2018

POV Clock

FINAL LAB REPORT

JEREMIAH KALMUS, DOMINIC BURGI

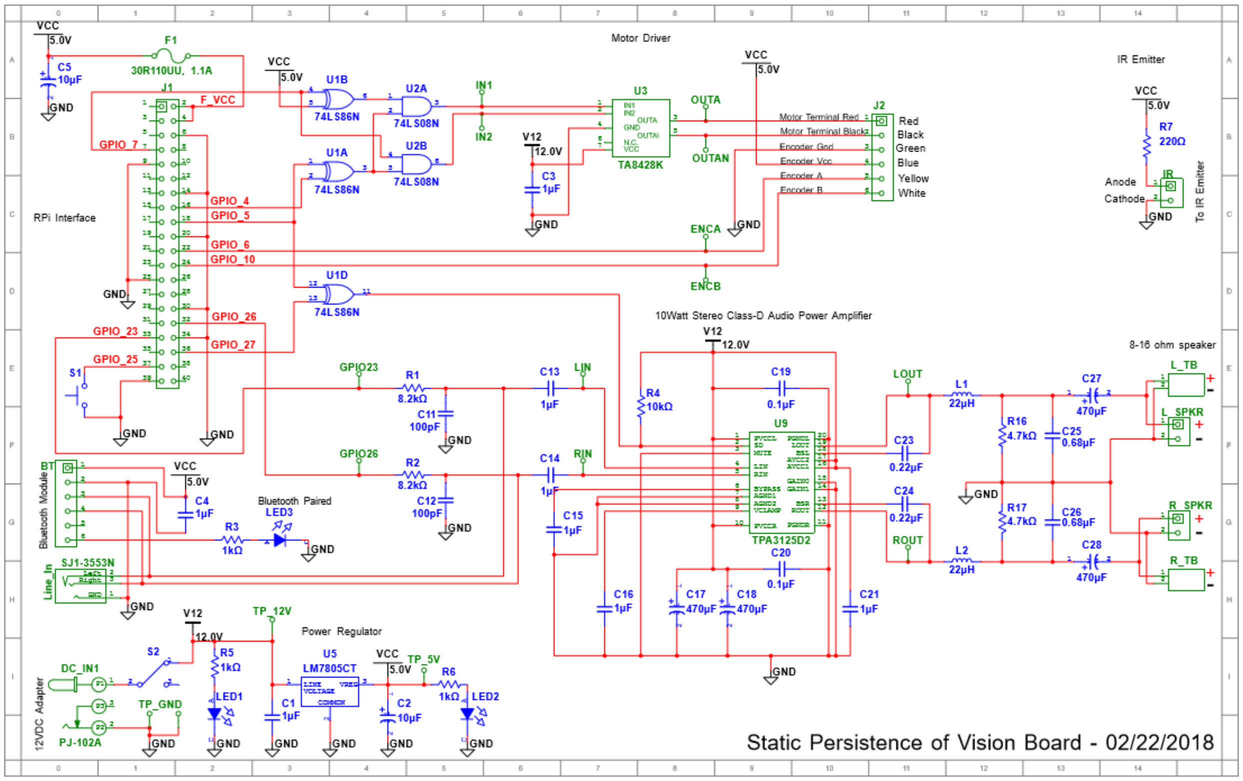
Table of Contents

Introduction

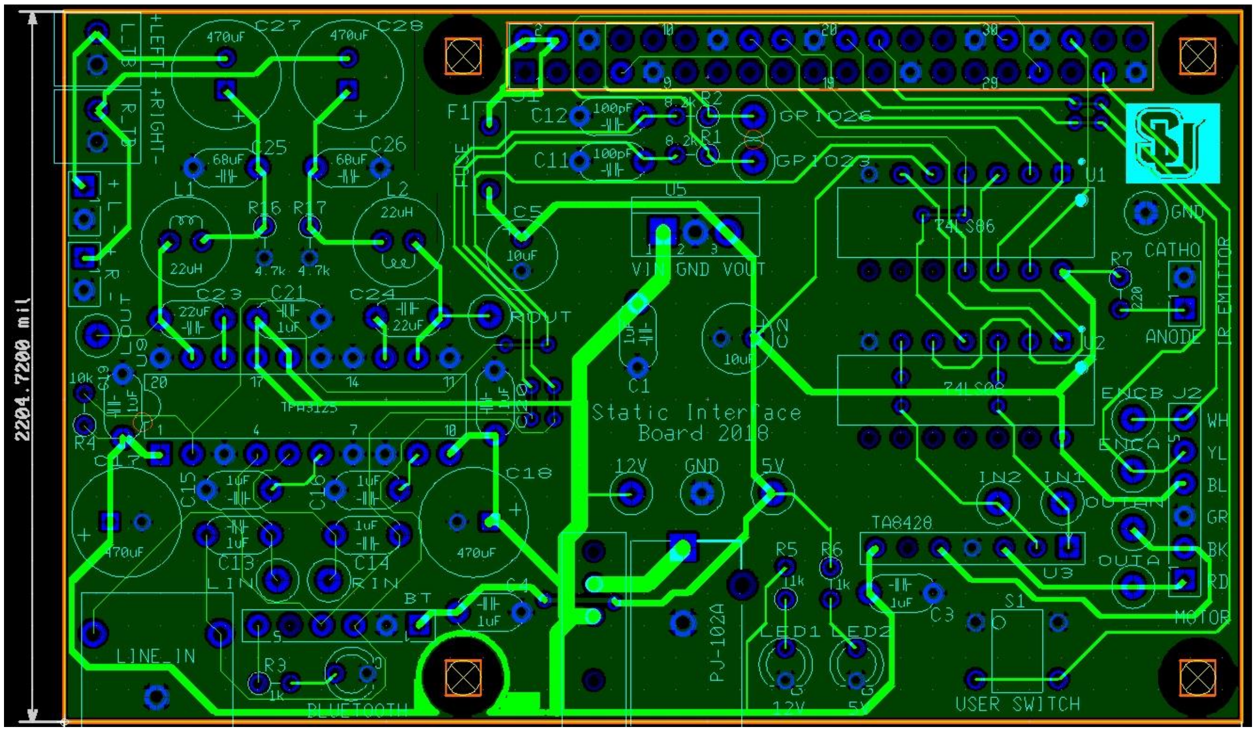
The objective of this project was to create a fully operational clock that used the concept of persistence of vision. This means that with the use of rapid and consistent motion, a matrix of LED's will create an image that is easily recognizable by the human eye. In our case, we created an analog clock that will accurately tell time and make sound.

The main components we used are a Raspberry Pi 3, a Raspberry Pi Zero, and a small DC motor. The Raspberry Pi 3, which we call the Static Pi, is used to keep track of sound, rotational speed of the motor, and the current time. The Raspberry Pi Zero, which we call the Rotor Pi, is attached to the DC motor and is rotating by command of the Static Pi. The Rotor Pi keeps track of the LED's that display the clock face. Since our design is of an analog clock, the Rotor Pi is responsible for dividing the clock face into equal sections and turning the LED's on or off during each given slice. Lastly, the DC motor is used to rotate the Rotor Pi and its LED's at a speed that makes the clock face easily readable.

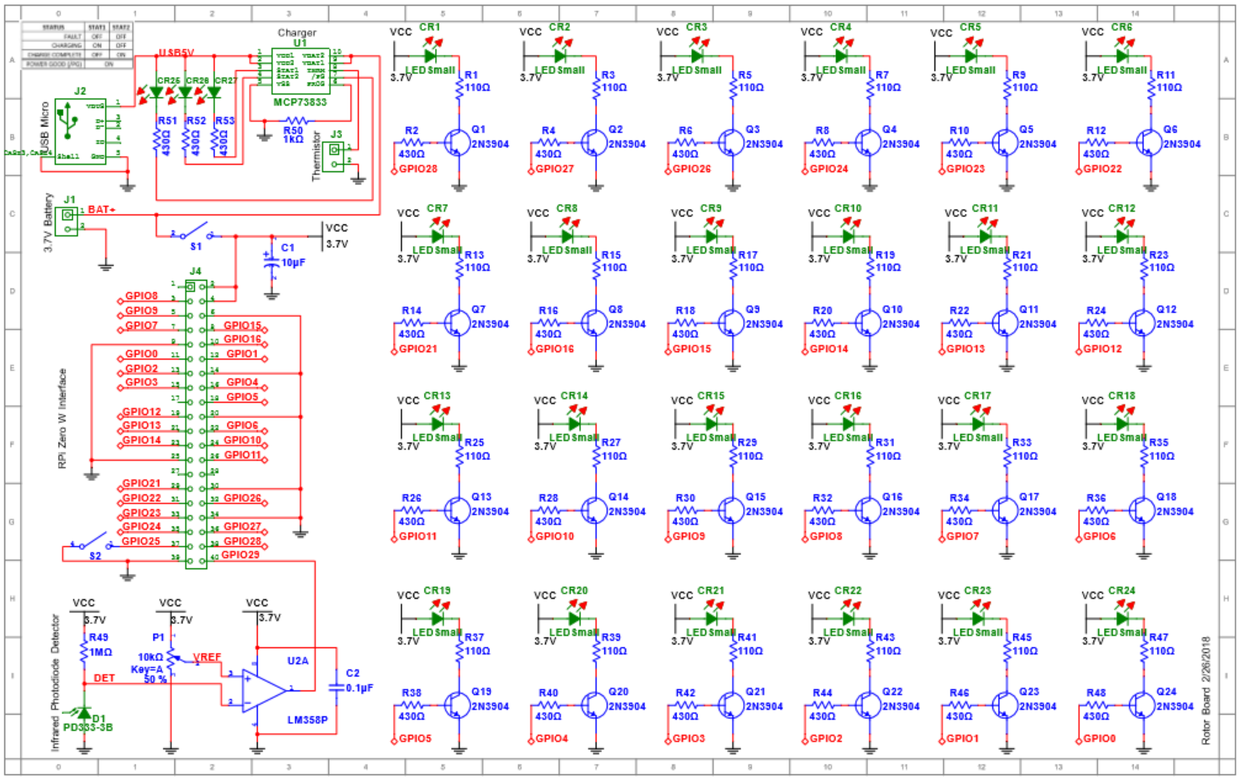
Schematic of StaticPi



Layout Diagram of StaticPi



Schematic of RotorPi



Layout Diagram of RotorPi

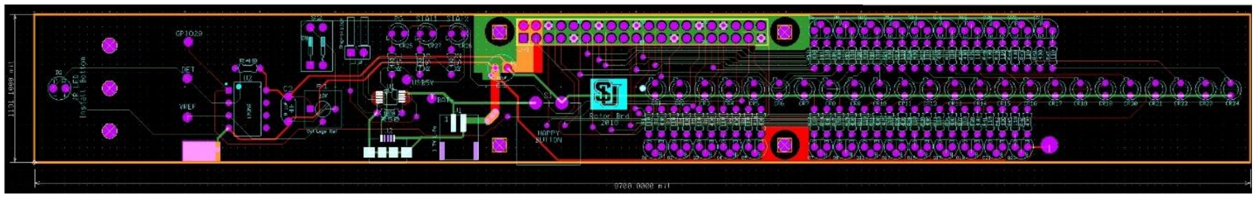


Photo of Front of Clock

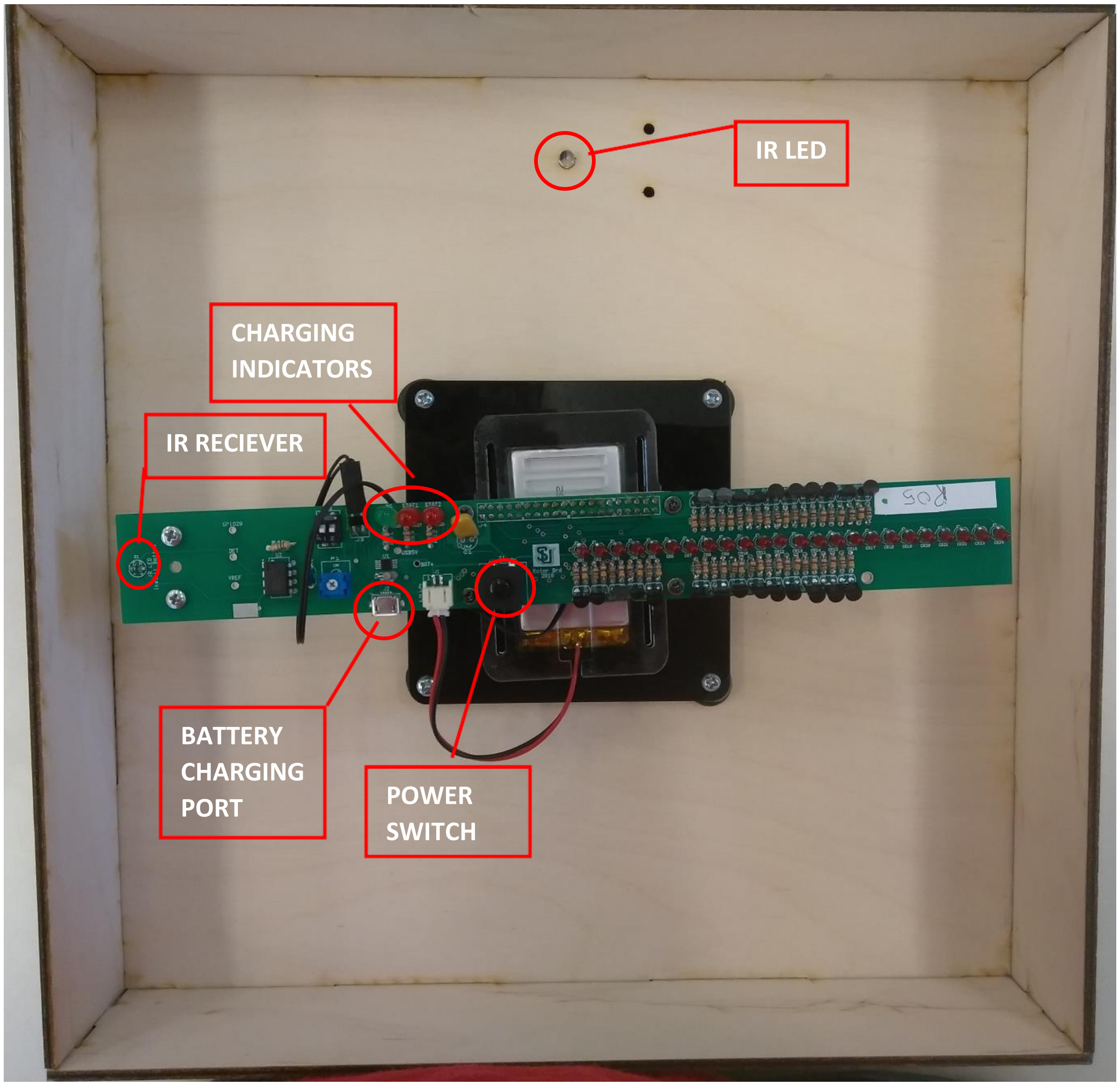
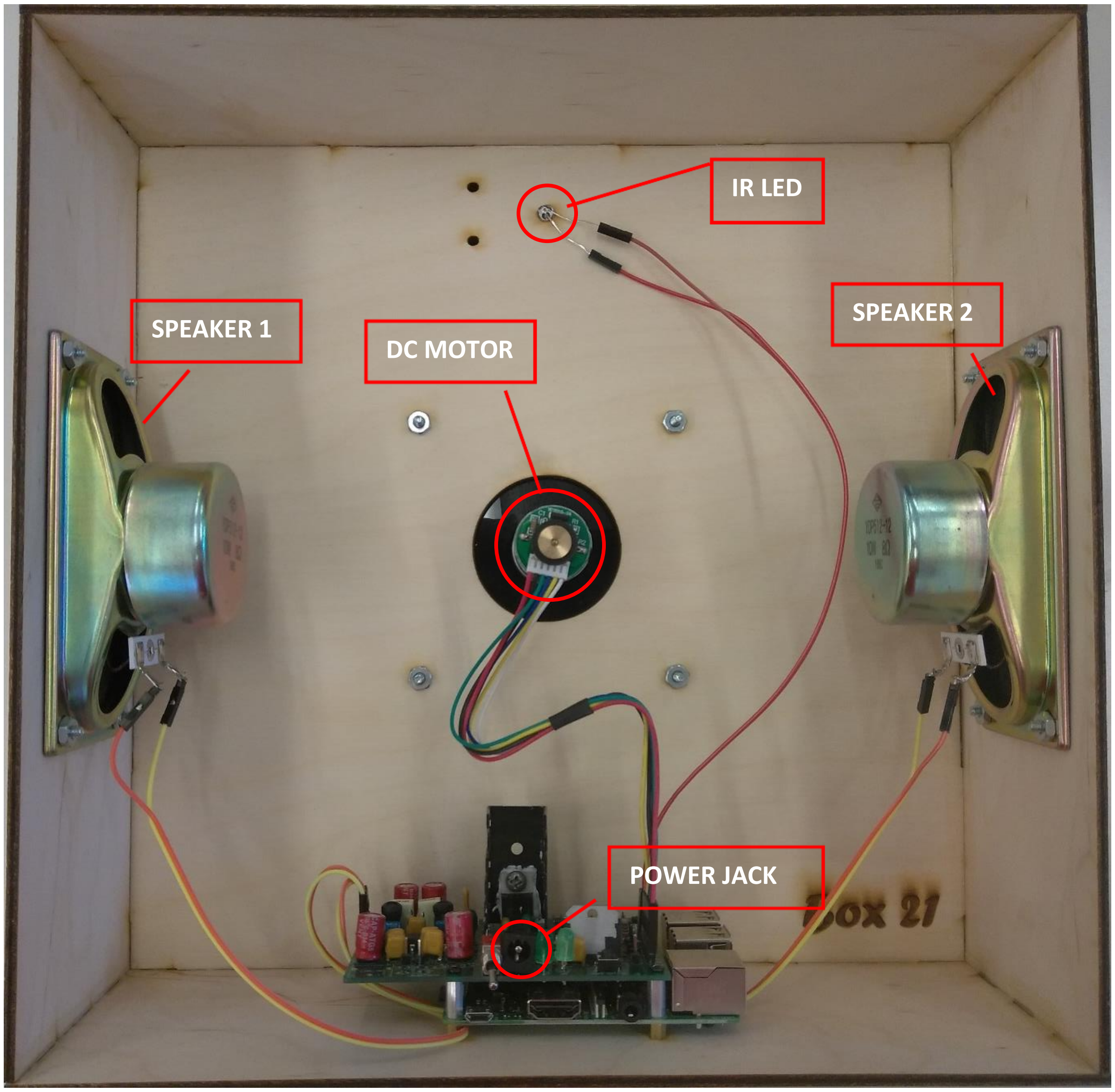


Photo of Back of Clock



Source Code for StaticPi

Include Files

motor.h

```
#ifndef MOTOR_H
#define MOTOR_H

/**
\file      motor.h
\brief     Motor Driver file.

This file provides functions to handle the motor and motor encoders.
*/

/// @brief Initialize motor pins, encoder counter ISR and RPM calulator thread
int initMotor(void);

/// @brief Stop the motor
void stopMotor(void);

/// @brief Fecth the current encoder count
/// @return motor encounter value
int getCount(void);

/// @brief Fecth the current motor RPM
/// @return motor encounter value
/// @todo Calculate RPM from count and diff_time_us (in microseconds) in
motors.c:rpmCalculatorThrFunc
int getRPM(void);

#endif
```

static_pinout.h

```
#ifndef STATIC_PINOUT_H
#define STATIC_PINOUT_H

/**
\file      static_pinout.h
\brief     GPIO Pin file of the static RPi.

This file maps all GPIO pins used by the hardware and RPi of the static portion of the POV
*/

// GPIO Input Pins
```

```
/// Email-My-IP and utility switch "S1" (input pin 37, GPIO_25)
#define IN_SW1          25
/// Motor encoder A
#define IN_MT_ENC_A     6


// GPIO Output Pins
#define OUT_DIS         5
#define OUT_PWM         4
#define OUT_AUDIO_EN    27
#define OUT_DIR         7

#endif
```

utils.h

```
#ifndef UTILS_H
#define UTILS_H

/**
\file    utils.h
\brief   Utility method file.

This file provides utility functions.
*/

#include <sys/time.h>   //struct timeval

/// @brief Method that calculated the time difference betwen two time values
/// @param end End time
/// @param start Start time
/// @return the time difference in micro-seconds
long time_diff_us(struct timeval end , struct timeval start);

/// @brief Method that returns a number from 0 to the maximum value define in the argument
/// @param max_range Maximum value of the range, must be greater than 0.
/// @return Positive random integer value, or -1 if max_range is invalid
int myrandom(int max_range);

#endif
```

web_client.h

```
#ifndef WEB_CLIENT_H
#define WEB_CLIENT_H
```

```
/**
\file      web_client.h
\brief     Web Client method file.

This file provides functions to handle the web client.
*/

/// Maximum character size of retrieved message string
#define MESSAGE_BUFFER_SIZE  2000

/// @brief This method configures and initialize the Web client, opens a socket from the local device to
a remote web server, port 10000.
/// @param ip The IP address of the web server
void initWebClient(char* ip);

/// @brief This method configures and initialize the Web client, opens a socket from the local device to
a remote web server, port as argument.
/// @param ip The IP address of the web server
/// @param newport The server port to use
void initWebClient_new_port(char* ip, unsigned short newport);


/// @brief This method retrieves the local IP address of a given network inferface (e.g 'wlan0')
/// @param interface The name of the local interface
/// @return IP address of local interface
char* getMyIP (char* interface);

/// @brief This method sends a plain text message to a pre-configured web server. All messages are sent
to port 10000.
/// @param mess The text message to send to the web server
void sendMessage(char *mess);

/// @brief This method retreives a message sent by the web server (from port 10000)
/// @return The message retrieved
const char* getMessage(void);

/// @brief Closes the socket connection to the web server
void stopWebClient(void);

#endif
```

Source Files

diagpov.h

```
/**
\file      diagpov.c
\author    Eddy Ferre - ferree@seattleu.edu
\date      01/13/2018

\brief     Main diagnostic file for the static RPi controllers..

This program sets up various sensors and actuators, then allows the user to interact from the console.
*/

// Linux C libraries
#include <stdio.h>     //printf, fprintf, stderr, fflush, scanf, getchar
#include <string.h>    //strncpy, strerror
#include <errno.h>     //errno
#include <stdlib.h>    //exit, EXIT_SUCCESS, EXIT_FAILURE
#include <signal.h>    //signal, SIGINT, SIGQUIT, SIGTERM
#include <wiringPi.h>  //wiringPiSetup, pinMode, delay, INPUT, OUTPUT, PWM_OUTPUT

// Local headers
#include "static_pinout.h"
#include "web_client.h"
#include "motor.h"


// Local function declaration
/// Function controlling exit or interrupt signals
void control_event(int sig);

/// Test function for the motor encoders sensors
int encoderTest(void);

/// Test function for the motor drive and encoders sensors
int timedMotorDriveTest();

/// Test function for the motor PWM values
int rampUpPwmTest(void);

/// Test function for the web client transmission
void messagingTest(void);


/**
```

```
main function - Entry point function for diagnostic

@param argc number of arguments passed
@param *argv array of string argumnents

@return number stdlib:EXIT_SUCCESS exit code when no error found.
*/
int main (int argc, char *argv[])
{
  // Inform OS that control_event() function will be handling kill signals
  (void)signal(SIGINT,control_event);
  (void)signal(SIGQUIT,control_event);
  (void)signal(SIGTERM,control_event);

  // Initialize the Wiring Pi facility
  if (wiringPiSetup() != 0)
  {
    // Handles error Wiring Pi initialization
    fprintf(stderr, "Unable to setup wiringPi: %s\n", strerror(errno));
    fflush(stderr);
    exit(EXIT_FAILURE);
  }

  // Parse the function argument
  int test_num;
  char c;
  if(argc < 2)
  {
    printf("  Select one of the test below:\n");
    printf("    0 - Encoder Test\n");
    printf("    1 - Timed Motor Drive Test\n");
    printf("    2 - Ramp-Up PWM Test\n");
    printf("    3 - Messaging Test\n");
    printf("  Type a test number then press ENTER... > ");
    fflush(stdout);
    scanf(" %c", &c);
    getchar();
    fflush(stdin);
  }
  else
  {
    c = argv[1][0];
  }
```

```
    if(c < '0' || c > '3')
    {
        // Handles argument type error
        fprintf(stderr, "Argument must be a number, received %c\n", c);
        fflush(stderr);
        exit(EXIT_FAILURE);
    }
    test_num = c - '0';

    // Select the test to run
    switch(test_num)
    {
    case 0:
        encoderTest();
        break;
    case 1:
        timedMotorDriveTest();
        break;
    case 2:
        rampUpPwmTest();
        break;
    case 3:
        messagingTest();
        break;
    default:
        fprintf(stderr, "Cannot handle argument \"%s\" (=%d)\n", argv[1], test_num);
        fflush(stderr);
        exit(EXIT_FAILURE);
    }
    printf("Test completed\n");
    return EXIT_SUCCESS;
}

/**
encoderTest function

Initialize the motor encoder ISR and monitors the encoder counter.
Displays the counter value

@return number '0' exit code when no error found.
*/
int encoderTest(void)
{
    printf("Running encoder Test\n");
    /// \todo
    /// * Initialize motor encoder (GPIO pin mode, GPIO pins)
    /// * Display initial counter value
    /// * Console instruction to move the motor rotor until 'ENTER' key pressed
    /// * Display the new counter value in the console
    /// * Display the motor RPM

    return 0 ;
}

/**
timedMotorDriveTest function

Initialize the motor encoder ISR and monitors the encoder counter.
Displays the counter value of the motor encoder
Displays the motor RPM

@return number '0' exit code when no error found.
*/
int timedMotorDriveTest(void)
{
    printf("Running Timed Motor Drive Test\n");

    /// \todo
    /// * Initialize motor encoder (variables, GPIO pins)
    /// * Display the current encoder count
    /// * Set the motor PWM to 100% duty-cycle
    /// * Wait for 2 seconds
    /// * Display the new counter value in the console
    /// * Display the motor RPM
    /// * Stop the motor

    return 0 ;
}

/**
rampUpPwmTest function

Initialize the motor encoder ISR and monitors the encoder counts.
Increments the motor duty-cycle by 5% every time 'ENTER' key is pressed.

@return number '0' exit code when no error found.
*/
```

```
int rampUpPwmTest(void)
{
   printf("Running Ramp Up PWM Test\n");

   /// \todo
   /// * Initialize motor encoder (variables, GPIO pins)
   /// * Until the ducty cycle (DC) is 100%, or 'q' is entered, repeat:
   ///   - Incremement DC by 5%
   ///   - wait until the "ENTER" key pressed
   /// * Stop the motor

   return 0;
}


void messagingTest(void)
{
   char ip[20], message[MESSAGE_BUFFER_SIZE];
   sprintf(message, "this is a test messages from %s", getMyIP("wlan0"));
   printf("Enter the IP to send to > ");
   fflush(stdout);
   scanf ("%s", ip);
   printf("Initialize Web Client\n");
   initWebClient(ip);
   printf("Send a message: \"%s\"\n", message);
   sendMessage(message);
   printf("Waiting for a response...\n");
   const char* response = getMessage();
   printf("Received: \"%s\"\n", response);
}


void control_event(int sig)
{
   printf("\b\b  \nExiting diagpov... ");

   //stop the motor
   stopMotor();

   delay(200);
   printf("Done\n");
   exit(EXIT_SUCCESS);
}
```

motor.c

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <wiringPi.h>
#include <pthread.h>
#include <softPwm.h>
#include <sys/time.h>

#include "motor.h"
#include "utils.h"

#define PWM_PIN           4
#define DIS_PIN          5
#define ENC_PIN           6
#define DIR_PIN          7

#define PULSE_SAMPLING_MS  250
#define MAX_PWM         1024

// Local function declaration
void* rpmCalculatorThrFunc (void* null_ptr);

volatile int pulseCounter, currentPulseCounter;

int pulseCounter_A;
pthread_t rpmCalculatorThr;
static int init_motors_done = 0;
float dutyCycle = 0;

// ISR function for the encoder counter
void counterA_ISR (void)
{
   ++pulseCounter_A;
}

// Thread function to calculate RPM
void* rpmCalculatorThrFunc (void* null_ptr)
{
   int value = 0;
   float KP = 1;
```

```
	float rpm = 1500;

   // Expected count:
   const int expected_cnt = (int)(rpm * 211.2 * (float)PULSE_SAMPLING_MS / (4000.0 * 60.0));

   while(1) {
      pulseCounter_A = 0;
      delay(PULSE_SAMPLING_MS);
      currentPulseCounter = pulseCounter_A;
      value += (int)((float)(expected_cnt - currentPulseCounter) * KP);
      value = value > MAX_PWM ? MAX_PWM : (value < 0 ? 0 : value);  // Bounds 0 <= value <=
MAX_PWM
      softPwmWrite(PWM_PIN, value);
   }
   return 0 ;
}


// init_motor: initializes motor pin (GPIO allocation and initial values of output)
// and initialize the elements of all motor control data structure
int initMotor(void)
{
	currentPulseCounter = 0;

   // Initialize GPIO pins
   pinMode(PWM_PIN, OUTPUT);
   pinMode(DIS_PIN, OUTPUT);
   pinMode(DIR_PIN, OUTPUT);
   pinMode(ENC_PIN, INPUT);
   digitalWrite(DIS_PIN, 0);  // Set disable pin to low
   digitalWrite(DIR_PIN, 1);  // set direction, 1: spin clockwise, 0: spin counter-clockwise

   if( softPwmCreate(PWM_PIN, 0, MAX_PWM) )  //Create a SW PWM, value from 0 to MAX_PWM
(=100% duty cycle)
   {
      fprintf(stderr,"Error creating software PWM: %s\n", strerror(errno));
      fflush(stderr);
      return -1;
   }

   if(!init_motors_done)
   {

      wiringPiISR(ENC_PIN, INT_EDGE_FALLING, &counterA_ISR);
```

```
        int ret = pthread_create( &(rpmCalculatorThr), NULL, rpmCalculatorThrFunc, NULL);
        if( ret )
        {
            fprintf(stderr,"Error creating rpmCalculatorThr - pthread_create() return code: %d\n",ret);
            fflush(stderr);
            return ret;
        }
    }
    init_motors_done = 1;
    return 0;
}

// stopMotor: stop the motor
void stopMotor(void)
{
            softPwmWrite(PWM_PIN, 0);
            pinMode(PWM_PIN, OUTPUT);
    digitalWrite(PWM_PIN, 0);
}


// getCount: accessor funtion of a motor encoder counter
int getCount(void)
{
    return currentPulseCounter;
}


// getRPM: accessor funtion of a motor encoder counter
int getRPM(void)
{
    return (int)((float)currentPulseCounter * 4000.0 * 60.0  / (211.2 * (float)PULSE_SAMPLING_MS));
}
```

pov.c

```
/**
\file      pov.c
\author    Eddy Ferre - ferree@seattleu.edu
\helpers            Dominic Burgi, Jeremiah Kalmus
\date      01/13/2018

\brief     Main persistence of vision file for the static RPi controllers.

This program will run the logic of the static RPi
```

```
*/

// Linux C libraries
#include <time.h>            //time
#include <stdbool.h>   //bool
#include <stdio.h>    //printf, fprintf, stderr, fflush, scanf, getchar
#include <string.h>    //strncpy, strerror
#include <errno.h>     //errno
#include <stdlib.h>    //exit, EXIT_SUCCESS, EXIT_FAILURE
#include <signal.h>    //signal, SIGINT, SIGQUIT, SIGTERM
#include <wiringPi.h>  //wiringPiSetup, pinMode, digitalWrite, delay, INPUT, OUTPUT, PWM_OUTPUT
#include <pthread.h>

// Local headers
#include "static_pinout.h"
#include "motor.h"      //initMotor
#include "web_client.h" //initWebClient_new_port

#define PORT 10000

// Local function declaration
/// Function controlling exit or interrupt signals
void control_event(int sig);
void* communicationThrFunc (void* null_ptr);
pthread_t communicationThr;

const int clock_face[]={
0x4E0000,       0x920000,       0x620000,       0x000000,       0x000000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x000000,       0xC00000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x000000,       0x000000,       0x000000,
        0x000000,       0x000000,       0xC00000,       0x000000,       0x000000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x400000,       0xA00000,       0x800000,
        0x400000,       0x800000,       0xA00000,       0x400000,       0x000000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x000000,       0xC00000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x000000,       0x000000,       0x000000,
        0x000000,       0x000000,       0xC00000,       0x000000,       0x000000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x640000,       0x920000,       0x7C0000,
        0x000000,       0x000000,       0x000000,       0x000000,       0x000000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x000000,       0xC00000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x000000,       0x000000,       0x000000,
        0x000000,       0x000000,       0xC00000,       0x000000,       0x000000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x400000,       0xA00000,       0x200000,
        0x600000,       0xA00000,       0xA00000,       0x400000,       0x000000,       0x000000,
        0x000000,       0x000000,       0x000000,       0x000000,       0xC00000,       0x000000,
        0x000000,     0x000000,     0x000000,     0x000000,     0x000000,     0x000000,
        0x000000,     0x000000,     0xC00000,     0x000000,     0x000000,     0x000000,
        0x000000,     0x000000,     0x000000,     0x420000,     0xFE0000,     0x020000,};


const int pin_table[] = {28, 27, 26, 24, 23, 22, 21, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

const int len_pin_table = 24;
const int len_value_table = 120;

const char rotor_ip[] = "10.133.0.20";
char server_ip[20];
char my_ip[20];
int hour, min, minIndex, hourIndex;
int prevMin = -1;
char buffer [MESSAGE_BUFFER_SIZE];

/**
main function - Entry point function for pov

@param argc number of arguments passed
@param *argv IP address of the rotor RPi

@return number stdlib:EXIT_SUCCESS exit code when no error found.
*/
int main (int argc, char *argv[])
{
  // Inform OS that control_event() function will be handling kill signals
  (void)signal(SIGINT, control_event);
  (void)signal(SIGQUIT, control_event);
  (void)signal(SIGTERM, control_event);

  //Local variable definition

  // Parse 1st argument: server IP
  if(argc < 2)
  {
    printf("Enter the server IP: > ");
    fflush(stdout);
    scanf(" %s", server_ip);
    getchar();
    fflush(stdin);
  }
  else
```

```
{
   sprintf(server_ip, "%s", argv[1]);
}
         sprintf(my_ip, "%s",  getMyIP("wlan0"));
printf("  rotor RPi IP     : %s\n", rotor_ip);
printf("  My wireless IP is: %s\n", my_ip);

// Initialize the Wiring Pi facility
         printf("Initialize Wiring Pi facility... ");
if (wiringPiSetup() != 0)
{
   // Handles error Wiring Pi initialization
   fprintf(stderr, "Unable to setup wiringPi: %s\n", strerror(errno));
   fflush(stderr);
   exit(EXIT_FAILURE);
}
         printf("Done\n");

// Initialize GPIO pins
printf("Initialize GPIO pins... ");
pinMode(IN_SW1, INPUT);
pinMode(OUT_AUDIO_EN, OUTPUT);
digitalWrite(OUT_AUDIO_EN, 0);
         system("gpio mode 23 ALT0");
system("gpio mode 26 ALT0");
         system("gpio mode 4 OUTPUT");
         system("gpio mode 5 OUTPUT");
         system("gpio mode 27 OUTPUT");
         system("gpio write 4 0");
         system("gpio write 5 0");
         system("gpio write 27 1");
         printf("Done\n");

// Start encoder counter ISRs, setting actual motor RPM
         printf("Initialize Motor... ");
initMotor();
         printf("Done\n");

         initWebClient_new_port(server_ip, PORT);

         int image_table[120];
int ret = pthread_create( &(communicationThr), NULL, communicationThrFunc, NULL);
if( ret )
{
   fprintf(stderr,"Error creating communicationThrFunc - pthread_create() return code: %d\n",ret);
   fflush(stderr);
   return ret;
}

printf("Start Main loop\n");
while(1){
			//Update the time, set command to display
			time_t ugly_time;
			struct tm *pretty_time;
			time(&ugly_time);
			pretty_time = localtime(&ugly_time);
			min = pretty_time->tm_min;
			hour = pretty_time->tm_hour;

			if (min != prevMin) {

				minIndex = min*2;
				hourIndex = (10*hour + min/6) % 120;
				for(int i = 0;i < 120;i++){
					image_table[i] = clock_face[i];
				}

				//make copy of clock_face and edit that copy
				image_table[minIndex] |= 0xFFFFFF;
				image_table[(minIndex+119) % 120] |= 0x1FFFFF;
				image_table[(minIndex+1) % 120] |= 0x1FFFFF;
				image_table[hourIndex] |= 0x000FFF;
				image_table[(hourIndex+119) % 120] |= 0x0001FF;
				image_table[(hourIndex+1) % 120] |= 0x0001FF;

				sprintf(buffer, "%s,%s,display",my_ip,rotor_ip);
				for (int i = 0; i < len_value_table; i++) {
					if (image_table[i] != 0) {
						sprintf(buffer, "%s,%d,%X", buffer, i, image_table[i]);
					}
				}
				sprintf(buffer, "%s\n", buffer);
				sendMessage(buffer);
				if(min % 15 == 0){
					system("omxplayer -o local --vol=-1000 hourlychimebeg.mp3");
				} else {
					system("omxplayer -o local --vol=-1000 Meep.mp3");
				}
```

```
                        }



                        prevMin = min;
        delay(500);  //Always keep a sleep or delay in infinite loop
    }
    return 0;
}

void control_event(int sig)
{
    printf("\b\b  \nExiting pov... ");
            sprintf(buffer, "%s,%s,display,0,0\n",my_ip,rotor_ip);
            sendMessage(buffer);
            delay(400);
    //stop the motor
    stopMotor();
    delay(400);
    printf("Done\n");
    exit(EXIT_SUCCESS);
}

void* communicationThrFunc (void* null_ptr)
{
            char *token, *token2;
            char rcv_rotor_ip[20];
            const char delim[] = ",\n";
            char message[MESSAGE_BUFFER_SIZE];

            // Lock remote device to this pi


            while(1)
            {
                        sprintf(message, "%s", getMessage());
                        //printf("received: %s", message);
                        //fflush(stdout);
                        token = strtok(message, delim);
                        sprintf(rcv_rotor_ip, "%s", token);
                        token = strtok(NULL, delim);

                        if(strcmp(rcv_rotor_ip, rotor_ip))
                        {
                                    printf("\nReceived message from unknown RPi \"%s\n", rcv_rotor_ip);
                                    fflush(stdout);
                        } else {
                                    token = strtok(NULL, delim);
                                    if (!strcmp(token, "response"))
                                    {
                                            token = strtok(NULL, delim);
                                            token2 = strtok(NULL, delim);
                                            if (strcmp(token2, "ok"))
                                            {
                                                    printf("\nCommand \%s returned : \%s\\n", token, token2);
                                                    fflush(stdout);
                                            }
                                    } else {
                                            printf("\nReceived unknown command \%s\n", token);
                                            fflush(stdout);
                                    }
                        }
            }
}
```

utils.c

```
#include <sys/time.h>    //struct timeval
#include <stdlib.h>     //rand, srand, RAND_MAX
#include <stdio.h>      //fprintf, stderr
#include <time.h>       //time

#include "utils.h"

long time_diff_us(struct timeval end , struct timeval start)
{
   return (end.tv_sec - start.tv_sec) * 1000000 + end.tv_usec - start.tv_usec;
}

int myrandom(int max_range)
{
   if(max_range <= 0 || max_range > RAND_MAX)
   {
      fprintf(stderr, "ERROR: Invalid argument max_range (=%d) of function myrandom", max_range);
      return -1;
   }
   srand(time(NULL));  //use current time as seed for random generator
   return rand() % max_range;
}
```

web_client.c

```
#include <stdio.h>
```

```
#include <string.h>    //strlen, strcmp
#include <stdlib.h>    //strlen
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h> //inet_addr
#include <unistd.h>    //write
#include <sys/types.h>
#include <ifaddrs.h>

#include "web_client.h"

static char myIP[NI_MAXHOST] = "";
unsigned short port = 10000;
struct sockaddr_in server;
int socket_desc;
int connected;
char serverIp[20];

char* getMyIP (char* interface)
{
   struct ifaddrs *ifaddr, *ifa;
   int s;

   if (getifaddrs(&ifaddr) == -1)
   {
      perror("getifaddrs");
      exit(EXIT_FAILURE);
   }
   for (ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next)
   {
      if (ifa->ifa_addr == NULL)
         continue;
      s=getnameinfo(ifa->ifa_addr,sizeof(struct sockaddr_in),myIP, NI_MAXHOST, NULL, 0,
NI_NUMERICHOST);
      if((strcmp(ifa->ifa_name,interface)==0)&&(ifa->ifa_addr->sa_family==AF_INET))
      {
         if (s != 0)
         {
            printf("getnameinfo() failed: %s\n", gai_strerror(s));
            exit(EXIT_FAILURE);
         }
         break;
      }
   }
```

```
    freeifaddrs(ifaddr);
    return myIP;
}


void connectToServer(void)
{
    if(!connected)
    {
        //Create socket
        printf("Create socket\n");
        socket_desc = socket(AF_INET, SOCK_STREAM , 0);
        if (socket_desc < 0)
        {
            fprintf(stderr, "connect error");
            return;
        }
        printf("Configure socket\n");
        server.sin_addr.s_addr = inet_addr(serverIp);
        server.sin_family = AF_INET;
        server.sin_port = htons( port );
        //Connect to remote server
        printf("Connect to remote server\n");
        if (connect(socket_desc , (struct sockaddr *)&server , sizeof(server)) < 0)
        {
            fprintf(stderr, "connect error\n");
            return;
        }
        printf("Connected to server %s:%d\n", serverIp, port);
        fflush(stdout);
        connected = 1;
    }
}


void initWebClient(char* ip)
{
    strncpy(serverIp, ip,  sizeof serverIp - 1);
    connected = 0;
    connectToServer();
}

void initWebClient_new_port(char* ip, unsigned short newport)
{
    strncpy(serverIp, ip,  sizeof serverIp - 1);
    connected = 0;
    port = newport;
    connectToServer();
}


void sendMessage(char *mess)
{
    if(connected)
    {
        if( send(socket_desc , mess , strlen(mess) , 0) < 0)
        {
            perror("send");
            connected = 0;
        }
    }
}


const char* getMessage(void)
{
    char *server_reply = (char*)malloc(MESSAGE_BUFFER_SIZE * sizeof(char));
    server_reply[0] = '\0';
    if(connected)
    {
        if( recv(socket_desc, server_reply , MESSAGE_BUFFER_SIZE , 0) < 0)
        {
            perror("recv");
            connected = 0;
        }
    }
    return server_reply;
}


void stopWebClient(void)
{
    if(connected)
    {
        if( close(socket_desc) < 0)
        {
            perror("close");
            connected = 0;
```

```
      }
   }
}
```

Source Code for RotorPi

Include Files

static_pinout.h

Same as listed above for StaticPi

utils.h

Same as listed above for StaticPi

web_client.h

Same as listed above for StaticPi

Source Files

utils.c

Same as listed above for StaticPi

web_client.c

Same as listed above for StaticPi

led.c

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <wiringPi.h>
#include <pthread.h>
#include <web_client.h>

int clock_face[120];

void counter_ISR(void);
void* communicationThrFunc(void* null_ptr);
pthread_t communicationThr;

const int pin_table[] = {28, 27, 26, 24, 23, 22, 21, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

int value_idx = 0;
int main()
{
        wiringPiSetup();
```

```
	const int len_pin_table = 24;
	const int len_value_table = 120;

	int table_idx = 0;
	int delay = 243;
	//for(int i = 0;i <

	for (int i = 0; i < len_pin_table; i++){
		pinMode(pin_table[i], OUTPUT);
	}

	for(int i = 0;i < 120;i++){
		clock_face[i] = 0;
	}

	wiringPiISR(29, INT_EDGE_RISING, &counter_ISR);
	initWebClient_new_port("10.122.61.34", 10000);
	int ret = pthread_create( &(communicationThr), NULL, communicationThrFunc, NULL);
  if( ret )
  {
    fprintf(stderr,"Error creating communicationThrFunc - pthread_create() return code: %d\n",ret);
    fflush(stderr);
    return ret;
  }

	while(1)
	{
		for(table_idx = 0; table_idx < len_pin_table; table_idx++)
		{
			digitalWrite(pin_table[table_idx], (clock_face[value_idx] & (1 << table_idx)));
		}
		delayMicroseconds(delay);
		value_idx++;
		value_idx %= len_value_table;
	}


	return 0;
}

void counter_ISR(void) {
	value_idx = 63;
}
```

```
void* communicationThrFunc(void* null_ptr)
{
	char response[MESSAGE_BUFFER_SIZE];
	char message[MESSAGE_BUFFER_SIZE];
	char* token;
	char static_ip[20];
	char my_ip[20];
	char delim[] = ",\n";

	while(1){
		sprintf(message, "%s", getMessage());
		token = strtok(message, delim);
		sprintf(static_ip, "%s", token);

		token = strtok(NULL, delim);
		sprintf(my_ip, "%s", token);

		token = strtok(NULL, delim);

		if(!strcmp(token, "display")){
			for(int i = 0;i < 120;i++){
				clock_face[i] = 0;
			}
			token = strtok(NULL, delim);
			while(token != NULL){
				int i = strtol(token, NULL, 10);
				token = strtok(NULL, delim);
				int val = strtol(token, NULL, 16);
				clock_face[i] = val;
				token = strtok(NULL, delim);
			}
			sprintf(response, "%s,%s,response,display,ok\n", my_ip, static_ip);
			sendMessage(response);
		}
	}

}
```

Source Code for POVserver.py

```
#!/usr/bin/python

from time import sleep
from traceback import format_exc
from socket import socket, error, AF_INET, SOCK_STREAM, SOL_SOCKET, SO_REUSEADDR
from select import select
from threading import Thread
from re import match


class Server(Thread):
  def __init__(self, port):
    Thread.__init__(self)
    self.daemon = True
    self.port = port
    self.srvsock = socket(AF_INET, SOCK_STREAM)
    self.srvsock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    self.srvsock.bind(("", port))
    self.srvsock.listen(5)
    self.descriptors = [self.srvsock]
    print('Server started on port {}'.format(port))

  def run(self):
    while 1:
      # Await an event on a readable socket descriptor
      (sread, swrite, sexc) = select(self.descriptors, [], [])
      # Iterate through the tagged read descriptors
      for sock in sread:
        # Received a connect to the server (listening) socket
        if sock == self.srvsock:
          self.accept_new_connection()
        else:
          try:
            # Received something on a client socket
            try:
              string = str(sock.recv(2500).decode('utf-8'))
            except UnicodeDecodeError:
              string = "invalid character transmitted"
              pass
            host, port = sock.getpeername()
            # Check to see if the peer socket closed
            if string == '' or string == 'q':
              print('Client left {0}:{1}'.format(host, port))
```

```
                        sock.close()
                        self.descriptors.remove(sock)
                    else:
                        print('{0} says: "{1}"'.format(host, string.rstrip()))
                        fwd_sock = None
                        if self.validate(string):
                            dest_ip = string.split(',')[1]
                            fwd_sock = self.get_sock(dest_ip)
                            if fwd_sock is not None:
                                try:
                                    fwd_sock.send(string.encode())
                                    print('  Forward successful')
                                except error:
                                    host, port = fwd_sock.getpeername()
                                    print('Socket Error - Client left {0}:{1}'.format(host, port))
                                    fwd_sock.close()
                                    self.descriptors.remove(fwd_sock)
                            else:
                                print('  No target found, message dismissed')
                                resp = 'Error: No target found, message dismissed\n'
                                sock.send(resp.encode())
                        else:
                            print('  Invalid command, message dismissed')
                            resp = 'Error: Invalid command, message dismissed\n'
                            sock.send(resp.encode())
                except (error, OSError):
                    host, port = sock.getpeername()
                    print('Socket Error - Client left {0}:{1}'.format(host, port))
                    sock.close()
                    self.descriptors.remove(sock)

def get_sock(self, ip):
    ret_sock = None
    for sock in self.descriptors:
        try:
            host, port = sock.getpeername()
            if host == ip:
                ret_sock = sock
                break
        except (error, OSError):
            pass
    return ret_sock

def validate(self, string):
        m =
match(r"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\,\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\,(test|display|response|lo
ck|unlock)\,.+\n", string)
        return m is not None

    def accept_new_connection(self):
        newsock, (remhost, remport) = self.srvsock.accept()
        self.descriptors.append(newsock)
        print('Client joined {0}:{1}'.format(remhost, remport))


def main():
    myServer = Server(10000)
    myServer.start()
    try:
        while 1:
            # Do nothing...
            sleep(1.0)
    except KeyboardInterrupt:
        print('Exiting...')


if __name__ == '__main__':
    main()
```

Instructions

Adjusting the Rotor Potentiometer

The potentiometer on the Rotor Pi is something that has to be set to ensure that the infrared transceiver is operating correctly. We would like to find the range of exposure for the potentiometer where when it's turned all the way to one side it is fully exposed and the opposite limit is no exposure. From this, we would like to find a reference voltage about halfway between the two limits for our potentiometer.

This reference voltage can be found by opening a putty session that is connected to the Static Pi and by using the WiringPi library, there is a command called "gpio readall". Use this command to open a window that monitors all input and output devices on the Static Pi. From here we simply passed the transmitter and receiver components over each other and found a position for the potentiometer that sent a high signal across the sender and receiver circuits when they passed each other and sent a low signal otherwise.

Powering the StaticPi and Charging RotorPi Battery

Powering the Static Pi is simple, we merely plug a 12-volt DC plug into the respective port on the Static Pi. For the Rotor Pi, it is powered using a battery that can be charged with a micro USB cord. Plug the micro USB into the port on the Rotor Pi and attach the other end to a USB power source. Most importantly, make sure to have the Rotor Pi POWERED OFF WHILE CHARGING.

Starting the Clock

To start our POV clock, we first need to power the Static Pi and have the Rotor Pi battery charged. The Static Pi and the Rotor Pi then need to be powered on. Next, we connected to our Static Pi and Rotor Pi by opening a putty session for each. We go to the directory in the Static Pi session that houses the pov executable and on the Rotor Pi session we go to the directory that houses the led executable. We then run the command "sudo ./led" on the Rotor Pi to activate the LED matrix needed for the clock face. Then, on the Static Pi we entered "sudo ./pov" to start the DC motor and keep track of the clock data.

Stopping the Clock

To stop the clock, we merely use ctrl+c to terminate the program. This is due to the fact that we have the clock in an infinite loop in the code. This is because the clock has the potential to run for a long period of time and we would like it to stop only when the user manually shuts the clock down.