Painting Monitoring System

Final Project Report
December 8, 2017
E155

Richard Liu and Kimberly Joly

Abstract

This system monitors temperature, light, and human proximity to the painting, responds accordingly to preserve the painting, and was implemented using an FPGA and Raspberry Pi. If the temperature, light, or proximity go over a certain threshold, an LED turns on to warn the user of possible environmental stresses. When the amount of light is above a certain threshold or the user manually controls the curtain, the system triggers a stepper motor via the FPGA to lower the curtain. The Raspberry Pi hosts the web server, reads sensor inputs, and sends to appropriate curtain control signal to the FPGA.

1. Introduction

Paintings may be subjected to different environmental stresses, such as being exposed to too much light, fluctuations in temperature, and human pollution through touch. A growing IoT field, cheap yet powerful computers, and an abundance of available sensors may help solve this problem. This project creates a system to monitor temperature, light, and human proximity to a painting. The system then responds accordingly based on the light sensor input or human input to “protect” the painting by notifying the user or lowering the curtain. For example, lowering the curtain due to high luminosity levels will protect the painting because prolonged exposure to light can lead to colors fading. It also alerts the user if a sensor value goes above a certain threshold via alert LEDs.

The Raspberry Pi reads data from sensors, notifies the user when a sensor value is above a certain threshold by lighting LEDs on the uMudd, hosts the Apache web server to interface with the system, and sends appropriate signals to the FPGA to control the percentage to which the curtain should close. The FPGA then controls the stepper motor based on the signal received. Figure 1 below details the relationship between each component of the system.

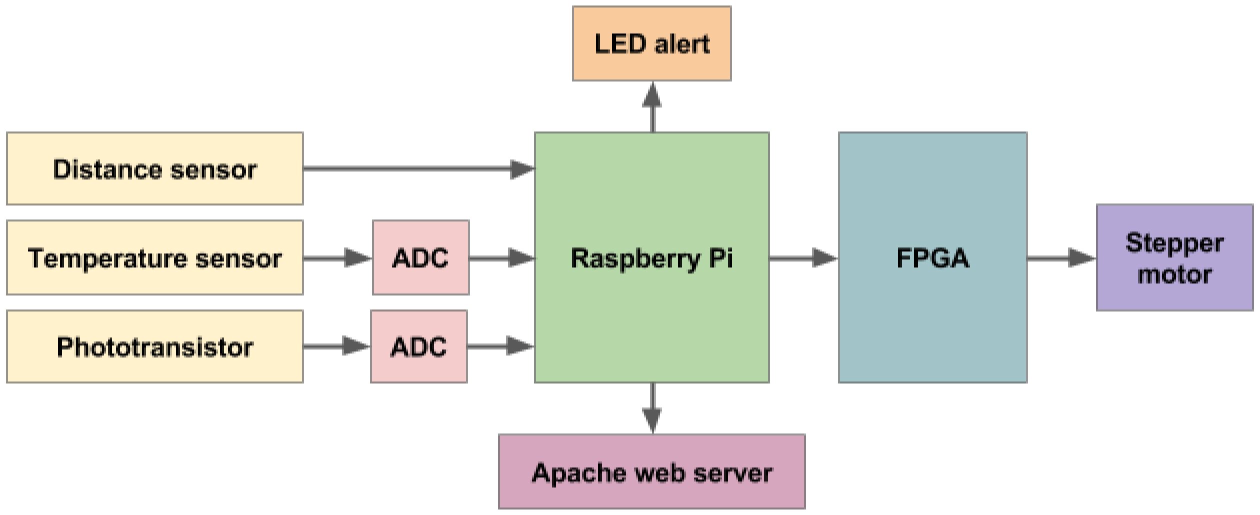


Figure 1: High-level structure of the system

2. New Hardware

This project uses a stepper motor found in the MicroPs lab. The motor is a Jameco 171601 unipolar stepper motor [1] controlled by the FPGA, and driven by a L293DNE H-bridge [2].

The stepper motor uses four wires to power the electromagnet. The wires are powered using an external voltage source due to the high current draw. To drive the motor clockwise using wave modulation, the team powered the motor with the four steps shown in Table 1. To make the motor rotate counterclockwise, the same steps are followed in reverse order.

	Wire color			
	Yellow	Red	Black	Gray
Step A	0V	0V	0V	5V
Step B	0V	0V	5V	0V
Step C	0V	5V	0V	0V
Step D	5V	0V	0V	0V

Table 1: The four steps used to power the motor in the clockwise direction

Wave modulation requires the least current, since only one coil of the electromagnet is powered at a time. An H-bridge was used so that the FPGA can drive the stepper motor while using an external voltage supply. The wiring of the stepper motor can be seen below in Figure 2.

3. Schematics

The schematic of the system is found in Figure 2 below.

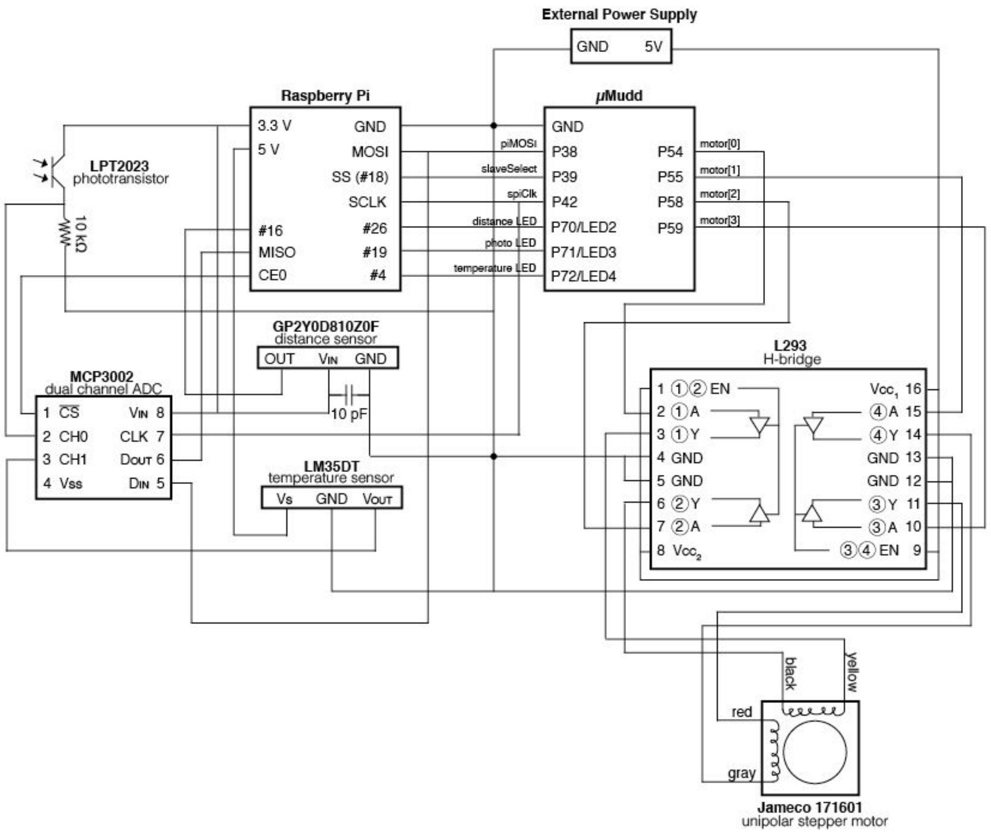


Figure 2: Schematic of the circuit design

4. Microcontroller (Raspberry Pi) Design

The Raspberry Pi completes a few tasks as a microcontroller. First, it reads the data from three sensors, directly as a digital signal for the distance sensor, or via SPI for the phototransistor and temperature sensor from an ADC. It then controls three LEDs on the uMudd based on the sensor input. Second, it sends a control signal to the FPGA via SPI, indicating to what percentage the curtain should be closed. Third, it hosts the web server used by the user to interact with the system. Figure 3 shows a high-level block diagram of this system.

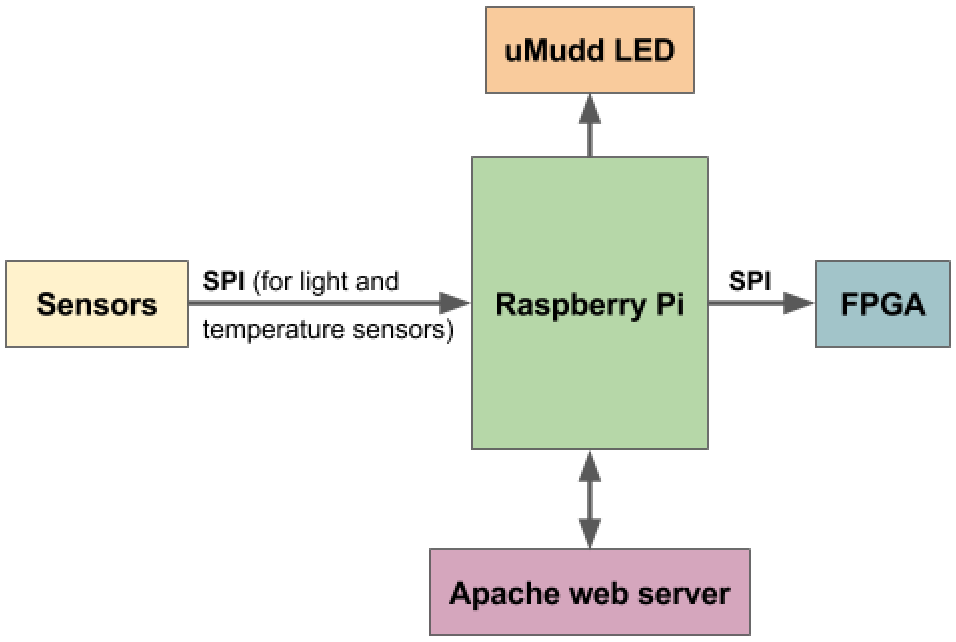


Figure 3: Microcontroller block diagram logic

4.1. Read sensor data

The Raspberry Pi reads data from the distance, temperature, and light sensors. The distance sensor (GP2Y0D810Z0F) is a digital sensor, so the Raspberry Pi reads in the signal of the pin connected to sensor “OUT” pin. The phototransistor (LPT2023) and temperature sensor (LM35DT) output an analog signal, which must be passed into channels 1 and 2 of the ADC, respectively. The sensor outputs are then sent to the Raspberry Pi pins via SPI. The code used for the phototransistor and temperature build of the SPI communication code written for Lab 6.

Three .c files, found in Appendix A, read data from each of the sensors, and display them on the webpage. If the sensors read data that is above a specified threshold, the Raspberry Pi lights up a uMudd LED as an “alert” to the user.

4.2. Send control signal to the FPGA

Whenever the user presses a button which controls the curtain, or the curtain is lowered due to high light levels, the Raspberry Pi sends a control signal to the FPGA via SPI, indicating to what percentage the curtain should be lowered.

The SPI clock sends packages of 16 bits at a time with the `spiSendReceive16()` function, after initializing SPI with the `spiInit()` function. The first 3 bits define to which percentage the curtain will close. The last 13 bits are ignored and have a value of 0s because 3 bits are sufficient to express the different percentages to which the curtain will close. Table 2 below details the 7 possible percentages to which the curtain can close and the corresponding configurations sent from the Raspberry Pi to the FPGA to control the motor.

Percentage (%)	**Configuration**		
	First bit	Second bit	Third bit
0	0	0	0
17	0	0	1
34	0	1	0
50	1	0	0
67	1	0	1
84	1	1	0
100	1	1	1

Table 2: Curtain percentage closed and corresponding configuration

The first bit corresponds to 50, the second bit corresponds to 34, and the third bit corresponds to 17. Each state is made up of a combination of the values when added. For example, 67 = 50+17, so the encoding for the state corresponding to 67% is 101.

For the FPGA to know when to read the SPI clock data being sent from the Raspberry Pi, the Raspberry Pi sets the slave select pin to low (0V) when the 16-bit serial data is being sent. The slave select pin will be set to high (3.3V) as soon as the packet has been sent. An example of this control signal is shown in Figure 4 below.

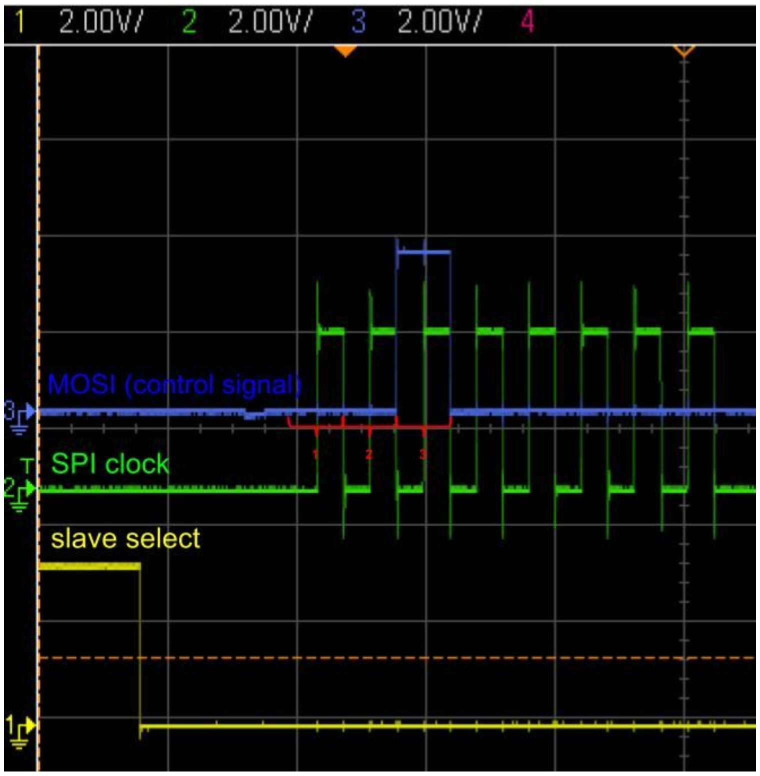


Figure 4: Example of the 17% (001) curtain control signal

The functions `curtain0`, `curtain17`, `curtai34`, `curtain50`, `curtain67`, `curtain84`, and `curtain100` sends the control signal, and takes in the percentage to which the curtain should be closed as an input. They are written in C and can be found in Appendix A.

4.3. Host the web server

The Raspberry Pi also hosts an Apache Web Server, so the user can interact with the system by viewing sensor data and controlling the curtain. The main page, `mainPage.html`, is found in Appendix A, and is store in the `/var/www/html` directory in the Raspberry Pi. The other files mentioned in this section, used to read sensor data and send curtain control signals, are stored as executables in the `/usr/lib/cgi-bin`, and their permissions modified with the `chown` and `chmod` commands.

5. FPGA Design

The main tasks for the FPGA include: reading the new curtain configuration sent from the Raspberry Pi, reading the current configuration from memory, writing the new configuration into memory, comparing the current and new configurations, calculating the number of rotations needed to modify the current to new configuration, and outputting pulses using wave modulation to control the stepper motor. A high-level block diagram is shown in Figure 5 below.

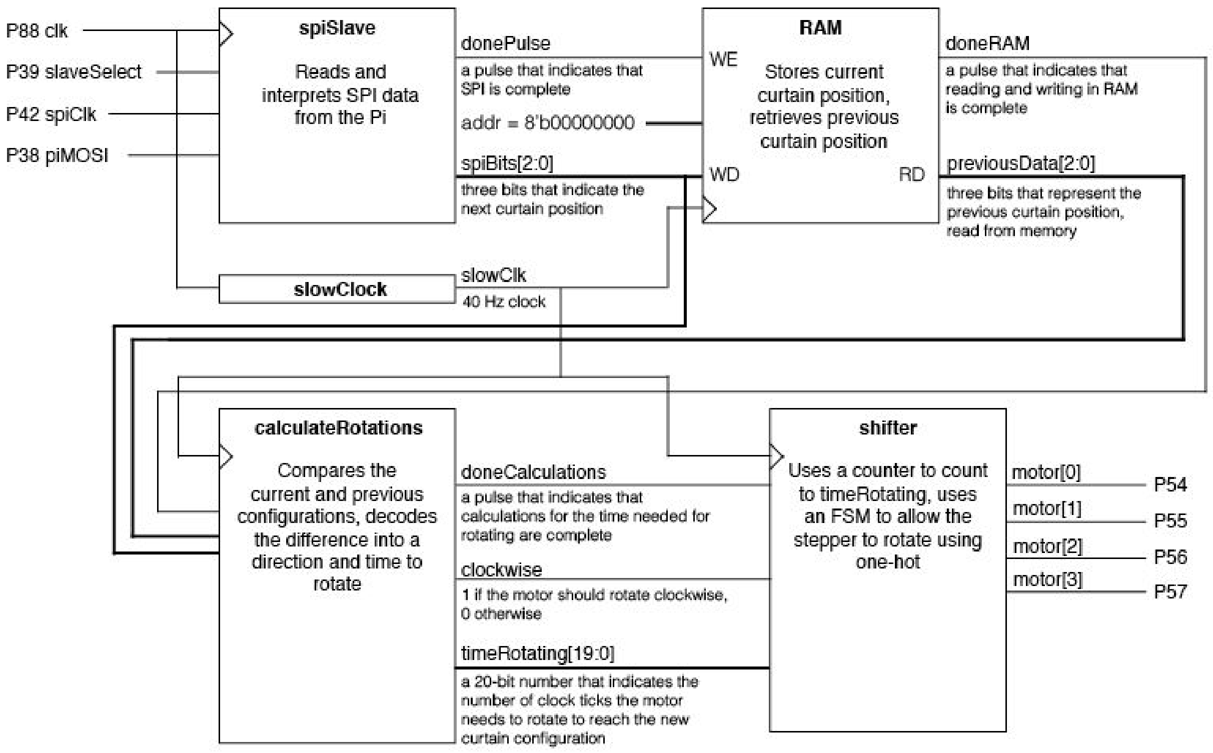

P88 clk
P39 slaveSelect
P42 spiClk
P38 piMOSI
spiSlave
Reads and interprets SPI data from the Pi
donePulse
a pulse that indicates that SPI is complete
addr = 8'b00000000
spiBits[2:0]
three bits that indicate the next curtain position
RAM
WE
Stores current curtain position, retrieves previous curtain position
WD
RD
doneRAM
a pulse that indicates that reading and writing in RAM is complete
previousData[2:0]
three bits that represent the previous curtain position, read from memory
slowClock
slowClk
40 Hz clock
calculateRotations
Compares the current and previous configurations, decodes the difference into a direction and time to rotate
doneCalculations
a pulse that indicates that calculations for the time needed for rotating are complete
clockwise
1 if the motor should rotate clockwise, 0 otherwise
timeRotating[19:0]
a 20-bit number that indicates the number of clock ticks the motor needs to rotate to reach the new curtain configuration
shifter
Uses a counter to count to timeRotating, uses an FSM to allow the stepper to rotate using one-hot
motor[0]
motor[1]
motor[2]
motor[3]
P54
P55
P56
P57

Figure 5: A block diagram of the FPGA modules

5.1. SPI Slave

The SPI slave is represented by the `spiSlave` module. The module utilizes a slave select pin, as the FPGA is not the only SPI slave. Until the slave select pin goes low, the FPGA waits for instructions from the Raspberry Pi. Since only three bits are relevant, only those bits are saved in memory. A finite state machine, shown in Figure 6, is used to record these bits.

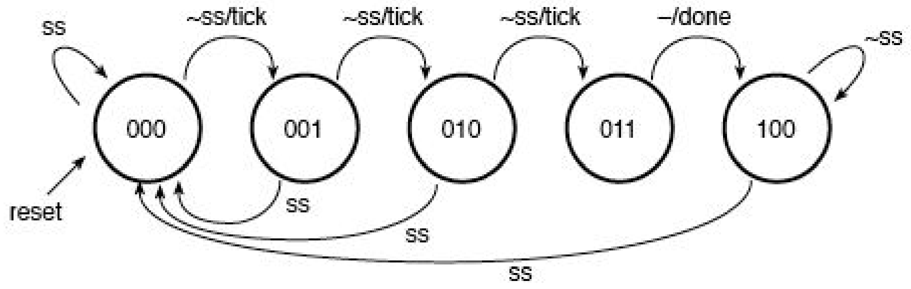


Figure 6: The FSM used in the `spiSlave` module (ss represents `slaveSelect`, tick enables a flip flop to store the value outputted by the Raspberry Pi, and done outputs `donePulse`)

Once the bits are successfully recorded, the module outputs a pulse (represented by `donePulse`) that acts as the write enable pin for memory.

5.2. Slow Clock

The faster a stepper motor rotates, the less torque it supplies. Since the stepper motor does not supply much torque in the first place, the team had to keep the rotation speed low. To do so, a 40 Hz clock is created mainly for motor controls. The slow clock module is represented by `slowClock`. It uses a counter and ticks at a threshold.

5.3. Memory

Since the Raspberry Pi outputs the desired curtain configuration, the FPGA stores the current curtain configuration to be compared. Since the RAM only reads and writes into one address, it performs the tasks of a 3-bit register. After saving new data into memory, a pulse (`doneRAM`) is outputted to start the next block to calculate the time needed to raise or lower the curtain. This modified version of RAM is represented by the `memory_KJRL` module.

5.4. Calculating Rotations

The `calculateRotations` module takes care of determining the direction the motor needs to rotate and for how much time. Depending on the difference between the current and new configurations, the motor will rotate clockwise or counterclockwise. The difference between the two 3-bit numbers is decoded into the amount of time needed for rotations. Once calculations are done, a pulse (`doneCalculations`) is sent to the next module to reset the rotation counter, signaling that the motor should start rotating.

5.5. Controlling the Stepper Motor

Wave modulation is used to control the stepper motor, which allows rotation in either direction. To implement wave modulation, the team used one-hot. An FSM, represented by the shifter module, is used to power the motor. Depending on the direction the motor needs to rotate, the module shifts the single high bit left or right. By encoding the state in one-hot, the output of the FSM is the state. The FSM is represented by Figure 7.

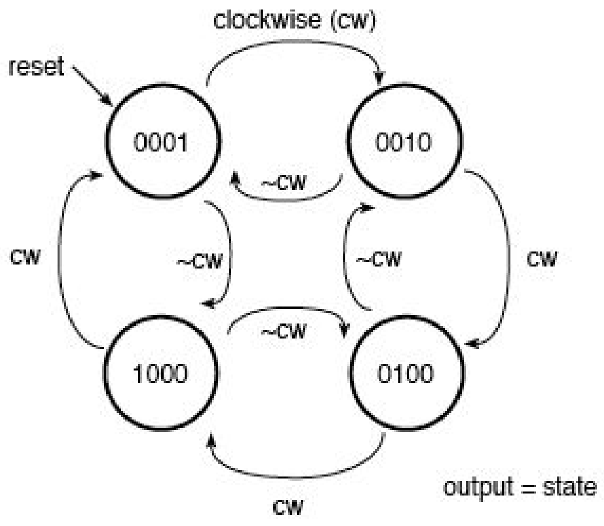
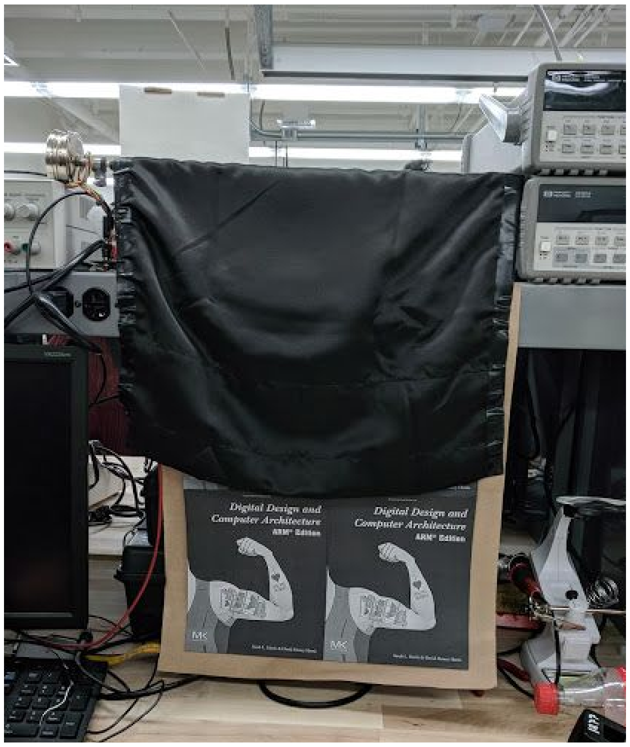


Figure 7: The FSM used in the shifter module. If cw reads 1, then the motor is rotating clockwise. The output is equivalent to the state.

6. Results

6.1. Curtain Controls

The stepper motor is successful in controlling the curtain. The user is able to click on the percentages on the website and the motor will rotate the dowel, causing the curtain to move up or down. If the user shines a light on the phototransistor, such as an cell phone LED flashlight, the Raspberry Pi will force the curtains to lower. A photo of the functioning project can be seen in Figure 8.


Digital Design and Computer Architecture ARM Edition
Digital Design and Computer Architecture ARM Edition
Figure 8: The curtain at the 50% configuration

However, after many iterations of controlling the curtain, the curtain will become uncalibrated from its reference point of 0%, by not being fully opened when the 0% button is pressed. Professor Spencer suggested that the stepper motor could be skipping due to its small torque. The team believes that curtain is lightweight and does not roll onto the dowel properly, causing folds in the material. To combat this problem, a weight is added to the bottom of the curtain, which forced the curtain to roll more smoothly when raised. These factors contributed to the imperfect motor controls and suggest that a better mechanical design may be more desirable.

References

[1] https://www.jameco.com/jameco/products/prodds/171601.pdf
[2] http://www.ti.com/lit/ds/symlink/l293.pdf

Parts List

The following table lists the components used for this project, other than standard resistors, capacitors, and parts available in the MicroP's lab:

Part	**Source**	**Vendor Part #**	**Price**
LM35DT temperature	Mouser Electronics	926-LM35DT/NOPB	2.90

sensor			
Sharp digital distance sensor with Pololu carrier	Adafruit	GP2Y0D810Z0F	6.95

Appendix A: Raspberry Pi Code

Code for the phototransistor:

```
#include “stdio.h”
#include “EasyPIO.h”
#include “GPIO.h”

int sample;
float voltage;
Float scaledvoltage;
int threshold = 96;

int turnOn(void) {
       setPinType(19,1);
       digitalWrite(19,1);
       return 0;
}

int turnOff(void) {
       setPinType(19,1);
       digitalWrite(19,0);
       return 0;
}

int main(void) {
       char received;
       pioInit();
       SetUpPerif();
       spiInit(244000,0);
       sample = spiSendReceive16(0x6000);
       voltage = (sample * 3.3) / 1024;
       scaledvoltage = voltage*100/3.3;

       if (scaledvoltage >= threshold) {
              turnOn();
              curtain100();
       }

       else if (scaledvoltage < threshold) {
              turnOff();
       }

       printf(“Content-type: text/html\n\n”);
       printf(“<html>\n”);
       printf(“<head>\n”);
       printf(“<meta http-equiv=’refresh’ content =’5’>”);
       printf(“</head>\n”);
       printf(“<body>\n”);
       printf(“<h1> The temperature is: “);
       printf(“%.3f</h1>\n”, temperature);
       printf(“</body>\n”);
       printf(“</html>\n”);
       return 0;
}
```

Code for the distance sensor:

```
#include “stdio.h”
#include “EasyPIO.h”
#include “GPIO.h”

int turnOn(void) {
	SetUpPerif();
	setPinType(26,1);
	digitalWrite(26,1);
	return 0;
}

int turnOff(void) {
	SetUpPerif();
	setPinType(26,1);
	digitalWrite(26,0);
	return 0;
}

char* status;
int i;

int main(void) {
	pioInit();
	pinMode(16,0);
	int dist = digitalRead(16);

	if (dist == 0) {
		status = “There is something there!”;
		turnOn();
	}

	else if (dist == 1) {
		status = “There isn’t anything there!”;
		turnOff();
	}

	printf(“Content-type: text/html\n\n”);
	printf(“<html>\n”);
	printf(“<head>\n”);
	printf(“<meta http-equiv=’refresh’ content =’5’>”);
	printf(“</head>\n”);
	printf(“<body>\n”);
	printf(“%.3f</h1>\n”, status);
	printf(“</body>\n”);
	printf(“</html>\n”);
	return 0;
}
```

Code for the temperature sensor:

```
#include “stdio.h”
#include “EasyPIO.h”
#include “GPIO.h”

int sample;
float temperature;
int threshold = 75;
int turnOn(void) {
       setPinType(4,1);
       digitalWrite(4,1);
       return 0;
}

int turnOff(void) {
       setPinType(4,1);
       digitalWrite(4,0);
       return 0;
}

int main(void) {
       char received;
       pioInit();
       SetUpPerif();
       spiInit(244000,0);
       sample = spiSendReceive16(0x7000);
       temperature = (sample * 3.3) / 1024;

       if (temperature >= threshold) {
              turnOn();
       }

       else if (temperature < threshold) {
              turnOff();
       }

       printf("Content-type: text/html\n\n");
       printf("<html>\n");
       printf("<head>\n");
       printf("<meta http-equiv='refresh' content ='5'>");
       printf("</head>\n");
       printf("<body>\n");
       printf("<h1> The temperature is: ");
       printf("%.3f</h1>\n", temperature);
       printf("</body>\n");
       printf("</html>\n");
       return 0;
}
```

Code for curtain control to 0%:

```
#include "stdio.h"
#include "EasyPIO.h"
#include "GPIO.h"

int percentage;

int main(void) {
       char received;
       pioInit();
       SetUpPerif();
       setPinType(18,1);
       digitalWrite(18,0);
       spiInit(244000,0);
       percentage = spiSendReceive16(0b000);
       digitalWrite(18,1)
       printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);
```

```
        printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/mainPage.html
                                                                      \">");
        return 0;
}
```

Code for curtain control to 17%:

```
#include "stdio.h"
#include "EasyPIO.h"
#include "GPIO.h"

int percentage;

int main(void) {
        char received;
        pioInit();
        SetUpPerif();
        setPinType(18,1);
        digitalWrite(18,0);
        spiInit(244000,0);
        percentage = spiSendReceive16(0b001);
        digitalWrite(18,1)
        printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);
        printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/mainPage.html\">");
        return 0;
}
```

Code for curtain control to 34%:

```
        #include "stdio.h"
        #include "EasyPIO.h"
        #include "GPIO.h"

        int percentage;

        int main(void) {
                char received;
                pioInit();
                SetUpPerif();
                setPinType(18,1);
                digitalWrite(18,0);
                spiInit(244000,0);
                percentage = spiSendReceive16(0b010);
                digitalWrite(18,1)
                printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);
                printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/mainPage.html\">");
                return 0;
}
```

Code for curtain control to 50%:

```
#include "stdio.h"
#include "EasyPIO.h"
#include "GPIO.h"

int percentage;

int main(void) {
        char received;
        pioInit();
```

```
              SetUpPerif();
              setPinType(18,1);
              digitalWrite(18,0);
              spiInit(244000,0);
              percentage = spiSendReceive16(0b100);
              digitalWrite(18,1)
              printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);
              printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/mainPage.html\">");
              return 0;
}
```

Code for curtain control to 67%:

```
       #include "stdio.h"
       #include "EasyPIO.h"
       #include "GPIO.h"

       int percentage;

       int main(void) {
              char received;
              pioInit();
              SetUpPerif();
              setPinType(18,1);
              digitalWrite(18,0);
              spiInit(244000,0);
              percentage = spiSendReceive16(0b101);
              digitalWrite(18,1)
              printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);
              printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/mainPage.html\">");
              return 0;
}
```

Code for curtain control to 84%:

```
       #include "stdio.h"
       #include "EasyPIO.h"
       #include "GPIO.h"

       int percentage;

       int main(void) {
              char received;
              pioInit();
              SetUpPerif();
              setPinType(18,1);
              digitalWrite(18,0);
              spiInit(244000,0);
              percentage = spiSendReceive16(0b110);
              digitalWrite(18,1)
              printf("%s%c%c\n", "Content-Type:text/html;charset=iso-8859-1",13,10);
              printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/mainPage.html
                                                                           \">");
              return 0;
}
```

Code for curtain control to 100%:

```
       #include "stdio.h"
       #include "EasyPIO.h"
```

```
#include “GPIO.h”

int percentage;

int main(void) {
       char received;
       pioInit();
       SetUpPerif();
       setPinType(18,1);
       digitalWrite(18,0);
       spiInit(244000,0);
       percentage = spiSendReceive16(0b111);
       digitalWrite(18,1)
       printf(“%s%c%c\n”, “Content-Type:text/html;charset=iso-8859-1”,13,10);
       printf(“<META HTTP-EQUIV=\”Refresh\” CONTENT=\”0;url=/mainPage.html
                                                                   \”>”);
       return 0;
}
```

Code for the main page:

```
<!DOCTYPE html>
<html>
<head>
       <title> Environment data page </title>
       <h1> <center> Monitor the environment </center> </h1>
       <meta http-equiv=”content-type” content=”text-html;charset=utf-8”>
</head>
<body>
       <form action=”cgi-bin/phototransistor” method=”POST”>
              <input type=”submit” value=”What’s the luminosity?”>
       </form>
       <br>
       <form action=”cgi-bin/temp” method=”POST”>
              <input type=”submit” value=”What’s the temperature?”>
       </form>
       <br>
       <form action=”cgi-bin/distance” method=”POST”>
              <input type=”submit” value=”Is there something close by?”>
       </form>
       <br> By how much do you want to close the curtain? <br>
       <form action=”cgi-bin/curtain0” method=”POST”>
              <input type=”submit” value=”0%”>
       </form>
       <form action=”cgi-bin/curtain17” method=”POST”>
              <input type=”submit” value=”17%”>
       </form>
       <form action=”cgi-bin/curtain34” method=”POST”>
              <input type=”submit” value=”34%”>
       </form>
       <form action=”cgi-bin/curtain50” method=”POST”>
              <input type=”submit” value=”50%”>
       </form>
       <form action=”cgi-bin/curtain67” method=”POST”>
              <input type=”submit” value=”67%”>
       </form>
       <form action=”cgi-bin/curtain84” method=”POST”>
              <input type=”submit” value=”84%”>
       </form>
       <form action=”cgi-bin/curtain100” method=”POST”>
              <input type=”submit” value=”100%”>
```

```
        </form>
</body>
</html>
```

Appendix B: FPGA Code

```
// Kimberly Joly and Richard Liu
// kjoly@hmc.edu and rliu@hmc.edu
// November 28, 2017
// This is the top level module that interfaces from a Raspberry Pi
// and controls a stepper motor using wave modulation. SPI is used
// to read bits from the Raspberry Pi to determine the number of
// rotations needed and the direction the rotations should be in.
module finalproject (input  logic clk, spiClock, slaveSelect, piMOSI,
                     output logic [3:0] motor);

       // create a slow clock
       logic slowClk;
       slowClock slowDownClock(clk, slowClk);

       // use SPI to get settings
       logic [2:0] currentSetting;
       logic doneSPIpulse;
       spiSlave readInfo(clk, spiClock, slaveSelect, piMOSI,
                     doneSPIpulse, currentSetting);

       // read previous configuration and store new configuration
       logic doneRAM;
       logic [2:0] previousSetting;
       logic [7:0] changingData;
       memory_KJRL ram(slowClk, doneSPIpulse, 6'b000000,
                     {5'b00000, currentSetting}, changingData,
                     {5'b00000, previousSetting}, doneRAM);

       // calculate the number of rotations needed to achieve desired state
       logic doneCalculations;
       logic clockwise;
       logic [2:0] modification;
       logic [19:0] timeRotating;
       calculateRotations getTimes(slowClk, doneRAM, previousSetting,
                                    currentSetting, clockwise, doneCalculations,
                                    modification, timeRotating);

       // control the motor using wave modulation
       shifter outputPulses(slowClk, doneCalculations, clockwise,
                                   timeRotating, motor);

endmodule

// Kimberly Joly and Richard Liu
// kjoly@hmc.edu and rliu@hmc.edu
// November 28, 2017
// This module slows down the FPGA clock to a much slower clock. This is
// the main clock of interest because the motor is the slowest. The slow
// speed is used to provide enough torque.
module slowClock(input  logic fastClock,
               output logic slowedClock);

       // counter for the slowed down clock
       logic [23:0] counter;

       always_ff @(posedge fastClock)
              if (counter > 20'b11111000110101000000) begin      // 40 Hz
                     counter <= 24'b0;
              // reset counter
                     slowedClock <= ~slowedClock;        // flip value
              end
```

```
            else counter <= counter + 1'b1;          // increment counter

endmodule

// Kimberly Joly and Richard Liu
// kjoly@hmc.edu and rliu@hmc.edu
// November 28, 2017
// The shifter module is an FSM that uses one hot encoding to
// use wave modulation on the stepper motor. Since the stepper
// motor consists of a magnet, the module steps through the four
// basic configurations.
module shifter (input  logic sck, startShifting, clockwise,
                         input  logic [19:0] timeToRotate,
                         output logic [3:0] shifted);

     // counter to see if the motor has rotated enough
     logic [19:0] counter;
     always_ff @(posedge sck) begin
          // since the enable is a pulse, we reset if enabled
          if (startShifting) counter <= 20'b0;

          // if we haven't exceeded rotations
          if (counter < timeToRotate) counter <= counter + 1'b1;
end

     // four states, using one hot encoding
     logic [3:0] state, nextState;

     always_ff @(posedge sck)
          // only rotate when under timeToRotate
          if (counter < timeToRotate) state <= nextState;

     // rotating clockwise and counterclockwise uses
     // different iterations of the four settings
     always_comb
          case(state)
               4'b0001: if (clockwise)      nextState = 4'b0010;
               else                         nextState = 4'b1000;
               4'b0010: if (clockwise)      nextState = 4'b0100;
               else                         nextState = 4'b0001;
               4'b0100: if (clockwise)      nextState = 4'b1000;
               else                         nextState = 4'b0010;
               4'b1000: if (clockwise)      nextState = 4'b0001;
               else                         nextState = 4'b0100;
               default:       nextState = 4'b0001;
          endcase

     // output the current wave pulse
     assign shifted = state;

endmodule


// Kimberly Joly and Richard Liu
// kjoly@hmc.edu and rliu@hmc.edu
// November 28, 2017
// The calculateRotations module takes in the previous setting and current
// setting to calculate the amount of time needed to rotate the motor.
module calculateRotations (input  logic sck, startCalculating,
                    input  logic [2:0] previousData, currentData,
                    output logic clockwise, doneCalcuating,
                    output logic [2:0] modification,
                    output logic [19:0] timeToRotate);

     // direction can be determined by the current and previous configs
     assign clockwise = (previousData > currentData);
```

```
        always_ff @(posedge sck) begin
                if (startCalculating)        begin

                        // the modification should be kept positive
                        if (previousData > currentData)
                                modification = previousData - currentData;
                        else
                                modification = currentData - previousData;

                        // arithmetic to determine how much the motor needs to rotate
                        timeToRotate = 20'b0;
                        if (modification[2])  timeToRotate = timeToRotate + 20'b100001100;
                        if (modification[1])  timeToRotate = timeToRotate + 20'b10000110;
                        if (modification[0]) timeToRotate = timeToRotate + 20'b1000011;

                        // short pulse when done
                        doneCalcuating = 1'b1;
                end

                // otherwise, set it to zero
                else   doneCalcuating = 1'b0;
        end

endmodule

// Kimberly Joly and Richard Liu
// kjoly@hmc.edu and rliu@hmc.edu
// November 28, 2017
// The SPI slave module makes the FPGA a slave and the Raspberry
// Pi the master. When the slave select (ss) bit is low, it
// listens for the master's instructions. The bits that are needed
// are stored and then passed on to other modules.
module spiSlave(input  logic clk, spiclk, ss, mosi,
                output logic donePulse,
                output logic [2:0] spiBits);

        // an equivalent of an enable pin
        logic tick;
        logic doneReadingSpi;

        // save the three bits of data that are relevant
        ff_with_en data0(spiclk, tick, mosi, spiBits[0]);
        ff_with_en data1(spiclk, tick, spiBits[0], spiBits[1]);
        ff_with_en data2(spiclk, tick, spiBits[1], spiBits[2]);

        // we have two asynchronous clocks with known frequencies.
        // to interface the data and switch from a faster clock to
        // a slower clock, we use the FPGA clock to count and
        // create a pulse that the slow clock can understand.
        logic [23:0] counter;
        always_ff @(posedge clk) begin
                if (doneReadingSpi) counter <= 24'b0;

                // the counter counts to half the frequency of the slow clock
                else if (counter < 21'b111110000000000000000) begin
                        donePulse <= 1'b1;    // creates the pulse in the meantime
                        counter <= counter + 1'b1;
                end

                else donePulse <= 1'b0;      // no pulse once done
        end

        // finite state machine to read bits
        logic [2:0] state, nextState;
        always_ff @(posedge spiclk) begin
                state <= nextState;
        end

        always_comb
                // the slave only listens when the slave select pin is low
                // when ss is high, then the results are forgotten
                case(state)
                        3'b000:         if (~ss)        nextState = 3'b001;
                                        else            nextState = 3'b000;
                        3'b001: if (~ss)                nextState = 3'b010;
                                        else            nextState = 3'b000;
                        3'b010: if (~ss)                nextState = 3'b011;
                                        else            nextState = 3'b000;
                        3'b011:         if (~ss)        nextState = 3'b100;
                                        else            nextState = 3'b000;
                        3'b100: if (~ss)                nextState = 3'b100;
                                        else            nextState = 3'b000;
                        default:                        nextState = 3'b000;
                endcase

        // Mealy FSM
        assign doneReadingSpi = ((state == 3'b011) & (~ss)) | ((state == 3'b011) & (ss));
        assign tick = (state == 3'b000 | state == 3'b001 | state == 3'b010) & (~ss);

endmodule

// Kimberly Joly and Richard Liu
// kjoly@hmc.edu and rliu@hmc.edu
// November 28, 2017
// The flip flop with enable pin has d, q = 1 bit.
module ff_with_en(input  logic clk, en, d,
                output logic q);

        always_ff @(posedge clk) begin
                if (en)         q <= d;
        end

endmodule

// Kimberly Joly and Richard Liu
// kjoly@hmc.edu and rliu@hmc.edu
// November 28, 2017
// The flip flop with enable pin has d, q = 8 bits.
module ff_with_en8(input  logic clk, en,
                input  logic [7:0] d,
                output logic [7:0] q);

        always_ff @(posedge clk) begin
                if (en)         q <= d;
        end

endmodule

// Kimberly Joly and Richard Liu
// kjoly@hmc.edu and rliu@hmc.edu
// November 17, 2017
// This module is RAM, but it also outputs previous data using
// two enabled flip flops. The two flip flops store the previous
// value even after new data has been written. A finite state
// machine is also used to pulse a done signal, meaning that the
// module has written new data successfully.
module memory_KJRL (input  logic clk, we,
                        input  logic [5:0] addr,
                        input  logic [7:0] wd,
                        output logic [7:0] rd,
                        output logic [7:0] previousData,
                        output logic doneStoring);

        // 64, 8-bit memory
        logic [7:0] memory [63:0];

        // states for FSM
        logic [1:0] state, nextState;

        always_ff @(posedge clk) begin
                rd <= memory[addr];             // if not write-enabled, read data
                if (we) memory[addr] <= wd;     // if write-enabled, store data
                state <= nextState;             // advance to next state
        end

        logic [7:0] intermediate;                       // intermediate value (rd)
        // the two enabled flip flops store the previous value even
        // after new data has been written, regardless of address
        ff_with_en8 getPrevious1(clk, we, wd, intermediate);
        ff_with_en8 getPrevious2(clk, we, intermediate, previousData);

        // FSM to pulse a done signal
        always_comb
                case(state)
                        2'b00:  if (we) nextState = 2'b01;
                                else    nextState = 2'b00;
                        2'b01:          nextState = 2'b10;
                        2'b10:  if (we) nextState = 2'b10;
                                else    nextState = 2'b00;
                        default:        nextState = 2'b00;
                endcase

        // pulse at the second state
        assign doneStoring = (state == 2'b01);

endmodule
```