The CATBOT - A web controlled robot with webcam and laser pointer

John Callow

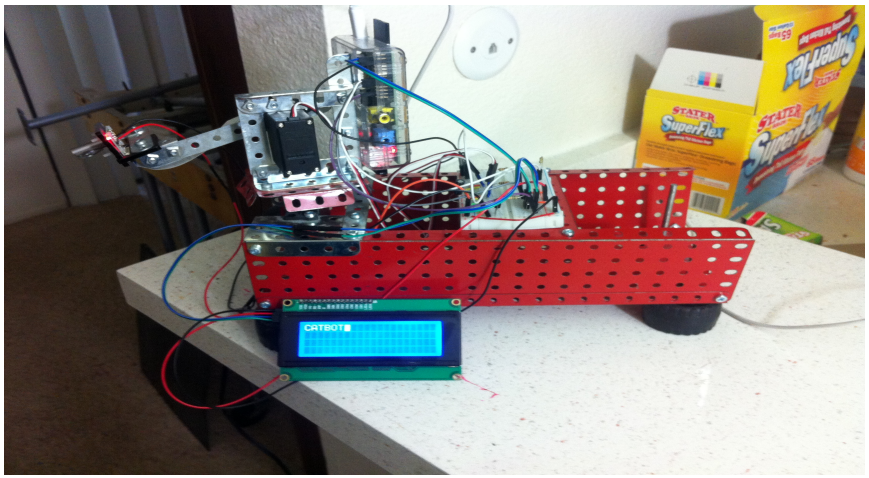
University of California Irvine Extension

john.j.callow@gmail.com

June 14, 2014

Table of Contents

Introduction



This document/rough guide goes over the steps for building a web controlled robot I've named the CATBOT. The robot has a camera and laser pointer mounted on a servo arm all controlled by a raspberry pi. There is also an I2C 20x4 LCD display that currently just displays CATBOT when its driver is loaded. The purpose of this robot is to help get my cats some much needed exercise.

When CATBOT is online you may try it out at 71.95.55.215:1234. This address will likely change over time. You can get the most recent address along with updated versions of this guide and source code at:

https://drive.google.com/folderview?id=0B5AYbBGUvSoacXVaYjctOFF3aGc&usp=sharing

Hardware Setup

For this project I want to focus more on the software side so I'll only do a brief overview of the hardware. The essential parts are:

1. Raspberry Pi with power supply
2. SDHC card with at about 2gb (Eventually I'll shrink the size down)
3. 2 Servo motors
4. 2nd power supply to drive peripherals (will eventually reduce to only one)
5. Laser pointer
6. Sainsmart 20x4 I2C lcd Display
7. Raspberry Pi camera module
8. breadboard and miscellaneous erector set parts
9. edimax wireless usb adapter

The servos and laser pointer are controlled with transistors that are connected to GPIO pins on the Pi. The LCD is connected to the SDA and SCL pins. The camera is connected to a MIPI port. A second 5v power supply is connected to a breadboard providing power to all peripherals except camera. The Pi

Compiling the Kernel

The first step is to compile the kernel for the raspberry pi on a Desktop. It is possible to compile the kernel natively on the pi, but this takes a long time. The following steps were done on my desktop running a fresh install of Ubuntu 14.04 without issue.

1. Grab the gcc compiler for arm architectures
   sudo apt-get install git gcc-arm-linux-gnueabi make ncurses-dev
2. Choose/create a directory to work in
   cd /opt
   sudo mkdir raspberrypi
   cd raspberrypi
3. Grab source code and tools
   sudo git clone git://github.com/raspberrypi/linux.git
   sudo git clone https://github.com/raspberrypi/tools.git
4. Setup the configuration using bcmrpi_cutdown_defconfig
   cd linux
   sudo cp arch/arm/configs/bcmrpi_cutdown_defconfig .config

Compiling Kernel Continued

5. Build the kernel
   sudo make ARCH=arm CROSS_COMPILE= /usr/bin/arm-linux-gnueabi- \
   bcmrpi_cutdown_defconfig

   sudo make ARCH=arm CROSS_COMPILE= /usr/bin/arm-linux-gnueabi-
6. Build Modules
   sudo mkdir ../modules
   sudo make modules_install ARCH=arm CROSS_COMPILE=/usr/bin/arm-linux-\
   gnueabi- INSTALL_MOD_PATH=../modules/
7. Make kernel.img
   cd ../tools/mkimage/
   sudo ./imagetool-uncompressed.py ../../linux/arch/arm/boot/Image

We are now done building the kernel. Now we will install Raspbian on an sd card and transfer our newly compiled kernel for use.

Installing Raspbian

We need to install Raspbian on an SD card. To do so I did the following in Ubuntu:

1. Download the zip containing the Raspbian OS from http://www.raspberrypi.org/downloads/
2. right click and extract the zip file
3. insert an SD card into the computer (Anything over 4GB should work)
4. find the drive name, I recommend the following
   sudo fdisk -l
5. install raspbian with the following
   sudo dd if="Path To Raspbian Image" of="Device (example /dev/sdc)"

If you prefer a GUI you could also right click the Raspian image and select open with image writer or your favorite GUI for writing images.

Install New Kernel

Now with Raspian installed, plug in an ethernet cable and power up the Raspberry pi. Then use the following commands from a terminal on the desktop to setup the new kernel. Note that the default password for the Pi is raspbian.

1. Copy files to home directory
   sudo scp -r /opt/raspberrypi/modules/ pi@raspberrypi:/home/pi/
   sudo scp /opt/raspberrypi/tools/mkimage/kernel.img pi@raspberrypi:/home/pi/
2. ssh into the pi from desktop ,rename old kernel, and copy our files to the proper directory
   ssh pi@raspberrypi
   sudo mv /boot/kernel.img /boot/Old_kernel.img
   sudo cp -r /home/pi/modules/lib /
   sudo cp /home/pi/kernel.img /boot/
3. Now reboot and ssh back into the Pi. It should now display your updated kernel version.
   sudo reboot
   ssh pi@raspberrypi

Raspi-config and wireless setup

We now need to configure Raspbian and change a few options for convenience. With the pi wired to a network using Ethernet, do the following:

1. ssh into pi with: ssh pi@raspberry
2. run raspi-config with: raspi-config
3. Select Expand FileSystem
4. Change user password if you want
5. Change Internationalisation Options, specifically the default keyboard is usually set to English UK, change to English US if you are working directly on the Pi. Not necessary if using ssh.
6. Select Enable Camera
7. Select Finish and then reboot with:sudo reboot

That does it for raspi-config. Wireless setup is going to be dependant on your home network configuration. I recommend following the instructions at:

https://learn.adafruit.com/adafruits-raspberry-pi-lesson-3-network-setup/setting-up-wifi-with-occidentalis

Setup ServoBlaster

Now we will install the ServoBlaster driver. There is an old kernel module version that you can cross-compile after modifying the makefile, but it is recommended to use the newer user space version. Log into your Pi and run the following commands:

1. install git on Pi (or grab files on desktop and transfer)
   sudo apt-get install git
2. Grab files
   git clone git://github.com/richardghirst/PiBits.git
3. Navigate to correct directory
   cd ./PiBits/ServoBlaser/user/
4. install with makefile (requires sudo/root)
   sudo make install

ServoBlaster is now installed. You can test it by connecting the control wire of a servo to a pin (say GPIO-25 as listed in hardware setup) and running command: echo 7=150 > /dev/servoblaster
You can try other postions instead of 150, the defaults run from 50-250.

Install RPi-Cam-Web-Interface

I tried several methods for streaming video from the Raspberry Pi camera module. Most ended up very choppy and unresponsive. I ended up using RPi-Cam-Web-Interface which allowed for high definition streaming at 30 fps. This software can also do more than just stream, such as detect motion. For more info check out

http://elinux.org/RPi-Cam-Web-Interface

To install, ssh into your pi and run the following:

- git clone https://github.com/silvanmelchior/RPi_Cam_Web_Interface.git
- cd RPi_Cam_Web_Interface
- chmod u+x RPi_Cam_Web_Interface_Installer.sh
- ./RPi_Cam_Web_Interface_Installer.sh install

Now just reboot your Pi. To check that it is working, on your desktop or computer on the same network as the Pi go to http://"your Pi's local IP":80. You should see a webpage with a box displaying your camera's current view.

RPi-Cam-Web-Interface Setup

I chose to change the default port used by RPi-Cam-Web-Interface. If you want to change from port 80 to say 1234, you can use the following steps once connected to the Pi through ssh.

1. sudo nano /etc/apache2/ports.conf
2. change "NameVirtualHost*:80" to "NameVirtualHost*:1234"
3. change "Listen 80" to "Listen 1234"
4. save (hit control x followed by y and enter)
5. sudo nano /etc/apache2/sites-enabled/000-default
6. change the line "<Virtual*:80>" to "<Virtual*:1234>"
7. save and restart apache with
   sudo /etc/init.d/apache2 restart

Now test that the changes worked by going to http://"your Pi's local IP":1234 in a web browser. It should display the same webpage with the streaming camera view as before.

Install Node.js

We will use Node.js with package socket.io to run server side javascript to control the servos and laser. To install ssh into your Pi and run the following commands

1. First install Node.js, at the time of writing v0.10.26 was the latest supported.
   wget
   http://nodejs.org/dist/v0.10.26/node-v0.10.26-linux-arm-pi.tar.gz tar

   -xvzf node-v0.10.26-linux-arm-pi.tar.gz
2. Next navigate to the directory containing our html file and install socket.io cd /var/www
   sudo /home/pi/node-v0.10.26-linux-arm-pi/bin/npm install socket.io

That should be it for installing Node. Next We'll modify the html file.

Modify html File

The default html file that comes with RPi-Cam-Web-Interface needs to be modified to use sockets. I also decided to remove everything from the default file except the script to load video from the webcam to make a very basic looking webpage. The code follows:

```
1  <!DOCTYPE html>
2
3  <!-- The html code below runs a script to grab the video and
       formats everything.  Currently the page is fairly simple-->
4  <html>
5    <head>
6      <title>Catbot View</title>
7      <script src="script.js"></script>
8    </head>
9    <body id="bodies", onload="setTimeout('init();', 100);">
10   <center>
11       <h1>Catbot View</h1>
12       <div><img id="mjpeg_dest"></div>
13       <p>Press the arrow keys on keyboard to control the view.
             Press enter to turn the laser pointer on and off<p>
14     </center>
15   </body>
16 </html>
```

HTML Code continued

```
17  <!-- grabs socket.io.js and attempts to connect -->
18  <script src="http://192.168.10.150:4321/socket.io/socket.io.js">
        </script>
19  <script><!--Replace 192... with your int/external address-->
20    var socket = io.connect('http://192.168.10.150:4321');
21    socket.on('connected',function()  {
22    console.log("connected");
23    });
24    var wait = 0; //Used to pace the number of events sent
25  document.getElementById('bodies').onkeydown = function(event) {
26    if (event.keyCode == 37 || event.keyCode == 65) {
27      if (wait == 0) {
28        wait = 1;
29        socket.emit('Right');
30        setTimeout(function() {wait = 0; }, 33);
31        }
32    }
33    else if (event.keyCode == 39 || event.keyCode == 68) {
34      if (wait == 0) {
35        wait = 1;
36        socket.emit('Left');
37        setTimeout(function() {wait = 0; }, 33);
38        }
39    }
```

HTML Code Continued

```
40      else if (event.keyCode == 38   || event.keyCode == 87){
41        if (wait == 0) {
42          wait = 1;
43          socket.emit('Up');
44          setTimeout(function() {wait = 0; }, 33);
45          }
46        }
47      else if (event.keyCode == 40 || event.keyCode == 83){
48        if (wait == 0) {
49          wait = 1;
50          socket.emit('Down');
51          setTimeout(function() {wait = 0; }, 33);
52          }
53        }
54      else if(event.keyCode == 13){
55        socket.emit('Laser');
56        }
57  }
58  </script>
```

Server Side Javascript

Now lets look at the server side javascript code. It is fairly simple, just note that each function used for movement/laser works by executing a shell command with "exec".

```
1  var io = require('socket.io').listen(4321);
2  var sys = require('sys')
3  var exec = require('child_process').exec;
4  var x = 120;
5  var y = 150;
6  var laser = 0;
7
8  io.sockets.on('connection', function (socket) {
9    socket.emit('connected');
10   //Listen for events from user and then execute function from
         Below
11   socket.on('Right',  Moveright );
12   socket.on('Left',  Moveleft );
13   socket.on('Up',  Moveup );
14   socket.on('Down',  Movedown );
15   socket.on('Laser',  ToggleLaser );
16 });
17 function puts(error, stdout, stderr) {sys.puts(stdout)}
```

Server Side Javascript Continued

```
18  function Moveleft() {
19    if (x > 82) {
20      x = x - 1;
21    }
22    exec("echo 6=" + x + " > /dev/servoblaster", puts);
23    console.log("moved left");
24  }
25  function Moveright() {
26    if (x < 182) {
27      x = x + 1;
28    }
29    exec("echo 6=" + x + " > /dev/servoblaster", puts);
30    console.log("moved right");
31  }
32  function Moveup() {
33    if (y < 230) {
34      y = y + 1;
35    }
36    exec("echo 5=" + y + " > /dev/servoblaster", puts);
37    console.log("moved up");
38  }
```

Server Side Javascript Continued

```
39  function Movedown() {
40    if (y > 130) {
41      y = y - 1;
42    }
43    exec("echo 5=" + y + " > /dev/servoblaster", puts);
44    console.log("moved down");
45  }
46  function ToggleLaser() {
47    if (laser == 1) {
48      laser = 0;
49      exec("echo 0 > /sys/class/gpio/gpio11/value", puts);
50      console.log("Turned off laser");
51    }
52    else {
53      laser = 1;
54      exec("echo 1 > /sys/class/gpio/gpio11/value", puts);
55      console.log("Turned on laser");
56    }
57  }
```

Power Button Shell Script

The Raspberry Pi has no power/shutdown button. So in the future I plan to add one. I'd a power button that behaves similar to modern pc power/reset buttons. A single push will start a shutdown sequence and the pi will run "shutdown -h now" and turn itself off when done. Holding down the power button should force the power off after a few seconds.

Server code Startup script

The following script loads the serverside javascript code on startup. It is a modified version of a CentOS script from https://gist.github.com/nariyu/1211413. I have ssh in required-start to make sure I can still log in if the script hangs. Had a typo originally without ssh and had to remove the SD card from pi and load in desktop to directly modify files.

```
#!/bin/sh
# /etc/init.d/Load_Node
### BEGIN INIT INFO
# Provides:          Load_Node
# Required-Start:    $remote_fs $syslog ssh
# Required-Stop:     $remote_fs $syslog
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Starts server code
### END INIT INFO
USER="pi"
DAEMON="/home/pi/node-v0.10.26-linux-arm-pi/bin/node"
ROOT_DIR="/var/www"
SERVER="$ROOT_DIR/myjavascript.js"
LOG_FILE="$ROOT_DIR/myjavascript.js.log"
LOCK_FILE="/var/lock/node-server"
```

Server code Startup script continued

```
do_start()
{
  if [ ! -f "$LOCK_FILE" ] ; then
    echo -n $"Starting $SERVER: "
    echo "" > "$LOG_FILE"
    sudo sh -c "$DAEMON $SERVER > $LOG_FILE &"
    echo "" > "$LOCK_FILE"
    RETVAL=0
  else
    echo "$SERVER is locked."
    RETVAL=1
  fi
}
do_stop()
{
  echo -n $"Stopping $SERVER: "
  pid=`ps -aefw | grep "$DAEMON $SERVER" | grep -v " grep " | awk
       '{print $2}'`
  kill -9 $pid > /dev/null 2>&1
  rm -f "$LOCK_FILE"
  RETVAL=0;
}
```

Server code Startup script continued

```
case "$1" in
  start)
    do_start
    ;;
  stop)
    do_stop
    ;;
  restart)
    do_stop
    do_start
    ;;
   *)
    echo "Usage: $0 {start|stop|restart}"
    RETVAL=1
esac

exit "$RETVAL"
```

To use the script, save it as Load_Node in the pi's /etc/init.d directory. Then make it executable and run on startup with the following commands:

```
sudo chmod 755 /etc/init.d/Load_Node
sudo update-rc.d Load_Node defaults
```

Enable I2C For LCD Display

I2C is disabled by default. Perform the following steps on the pi to enable it.

1. open /etc/modules
   sudo nano /etc/modules
2. In the file add the following two lines:
   i2c-bcm2708
   i2c-dev
3. Save the file and reboot
4. open etc/modprobe.d/raspi-blacklist.conf with sudo if it exists and comment out the following line (put a # in front of them) blacklist i2c-bcm2708
5. save the file. I2C is now enabled. While we're at it install i2ctools
   sudo apt-get install i2c-tools

Now if you haven't already, connect the sainsmart 20x4 lcd display to the pi's I2C pins (SDA and SCL). Also make sure the screen is powered by a 5v source. (The pi's 5v output seems to work, but I have it powered externally along with the servos and laser).

On the pi run "sudo i2cdetect -y 1" and it will display the address in hex of any connections detected on I2C. My display had address 0x27.

I2C Sainsmart LCD Device Driver Code

Below is my currently unfinished I2C device driver code. As of now it properly detects and initializes the display and outputs "CATBOT". Eventually I will set it up to work as a character driver so that any writes to a file will be displayed on the screen.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/i2c.h>
#include <linux/delay.h>

/* Addresses to scan */
 static const unsigned short normal_i2c[] = { 0x27, I2C_CLIENT_END };

/* possible lcds */
static struct i2c_device_id sainsmart_id[] = {
  { "sainsmart", 0},
  {}
};
```

I2C Sainsmart LCD Device Driver Code Continued

```
/* helper commands for writing (4-bit mode) */
int strobe(struct i2c_client *client, u8 value)
{
  u8 temp;
  temp = (value | 0x0C);
  i2c_smbus_write_byte(client, temp);
  mdelay(1);
  temp = (value & 0xFB);
  i2c_smbus_write_byte(client, temp);
  mdelay(1);
  return 0;
}
int write_char(struct i2c_client *client, u8 value)
{
  u8 temp_value;
  temp_value = ((value >> 4) << 4) | 0x09;
  i2c_smbus_write_byte(client, value);
  strobe(client, temp_value);
  temp_value = ((value & 0x0F) << 4) | 0x09;
  i2c_smbus_write_byte(client, value);
  strobe(client, temp_value);
  i2c_smbus_write_byte(client, 0x08);
  return 0;
}
```

I2C Sainsmart LCD Device Driver Code Continued

```
int write_command(struct i2c_client *client, u8 value)
{
  u8 temp_value;
  temp_value = (value & 0xF0) | 0x08;
  i2c_smbus_write_byte(client, value);
  strobe(client, temp_value);
  temp_value = ((value & 0x0F)  << 4) | 0x08;
  i2c_smbus_write_byte(client, value);
  strobe(client, temp_value);
  i2c_smbus_write_byte(client, 0x08);
  return 0;
}
/* initialize lcd to 4-bit mode, required to use i2c backpack */
int init_lcd(struct i2c_client *client)
{
  mdelay(100);
  i2c_smbus_write_byte(client, 0x30);
  strobe(client, 0x30);
  mdelay(5);
  strobe(client, 0x30);
  mdelay(5);
  strobe(client, 0x30);
  mdelay(5);
```

I2C Sainsmart LCD Device Driver Code Continued

```
i2c_smbus_write_byte(client, 0x20);
  strobe(client, 0x20);
  mdelay(5);
  write_command(client, 0x28);
  write_command(client, 0x08);
  write_command(client, 0x01);
  write_command(client, 0x06);
  write_command(client, 0x0C);
  write_command(client, 0x0F);

  return 0;
}
/* driver initialization/remove functions */
MODULE_DEVICE_TABLE(i2c, sainsmart_id);
static int sainsmart_probe(struct i2c_client *client,
                           const struct i2c_device_id *id)
{
  pr_info("called␣probe");

  init_lcd(client);
  write_char(client, 'C');
  write_char(client, 'A');
  write_char(client, 'T');
```

I2C Sainsmart LCD Device Driver Code Continued

```
 write_char(client, 'B');
  write_char(client, 'O');
  write_char(client, 'T');
  return 0;
}
static int sainsmart_remove(struct i2c_client *client)
{
  pr_info("called␣remove");
  return 0;
}

static struct i2c_driver sainsmart_driver = {
  .driver = {
    .name   = "sainsmart",
    .owner  = THIS_MODULE,
  },
  .probe          = sainsmart_probe,
  .remove         = sainsmart_remove,
  .id_table       = sainsmart_id
};
```

I2C Sainsmart LCD Device Driver Code Continued

```
static int __init sainsmart_init(void)
{
  return i2c_add_driver(&sainsmart_driver);
}
module_init(sainsmart_init);
static void __exit sainsmart_exit(void)
{
  i2c_del_driver(&sainsmart_driver);
}
module_exit(sainsmart_exit);
MODULE_LICENSE("GPL");
```

This code took much longer to get running than expected, and as mentioned isn't finished. Two sources finally got me to the point of having code do anything:
Free electrons lab course - http://free-electrons.com/doc/training/linux-kernel/
I2C documentation - https://www.kernel.org/doc/Documentation/i2c/
For now I run the following in command line for instantiating my lcd:

echo sainsmart 0x27 /sys/class/i2c-adapter/i2c-1/new_device

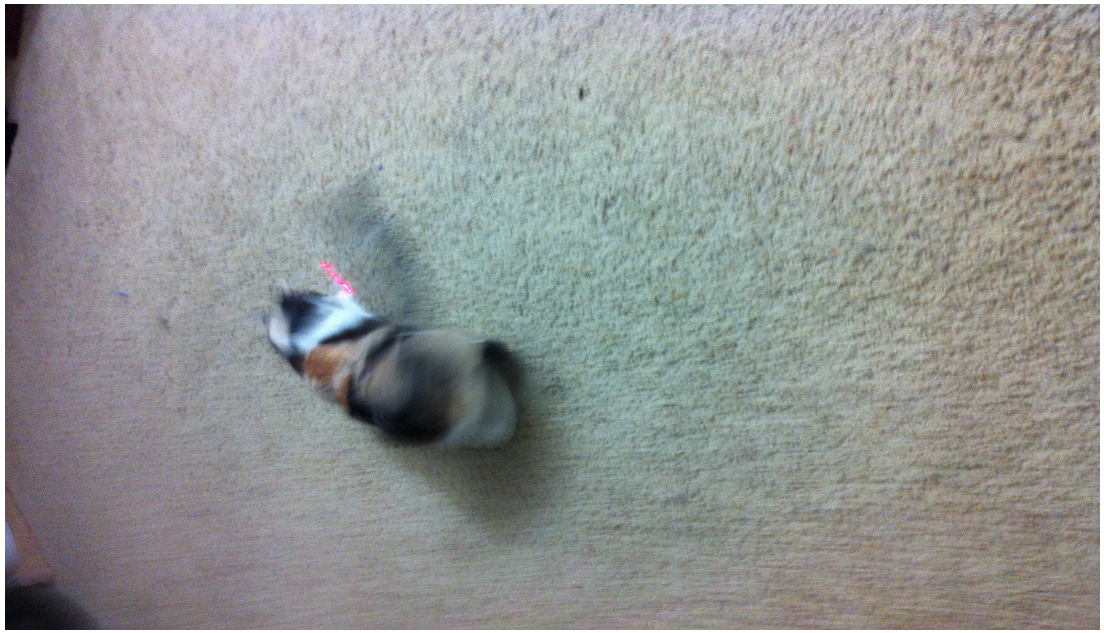
Eventually will use i2c_board_info method instead. The first thing I did was get probe command called and output some text to console. This took a few days of reading the documentation. Once I got that things started moving more quickly.

Makefile

The following makefile was used to compile the above code:

```
KERNEL_TREE := /opt/raspberrypi/linux/
PWD := $(shell pwd)

export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-

obj-m := sainsmart_lcd_driver.o

default:
@[ -d ${KERNEL_TREE} ] || { echo "Edit Makefile to set KERNEL_TREE to point at \
 your kernel"; exit 1; }
@[ -e ${KERNEL_TREE}/Module.symvers ] || { echo "KERNEL_TREE/Module.symvers does \
 not exist, you need to configure and compile your kernel"; exit 1; }
make -C ${KERNEL_TREE} M=$(PWD) modules

clean:
make -C ${KERNEL_TREE} M=$(PWD) clean
```

Conclusion



The CATBOT was a fun project that ended up with me learning a lot. I feel much more comfortable compiling the kernel, programming websockets in javascript, writing shell scripts, and writing device drivers. There are still many improvements to make such as completing the lcd device driver, adding power button, and adding a que system for users. I'll try to continue updating the guide and robot over the summer and post the updates to my shared google docs for anyone that is interested.

References

RaspberryPi: Compile Kernel Linux

http://opensource.telkomspeedy.com/wiki/index.php/RaspberryPi:_Compile_Kernel_Linux

Stack Overflow: Cross compiling a kernel module

http://stackoverflow.com/questions/3467850/cross-compiling-a-kernel-module

RPi Cam Web Interface

http://www.raspberrypi.org/forums/viewtopic.php?f=43&t=63276

socket.io documentation

http://socket.io/docs/

Node.js sample startup script

https://gist.github.com/nariyu/1211413

Adafruit: Configuring I2C

https://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c

HD44780U Data Sheet

https://www.sparkfun.com/datasheets/LCD/HD44780.pdf

Linux kernel and driver development training

http://free-electrons.com/doc/training/linux-kernel/labs.pdf

I2C Documentation

https://www.kernel.org/doc/Documentation/i2c/