were the video feed, only reaching 900p video streaming and frame rates of 15 due to limitations of MJPEG and the upload speed of home network. If we had used a more capable video compression technique such as H.264 advanced video streaming codex, we could have easily reached this goal. We also could not get I^2C to correctly read the temperature and light levels. We believe this is due a lack of pullup resistors on the portion of the circuit containing the ADC. With the core functions of our design working as intended, adding in these secondary features would be simple enough given more time to continue working on the project.

All documents and codes of the system design can be found on our github page: https://github.com/MichaelRossPSU/JAM.git

Appendices

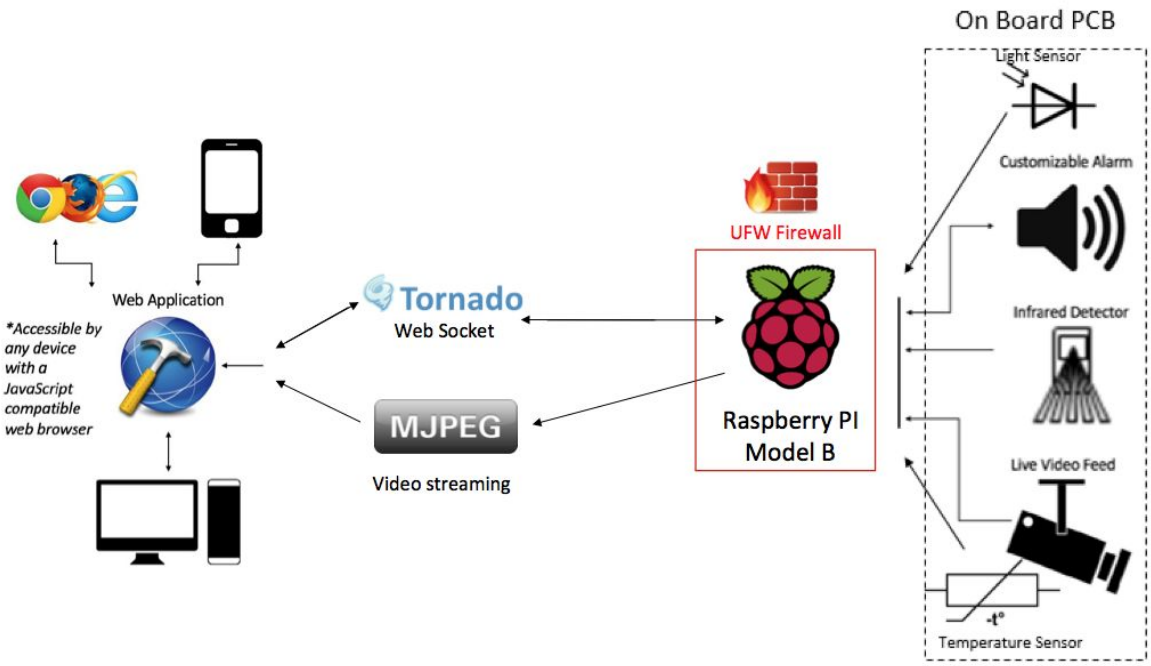

Figure A - Design Flow

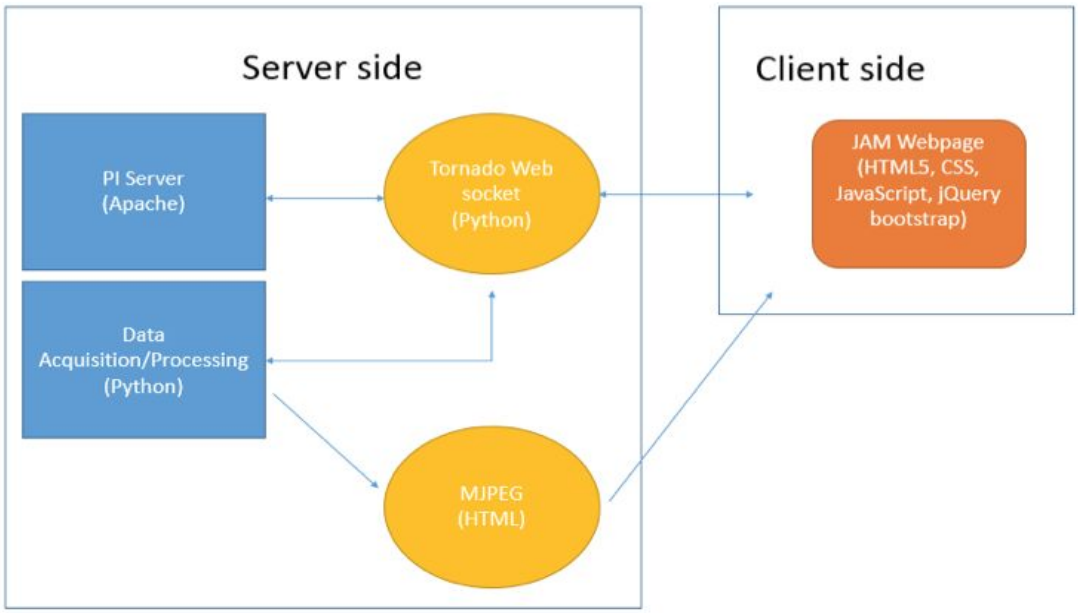

Figure B - Software Flow

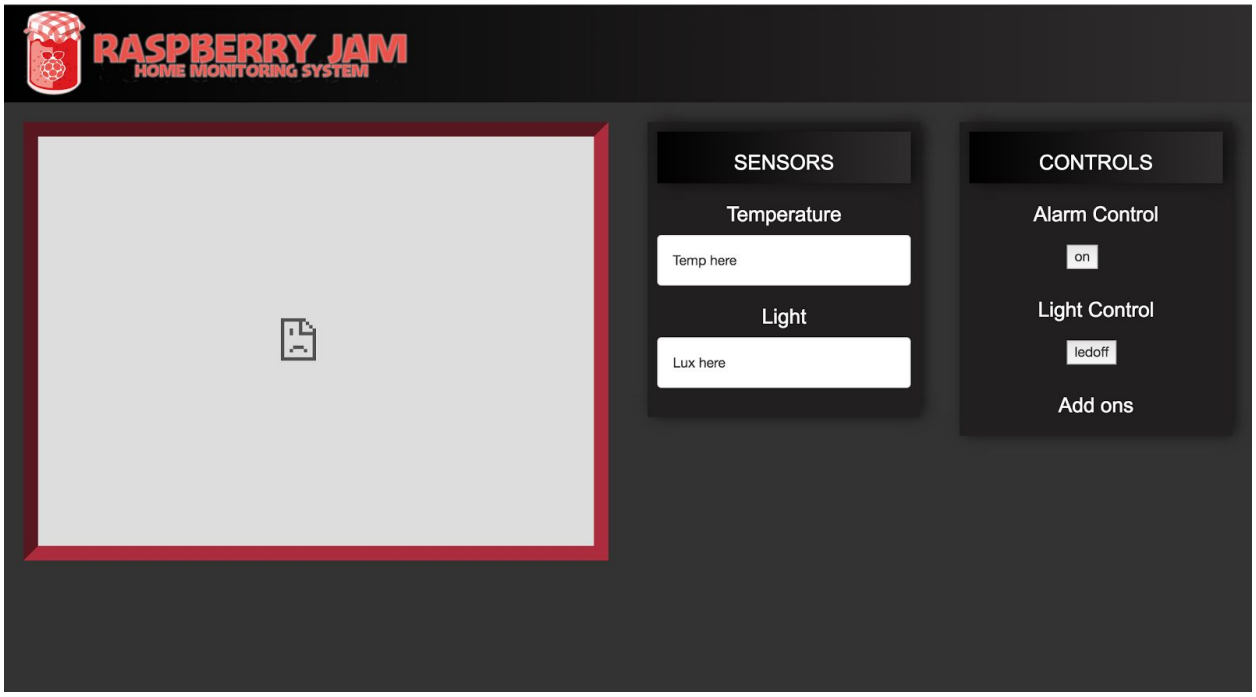

Figure C - Website Screenshot using Chrome on OS X

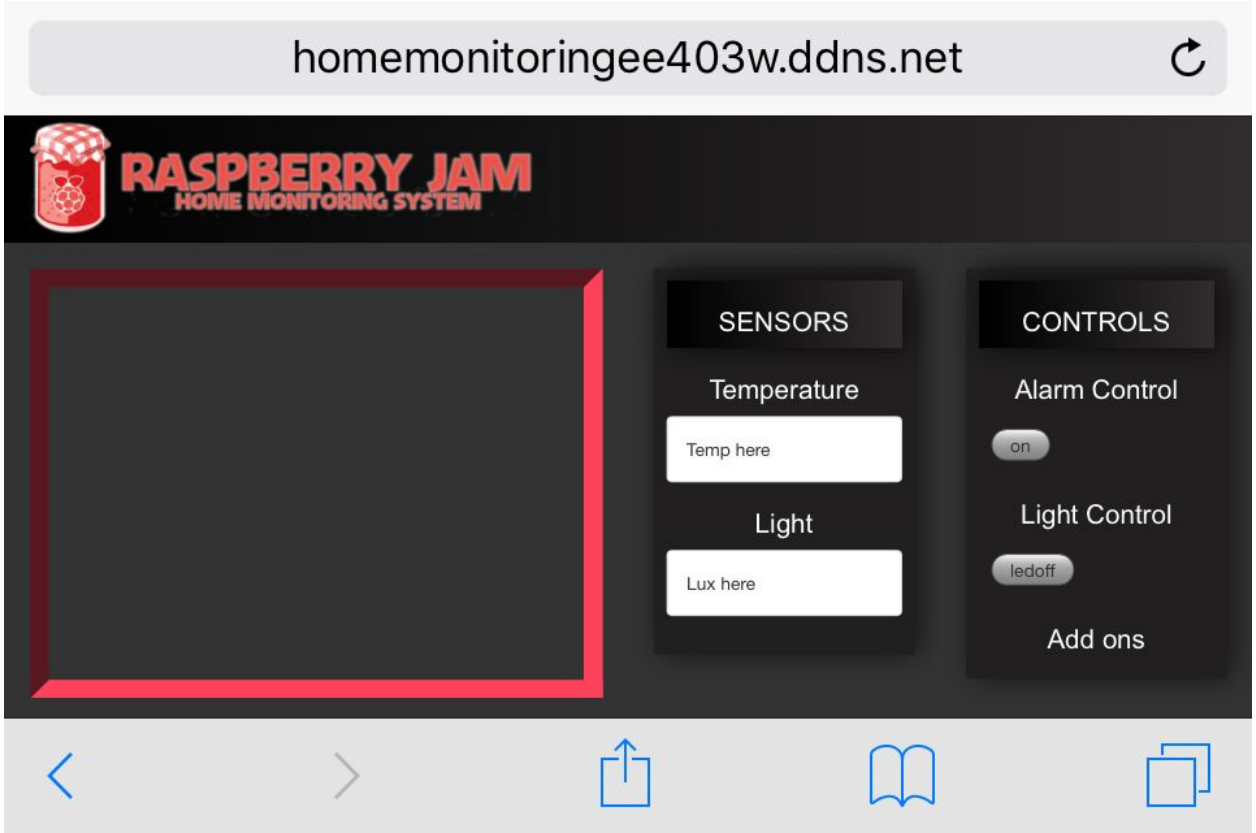

Figure D - Website Screenshot using Safari on iOS

Componenet	Cost
Raspberry Pi Model B	$25.00
Raspberry Pi Camera Board	$15.99
PIR Sensor	$9.99
2N2222A BJT	$0.39
JQC-3F Relay	$0.60
LM358 Op Amp	$0.15
MAX11645 ADC	$1.24
LM335Z Temp Sensor	$0.55
SFH229 Photodiode	$0.27
1N4004 Diode	$0.01
Speaker	$1.98
Other Bulk Components	$1.00
Total:	$57.17

**Note: Could not find bulk prices for Pi, Camera, and IR Sensor, so they might be lower*

Figure E - Bill of Materials

References:

- *Adafruit for Pi and infrared sensor: https://www.adafruit.com/*
- *Digitkeys for the hardwares: http://www.digikey.com/*
- *Osh park for PCB board: https://oshpark.com/*
- *Eagle CadSoft for PCB designs: http://www.cadsoftusa.com/download-eagle/*
- *Apache server documentations: https://www.raspberrypi.org/documentation/remote-access/web-server/apache.md*
- *JQuery library: https://jquery.com/*
- *Bootstrap framework: http://getbootstrap.com/*
- *HTML5 resources: http://www.w3schools.com/html/html5_intro.asp*
- *CSS resources: http://www.w3schools.com/css/default.asp*
- *MJPEG streaming: https://en.wikipedia.org/wiki/Motion_JPEG*
- *Websockets documentations: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API*
- *Tornado websocket: http://www.tornadoweb.org/en/stable/websocket.html*
- *Uncomplicated firewall: https://wiki.ubuntu.com/UncomplicatedFirewall*
- *Python resources: https://www.python.org/*