|  |
| --- |
| Code: |
| #include <PID\_v1.h>  //ButtonConstants  #define debounceDelay 90  #define selectC 0x02  #define upC 0x10  #define downC 0x04  #define backC 0x01  #define pauseC 0x08  #define dispensePin 40  #define redPin 38  #define greenPin 39  // Define ADC prescaler  const unsigned char PS\_16 = (1 << ADPS2);  const unsigned char PS\_128 = (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);  //MotorConstants  #define motorPin 46  #define motorInA 50  #define motorInB 53  #define Kp .8  #define Ki .01  #define Kd .005  #define minPWM 32  #define sTime 20  #define numOfSamples 500  #define fourMMNutSetPoint 950  #define tenMMNutSetPoint 934  #define fiftyMMScrewSetPoint 952  #define twentyFiveMMScrewSetPoint 943  #define wallAnchorSetPoint 950  #define dryWallScrewSetPoint 952  #define shakeDelay 700  #define lightJam 830  #define heavyJam 640  #define blink\_time 750  //Custom datatypes for LCD, motor, leds, and sensor states  typedef enum STATE\_LCD  {  WELCOME,  SELECT\_PART,  SELECT\_AMOUNT,  CONFIRM,  LOADING,  COUNTING,  COMPLETE,  PAUSE,  ERROR\_SENSOR,  ERROR\_MOTOR,  ERROR\_USER  } state\_lcd;    typedef enum STATE\_MOTOR  {  ON,  OFF,  } state\_motor;    typedef enum STATE\_SENSOR  {  ACTIVE,  STANDBY  } state\_sensor;    typedef enum STATE\_LED  {  ENABLED,  DISABLED,  BLINK,  } state\_led;  //Structures for all controllers  typedef struct LCD\_Controller  {  short amount;  short part;  boolean confirmation;  state\_lcd state;  } LCD\_Controller;    typedef struct Buttons  {  byte select;  long selectDB;  boolean selectF;  byte up;  long upDB;  boolean upF;  byte down;  long downDB;  boolean downF;  byte back;  long backDB;  boolean backF;  byte pause;  long pauseDB;  boolean pauseF;  byte dispense;  long dispenseDB;  boolean dispenseF;  } Buttons;    typedef struct Motor\_Controller  {  state\_motor state;  double peak;  double pwm;  double current;  double set\_point;  int sample\_num;  } Motor\_Controller;    typedef struct SENSOR\_CONTROLLER  {  boolean topHi;  boolean btmHi;  short top;  short btm;  short tempCount;  short partCount;  state\_sensor state;  } sensor\_controller;    typedef struct INDICATOR\_CONTROLLER  {  boolean red;  long red\_time;  state\_led red\_state;  boolean green;  long green\_time;  state\_led green\_state;  } indicator\_controller;  //Initialization of controllers  LCD\_Controller LCD = {1,1,false,WELCOME};  Buttons BTNS = {0,0,false,0,0,false,0,0,false,0,0,false,0,0,false,0,0,false};  Motor\_Controller motor = {OFF,0,0,0,SPT,0};  sensor\_controller sensors = {false,false,0,0,0,0,STANDBY};  indicator\_controller leds = {false,0,ENABLED,true,0,DISABLED};  PID PIDcontroller = PID(&motor.peak,&motor.pwm,&motor.set\_point,Kp,Ki,Kd,DIRECT);  String part\_name[] = {"50mm Cap Screw","4mm Hex Nut","25mm Cap Screw","10mm Hex Nut","Wall Anchor","Dry-wall Screw"};  state\_lcd previous\_state;  int jamReading = lightJam;  void setup()  {  Serial.begin(9600);  setup\_adc();  setup\_BTNS();  setup\_motor();  setup\_sensors();  lcd\_setup();  update\_LCD();  }  void loop()  {  process\_input();  get\_count();  control\_motor();  control\_leds();  }  //Displays correct text on LCD according to the state of the LCD  void update\_LCD()  {  switch(LCD.state)  {  case WELCOME:  clear\_lcd();  writeLCD(0x84);  writeLCD("Welcome!");  nextLine();  for(int i = 0;i<16; i++)  {  writeLCD("\*");  delay(100);  }  LCD\_next\_screen();  update\_LCD();  break;  case SELECT\_PART:  clear\_lcd();  writeLCD("Select Part:");  nextLine();  printLCD(part\_name[LCD.part]);  break;  case SELECT\_AMOUNT:  clear\_lcd();  writeLCD("Select Amount:");  nextLine();  printLCD(String(LCD.amount));  break;  case CONFIRM:  clear\_lcd();  writeLCD("Ready to begin?");  nextLine();  if(LCD.confirmation == false)  {  writeLCD("X No Yes");  }  else  {  writeLCD(" No X Yes");  }  break;  case COUNTING:  clear\_lcd();  writeLCD("Counting...");  nextLine();  printLCD("Parts: " + String(sensors.partCount));  break;  case COMPLETE:  clear\_lcd();  writeLCD(" \* Ready to \*");  nextLine();  writeLCD(" \* Dispense! \*");  break;  case PAUSE:  clear\_lcd();  writeLCD("Count paused.");  nextLine();  printLCD("Parts: " + String(sensors.partCount));  break;  case ERROR\_SENSOR:  clear\_lcd();  setLCDCursor(5,1);  writeLCD("ERROR!");  nextLine();  writeLCD("Sensor failure.");  break;  case ERROR\_MOTOR:  clear\_lcd();  setLCDCursor(5,1);  writeLCD("ERROR!");  nextLine();  writeLCD("Motor failure.");  break;  case ERROR\_USER:  clear\_lcd();  setLCDCursor(5,1);  writeLCD("ERROR!");  nextLine();  writeLCD("Dispenser open.");  break;  }  }  //Transitions the LCD and controllers to the next state  void LCD\_next\_screen()  {  switch(LCD.state)  {  case WELCOME:  LCD.state = SELECT\_PART;  break;  case SELECT\_PART:  LCD.state = SELECT\_AMOUNT;  break;  case SELECT\_AMOUNT:  LCD.state = CONFIRM;  break;  case CONFIRM:  sensors.state = ACTIVE;  LCD.state = COUNTING;  break;  case COUNTING:  sensors.state = STANDBY;  LCD.state = COMPLETE;  break;  case PAUSE:  sensors.state = ACTIVE;  LCD.state = COUNTING;  break;  case COMPLETE:  sensors.state = ACTIVE;  LCD.state = COUNTING;  break;  case ERROR\_SENSOR:  sensors.state = STANDBY;  LCD.state = SELECT\_PART;  break;  case ERROR\_MOTOR:  sensors.state = STANDBY;  LCD.state = SELECT\_PART;  break;  case ERROR\_USER:  sensors.state = STANDBY;  LCD.state = SELECT\_PART;  break;  }  }    //Transitions LCD to previous state  void LCD\_previous\_screen()  {  switch(LCD.state)  {  case WELCOME:  break;  case SELECT\_PART:  break;  case SELECT\_AMOUNT:  LCD.state = SELECT\_PART;  break;  case CONFIRM:  LCD.state = SELECT\_AMOUNT;  break;  case COUNTING:  break;  case COMPLETE:  break;  case ERROR\_SENSOR:  break;  case ERROR\_MOTOR:  break;  case ERROR\_USER:  break;  }  }  void process\_input()  {  //registers input for all buttons on register A  byte buttonState = PINA;  switch(LCD.state)  {  case SELECT\_PART:  if(upBTN(buttonState))  {  if(LCD.part != 5)  {  LCD.part++;  }  else  {  LCD.part = 0;  }  changeSetPoint();  update\_LCD();  }  if(downBTN(buttonState))  {  if(LCD.part != 0)  {  LCD.part--;  }  else  {  LCD.part = 5;  }  changeSetPoint();  update\_LCD();  }  if(selectBTN(buttonState))  {  LCD\_next\_screen();  update\_LCD();  }  break;  case SELECT\_AMOUNT:  if(upBTN(buttonState))  {  if(LCD.amount != 6)  {  LCD.amount++;  }  else  {  LCD.amount = 1;  }  update\_LCD();  }  if(downBTN(buttonState))  {  if(LCD.amount != 1)  {  LCD.amount--;  }  else  {  LCD.amount = 6;  }  update\_LCD();  }  if(selectBTN(buttonState))  {  LCD\_next\_screen();  update\_LCD();  }  if(backBTN(buttonState))  {  LCD\_previous\_screen();  update\_LCD();  }  break;  case CONFIRM:  if(upBTN(buttonState))  {  LCD.confirmation = !LCD.confirmation;  update\_LCD();  }  if(downBTN(buttonState))  {  LCD.confirmation = !LCD.confirmation;  update\_LCD();  }  if(selectBTN(buttonState))  {  if(LCD.confirmation)  {  leds.red\_state = DISABLED;  leds.green\_state = BLINK;  LCD\_next\_screen();  update\_LCD();  motor.state = ON;  switch\_motor();  }  }  if(backBTN(buttonState))  {  LCD\_previous\_screen();  update\_LCD();  }  break;  case COUNTING:  if(pauseBTN(buttonState))  {  leds.red\_state = ENABLED;  leds.green\_state = DISABLED;  leds.green = true;  motor.state = OFF;  switch\_motor();  LCD.state = PAUSE;  update\_LCD();  }  break;  case PAUSE:  if(pauseBTN(buttonState))  {  leds.red\_state = DISABLED;  leds.green\_state = BLINK;  motor.state = ON;  switch\_motor();  LCD.state = COUNTING;  update\_LCD();  }  case COMPLETE:  if(dispenseBTN(digitalRead(dispensePin)))  {  delay(2000);  leds.red\_state = DISABLED;  leds.green\_state = BLINK;  sensors.partCount = 0;  sensors.state = ACTIVE;  motor.state = ON;  switch\_motor();  LCD\_next\_screen();  update\_LCD();  }  break;  case ERROR\_SENSOR:  break;  case ERROR\_MOTOR:  break;  case ERROR\_USER:  break;  }  }    void control\_leds()  {  if(leds.red\_state == ON && leds.red == false)  {  digitalWrite(redPin,HIGH);  leds.red = true;  }  if(leds.red\_state == OFF && leds.red == true)  {  digitalWrite(redPin,LOW);  leds.red = false;  }  if(leds.green\_state == ON && leds.green == false)  {  digitalWrite(greenPin,HIGH);  leds.green = true;  }  if(leds.green\_state == OFF && leds.green == true)  {  digitalWrite(greenPin,LOW);  leds.green = false;  }  if(leds.green\_state == BLINK)  {  if(millis()-leds.green\_time > blink\_time)  {  digitalWrite(greenPin,leds.green);  leds.green = !leds.green;  leds.green\_time = millis();  }  }  if(leds.red\_state == BLINK)  {  if(millis() - leds.red\_time > blink\_time)  {  digitalWrite(redPin,!leds.red);  leds.red = !leds.red;  leds.red\_time = millis();  }  }  }  //Sets register A as input and sets up LED Pins as outputs  void setup\_BTNS()  {  DDRA = 0x00;  pinMode(dispensePin,INPUT);  pinMode(greenPin,OUTPUT);  pinMode(redPin,OUTPUT);  }  //All button methods. Debounces and prevents double button presses.  boolean selectBTN(byte buttonVals)  {  byte btnstate = buttonVals&selectC;  if (btnstate != BTNS.select)  {  BTNS.selectDB = millis();  BTNS.select = btnstate;  BTNS.selectF = true;  }    if ((millis() - BTNS.selectDB) > debounceDelay)  {  if(BTNS.selectF && BTNS.select != 0x00)  {  BTNS.selectF = false;  return true;  }  else  {  return false;  }  }  else  {  return false;  }  }  boolean upBTN(byte buttonVals)  {  byte btnstate = buttonVals&upC;    if (btnstate != BTNS.up)  {  BTNS.upDB = millis();  BTNS.up = btnstate;  BTNS.upF = true;  }    if ((millis() - BTNS.upDB) > debounceDelay)  {  if(BTNS.upF && BTNS.up != 0x00)  {  BTNS.upF = false;  return true;  }  else  {  return false;  }  }  else  {  return false;  }  }  boolean downBTN(byte buttonVals)  {  byte btnstate = buttonVals&downC;    if (btnstate != BTNS.down)  {  BTNS.downDB = millis();  BTNS.down = btnstate;  BTNS.downF = true;  }    if ((millis() - BTNS.downDB) > debounceDelay)  {  if(BTNS.downF && BTNS.down != 0x00)  {  BTNS.downF = false;  return true;  }  else  {  return false;  }  }  else  {  return false;  }  }  boolean backBTN(byte buttonVals)  {  byte btnstate = buttonVals&backC;    if (btnstate != BTNS.back)  {  BTNS.backDB = millis();  BTNS.back = btnstate;  BTNS.backF = true;  }    if ((millis() - BTNS.backDB) > debounceDelay)  {  if(BTNS.backF && BTNS.back != 0x00)  {  BTNS.backF = false;  return true;  }  else  {  return false;  }  }  else  {  return false;  }  }  boolean pauseBTN(byte buttonVals)  {  byte btnstate = buttonVals&pauseC;    if (btnstate != BTNS.pause)  {  BTNS.pauseDB = millis();  BTNS.pause = btnstate;  BTNS.pauseF = true;  }    if ((millis() - BTNS.pauseDB) > debounceDelay)  {  if(BTNS.pauseF && BTNS.pause != 0x00)  {  BTNS.pauseF = false;  return true;  }  else  {  return false;  }  }  else  {  return false;  }  }    boolean dispenseBTN(boolean buttonVals)  {  boolean btnstate = !buttonVals;    if (btnstate != BTNS.dispense)  {  BTNS.dispenseDB = millis();  BTNS.dispense = btnstate;  BTNS.dispenseF = true;  }    if ((millis() - BTNS.dispenseDB) > debounceDelay)  {  if(BTNS.dispenseF && BTNS.dispense != 0x00)  {  BTNS.dispenseF = false;  return true;  }  else  {  return false;  }  }  else  {  return false;  }  }  //LCD Methods  void lcd\_setup()  {  Serial1.begin(19200);  writeLCD(0x16);  writeLCD(0x11);  }  void clear\_lcd()  {  writeLCD(0x0C);  }  void setLCDCursor(int x, int y)  {  writeLCD(128+20\*y+x);  }  void nextLine()  {  writeLCD(0x0D);  }  void printLCD(String text)  {  Serial1.print(text);  }  void writeLCD(char text[])  {  Serial1.write(text);  }  void writeLCD(int text)  {  Serial1.write(text);  }  //Motor Control Methods  //Sets the PWM to a certain frequency  void setPwmFrequency(int pin, int divisor)  {  byte mode;  if(pin == 5 || pin == 6 || pin == 9 || pin == 10) {  switch(divisor) {  case 1: mode = 0x01; break;  case 8: mode = 0x02; break;  case 64: mode = 0x03; break;  case 256: mode = 0x04; break;  case 1024: mode = 0x05; break;  default: return;  }  if(pin == 5 || pin == 6) {  TCCR0B = TCCR0B & 0b11111000 | mode;  } else {  TCCR1B = TCCR1B & 0b11111000 | mode;  }  } else if(pin == 3 || pin == 11) {  switch(divisor) {  case 1: mode = 0x01; break;  case 8: mode = 0x02; break;  case 32: mode = 0x03; break;  case 64: mode = 0x04; break;  case 128: mode = 0x05; break;  case 256: mode = 0x06; break;  case 1024: mode = 0x7; break;  default: return;  }  TCCR2B = TCCR2B & 0b11111000 | mode;  }  }    //Increases sampling rate of ADC with little or not effect on accuracy  void setup\_adc()  {  ADCSRA &= ~PS\_128;  ADCSRA |= PS\_16;  }  //Samples current for numOfSamples and uses peak as input for PID controller  void control\_motor()  {  if(motor.state == ON)  {  int reading = analogRead(15);  if(reading > motor.peak)  {  motor.peak = reading;  }  motor.sample\_num++;  if(motor.sample\_num >= numOfSamples)  {  motor.sample\_num = 0;  motor.peak = 1024-motor.peak;  if(motor.peak > jamReading)  {  PIDcontroller.Compute();  analogWrite(motorPin,motor.pwm);  }  else  {  fix\_jam();  }  motor.peak = 0;  }  }  }    //Sets ADC to use 1.1V reference, Sets A,B,and PWM pins as output, Sets PWM frequency to max, Sets PID controller output constraints, sample time, and mode.  void setup\_motor()  {  analogReference(INTERNAL1V1);  pinMode(motorPin,OUTPUT);  pinMode(motorInA,OUTPUT);  pinMode(motorInB,OUTPUT);  setPwmFrequency(motorPin,1);  PIDcontroller.SetMode(AUTOMATIC);  PIDcontroller.SetOutputLimits(minPWM,255);  PIDcontroller.SetSampleTime(sTime);  }  void resetMotorVals()  {  motor.peak = 0;  motor.pwm = 0;  motor.sample\_num = 0;  }  //Turns on and off motor according to the motor state  void switch\_motor()  {  if(motor.state == ON)  {  resetMotorVals();  PIDcontroller.SetMode(AUTOMATIC);  digitalWrite(motorInB,LOW);  digitalWrite(motorInA,HIGH);  }  else  {  PIDcontroller.SetMode(MANUAL);  digitalWrite(motorInB,HIGH);  digitalWrite(motorInA,HIGH);  }  }    //Sensor Module Methods  //Sets register C as input  void setup\_sensors()  {  DDRC = 0x00;  }  //Gets values from register C and applies bitmasks for upper and lower sensors groups  void readSensors()  {  byte sensorsRead = PINC ;  sensors.top = sensorsRead & 0xF0;  sensors.btm = sensorsRead & 0x00; //0x0F  }  void get\_count()  {  if(sensors.state == ACTIVE)  {  readSensors();    //Uses double level algorithm for the small nuts  if(false)//LCD.part == 1 | LCD.part == 3)  {  if (false)//sensors.top != 0x00 && !sensors.topHi)  {  sensors.topHi=true;  sensors.tempCount ++;  delay(10);  }  else  {  if(sensors.top == 0x00)  {  sensors.topHi=false;  }  }  if (sensors.btm != 0x00 && !sensors.btmHi)  {  sensors.btmHi=true;  sensors.tempCount ++;  delay(10);  }  else  {  if(sensors.btm == 0x00)  {  sensors.btmHi=false;  }  }    if (sensors.tempCount!=0 && sensors.tempCount%2==0)  {  sensors.partCount ++;  sensors.tempCount = 0;  if(sensors.partCount < LCD.amount)  {  update\_LCD();  }  else  {  LCD\_next\_screen();  leds.green\_state = ENABLED;  leds.red\_state = DISABLED;  update\_LCD();  motor.state = OFF;  switch\_motor();  }  }  }    //Uses single detection algorithm for longer parts  else  {  if (sensors.btm != 0x00 | sensors.top !=0x00)  {  sensors.partCount ++;  if(sensors.partCount < LCD.amount)  {  update\_LCD();  }  else  {  LCD\_next\_screen();  leds.green\_state = ENABLED;  leds.red\_state = DISABLED;  motor.state = OFF;  switch\_motor();  update\_LCD();  }  delay(350);  }  }  }  }  void fix\_jam()  {  digitalWrite(motorInB,HIGH);  digitalWrite(motorInA,HIGH);  delay(50);  analogWrite(motorPin,150);  for(int k = 0; k<5; k++)  {  digitalWrite(motorInB,HIGH);  digitalWrite(motorInA,LOW);  for(int i=0;i<shakeDelay;i++)  {  get\_count();  }  digitalWrite(motorInA,HIGH);  delay(50);  digitalWrite(motorInB,LOW);  for(int i=0;i<shakeDelay/2;i++)  {  get\_count();  }  delay(50);  }  analogWrite(motorPin,0);  PIDcontroller.Compute();    digitalWrite(motorInB,LOW);  digitalWrite(motorInA,HIGH);  }  void changeSetPoint()  {  switch(LCD.part)  {  case 0:  motor.set\_point = fiftyMMScrewSetPoint;  jamReading = lightJam;  break;  case 1:  motor.set\_point = fourMMNutSetPoint;  jamReading = lightJam;  break;  case 2:  motor.set\_point = twentyFiveMMScrewSetPoint;  jamReading = heavyJam;  break;  case 3:  motor.set\_point = tenMMNutSetPoint;  jamReading = heavyJam;  break;  case 4:  motor.set\_point = wallAnchorSetPoint;  jamReading = lightJam;  break;  case 5:  motor.set\_point = dryWallScrewSetPoint;  jamReading = lightJam;  break;  }  } |