**Table of Contents**

## System Block Diagram 2

## Description 3

## Flowchart 5

## Appendix A – Program Code (PID Controller) 6

## Appendix B – LM35 Temperature Sensor Data Sheet 13

## Appendix C – Anode RGB LED Data Sheet 15

## Appendix D – PID Controller Description 16

## Appendix E – Arduino PID Library 18

## Appendix F – Arduino LM35 Temperature Sensor Library 27

## Appendix G – PID Controller Graphing Monitor 31

**System Block Diagram**



**Description of Project**

This project is a smart fan that acts as a laptop cooler. Attached to the Arduino circuit board are five components: a four-wire PWM fan, a RGB anode LED, a LM35 temperature sensor, a 10k potentiometer and a LCD screen. A breadboard is used to provide space for certain components, along with wires and other circuit essentials such as resistors or diodes.

The fan has four pins. One pin goes to the GND pin on the Arduino, one goes to a power pin, one pin deals with fan speed with PWM, and the last pin deals with temperature control. The fan’s power will adjust depending on the temperature reading.

The RGB LED has four pins. The longest pin goes to the ground, while the other three set the red, green and blue values of the led. These go to the digital pins of the Arduino, where the code can set the values of each colour. The colour of the LED correlates with the current temperature reading the sensor is receiving. Blue describes a lower-than-setpoint temperature, green describes a within-optimal-range temperature, red describes an above-optimal-range temperature and blinking red describes an above-critical-zone temperature.

The LM35 temperature sensor has three pins. One pin is for power, one is for ground, and one is to send the temperature to an Arduino digital analog pin. The sensor simply reads the temperature of its surroundings.

The 10k potentiometer allows a person to manually control the setpoint for the wanted temperature. It adjusts the set point temperature, which tells the fan that it needs to adjust its speed in order for the current temperature reading to reach that point. If the temperature is already below the setpoint, the fan does not increase its speed as the temperature is cold enough. If the temperature is above the setpoint, the fan accelerates to meet setpoint temperature.

The fan speed is controlled via a PID controller implemented in the code. PID stands for proportional-integral-derivative. A PID controller is a control loop feedback mechanism that minimizes error between a measured process variable (temperature) and a setpoint (desired temperature) by automatically adjusting the process control inputs (PWM output to fan PWM input). By tuning three constant parameters: the proportional gain, the integral gain, and the derivative gain, the desired temperature can be reached with minimal offshoot and oscillation. This is further explained in Appendix D.

The LCD screen displays the current temperature reading, the set point temperature, the current output (0 - 255 in PWM) and the current RPM of the fan. There are twelve pins on the fan that are required to be connected to the Arduino. They are used to connect to power, to digital pins to adjust the output, and to control the backlight of the screen.

**Flowchart**



**Appendix A – Program Code (PID Controller)**

/\*

March 2013

This sketch uses a PID controller to control the PC temperature and displays the result on an LCD.

Setpoint for temperature is read from a potentiometer. Data is also sent through serial to PC

running a real-time graph under Processing software.

Author: Matthew Tesfaldet

\*/

#include <LiquidCrystal.h>

#include <LM35.h>

#include <PID\_v1.h>

// If this is defined it prints out the data that should be sent to the PC

// via Processing software

//#define CHECK\_DATA

#define CRITICAL 30.00 // define critical temperature (degrees celsius)

/\* CONTROL DEVICES AND SENSORS \*/

int tachPin = 8; // fan tachometer connected to digital pin 8

int fanPin = 11; // fan drive connected to digital pin 11

int potPin = 0; // analog input 0 for the potentiometer

int FETPin = 9; // FET conected to digital pin 9

// (MOSFET IS OPTIONAL IF WANT TO CONTROL FAN SPEED VIA PWM ON POWER SOURCE)

int tempPin = 5; // temp sensor connected to analog pin 5

int redPin = 13; // red anode for RGB LED

int grnPin = 12; // green anode

int bluPin = 10; // blue anode

LM35Sensor sensor;

/\* MISC. VARS \*/

int potValue; // variable to store the value coming from the pot (for setting wanted temperature)

unsigned long duration; // time duration between tachometer pulses

double setpoint; // setpoint in degrees celsius

double temperature; // current temp speed in degrees celsius

float rpm; // fan speed in RPM

boolean beginning = true; // to kickstart the fan at the start of the process

/\* PID CONTROL \*/

double aggKp=40, aggKi=2, aggKd=10; // aggressive tuning

double consKp=20, consKi=1, consKd=5; // conservative tuning

double out; // PID controller output (0 - 255)

double error;

PID myPID(&temperature, &out, &setpoint, consKp, consKi, consKd, REVERSE); // Initialize PID

boolean auto\_manual=1; // setpoint auto or from manually from pot (1 - manual, 0 - auto)

int auto\_setpoint=20;

int sampletime=40; // Number of sample to average-out sampling values

unsigned long val; // Working variable to hold the accumulated value

/\* vars to send to PC Processing software graph \*/

unsigned int an0=0; // Variables for sending data to PC

unsigned int an1=0;

unsigned int an2=0;

unsigned int an3=0;

unsigned int an4=0;

unsigned int an5=0;

unsigned int offset=2;

// initialize the LCD library routine with the numbers of the interface pins

LiquidCrystal lcd(7, 6, 5, 4, 3, 2);

void setup() {

lcd.begin(16, 2); // set up the LCD's number of rows and columns:

pinMode(fanPin, OUTPUT);

pinMode(FETPin, OUTPUT);

pinMode(redPin, OUTPUT);

pinMode(grnPin, OUTPUT);

pinMode(bluPin, OUTPUT);

myPID.SetMode(AUTOMATIC);

Serial.begin(9600); // Serial port baud rate

}

void loop() {

unsigned int startTag = 0xDEAD; // Analog port maxes at 1023 so this is a safe termination value

// for sending values to PC.

val=0;

if (auto\_manual) {

for (int i=0;i<sampletime;i++) {

potValue = analogRead(potPin); // Read the value from the pot. AnalogRead values go from 0 to 1023.

val=val+potValue;

}

potValue=val/sampletime; // averaging out pot value over sample time

// convert to percentage (0% - 30% translates to 0 - 30 degrees celsius)

setpoint=float(potValue)\*30/1024;

}

else {

setpoint=auto\_setpoint; // in degrees celsius

}

duration=0;

// skip at beginning since fan isn't spinning, which will take pulseIn 1 second to

// timeout each time (no tach signal yet)

if (!beginning) {

for (int i = 0; i < sampletime; i++) { // Average out pulseIn time reading in microseconds

duration = duration+pulseIn(tachPin, HIGH);

}

}

beginning = false;

rpm=600000000/float(duration); // in RPM

sensor.read(tempPin); // reading from the temperature sensor

temperature = sensor.getCelsius(); // getting the temperature in degrees celsius

error=abs(setpoint-temperature); // distance away from setpoint

// compute PID value

if (error<1) {

// close to setpoint, be conservative

myPID.SetTunings(consKp, consKi, consKd);

}

else {

// far from setpoint, be aggressive

myPID.SetTunings(aggKp, aggKi, aggKd);

}

myPID.Compute();

if (temperature < CRITICAL) {

if (temperature < (int) (setpoint - 1)) { // well below setpoint temperature, shine blue

digitalWrite(redPin, HIGH);

digitalWrite(grnPin, HIGH);

digitalWrite(bluPin, LOW);

}

// optimal zone, shine green

else if (temperature >= (int) (setpoint - 1) && temperature <= (int) (setpoint + 1)) {

digitalWrite(redPin, HIGH);

digitalWrite(grnPin, LOW);

digitalWrite(bluPin, HIGH);

}

else { // near critical zone, shine red

digitalWrite(redPin, LOW);

digitalWrite(grnPin, HIGH);

digitalWrite(bluPin, HIGH);

}

analogWrite(fanPin, out); // analogWrite values from 0 to 255, PWM output

}

else { // if critical, fan on full blast and flash LED blinking red

int pinSetVal = LOW;

for (int i = 0; i < 5; i++) {

digitalWrite(redPin, pinSetVal);

digitalWrite(grnPin, HIGH);

digitalWrite(bluPin, HIGH);

if (pinSetVal == LOW) pinSetVal = HIGH;

else pinSetVal = LOW;

delay(100);

}

out = 255;

analogWrite(fanPin, out);

}

analogWrite(FETPin, 255); // keep MOSFET gate closed

lcd.setCursor(0, 0); // Set for first line

lcd.print(int(rpm)); // displays RPM value

lcd.print(" RPM CO:");

lcd.print((int)out);// displays Controller Output value

lcd.print(" ");

lcd.setCursor(0, 1); //set for second line

lcd.print("SP:");

lcd.print((int)setpoint);// displays setpoint value

lcd.print("C ");

lcd.print("Tmp:");// displays current temperature

lcd.print((int)temperature);

lcd.print("C ");

if (auto\_manual) { // if set-point in Auto, display "A"

lcd.setCursor(15, 1);

lcd.print("M");

}

else {

lcd.setCursor(15, 1);// else display "M" for Manual

lcd.print("A");

}

// scaling for graph on PC

an0=setpoint\*9+offset;

an1=temperature\*9+offset;

an2=out+offset;

Serial.write((unsigned byte\*)&startTag, 2); // Send serial data to PC

Serial.write((unsigned byte\*)&an0, 2);

Serial.write((unsigned byte\*)&an1, 2);

Serial.write((unsigned byte\*)&an2, 2);

Serial.write((unsigned byte\*)&an3, 2);

Serial.write((unsigned byte\*)&an4, 2);

Serial.write((unsigned byte\*)&an5, 2);

// for testing purposes

#ifdef CHECK\_DATA

Serial.print(" - RPM: ");

Serial.println(rpm);

Serial.print(" - setpoint: ");

Serial.println(setpoint);

Serial.print(" - output: ");

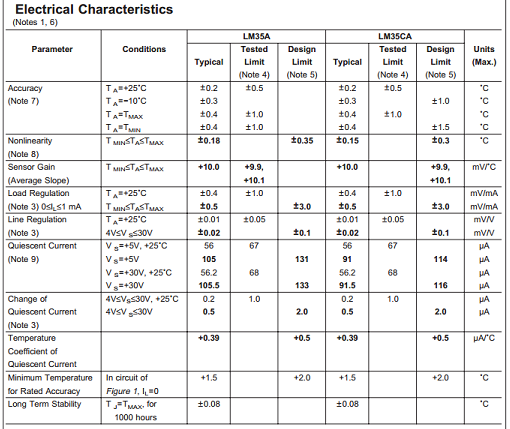
Serial.println(out);

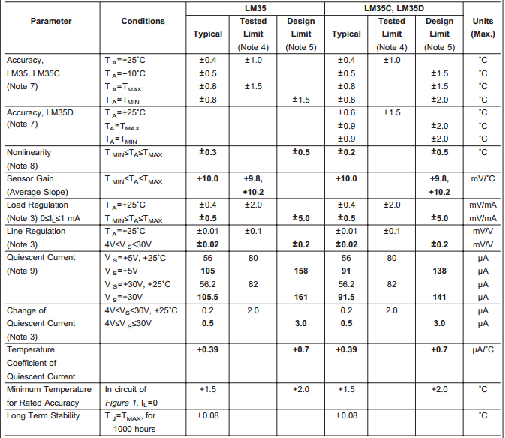
Serial.println("");

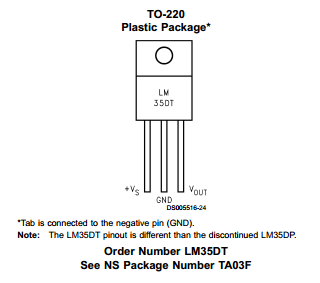
#endif

}

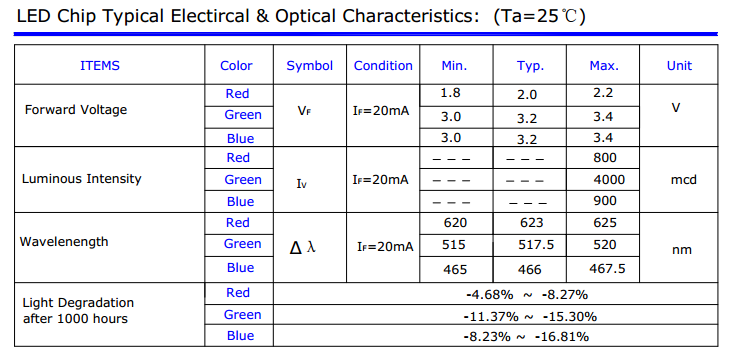
**Appendix B – LM35 Temperature Sensor Data Sheets**

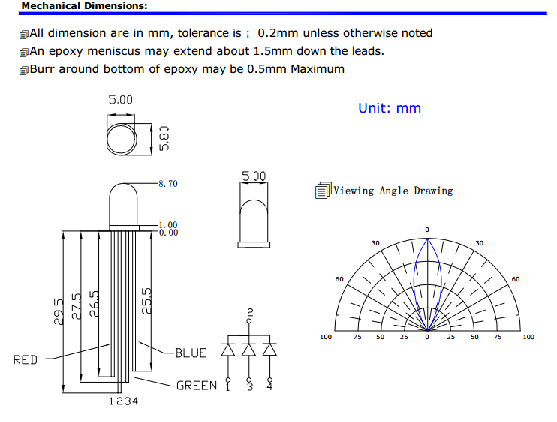




****

**Appendix C – Anode RGB LED Data Sheets**

****

****

**Appendix D – PID Controller Description**

**-** The process variable, i.e. the output to control the fan speed.

**-** The current error, measured as the difference between the setpoint (desired temperature) and the current input (the current temperature).

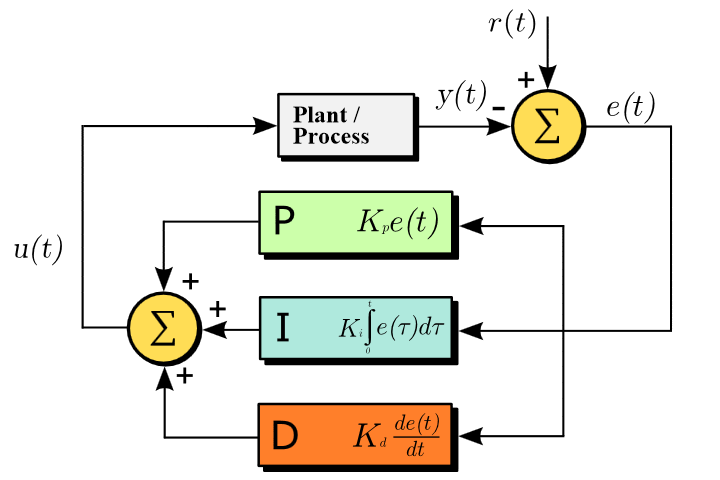
**-** The sum of instantaneous errors over time.

- The rate of change of error.

**-** The proportional gain. Combined with the current error, the proportional part of the equation produces and output proportional to the current error value. This contributes to the bulk of output change and is one of the tuning parameters that the user can modify. Increasing this parameter will decrease rise time, increase overshoot, and cause a small change in the settling time.

- The integral gain. Combined with the sum of instantaneous errors over time, the integral part of the equation produces the accumulated offset that should have been corrected previously. This causes an accelerated movement of the output towards the setpoint. It may also cause an overshoot beyond the setpoint and system instability. As like above, this is a tuning parameter. Increasing this parameter will decrease rise time, increase overshoot, and increase settling time.

- The derivative gain. Combined with the rate of change of error, the derivative part of the equation predicts future errors based on the current rate of change and improves settling time and stability of the system (caused by the integral portion). This is the final tuning parameter. Increasing this parameter will cause a small change in rise time, decrease overshoot, and decrease settling time.



**Appendix E – Arduino PID Library**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Arduino PID Library - Version 1.0.1

\* by Brett Beauregard <br3ttb@gmail.com> brettbeauregard.com

\*

\* This Library is licensed under a GPLv3 License

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#if ARDUINO >= 100

#include "Arduino.h"

#else

#include "WProgram.h"

#endif

#include <PID\_v1.h>

/\*Constructor (...)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* The parameters specified here are those for for which we can't set up

\* reliable defaults, so we need to have the user set them.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

PID::PID(double\* Input, double\* Output, double\* Setpoint,

double Kp, double Ki, double Kd, int ControllerDirection)

{

myOutput = Output;

myInput = Input;

mySetpoint = Setpoint;

inAuto = false;

PID::SetOutputLimits(0, 255); //default output limit corresponds to the arduino pwm limits

SampleTime = 100; //default Controller Sample Time is 0.1 seconds

PID::SetControllerDirection(ControllerDirection);

PID::SetTunings(Kp, Ki, Kd);

lastTime = millis()-SampleTime;

}

/\* Compute() \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* This, as they say, is where the magic happens. this function should be called

\* every time "void loop()" executes. the function will decide for itself whether a new

\* pid Output needs to be computed. returns true when the output is computed,

\* false when nothing has been done.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool PID::Compute()

{

if(!inAuto) return false;

unsigned long now = millis();

unsigned long timeChange = (now - lastTime);

if(timeChange>=SampleTime)

{

/\*Compute all the working error variables\*/

double input = \*myInput;

double error = \*mySetpoint - input;

ITerm+= (ki \* error);

if(ITerm > outMax) ITerm= outMax;

else if(ITerm < outMin) ITerm= outMin;

double dInput = (input - lastInput);

/\*Compute PID Output\*/

double output = kp \* error + ITerm- kd \* dInput;

if(output > outMax) output = outMax;

else if(output < outMin) output = outMin;

\*myOutput = output;

/\*Remember some variables for next time\*/

lastInput = input;

lastTime = now;

return true;

}

else return false;

}

/\* SetTunings(...)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* This function allows the controller's dynamic performance to be adjusted.

\* it's called automatically from the constructor, but tunings can also

\* be adjusted on the fly during normal operation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void PID::SetTunings(double Kp, double Ki, double Kd)

{

if (Kp<0 || Ki<0 || Kd<0) return;

dispKp = Kp; dispKi = Ki; dispKd = Kd;

double SampleTimeInSec = ((double)SampleTime)/1000;

kp = Kp;

ki = Ki \* SampleTimeInSec;

kd = Kd / SampleTimeInSec;

if(controllerDirection ==REVERSE)

{

kp = (0 - kp);

ki = (0 - ki);

kd = (0 - kd);

}

}

/\* SetSampleTime(...) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* sets the period, in Milliseconds, at which the calculation is performed

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void PID::SetSampleTime(int NewSampleTime)

{

if (NewSampleTime > 0)

{

double ratio = (double)NewSampleTime

/ (double)SampleTime;

ki \*= ratio;

kd /= ratio;

SampleTime = (unsigned long)NewSampleTime;

}

}

/\* SetOutputLimits(...)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* This function will be used far more often than SetInputLimits. while

\* the input to the controller will generally be in the 0-1023 range (which is

\* the default already,) the output will be a little different. maybe they'll

\* be doing a time window and will need 0-8000 or something. or maybe they'll

\* want to clamp it from 0-125. who knows. at any rate, that can all be done

\* here.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void PID::SetOutputLimits(double Min, double Max)

{

if(Min >= Max) return;

outMin = Min;

outMax = Max;

if(inAuto)

{

if(\*myOutput > outMax) \*myOutput = outMax;

else if(\*myOutput < outMin) \*myOutput = outMin;

if(ITerm > outMax) ITerm= outMax;

else if(ITerm < outMin) ITerm= outMin;

}

}

/\* SetMode(...)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Allows the controller Mode to be set to manual (0) or Automatic (non-zero)

\* when the transition from manual to auto occurs, the controller is

\* automatically initialized

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void PID::SetMode(int Mode)

{

bool newAuto = (Mode == AUTOMATIC);

if(newAuto == !inAuto)

{ /\*we just went from manual to auto\*/

PID::Initialize();

}

inAuto = newAuto;

}

/\* Initialize()\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* does all the things that need to happen to ensure a bumpless transfer

\* from manual to automatic mode.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void PID::Initialize()

{

ITerm = \*myOutput;

lastInput = \*myInput;

if(ITerm > outMax) ITerm = outMax;

else if(ITerm < outMin) ITerm = outMin;

}

/\* SetControllerDirection(...)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* The PID will either be connected to a DIRECT acting process (+Output leads

\* to +Input) or a REVERSE acting process(+Output leads to -Input.) we need to

\* know which one, because otherwise we may increase the output when we should

\* be decreasing. This is called from the constructor.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void PID::SetControllerDirection(int Direction)

{

if(inAuto && Direction !=controllerDirection)

{

kp = (0 - kp);

ki = (0 - ki);

kd = (0 - kd);

}

controllerDirection = Direction;

}

/\* Status Funcions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Just because you set the Kp=-1 doesn't mean it actually happened. these

\* functions query the internal state of the PID. they're here for display

\* purposes. this are the functions the PID Front-end uses for example

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double PID::GetKp(){ return dispKp; }

double PID::GetKi(){ return dispKi;}

double PID::GetKd(){ return dispKd;}

int PID::GetMode(){ return inAuto ? AUTOMATIC : MANUAL;}

int PID::GetDirection(){ return controllerDirection;}

**Appendix F – Arduino LM35 Temperature Sensor Library**

/\*

LM35.cpp - Temperature Sensor Library for use LM35 - implementation

Copyright (c) 2012 Levi Mota. All right reserved.

\*/

// include core Wiring API

#if defined(ARDUINO) && ARDUINO < 100

#include "WProgram.h"

#else

#include "Arduino.h"

#endif

// include this library's description file

#include "LM35.h"

const double DIVIDER\_HIGH\_RES = 9.309;

const double DIVIDER\_LOW\_RES = 2.048;

// Constructors /////////////////////////////////////////////////////////////////

// Function that handles the creation and setup of instances

LM35Sensor::LM35Sensor(void) {

// initialize this instance's variables

setSamples(500);

setHighRes(false);

}

LM35Sensor::LM35Sensor(int pSamples) {

// initialize this instance's variables

setSamples(pSamples);

setHighRes(false);

}

LM35Sensor::LM35Sensor(int pSamples, bool pHighRes) {

// initialize this instance's variables

setSamples(pSamples);

setHighRes(pHighRes);

}

// Public Methods //////////////////////////////////////////////////////////////

// Functions available in Wiring sketches, this library, and other libraries

void LM35Sensor::setSamples(int pSamples) {

samples = pSamples;

}

void LM35Sensor::setHighRes(bool pHighRes) {

highRes = pHighRes;

if (highRes) {

divider = DIVIDER\_HIGH\_RES;

analogReference(HIGH\_RES);

} else {

divider = DIVIDER\_LOW\_RES;

analogReference(LOW\_RES);

}

}

void LM35Sensor::read(int port) {

double tempSum = 0.0;

for (int i = 0; i < samples; i++) {

tempSum = tempSum + (analogRead(port) / divider);

}

celsius = tempSum / samples;

fahrenheit = 1.8 \* celsius + 32;

kelvin = celsius + 273.15;

}

double LM35Sensor::getCelsius(void) {

return celsius;

}

double LM35Sensor::getFahrenheit(void) {

return fahrenheit;

}

double LM35Sensor::getKelvin(void) {

return kelvin;

}

// Private Methods /////////////////////////////////////////////////////////////

// Functions only available to other functions in this library

**Appendix G – PID Controller Graphing Monitor**

// Maurice Ribble

// 6-28-2009

// http://www.glacialwanderer.com/hobbyrobotics

// This takes data off the serial port and graphs it.

// There is an option to log this data to a file.

// I wrote an arduino app that sends data in the format expected by this app.

// The arduino app sends accelerometer and gyroscope data.

//

// Modified by Bro\_Az August 2010 for PID monitoring.

// Modified by Matthew Tesfaldet March 2013 for compatibility

import processing.serial.\*;

// Globals

int g\_winW = 820; // Window Width

int g\_winH = 600; // Window Height

boolean g\_dumpToFile = false; // Dumps data to c:\\output.txt in a comma seperated format (easy to import into Excel)

boolean g\_enableFilter = false; // Enables simple filter to help smooth out data.

cDataArray g\_xAccel = new cDataArray(200);

cDataArray g\_yAccel = new cDataArray(200);

cDataArray g\_zAccel = new cDataArray(200);

cDataArray g\_vRef = new cDataArray(200);

cDataArray g\_xRate = new cDataArray(200);

cDataArray g\_yRate = new cDataArray(200);

cGraph g\_graph = new cGraph(10, 190, 800, 400);

Serial g\_serial;

PFont g\_font;

void setup()

{

size(g\_winW, g\_winH, P2D);

println(Serial.list());

g\_serial = new Serial(this, Serial.list()[0], 9600, 'N', 8, 1.0);

g\_font = loadFont("ArialMT-20.vlw");

textFont(g\_font, 20);

// This draws the graph key info

strokeWeight(1.5);

stroke(255, 0, 0); line(20, 420, 35, 420);

stroke(0, 255, 0); line(20, 440, 35, 440);

stroke(0, 0, 255); line(20, 460, 35, 460);

stroke(255, 255, 0); line(20, 480, 35, 480);

stroke(255, 0, 255); line(20, 500, 35, 500);

stroke(0, 255, 255); line(20, 520, 35, 520);

fill(0, 0, 0);

text("Temperature Setpoint", 40, 430);

text("Actual Temperature", 40, 450);

text("Controller Output", 40, 470);

text("Analog 3", 40, 490);

text("Analog 4", 40, 510);

text("Analog 5", 40, 530);

if (g\_dumpToFile)

{

// This clears deletes the old file each time the app restarts

byte[] tmpChars = {'\r', '\n'};

saveBytes("c:\\output.txt", tmpChars);

}

}

void draw()

{

// We need to read in all the avilable data so graphing doesn't lag behind

while (g\_serial.available() >= 2\*6+2)

{

processSerialData();

}

strokeWeight(1);

fill(255, 255, 255);

g\_graph.drawGraphBox();

strokeWeight(1.5);

stroke(255, 0, 0);

g\_graph.drawLine(g\_xAccel, 0, 1024);

stroke(0, 255, 0);

g\_graph.drawLine(g\_yAccel, 0, 1024);

stroke(0, 0, 255);

g\_graph.drawLine(g\_zAccel, 0, 1024);

stroke(255, 255, 0);

g\_graph.drawLine(g\_vRef, 0, 1024);

stroke(255, 0, 255);

g\_graph.drawLine(g\_xRate, 0, 1024);

stroke(0, 255, 255);

g\_graph.drawLine(g\_yRate, 0, 1024);

}

// This reads in one set of the data from the serial port

void processSerialData()

{

int inByte = 0;

int curMatchPos = 0;

int[] intBuf = new int[2];

intBuf[0] = 0xAD;

intBuf[1] = 0xDE;

while (g\_serial.available() < 2); // Loop until we have enough bytes

inByte = g\_serial.read();

// This while look looks for two bytes sent by the client 0xDEAD

// This allows us to resync the server and client if they ever

// lose sync. In my testing I haven't seen them lose sync so

// this could be removed if you need to, but it is a good way to

// prevent catastrophic failure.

while(curMatchPos < 2)

{

if (inByte == intBuf[curMatchPos])

{

++curMatchPos;

if (curMatchPos == 2)

break;

while (g\_serial.available() < 2); // Loop until we have enough bytes

inByte = g\_serial.read();

}

else

{

if (curMatchPos == 0)

{

while (g\_serial.available() < 2); // Loop until we have enough bytes

inByte = g\_serial.read();

}

else

{

curMatchPos = 0;

}

}

}

while (g\_serial.available() < 2\*6); // Loop until we have a full set of data

// This reads in one set of data

{

byte[] inBuf = new byte[2];

int xAccel, yAccel, zAccel, vRef, xRate, yRate;

g\_serial.readBytes(inBuf);

// Had to do some type conversion since Java doesn't support unsigned bytes

xAccel = ((int)(inBuf[1]&0xFF) << 8) + ((int)(inBuf[0]&0xFF) << 0);

g\_serial.readBytes(inBuf);

yAccel = ((int)(inBuf[1]&0xFF) << 8) + ((int)(inBuf[0]&0xFF) << 0);

g\_serial.readBytes(inBuf);

zAccel = ((int)(inBuf[1]&0xFF) << 8) + ((int)(inBuf[0]&0xFF) << 0);

g\_serial.readBytes(inBuf);

vRef = ((int)(inBuf[1]&0xFF) << 8) + ((int)(inBuf[0]&0xFF) << 0);

g\_serial.readBytes(inBuf);

xRate = ((int)(inBuf[1]&0xFF) << 8) + ((int)(inBuf[0]&0xFF) << 0);

g\_serial.readBytes(inBuf);

yRate = ((int)(inBuf[1]&0xFF) << 8) + ((int)(inBuf[0]&0xFF) << 0);

g\_xAccel.addVal(xAccel);

g\_yAccel.addVal(yAccel);

g\_zAccel.addVal(zAccel);

g\_vRef.addVal(vRef);

g\_xRate.addVal(xRate);

g\_yRate.addVal(yRate);

if (g\_dumpToFile) // Dump data to a file if needed

{

String tempStr;

tempStr = xAccel + "," + yAccel + "," + zAccel + "," + vRef + "," + xRate + "," + yRate + "\r\n";

FileWriter file;

try

{

file = new FileWriter("c:\\output.txt", true); //bool tells to append

file.write(tempStr, 0, tempStr.length()); //(string, start char, end char)

file.close();

}

catch(Exception e)

{

println("Error: Can't open file!");

}

}

/\*

print(xAccel); print(" "); print(yAccel); print(" "); print(zAccel); print(" ");

print(vRef); print(" "); print(xRate); print(" "); println(yRate);

\*/

}

}

// This class helps manage the arrays of data I need to keep around for graphing.

class cDataArray

{

float[] m\_data;

int m\_maxSize;

int m\_startIndex = 0;

int m\_endIndex = 0;

int m\_curSize;

cDataArray(int maxSize)

{

m\_maxSize = maxSize;

m\_data = new float[maxSize];

}

void addVal(float val)

{

if (g\_enableFilter && (m\_curSize != 0))

{

int indx;

if (m\_endIndex == 0)

indx = m\_maxSize-1;

else

indx = m\_endIndex - 1;

m\_data[m\_endIndex] = getVal(indx)\*.5 + val\*.5;

}

else

{

m\_data[m\_endIndex] = val;

}

m\_endIndex = (m\_endIndex+1)%m\_maxSize;

if (m\_curSize == m\_maxSize)

{

m\_startIndex = (m\_startIndex+1)%m\_maxSize;

}

else

{

m\_curSize++;

}

}

float getVal(int index)

{

return m\_data[(m\_startIndex+index)%m\_maxSize];

}

int getCurSize()

{

return m\_curSize;

}

int getMaxSize()

{

return m\_maxSize;

}

}

// This class takes the data and helps graph it

class cGraph

{

float m\_gWidth, m\_gHeight;

float m\_gLeft, m\_gBottom, m\_gRight, m\_gTop;

cGraph(float x, float y, float w, float h)

{

m\_gWidth = w;

m\_gHeight = h;

m\_gLeft = x;

m\_gBottom = g\_winH - y;

m\_gRight = x + w;

m\_gTop = g\_winH - y - h;

}

void drawGraphBox()

{

stroke(0, 0, 0);

rectMode(CORNERS);

rect(m\_gLeft, m\_gBottom, m\_gRight, m\_gTop);

}

void drawLine(cDataArray data, float minRange, float maxRange)

{

float graphMultX = m\_gWidth/data.getMaxSize();

float graphMultY = m\_gHeight/(maxRange-minRange);

for(int i=0; i<data.getCurSize()-1; ++i)

{

float x0 = i\*graphMultX+m\_gLeft;

float y0 = m\_gBottom-((data.getVal(i)-minRange)\*graphMultY);

float x1 = (i+1)\*graphMultX+m\_gLeft;

float y1 = m\_gBottom-((data.getVal(i+1)-minRange)\*graphMultY);

line(x0, y0, x1, y1);

}

}

}