Home Automation with Raspberry Pi

How-to project: time the lights in your home and control them remotely over the Internet.

BHARATH BHUSHAN LOHRAY

The Raspberry Pi has been very popular among hobbyists and educators ever since its launch in 2011. The Raspberry Pi is a credit-card-sized single-board computer with a Broadcom BCM 2835 SoC, 256MB to 512MB of RAM, USB ports, GPIO pins, Ethernet, HDMI out, camera header and an SD card slot. The most attractive aspects of the Raspberry Pi are its low cost of $35 and large user community following.

The Pi has several pre-built images for various applications (http://www.raspberrypi.org/downloads), such as the Debian-based Raspbian, XBMC-based (now known as Kodi) RASPBMC, OpenELEC-based Plex Player, Ubuntu Core, RISC OS and more. The NOOBS (New Out Of the Box Setup) image presents a user-friendly menu to select and install any of the several distributions and subsequently boot into any of the installed OSes. The Raspbian image comes with the Wolfram language as part of the setup.

Since its initial launch in February 2011, the Raspberry Pi has been revised four times, each time receiving upgrades but maintaining the steady price of $35. The newest release of the Pi (the Raspberry Pi 2) boasts a 900MHz quad core cortex A7 and 1GB of RAM. Moreover, Microsoft announced Windows 10 for the Raspberry Pi 2 through its IoT developer program for no charge (https://dev.windows.com/en-us/featured/raspberrypi2support). This, in addition to its versatile features, has caused fans like me to upgrade to the Raspberry Pi 2. With a few new Raspberry Pi 2 boards in hand, I set out to find some useful ways to employ my older Pi boards.

In this article, I briefly describe the requirements of the project that I outlined, and I explain the various tools I decided to use to build it. I then cover the hardware I chose and the way to assemble the parts to realize the system. Next, I continue setting up the development environment on the Raspbian image, and I walk through the code and bring everything together to form the complete system. Finally, I conclude with possible improvements and hacks that would extend the usefulness of a Pi home automation system.

The Internet of Things

An ongoing trend in embedded devices is to have all embedded devices connected to the Internet. The Internet was developed as a fail-safe network that could survive the destruction of several nodes. The Internet of Things (IoT) leverages the same redundancy. With the move to migrate to IPv6,

A multitude of IoT-connected devices in a home has the potential to act as a living entity that exhibits response to stimuli.

the IP address space would be large enough for several trillion devices to stay connected. A connected device also makes it very convenient to control it from anywhere, receive inputs from various sensors and respond to events. A multitude of IoT-connected devices in a home has the potential to act as a living entity that exhibits response to stimuli.

Raspberry Pi Home Automation

Inspired by the idea of having a home that has a life of its own, I settled on a home automation project to control the lights in my living room. The goal of my project was to be able to time the lights in my living room and control them remotely over the Internet using a Web browser. I also wanted to expose an API that could be used to control the device from other devices programatically.

The interesting part of this project is not the hardware, which is fairly simple and easy to construct, but the UI. The UI that I had in mind would support multiple users logged in to the same Pi server. The UI state had to keep up with the actual state of the system in real time indicating which lights actually were on when multiple users operated the system simultaneously. Apart from this, the lights may toggle on or off when triggered by the timer. A UI running on a device, such as a phone or a tablet, may be subject to random connection drops. The UI is expected to handle this and attempt to reconnect to the Pi server.

Hardware

Having outlined the requirements, I began to build the hardware. Table 1 shows the bill of materials that I used to build the hardware part of the system, and Figure 1 shows a block diagram of the hardware system.

Wiring this is time-consuming but easy. First, wire the SMPS to the wall outlet by cutting off an extension cord

Table 1. Bill of Materials

COMPONENT	QUANTITY	APPROXIMATE PRICE	PROCURED FROM	FUNCTION
Raspberry Pi	1	$35	Newark	The CPU
SD card	1	$25	amazon.com	To boot the RPi
Edimax WiFi	1	$10	amazon.com (http://www.amazon.com/Edimax-EW-7811Un-150Mbps-Raspberry-Supports/dp/B003MTTJOY)	To give the RPi wireless connectivity
Relay module	1	$10	amazon.com (http://www.amazon.com/JBtek-Channel-Relay-Arduino-Raspberry/dp/B00KTELP3I)	Used for switching
Ribbon cable	1	$7	amazon.com (http://www.amazon.com/Veewon-Flexible-Multicolored-Breadboard-Jumper/dp/B00N7XWXRK)	To connect the RPi header to the relay module
Power supply	1	$8	amazon.com (http://www.amazon.com/gp/product/B00HF3G7NO)	To power the RPi and the relay module
Extension cord	9	$54	Walmart (http://www.walmart.com/ip/Qvs-PC3PX-10-10ft-3-Outlet-3-Prong-Power-Cabl-Extension-Cord-Ac-Male-To-Female/41440394)	To power the SMPS and to provide a plug interface to the relays
Pencil box	1	$2	Walmart	To house the entire setup
USB cable	1	$5	amazon.com (http://www.amazon.com/AmazonBasics-USB-Cable-Micro-Meters/dp/B003ES5ZSW)	To power the RPi
14 gauge wire	1	6	Home Depot	To wire the relay terminals to the live wire from the wall outlet
Cable clamp	1	$2	Home Depot	As a strain relief

at the socket end. Strip the wires and screw them into the screw terminals of the SMPS. Next, wire the Raspberry Pi to the SMPS by cutting off the type A end of the USB cable and wiring it to the wire ends of the SMPS and the micro B end to the RPi. Strip out two strands of wires from the ribbon cable, and wire

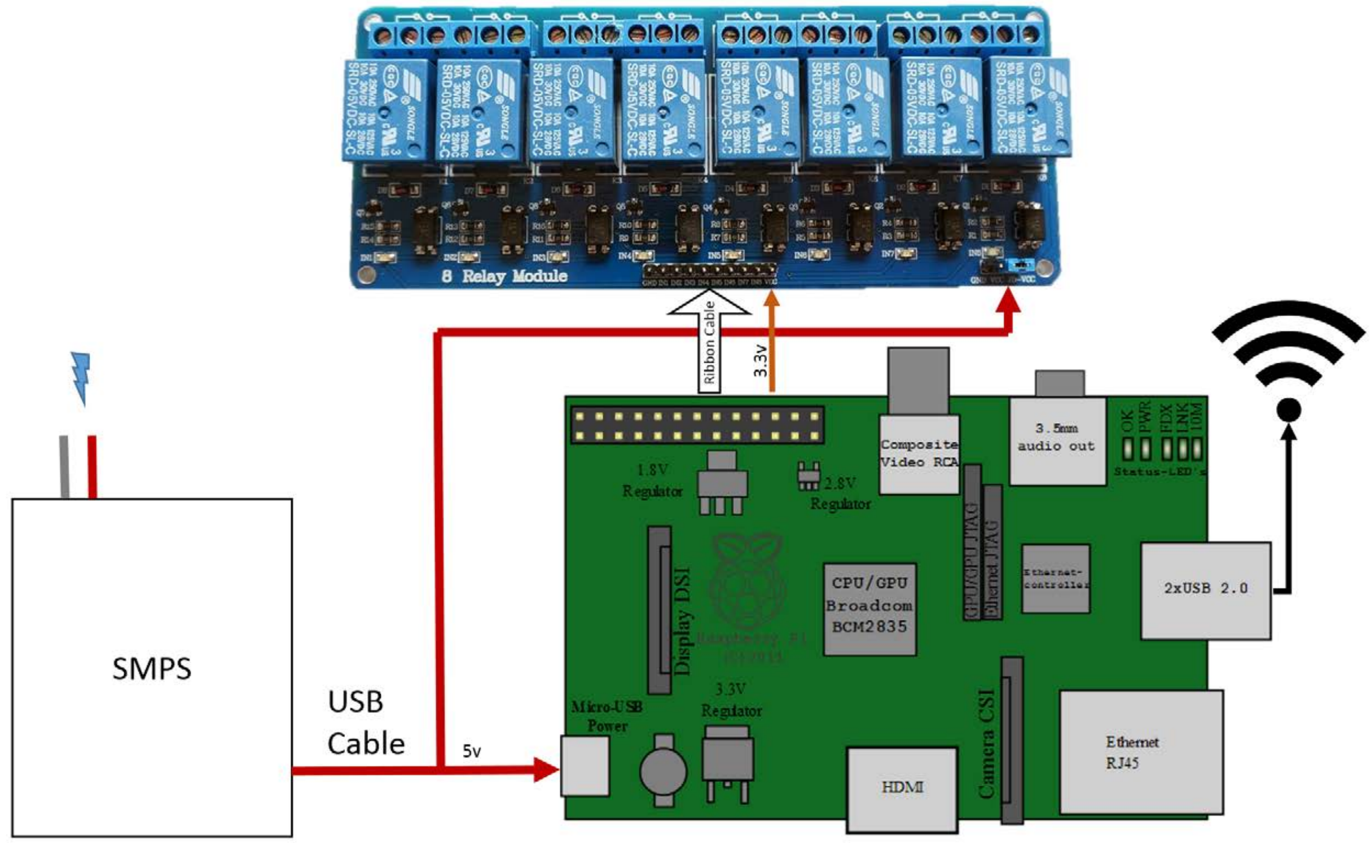
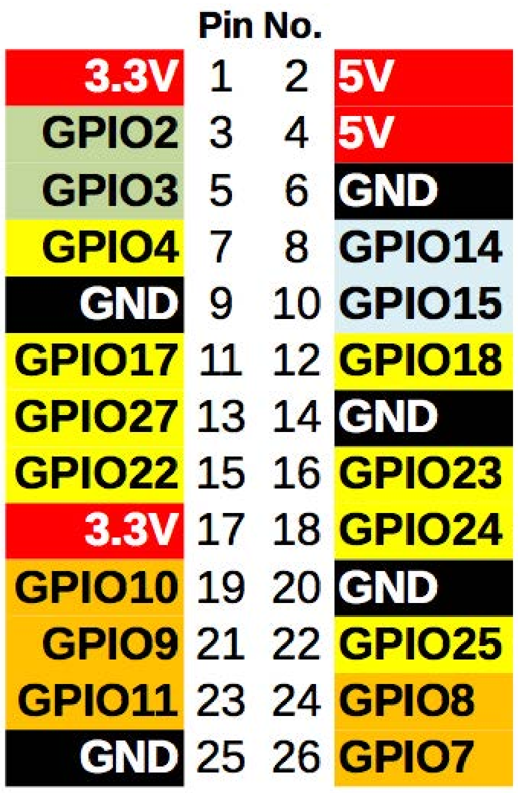


Figure 1. Block Diagram of the Hardware System

the appropriate terminals to GND and JDVcc. Remove the jumper that connects the JDVcc and Vcc. Not removing this jumper will feed back 5v to the 3.3v pins of the Pi and damage it.

Now that all the terminals are wired for power, connect the IN1-IN8 lines of the relay module to the appropriate GPIO pins of the RPi using more of the ribbon cable as shown in Figure 2. The code I present here is written for the case where I wire IN1-IN8 to GPIO1-GPIO7. Should you decide to wire them differently, you will need to modify your code accordingly.

The RPi's GPIO pins are shown in Figure 2. The RPi's IO ports operate at 3.3v, and the relay module works at 5v. However, the relays are isolated from the RPi's GPIO pins using optocouplers. The optocouplers may be supplied 3.3v over the Vcc pin. The Vcc pin of the relay module may be supplied 3.3v from the GPIO header of the Pi. *Make sure you have removed the jumper that bridges the Vcc and JDVcc on the relay module board.* The JDVcc pin should be supplied 5v for proper operation of the relay. The relay module is designed to be active low. This means that you



	Pin No.		
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

Figure 2. The RPi's GPIO Pins

have to ground the terminals IN1-IN8 to switch on a relay.

Warning: handle all wiring with caution. Getting a shock from the line can be fatal!

Cut the remaining extension cables at the plug end, and screw in the wire end to the relay. Also daisy-chain the live wire from the wall outlet to the relay terminals. The entire setup can be housed in a pencil box or something similar. Plan this out in advance to avoid having to unwire and rewire the terminals. Additionally, I added a few screw cable clamps to the holes I made in my housing to act as a strain relief (Figure 3).

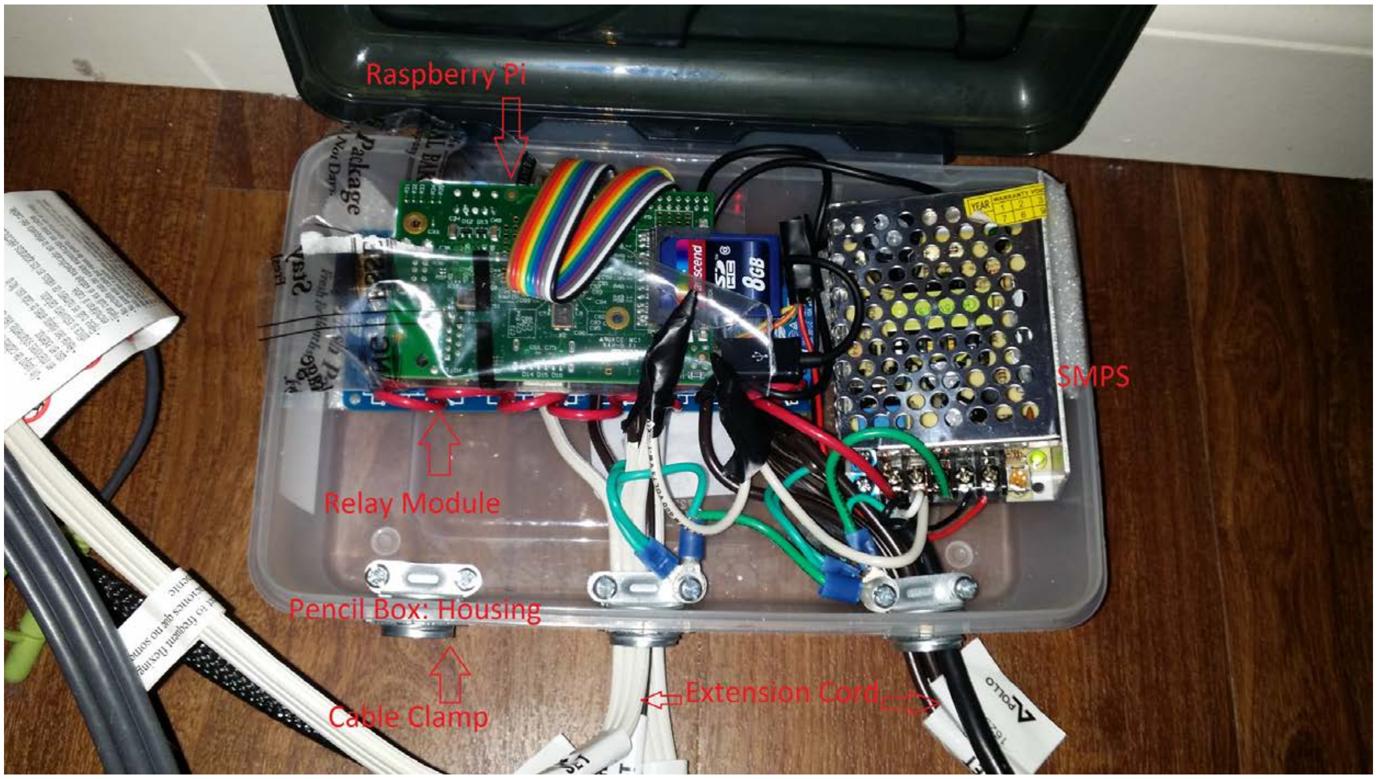


Figure 3. The Hardware Setup

Environment

I built my environment starting with a fresh install of Raspbian. For the initial installation, you need an HDMI-capable display, a USB keyboard, mouse and a wired Ethernet connection. You also optionally may connect a Wi-Fi adapter. Build the SD card for the first boot by following the instructions given at http://www.raspberrypi.org/documentation/installation/installing-image. During the first boot, the installer sets up the OS and expands the image to fill the entire card. After the first boot, you should be able to log in using the default credentials (user "pi" and password "raspberry").

Once you successfully log in, it's good practice to update the OS. The Raspbian image is based on Debian and uses the aptitude package manager. You also will need `python`, `pip` and `git`. I also recommend installing Webmin to ease administration processes. Instructions for installing Webmin are at http://www.webmin.com/deb.html (follow the directions in the "Using the Webmin APT repository" section):

```
sudo apt-get update && sudo apt-get dist-upgrade
sudo apt-get install python python-pip git git-core
```

Next, you need to set up the Wi-Fi connection. You can find detailed instructions for this at http://www.raspberrypi.org/documentation/configuration/wireless. I recommend the `wicd-curses` option. At this point, you can make changes to the RPi setup using the `sudo raspi-config` command. This will bring up a GUI that lets you choose options like the amount of RAM you share with the GPU, overclocking, GUI Boot and so on.

Another useful tool is the Cloud 9 IDE (https://github.com/c9/core). The Cloud9 IDE allows you to edit your code on the RPi using a Web browser. It also gives you a shell interface in the browser. You can develop and execute all your code without leaving the Web browser. The Cloud 9 IDE requires a specific version of NodeJS. Using the wrong version will cause frequent crashes of the Cloud 9 server, resulting in constant frustration. Instructions for installing NodeJS on the Raspberry Pi are outlined at http://weworkweplay.com/play/raspberry-pi-nodejs.

Software

I decided to build my front-end UI using HTML5, CSS3 and JavaScript. The combination of these three form a powerful tool for building

The back-end server on the Raspberry Pi needs to control the GPIO pins on the Raspberry Pi board. It also needs an HTTP interface to serve the UI and a WebSocket interface to pass command and status messages.

UIs. JavaScript provides easy communication APIs to servers. There also are a lot of JavaScript libraries like JQuery, Bootstrap and so on from which to choose. HTML5 supports the WebSocket API that allows the browser to keep a connection alive and receive communication over this connection. This makes WebSocket useful for implementing live and streaming apps, such as for games and chat interfaces. CSS is useful for styling the various HTML elements. When used properly, it lets one build dynamic UIs by switching the styles on an element in response to events. For this project, I chose JQuery to handle events, Bootstrap CSS (http://getbootstrap.com/css) to lay out the buttons in a grid form and pure JavaScript to handle WebSocket communications.

Libraries

The back-end server on the Raspberry Pi needs to control the GPIO pins on the Raspberry Pi board. It also needs an HTTP interface to serve the UI and a WebSocket interface to pass command and status messages. Such a specific server did not exist for off-the-shelf deployment, so I decided to write my own using Python. Python has prebuilt modules for the Raspberry Pi GPIO, HTTP server and WebSockets. Since these modules are specialized, minimum coding was required on my part.

However, these modules are not a part of Python and need to be installed separately. First, you need to be able to control the RPi's GPIO pins. The easiest way to do this from Python is by using the RPi.GPIO library from https://pypi.python.org/pypi/RPi.GPIO. Install this module with:

```
sudo pip install RPi.GPIO
```

Using the RPi.GPIO module is very simple. You can find examples of its usage at http://sourceforge.net/p/raspberry-gpio-python/wiki/Examples. The first step in using the module is to import it into the code. Next, you need to select the mode. The mode can be either GPIO.BOARD or GPIO.BCM. The mode decides whether the pin number references in the subsequent commands will be based on the BCM chip or the IO pins on the board. This is followed by setting pins as either input or output. Now you can use the IO pins as required. Finally, you need to clean up to release the GPIO pins. Listing 1 shows examples of using the RPi.GPIO module.

CherryPy is a Web framework module for Python (http://www.cherrypy.org). It is easily extendible to support WebSocket using the ws4py module (https://github.com/Lawouach/WebSocket-for-Python). CherryPy and ws4py also can be installed using `pip`:

```
pip install cherrypy
pip install ws4py
```

Examples of using the CherryPy framework and the ws4py plugin can be found in the CherryPy docs (https://cherrypy.readthedocs.org/en/latest) and the ws4py docs (http://ws4py.readthedocs.org/en/latest). A basic CherryPy server can be spawned using the code shown in Listing 2.

Slightly more advanced code would pass the `quickstart` method an object with configuration. The partial code in Listing 3 illustrates this. This code serves requests to /js from the js folder. The js folder resides in the

Listing 1. Using the RPi.GPIO Module

```
import RPi.GPIO as GPIO          # import module
GPIO.setmode(GPIO.BOARD)         # use board pin numbering
GPIO.setup(0, GPIO.IN)           # set ch0 as input
GPIO.setup(1, GPIO.OUT)          # set ch1 as output
var1=GPIO.input(0)                       # read ch0
GPIO.output(1, GPIO.HIGH)        # take ch1 to high state
GPIO.cleanup()                           # release GPIO.
```

home directory of the server code.

To add WebSocket support to the CherryPy server, modify the code as shown in Listing 4. The WebSocket handler class needs to implement three methods: `opened`, `closed` and `received_message`. Listing 4 is a basic WebSocket server that has been kept small for the purpose of explaining the major functional parts of the code; hence, it does not actually do anything.

On the client side, the HTML needs to implement a function to connect to a WebSocket and handle incoming messages. Listing 5 shows simple HTML that would do that. This code uses the `jQuery.ready()`

Listing 2. Spawning a Basic CherryPy Server

```
# From the CherryPy Docs at
# https://cherrypy.readthedocs.org/en/latest/tutorials.html

import cherrypy    # import the cherrypy module

class HelloWorld(object):      #
    @cherrypy.expose           # Make the function available
    def index(self):           # Create a function for each request
        return "Hello world!"  # Returned value is sent to the browser

if __name__ == '__main__':
   cherrypy.quickstart(HelloWorld())    # start the CherryPy server
                                        # and pass the class handle
                                        # to handle request
```

Listing 3. Passing the `quickstart` Method

```
cherrypy.quickstart(HelloWorld(), '', config={
   '/js': {          # Configure how to serve requests for /js
   'tools.staticdir.on': True,     # Serve content statically
                                   # from a directory
   'tools.staticdir.dir': 'js'     # Directory with respect to
                                   # server home.
   }
});
```

event to start connecting to the WebSocket server. The code in this Listing implements methods to handle all events: `onopen()`, `onclose()`, `onerror()` and `onmessage()`. To extend this example, add code to the `onmessage()` method to handle messages.

Listing 4. Basic WebSocket Server

```
import cherrypy                  # Import CherryPy server module
# Import plugin modules for CherryPy
from ws4py.server.cherrypyserver  import WebSocketPlugin, WebSocketTool
from ws4py.websocket import WebSocket   # Import modules for
                                        # the ws4py plugin.
from ws4py.messaging import TextMessage

class ChatWebSocketHandler(WebSocket):
        def received_message(self, m):
                msg=m.data.decode("utf-8")
                print msg
                cherrypy.engine.publish('websocket-broadcast',
                 ➥"Broadcast Message: Received a message")

        def closed(self, code, reason="A client left the room
         ➥without a proper explanation."):
                cherrypy.engine.publish('websocket-broadcast',
                 ➥TextMessage(reason))

class Root(object):
    @cherrypy.expose
    def index(self):
        return "index"

    @cherrypy.expose
    def ws(self):
        print "Handler created: %s" % repr(cherrypy.request.ws_handler)


if __name__ == '__main__':
    WebSocketPlugin(cherrypy.engine).subscribe()   # initialize websocket
                                                   # plugin
    cherrypy.tools.websocket = WebSocketTool()          #
    cherrypy.config.update({'server.socket_host': '0.0.0.0',
        'server.socket_port': 9003,
        'tools.staticdir.root': '/home/pi'})
    cherrypy.quickstart(Root(), '', config={
             '/ws': {
                     'tools.websocket.on': True,
                     'tools.websocket.handler_cls': ChatWebSocketHandler
               }
        });
```

Listing 5. Connecting to WebSocket and Handling Incoming Messages

```
<html>
    <head></head>
    <body>

    <script src="/js/jquery.min.js"></script>
    <script type="text/javascript">
    var ws;
    var addr="ws://127.0.0.1:9000";
    $(document).ready(function (){
            connectWS();
    });
    function dbg(m){
            console.log(m);
    }
    function connectWS(){
            dbg('Connecting...');
            if (window.WebSocket) {
                    ws = new WebSocket(addr);
            }
            else if (window.MozWebSocket) {
                    ws = MozWebSocket(addr);
            }
            else {
                    alert('Your archaic browser does not support
                     ➥WebSockets.');
                    dbg('WebSocket Not Supported');
                    return;
            }

            /* on websocket close */
            ws.onclose=function(){
                    dbg('Connection Closed.');
                    reconnect=setTimeout(connectWS,6000); //try to
                                                          //reconnect
                                                          //every 6 secs.
            }

            /* on websocket connection */
            ws.onopen=function(){
                    dbg('Connected.');
                    ws.send('Some message to send to the
                     ➥WebSocket server.');
            }

            /* on websocket error */
            ws.onerror=function(e){
                    dbg("Socket error: " + e.data);
            }

            /* on websocket receiving a message */
            ws.onmessage = function (evt) {
                    dbg(evt.data);
                    //add functions to handle messages.
            }
            return 0;
    }
    </script>
    </body>
</html>
```

Pi Home Automation

Now that you've seen the basics of WebSockets, CherryPy and the HTML front end, let's get to the actual code. You can get the code from the Git repository at https://bitbucket.org/lordloh/pi-home-automation. You can clone this repository locally on your RPi, and execute it out of the box using the command:

```
git clone https://bitbucket.org/lordloh/pi-home-automation.git
git fetch && git checkout LinuxJournal2015May
cd pi-home-automation
python relay.py
```

The relayLabel.json file holds the required configuration, such as labels for relays, times for lights to go on and off and so on. Listing 6 shows the basic schema of the configuration. Repeat this pattern for each relay. The `dow` property is formed by using one bit for each day of the week starting from Monday for the LSB to Sunday for the MSB.

Figure 4 shows the block diagram of the system displaying the major functional parts. Table 2 enumerates all the commands the client may send to the server and the action that the server is expected to take. These commands are sent from the browser to the server in JSON format. The command schema is as follows:

```
{
    "c":"<command form TABLE 2>",
    "r":<relay Number>
}
```

The `update` and `updateLabels` commands do not take a relay number. Apart from relay.py and relayLabel.json, the only other file required is index.html. The relay.py script reads this file and serves it in response to HTTP requests. The index.html file contains the HTML, CSS and JavaScript to render the UI.

Once the system is up and running, you'll want to access it from over the Internet. To do this, you need to set a permanent MAC address and reserved IP address for the Raspberry Pi on your

Listing 6. Basic Schema of the Configuration

```
{
  "relay1": {
    "times": [
      {
        "start": [
          <hour>,
          <minute>,
          <second>
        ],
        "end": [
          <hour>,
          <minute>,
          <second>
        ],
        "dow":
<Monday<<0|Tuesday<<1|Wednesday<<2|Thursday<<3|
➥Friday<<4|Saturday<<5|Sunday<<6>
      }
    ],
    "id": 1,
    "label": "<Appliance Name>"
  }
}
```

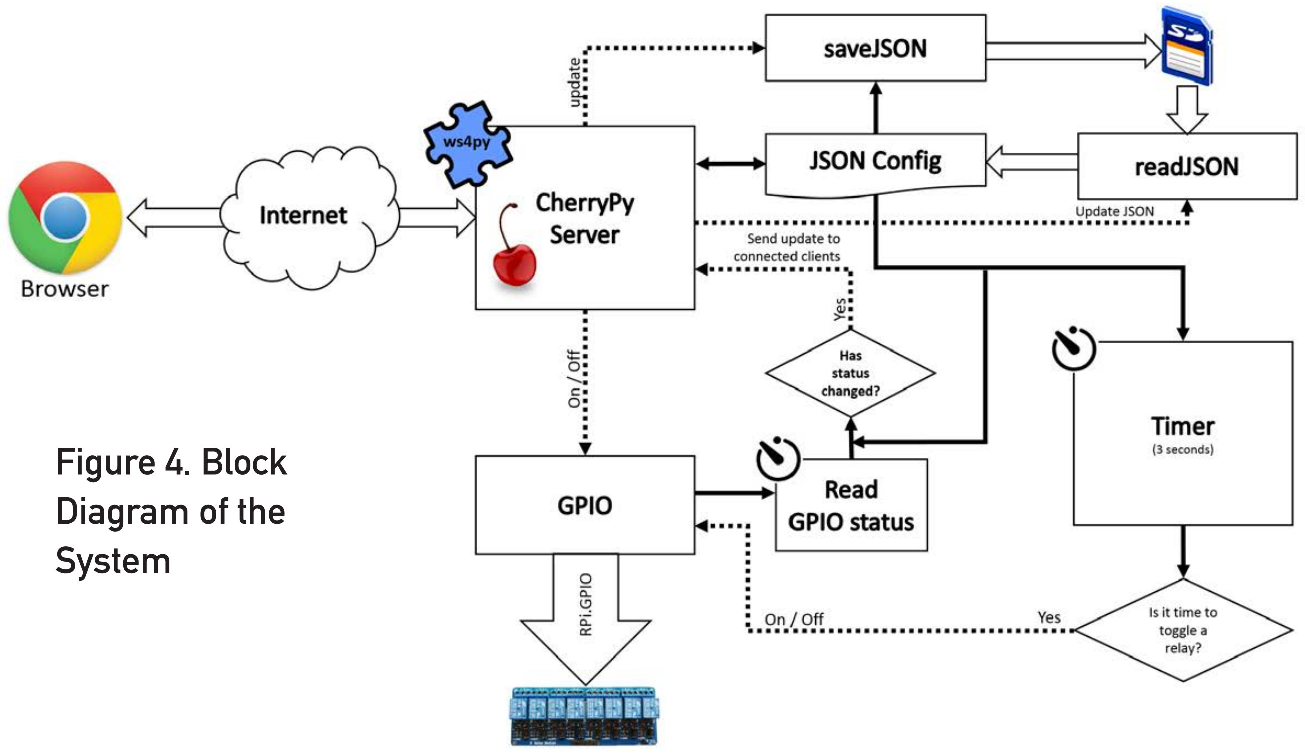


Figure 4. Block Diagram of the System

local network, and set up port forwarding on your router. The process for doing this varies according to router, and your router manual is the best reference for it. Additionally, you can use a dynamic domain name service so that you do not need to type your IP address to access your Pi every time. Some routers include support for certain dynamic DNS services.

Conclusion

I hope this article helps you to build this or other similar projects. This project can be extended to add new features, such as detecting your phone connected to your Wi-Fi and switching on lights. You also could integrate this with applications, such as OnX and Android Tasker. Adding password protection for out-of-network access is beneficial. Feel free to mention any issues, bugs and feature requests at http://code.lohray.com/pi-home-automation/issues.■

Table 2. Commands

COMMAND	DESCRIPTION
`on`	Switch a relay on
`off`	Switch a relay off
`update`	Send status of GPIO pins and relay labels
`updateLabels`	Save new labels to JSON files

Bharath Bhushan Lohray is a PhD student working on his dissertation on image compression techniques at the Department of Electrical & Computer Engineering, Texas Tech University. He is interested in machine learning.

Send comments or feedback via http://www.linuxjournal.com/contact or to ljeditor@linuxjournal.com.