Fire! Fire!

Team Members:

Tatianna Ensslin
Giovanni Soto
Christina Tobias

Table of Contents

Introduction 3
Methods 3
Thermistor Circuit 3
BeagleBone, GPIO Expander, LCD Circuit 3
Establish a Wi-Fi Connection with the BeagleBone 4
Create a Node.js Server Using Express 4
Use Route Directory to Save Temperature Data 5
Establish a Real-Time Data Feed on the Website 5
Save Temperature Data into a File to be Accessed via cURL 6
Call a cURL Request within the C++ Program 6
Results 6
Thermistor Circuit 6
BeagleBone, GPIO Expander, LCD Circuit 7
Create a Node.js Server Using Express 7
Establish a Real-Time Data Feed on the Website 7
Call a cURL Request within the C++ Program 8
Discussion 8
Appendix A – Altera FPGA Circuit VHDL Code 10
Appendix B – Node.js Server code 11
Appendix C – BeagleBone Program Code 13

Introduction

Our project is a heat based fire alarm system. After our device detects a temperature, the danger level will be sent to a website and displayed. These levels range from normal to fire and are based on the temperature the device receives. In order to detect the temperature, we are using a thermistor with a nichrome wire wrapped around it. When voltage is sent through the nichrome wire, the thermistor will heat up and the resistance through it will increase. This change in resistance will be measured by the change in voltage. This voltage can then be converted to a temperature through an equation we found by plotting resistance data, and thus be used to detect a fire.

Heat detectors are extremely versatile. Some other uses of the implementation of our project include heat detection in hardware, temperature levels in senior living homes, and as a base for climate control systems such as greenhouses or residential homes. Usually fire alarms detect smoke, but heat-based fire alarms are seen as more useful in areas of high dust or dirt accumulation.

In order to alert the user promptly that their system is overheating, we decided it would be useful to send the data from our heat system to a web browser. In order to transfer the information, we used an OURLiNK Wi-Fi Adapter to connect the BeagleBone to a wireless network. This allowed for us to send a cURL request to a cloud server droplet hosted by DigitalOcean. cURL is a cross platform command line for getting and sending requests through URL syntax. It stands for Curl URL Request Library and supports a wide range of internet protocols ranging from HTTP to RTSP. In this project, we use a cURL url-encode request from our C++ code on the BeagleBone to access and post the current heat temperature to a Node.js server running on the droplet server. It does this by performing the cat system call from a file on the BeagleBone, which copies the data to the server as a variable, which is then parsed and displayed.

Methods

Thermistor Circuit

For this experiment, we implemented a voltage divider circuit with a 640Ω thermistor and 100Ω resistor. The voltage source being applied to one end of the resistor is the 1.8 V found on the BeagleBone Black. The other end of the resistor is connected to the thermistor, and finally the other end of the thermistor is connected to ground. We connected a wire between the resistor and thermistor, allowing us to receive the reduced voltage as a result of the change in the thermistor's resistance when heated. This voltage is sent to an analog-in (AIN#) pin on the BeagleBone Black.

A nichrome wire is used as a heat source for the thermistor, and is wrapped around the thermistor. One end of the wire is connected to the power supply, which will supply voltage varying from 0 - 2.5 V. Finally, the other end of the wire is connected to ground. As more voltage is applied to the nichrome wire, it becomes more hot, and alters the resistance of the thermistor.

BeagleBone, GPIO Expander, LCD Circuit

In order to display the heat status, analog voltage and temperature on the LCD, we utilized the MCP23017 I2C 16-bit I/O expander. This component provides us with more GPIO pins to use for the LCD pins and only use (in our case) two pins from the BeagleBone Black - the I2C1_SDA and I2C1_SCL pins. We provided the GPIO Expander with 3.3 V of power from the BeagleBone Black. On the expander, we used both GPIOA and GPIOB sets of pins - GPIOB[0] - GPIOB[7] for the eight LCD data pins, GPIOA[7] for LCD_EN and GPIOA[6] for LCD_RS. The I2C1_SDA and I2C1_SCL pins are connected to pins SDA and SCL, respectively. Pins A2-A0

are all hardware address pins; we connected them all to ground, setting the device address to 0x20 (this could be verified on the BeagleBone).

Before the BeagleBone could connect to the GPIO expander, we had to initialize the device tree overlay. We echoed BB-I2C1 into $SLOTS to be able to use the i2c component of the BeagleBone. We found the address of the device, which is 0x20, by using the command i2cdetect. In order to control the data sent to the GPIO expander and through to the LCD display, we used the i2c class given to us. We wrote code that allowed the appropriate functions to be sent in and transferred to the LCD display. This included clearing the screen, setting the function, turning the display on, and setting the entry mode. After this we could send the data. Before sending new data to the GPIO expander it is essential to give the LCD display time to receive the information, and thus we used usleep() to create this delay.

To use the LCD display, we needed to use the Altera FPGA board. We sent the data from the BeagleBone to the GPIO expander, and then to the Altera FPGA, which connects to the LCD display. The code sending the data from the FPGA to the LCD display was simple, but every wire needed to be accounted for and all the pins needed to be correct (Appendix A).

In addition, we had to initialize the analog to digital device tree overlay using BB-ADC and echoing it into $SLOTS. We took in the analog input and parsed it to see it as a voltage ranging from 0V to 1.8V. This voltage was the output of the thermistor. Depending on the voltage received, we could set this to a degree in Fahrenheit. To do this, we knew room temperature, or 70 degrees, had an analog output of 1.5V. After running through the data and taking in different points of voltage to the heat from the nichrome wire, we found that about 150 degrees outputted 0.5V. This led us to creating an equation that could give us the temperature of any outputted voltage. This linear equation is y = -80x + 190.

Using the equation, we found that we could output the temperature and the range of danger, which starts at normal and ends at Fire!, to the LCD display. This data constantly changes with the heating up and cooling down of the nichrome wire, which heats up the thermistor. In this way, we had all the information we needed to send to a website that could be accessed from anywhere (Appendix B).

Establish a Wi-Fi Connection with the BeagleBone

Begin by establishing a Wi-Fi connection with the BeagleBone. In previous experiments, we have used the ESP01 Wi-Fi Module to connect, however, in this project we will use an OURLiNK USB Wi-Fi Adapter. It is important to note that when using a Wi-Fi Adapter with the BeagleBone, ensure that there is a pre-existing Linux driver installed on the device. It is an extremely difficult task to download a driver onto a USB without Wi-Fi/Ethernet connection! We set up the OURLiNK by opening the Wi-Fi Configuration window within the XLaunch GUI. We navigated to network > properties then sign into your established Wi-Fi Network. You can check that the connection works by connecting to a web page, such as a www.google.com.

Create a Node.js Server Using Express

In order to visually display the temperature and warning level of the heat system, we must create a client-server setup, with the client, the BeagleBone, feeding the data to the server. In order to create a server, create an account with a cloud server host. We will use DigitalOcean, however, Amazon and other companies also offer these services. Set up your server using an (Apache) LAMP Stack (if using a Mac). Guides can be found on the web. Once this has been created, download and install Node.js and Express Generator, using command line, or by accessing their website and downloading the package installer. Express Generator gives routes and templates allowing for simplified framework when establishing your

Node.js server. Use the following commands while you are SSH into your cloud server to install both Node.js and Express.

Downloading a stable version of Node.js:

```
cd /tmp
wget http://nodejs.org/dist/v0.10.32/node-v0.10.32-linux-x64.tar.gz
tar xvf node-v0.10.32-linux-x64.tar.gz
```

Installing Node.js:

```
cd node-v0.10.32-linux-x64/
cp * /usr/local/ -r
cd ~
```

Check version by running node –v. If you see "v0.10.32", you have successfully installed Node.js.

Next, install Express Generator:

```
npm install -g express-generator
```

Check that you have successfully installed it by viewing a list of available options by running express -h. Use the command express myAppName to create a new Node.js application. There are tutorials online for additional guidance if required. Our applications name is app.js. Once you completed this command, establish your application's dependencies by running the commands cd myAppName and npm install. You can run your new application (Node.js server) using npm start. When npm start is running, your server is running, which means you can also test your current website by typing your IP-address followed by :3000 (3000 is the standard port used with Express, but can be modified if desired). You can view the requests entering the Node.js server using the command screen -r.

Use Route Directory to Save Temperature Data

Now that the foundation of our server is created, navigate to your application directory and cd into your Routes folder. Inside this folder is an index.js file. This file contains the JavaScript which controls the paths that requests take into the server. In this file, create a variable which will be where the data is saved when the Post request is sent from the BeagleBone via cURL. In this file is also where you want to parse the data which is received from the BeageBone. Once it has been parsed, save the data to your variable to be sent through the Get request route in the same file (Appendix C).

Establish a Real-Time Data Feed on the Website

Now that we have established proper routes for our Post and Get requests on the Node.js server (our application), it is time to handle our data on the client end--the website. Navigate into your application directory, and run cd public/javascripts/action.js. This file contains any client (web browser) side JavaScript. Create a document function, say its named display, which uses $.get("/getVARNAME,function(data)){} to display the information using .html(data). Then, using window.setInterval(function()){}, call display() with a 2000ms interval. This implements an AJAX request within the JavaScript file, allowing for the data to be shown on the web page without having to refresh it (Appendix C).

NOTE: Navigating to your app directory>views>index.ejs is the file that contains the primary user-side of the web page. Here is where you can include HTML/CSS, and make your server look as appealing as you desire.

Save Temperature Data into a File to be Accessed via cURL

Now that the server is running, we must save the temperature data into a file so that we can send the contents of the file in a Post request to our Node.js server - this is done on the BeagleBone. The temperature value found from the equation y = -80x + 190 is stored into a .txt file ("fire.txt", in our case) with the C++ ofstream class via the iostream library.

Call a cURL Request within the C++ Program

Now that we have established our server and our data is being saved into a file to be Post, we can make cURL requests from other scripts/applications/and even via command line. cURL is a tool to transfer data from or to a server, using one of the supported protocols, such as (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET and TFTP). In our case, we use HTTP. In specific, the cURL command we will be calling is:

```
curl --data-urlencode "file=`cat YOURFILE.txt" http://IPNUMBER:3000/APPNAME
```

The "curl --data-urlencode" is the Post request to the Node.js server, whose IP address is given in the request as well, followed by the port and application name. When we send this request, we use the system call cat to copy the data inside the file and post it to the Node.js server. In order for our AJAX request to update as the temperature reading transitions, the cURL request needs to be consistently sending from the BeagleBone. In order to consistently send the cURL request, create a while loop in your C++ code which will use the system() function to call it and then sleep() for 2000ms, until being called again.

```
system("curl --data-urlencode "file=`cat YOURFILE.txt" http://IPNUMBER:3000/APPNAME");
```

Now restart your server using "npm install" and pull up your web page on a web browser. You can now run the script on the BeagleBone, calling the cURL request and running the heat detecting circuit. Within moments, you should be able to view the temperature data on the website, updating in real time on the web page, as well as view the get and post requests on the npm screen.

Results

Thermistor Circuit

In order to understand how the temperature would affect the thermistor, we first had to test the voltages. To do this, we used the GDM-8034 Multi-meter and connected it to the thermistor. We could then see the change in voltage across the thermistor as the nichrome wire heated it up. Using this method, we tested to see what the maximum voltage we could send through the nichrome wire without damaging the thermistor, which was 2.5V. We also found that without the thermistor being heated up, it outputted 1.5V.

When we heated the nichrome wire and had 2.5V going through it, the output of the thermistor changed to below .5V. This showed us that at 1.5V through the thermistor correlated to 70 degrees Fahrenheit and .5V correlated with around 150 degrees Fahrenheit. This allowed us to create a linear graph comparing the output voltage of the thermistor with a temperature. Using the two data points, we found that the temperature equaled -80 times the output voltage of the thermistor plus 190.

BeagleBone, GPIO Expander, LCD Circuit

After that we needed to receive the voltage from the thermistor through the BeagleBone. We did this using AIN0 to receive the analog voltage. We could see the voltage through a function we made that outputted it in the terminal.

Figure 1. LCD displays a test string "Fire!", which is printed continuously.

Figure 2. LCD displays the status, analog voltage, and temperature.

We wrote the code to display writing to the LCD display and tested it by sending out Fire! repeatedly. We then used our data to display the ranges of temperatures with the warning level, the voltage, and the temperature.

Create a Node.js Server Using Express

```
Tatianas-MacBook-Pro:~ Tatiana$ SSH root@107.170.133.224
root@107.170.133.224's password:
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-52-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Sat Apr 23 12:25:35 EDT 2016

  System load:  0.0                Processes:           87
  Usage of /:   52.3% of 19.56GB   Users logged in:     0
  Memory usage: 27%                IP address for eth0: 107.170.133.224
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

166 packages can be updated.
88 updates are security updates.

Last login: Sat Apr 23 12:25:40 2016 from 128.230.142.128
root@Otto:~# cd curl
root@Otto:~/curl# ls
app.js  bin  node_modules  npm-debug.log  package.json  public  routes  views
root@Otto:~/curl# npms start
No command 'npms' found, did you mean:
 Command 'pms' from package 'pms' (universe)
 Command 'npm' from package 'npm' (universe)
npms: command not found
root@Otto:~/curl# npm start

> curl@0.0.0 start /root/curl
> node ./bin/www
```

Figure 3. Terminal displays starting the Node.js server.

To connect to the cloud server, SSH into it via your username and the IP Address. Navigate into your application file ("curl" in this example), and start the Node.js server using npm start.

Establish a Real-Time Data Feed on the Website

In Figure 4, we see the POST request and the temperature data that is being sent over; "89" is the temperature being sent from the cURL request via the BeagleBone. When you run your server, this should display. If it doesn't, you can manually pull up the screen using screen -r. This is then parsed in the index.js file and refreshed in the action.js file, each can be found in the appendix. As the temperature is rendered through the /log endpoint, the AJAX request is triggered and the data is returned.

```
89
POST /log 200 1.982 ms - -
```

Figure 4. POST request going to the server

Call a cURL Request within the C++ Program

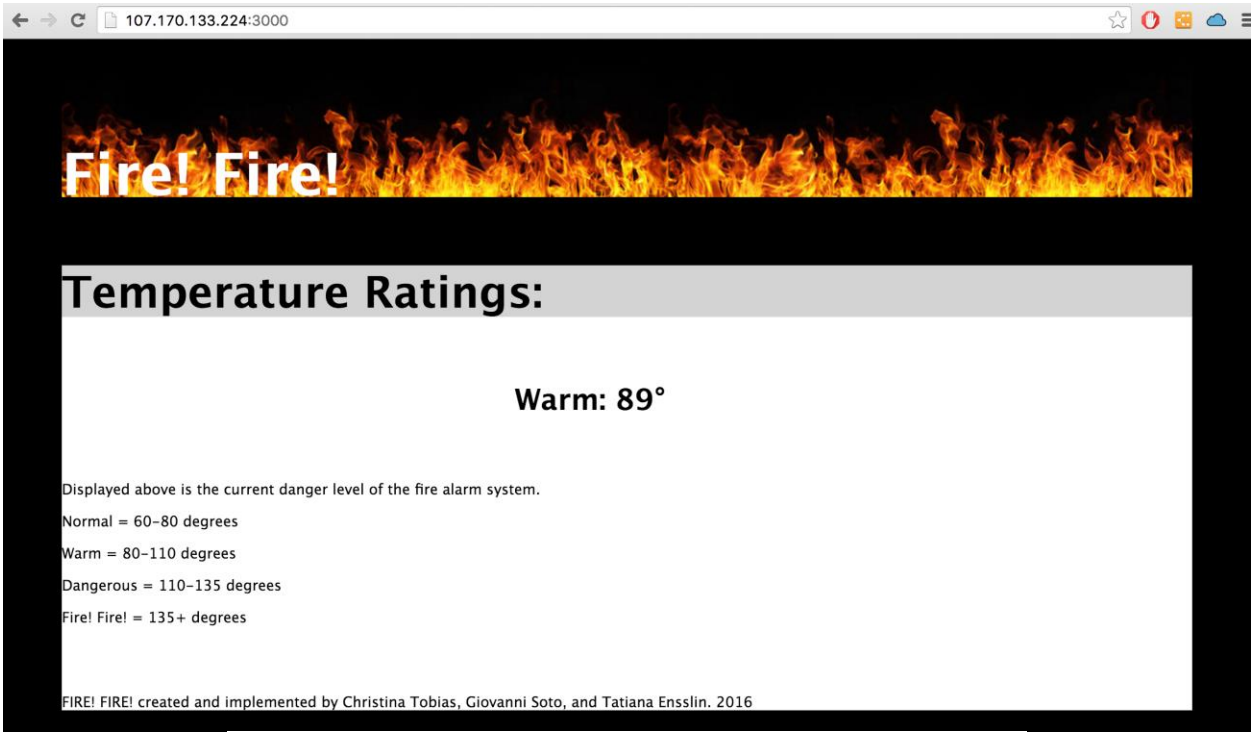

Figure 5. The website displays both the temperature and corresponding status.

You can see the temperature and the warning level displayed on the website (using the marquee function it traverses across the website).

The cURL request used from the BeagleBone to send the data within the C++ code is:

```
system(curl --data-urlencode "file=`cat fire.txt`" http://107.170.133.224:3000/log);
sleep(2000);
```

Discussion

The goal of this project was to use the knowledge we have gained this year to create a fully functioning device. Using different hardware components and ideas, we came up with the idea of a temperature based fire alarm.

However, throughout the lab we found that the hardest part was getting each of these components to work with each other. We experienced many problems with the Wi-Fi modules. We first wanted to use the ESP-01 to connect to the web server so that we could display the information on a website. However, we were confined to the local network, so we discovered that would not work. We then tried two different Wi-Fi adapters, both of which failed. The Netgear N900 caused us to have a driver problem; the driver of the device was not installed and the proper software was not in place to allow us to install the driver. Therefore, we could not get the BeagleBone Black to recognize the device and connect to the internet. The next Wi-Fi adapter we tried was the Belkin N150, which the BeagleBone Black recognized and displayed, but we still could not connect to the internet. We got as far as typing in our username and password, but every time it saw the password as being incorrect and would not finish connecting.

We learned a lot through designing and implementing this lab. The first lesson was that the hardest part is getting pieces that work together seamlessly. We spent a lot of time researching how to get the Wi-Fi adapters to work and which ones to buy, but that time was wasted when both of the ones we bought failed. However, we were given one from the lab that worked without any setup. We also learned that it is ideal to split up the work and have everyone work on something different. This allowed us to be as efficient as possible and have one person focused on each aspect, then in the end we could add everything together. This allowed us to get the work done in time after having a hard time with the Wi-Fi adapters and we were successful in implementing every part of the lab.

Appendix A – Altera FPGA Circuit VHDL Code

```vhdl
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity JuniorDesign is
    port(
        -- LCD Display
        Set_LCD_Data : in  std_logic_vector (7 downto 0); -- goes to PIN_P18 -> PIN_K21
        Set_LCD_EN, Set_LCD_RS : in  std_logic; -- goes to PIN_J22 & PIN_E25, respectively
        LCD_Dataz : out std_logic_vector (7 downto 0); -- goes to LCD_D[7] -> LCD_D[0]
        LCD_EN, LCD_RS, LCD_ON, LCD_RW : out std_logic);
end entity;

architecture HardwareSettings of JuniorDesign is
begin
    -- Set up LCD Display
    LCD_Data <= Set_LCD_Data;
    LCD_EN <= Set_LCD_EN;
    LCD_RS <= Set_LCD_RS;
    LCD_ON <= '1';
    LCD_RW <= '0';
end architecture;

```

Appendix B – Node.js Server code

```
//Action.js
$(document).ready(function() {

  var display = function(){

  $.get( "/getlog", function(data) {
    $( "#log" ).html(data);
  });
  }

  window.setInterval(function(){
    display();
  }, 2000);
});

```

```
//Index.Js
var express = require('express');
var router = express.Router();
var log;
var temp;
var level;
var rate;

router.get('/', function (req, res, next){
    res.render('index', {title:'Fire! Fire!', log: log});
});

router.post('/log', function (req, res, next) {
    temp = parseInt(req.body.file);

    console.log(temp);

  if(temp >= 60 && temp < 80){
    log = "Normal: "+ temp + '&#176';
    }
    if(temp >= 80 && temp < 110){
        log = "Warm: "+ temp + '&#176';
    }
  if(temp >= 110 && temp <= 135){
        log = "Dangerous: " + temp + '&#176';
    }
    if(temp > 135){
        log = "Fire! Fire!: " + temp + '&#176';
    }

  res.end();
});

router.get('/getlog', function (req, res, next) {
    res.send(log);
});
module.exports = router;

```

```
<!--Index.ejs-->
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
      <script src='/javascripts/jquery-2.2.1.min.js'></script>
      <script src='/javascripts/action.js'></script>
      <style>
        h1 {
          font-size: 400%
        }
        h2 {
          background-color: lightgray;
          font-size: 300%
        }
        h3 {
          font-size: 200%
        }
        #header {
          background-image: url("/images/fire-flames-black-background-texture-45874484.jpg");
          padding-top: 48px;
        }
        body {
          background-color: black;
        }
        #section {
          background-color: white;
        }
        #footer {
          background-color: white;
        }
         #Dangerous{
          font-color:orange;
        }
    </style>
  </head>
  <body>
    <div id="header">
      <h1> <p class="triple"> <font color="white"><b><%= title %></b></font> </p></h1>
    </div>
    <div id="section">
      <h2> Temperature Ratings: </h2>
      <h3> <marquee scrollamount="7"><p id="log"></p></marquee> </h3>
      <p> Displayed above is the current danger level of the fire alarm system. </p>
      <p> Normal = 60-80 degrees </p>
      <p> Warm = 80-110 degrees</p>
      <p> Dangerous = 110-135 degrees</p>
      <p> Fire! Fire! = 135+ degrees</p>
      <br>
      <br>
      <br>
    </div>
    <div id="footer">
      FIRE! FIRE! created and implemented by Christina Tobias, Giovanni Soto, and Tatiana Ensslin. 2016
    </div>
    <br>
    <br>
  </body>
</html>

```

Appendix C – BeagleBone Program Code

```
//I2CDevice.h
#ifndef I2C_H_
#define I2C_H_

#define BBB_I2C_0 "/dev/i2c-0"
#define BBB_I2C_1 "/dev/i2c-1"
#define BBB_I2C_2 "/dev/i2c-2"

class I2CDevice{
public:
    I2CDevice(unsigned int _bus, unsigned int _device);
    int open();
    int write(unsigned char value);
    unsigned char readRegister(unsigned int registerAddress);
    unsigned char* readRegisters(unsigned int number, unsigned int fromAddress = 0);
    int writeRegister(unsigned int registerAddress, unsigned char value);
    void debugDumpRegisters(unsigned int number = 0xff);
    void close();
    ~I2CDevice();

    //functions for Junior Lab Project
    char* getVoltage();
    void getAnalogVoltage(int AIN);
    void sendByteToLCD(int en, int rs, int value);
    void sendStringToLCD(int en, int rs, char* string);

private:
    unsigned int bus;    //the bus number
    unsigned int device; //the device number on the bus
    int file;            //the file handle to the device

    //variables for Junior Design Project
    char analogVoltage[4];
};
#endif

```

```cpp
//I2CDevice.cpp
#include "I2CDevice.h"
#include <cstdlib>
#include <fcntl.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <unistd.h>
using namespace std;

#define HEX(x) setw(2) << setfill('0') << hex << (int)(x)
#define CLEAR 0x01
#define FUNCTIONSET 0x38
#define DISPLAYON 0x0E
#define ENTRYMODESET 0x06

I2CDevice::I2CDevice(unsigned int _bus, unsigned int _device) {
    bus = _bus;
    device = _device;
    this->open();
}

int I2CDevice::open() {
    string name;
    if(this->bus == 0) name = BBB_I2C_0;
    else if(this->bus == 1) name = BBB_I2C_1;
    else name = BBB_I2C_2;

    if((this->file=::open(name.c_str(), O_RDWR)) < 0) {
        perror("I2C: failed to open the bus\n");
        return 1;
    }
    if(ioctl(this->file, I2C_SLAVE, this->device) < 0) {
        perror("I2C: Failed to connect to the device\n");
        return 1;
    }
    return 0;
}

int I2CDevice::writeRegister(unsigned int registerAddress, unsigned char value) {
    unsigned char buffer[2];
    buffer[0] = registerAddress;
    buffer[1] = value;
    if(::write(this->file, buffer, 2) != 2){
        perror("I2C: Failed write to the device\n");
        return 1;
    }
    return 0;
}

int I2CDevice::write(unsigned char value) {
    unsigned char buffer[1];
    buffer[0] = value;
    if (::write(this->file, buffer, 1) != 1){
        perror("I2C: Failed to write to the device\n");
        return 1;
    }
    return 0;
}

unsigned char I2CDevice::readRegister(unsigned int registerAddress) {
    this->write(registerAddress);
    unsigned char buffer[1];
    if(::read(this->file, buffer, 1) != 1){
        perror("I2C: Failed to read in the value.\n");
        return 1;
```

```
    }
    return buffer[0];
}

unsigned char* I2CDevice::readRegisters(unsigned int number, unsigned int fromAddress) {
    this->write(fromAddress);
    unsigned char* data = new unsigned char[number];
    if(::read(this->file, data, number) != (int)number) {
        perror("IC2: Failed to read in the full buffer.\n");
        return NULL;
    }
    return data;
}

void I2CDevice::debugDumpRegisters(unsigned int number) {
    cout << "Dumping Registers for Debug Purposes:" << endl;
    unsigned char *registers = this->readRegisters(number);
    for(int i = 0; i < (int)number; i++){
        cout << HEX(*(registers + i)) << " ";
        if (i % 16 == 15) cout << endl;
    }
    cout << dec;
}

void I2CDevice::close() {
    ::close(this->file);
    this->file = -1;
}

I2CDevice::~I2CDevice() {
    if(file != -1) this->close();
}

char* I2CDevice::getVoltage() {
    return analogVoltage;
}

void I2CDevice::getAnalogVoltage(int AIN) {
    char directory[40];
    sprintf(directory, "/sys/devices/ocp.3/helper.12/AIN%d", AIN);
    std::fstream file(directory,std::ifstream::in);
    if(!file.good()) {
        cout << "LCD::getAnalogVoltage(int): File did not open" << endl;
        exit(1);
    }
    file >> this->analogVoltage;
    file.close();
}

void I2CDevice::sendByteToLCD(int en, int rs, int value) {
    this->writeRegister(0x12, (en + rs));
    this->writeRegister(0x13, value);
    this->writeRegister(0x12, rs);
    usleep(5000);
}

void I2CDevice::sendStringToLCD(int en, int rs, char* string) {
    for(int i = 0; string[i] != '\0'; i++) {
        this->writeRegister(0x12, (en + rs));
        this->writeRegister(0x13, string[i]);
        this->writeRegister(0x12, rs);
        usleep(5000);
    }
}

#ifdef MAIN_CPP
int main() {
    double AINVoltage, temperature;
```

```
    char *pEnd, tempV[6], tempT[6];
    char *fileString = "/home/debian/Desktop/fire.txt";

    //bus number is 2, device number is 0x20
    I2CDevice *i2c = new I2CDevice(2, 0x20);

    //set GPIO A & B as outputs
    i2c->writeRegister(0x00, 0x00);
    i2c->writeRegister(0x01, 0x00);

    //initialize LCD display on Altera board
    i2c->sendByteToLCD(0x80, 0x00, CLEAR);
    i2c->sendByteToLCD(0x80, 0x00, FUNCTIONSET);
    i2c->sendByteToLCD(0x80, 0x00, DISPLAYON);
    i2c->sendByteToLCD(0x80, 0x00, ENTRYMODESET);

    for(;;) {
        //convert voltage from AIN0 pin to temperature value
        i2c->getAnalogVoltage(0);
        AINVoltage = (double)strtol(i2c->getVoltage(), &pEnd, 10) / 1000;
        temperature = -76.699 * AINVoltage + 188.35;

        //save temperature value to fire.txt file
        std::fstream file(fileString, std::ofstream::out);
        if(!file.good()) {
            cout << "File \"fire.txt\" did not open" << endl;
            exit(1);
        }
        file << temperature;
        file.close();

        //LCD - display status on top line
        i2c->sendByteToLCD(0x80, 0x00, CLEAR);

        if(AINVoltage > 1.375)
            i2c->sendStringToLCD(0x80, 0x40, "Normal");
        else if(AINVoltage > 1 && AINVoltage <= 1.375)
            i2c->sendStringToLCD(0x80, 0x40, "Warm");
        else if(AINVoltage > 0.6875 && AINVoltage <= 1)
            i2c->sendStringToLCD(0x80, 0x40, "Dangerous");
        else //AINVoltage < 0.6875
            i2c->sendStringToLCD(0x80, 0x40, "Fire! Fire!");
        i2c->sendByteToLCD(0x80, 0x00, 0xC0); //Newline

        //LCD - display voltage on bottom line
        sprintf(tempV, "%0.2lf", AINVoltage);
        i2c->sendStringToLCD(0x80, 0x40, tempV);
        i2c->sendByteToLCD(0x80, 0x40, 0x56); //V
        i2c->sendByteToLCD(0x80, 0x40, 0x20); //[SPACE]
        i2c->sendByteToLCD(0x80, 0x40, 0x20); //[SPACE]

        //LCD - display temperature on bottom line
        sprintf(tempT, "%0.1lf", temperature);
        i2c->sendStringToLCD(0x80, 0x40, tempT);
        i2c->sendByteToLCD(0x80, 0x40, 0xDF); //[Degree symbol]
        i2c->sendByteToLCD(0x80, 0x40, 0x46); //F

        //command to send file data to website
        system("curl --data-urlencode \"file=`cat fire.txt`\" http://107.170.133.224:3000/log");
        usleep(500000);
    }
}
#endif
```