Alex Brinkman

Team D: Project HARP (Human Assistive Robotic Picker)

Teammates: Abhishek Bhatia, Lekha Mohan, Feroze Naina, Abhishek Bhatia

ILR #1: Sensors and Motor Control Lab

Submitted 10/16/2015

1. Individual progress

   My individual contribution to the sensors and motor control lab was in creating the DC motor speed controller, helping with GUI development, and establishing the serial data communication. I have implemented PID controllers for other projects and knew I could leverage that experience for this project. We decided to use the Encoder Arduino library to interface with the integrated encoder on the DC motor to enable closed-loop feedback on both the speed and position controllers. I set a configurable speed controller task rate to ensure enough time passed to get a smooth speed reading from the encoder counts. Next, I implemented a configurable gain PI controller with integral burn-off and chose to omit the derivative term since it is most sensitive to sensor noise. The controller gains were tuned and evaluated using a potentiometer sensor as desired speed where the sampled potentiometer voltage was mapped to motor speed in rpm. The potentiometer signal showed low noise and provided an easy way to modify the controller speed set point, which allowed for rapid controller evaluation and modifications. To ease integration, the controller setpoint was implemented so we could direct signals from the GUI controls or any other sensors by changing only one line of code.

   Next, I began work on the QT GUI for the project. Feroze and Lekha set up a working outline for the GUI using QT Developer, PyQt, and Pyserial. Using QT Developer, I was able to create a user interface and define the graphical control objects that would display data from the Arduino and send commands from the GUI. Once these features were defined, I assigned callback functions to execute once the user interacted with the control elements or when serial data periodically arrived from the Arduino. In order to test the GUI elements were working correctly, I had to connect to the Arduino and stream test data. This forced me to define a method for passing data to and from the Arduino to continue the GUI design. Once I was able to show the GUI could accept data from the Arduino, I had to send data over the serial bus to control the Arduino from the GUI control elements. Figure X depicts the final version of the GUI.

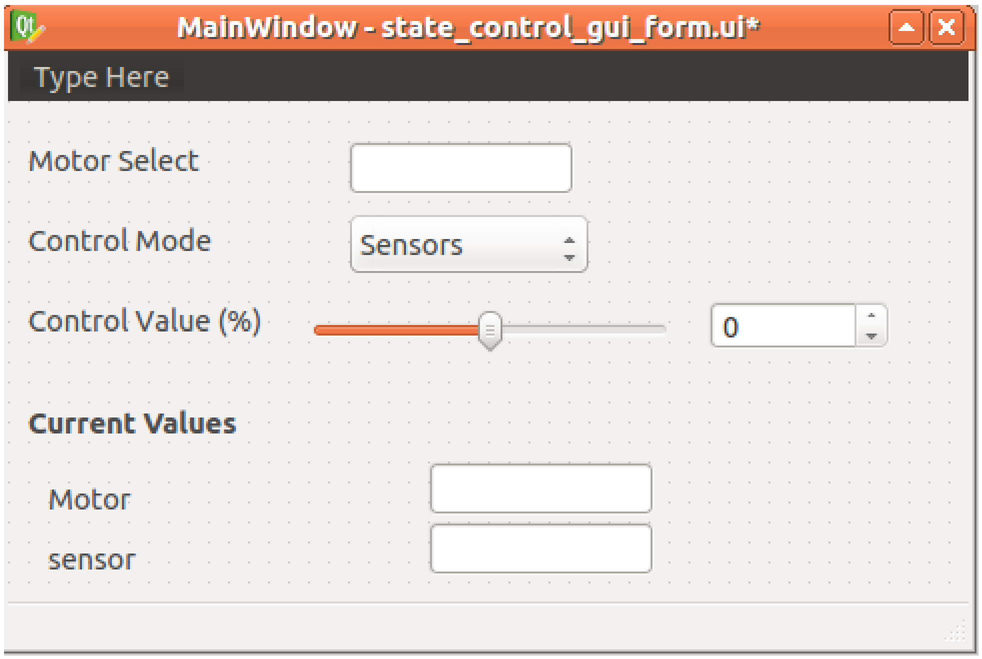

Figure 1: Motor Lab GUI

The control elements consist of a drop-down menu that is used to switch between sensor control and GUI control modes, a slider that updates the motor control signal when configured for GUI control, and a integer text box that numerically defines the motor control signal when configured for GUI control. The fields shown as ‘Motor’ and ‘sensor’ in Figure 1 changed depending on the motor select state and control mode to reflect the correct units and appropriate readings. After these elements were debugged and shown to work, I attempted to use a Qwt plot to show a live data stream, but this task proved to be out of my abilities.

After Abhishek integrated the Arduino code for each individual sensor, I worked with Abhishek and Rick to further debug the serial interface between the Arduino and GUI to improve packet loss and erratic signal passing.

2. Challenges

When developing the DC motor speed controller, I quickly found the motor to exhibit poor controllability at low speeds due to the ratio of the static and dynamic coefficients of friction and tried to tune the controller to account for this. I eventually had to saturate the desired speed signal to zero for low desired speeds to avoid erratic stick-slip behavior. The final implementation saturated desired speeds lower than a magnitude of 18 rpm to zero.

I had experienced several problems when setting up Qt and PyQt. I have developed GUIs in MATLAB’s GUIDE, so I was comfortable with design of the callbacks program flow. It turned out PyQt did not run on my laptop for still unknown reasons. I was, however, able to get everything working on my home desktop and learn Qt, so I developed as much as I could there and then worked in the lab with the team to finalize the GUI.

Serial data passing required significant debugging on both the GUI client side and the Arduino when reading from the serial data buffer. The Qt GUI would sometimes stream the data flawlessly and other times stop displaying the data stream. We ensured we could both simultaneously read and write to the serial port was supported in PySerial so that was not the issue. Next, we thought the problem could be the serial port was not held open long enough before reading the buffer, so we changed the code accordingly and performance improved significantly. On the Arduino serial receiving logic, we noticed after integrating the code the Arduino would show intermittent or delayed response to GUI motor commands. We first increased the baud rate from 9600 to 115200 bps to no effect. We then tried to add a delay after sensing the serial buffer contained new data. This allowed the buffer to fill with data from the GUI before reading the buffer and the Arduino response improved drastically.

3. Teamwork

To begin, we decided to delegate tasks individually and then come together once our individual components were completed to integrate the system. We created a Github repo to both organize the code and familiarize our team with the Github workflow in preparation for the software development needed for our project. We also agreed to an Arduino pin out map to help integration down the line. Feroze and Lekha started the GUI development, Abhishek tackled the stepper motor, Rick created the DC position speed controller and again, I made the DC speed controller. Once these individual tasks were completed, Abhishek integrated the Arduino code segments and I worked on finalizing the GUI and serial communication. Finally, we all worked over the next few days to improve the serial communication, controller performance. Some of these tasks proved easier than others, but I was impressed with the team’s willingness to always be willing to meet and work together to meet our goal.

4. Figures

The motor control hardware and electronics can be seen in Figure 2.

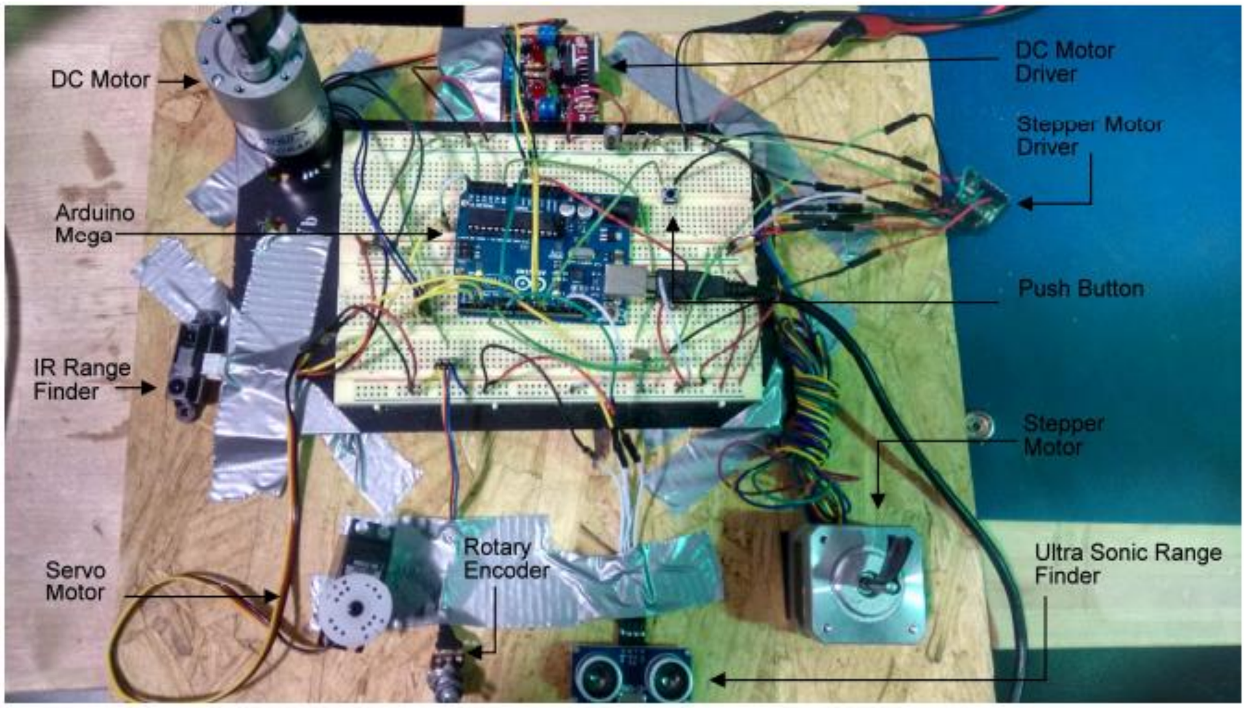

Figure 2: Motor Board

The IR sensor Transfer function was taken from the manufacturers data sheet and adapted to use with the Arduino ADC, seen in Figure 3.

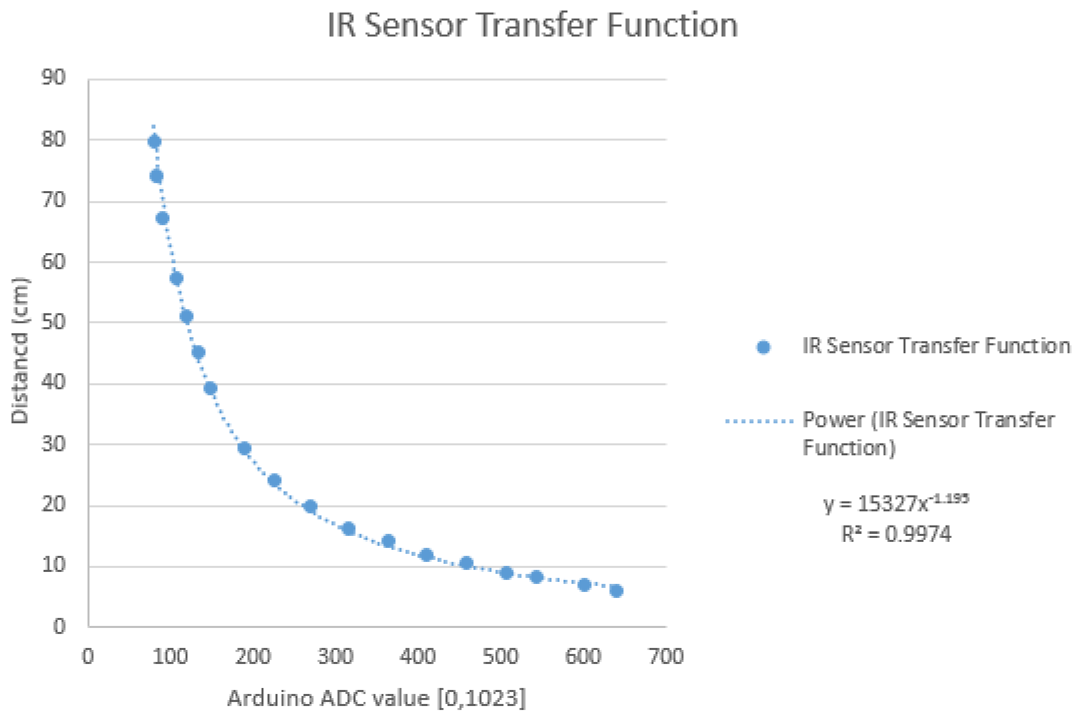

Figure 3: IR Sensor Transfer Function

5. Plans

For the next design review, I will be working on establishing the Git repo and creating a formal Software specification for our ROS software. In two weeks, Feroze and I plan to demo a working software system that runs a state controller that will control the robot during autonomous operation and show a PR2 moving in Rviz. Finally, I will update our work breakdown structure and schedule to reflect changes made from our latest meeting with Prof. Maxim.

6. Code

```
#include <Encoder.h>
#include <Servo.h>

Encoder myEnc(2, 3);
Servo myservo;  // create servo object to control a servo


/* PIN DEFINITIONS */
//Digital
const int pin_DC_mod = 10;
const int pin_DC_H1  = 11;
const int pin_DC_H2  = 6;
const int stepper_dir = 4;
const int stepper_step = 5;
const int echoPin = 7;
const int trigPin = 8;
const int button_pin = 12;
const int servo_pin = 9;

//Analog
const int potpin = A4;
const int IR_pin = A5;

void setup() {

  Serial.begin(115200);
  pinMode(5,OUTPUT); // Step
  pinMode(4,OUTPUT); // Dir
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(button_pin, INPUT);
  myservo.attach(servo_pin);

}

/* GLOBAL VARIABLES */
int val = 0;
unsigned int integerValue = 0;
```

```
char incomingByte;
int control_value_pct = 0;
int control_mode = 0;
int motor_control_value = 0;
int servo_val = 0;
int stepper_val = 0;
int dc_pos_val = 0;
int dc_speed_val = 0;
int button_value_old = 0;
int sensor_value;
int control_mode_old = 0;
int button_state = 0;
int next_state = 0;
static unsigned long last_interrupt_time0 = 0;
unsigned long interrupt_time0;
int maximumRange = 200; // Maximum range needed
int minimumRange = 0; // Minimum range needed
long duration, distance; // Duration used to calculate distance
int motor_control_value_new = 0;
float DC_desired_speed_rpm = 0;
float DC_actual_speed_rpm = 0;
float DC_actual_pos_revs = 0;
int   DC_motor_command_value = 0;
float DC_speed_error_sum = 0;
unsigned long last_millis = 0;
float last_DC_actual_pos_revs = 0;

long DC_motor_command_pos = 0;
long DC_actual_pos = 0;
long DC_last_pos = 0;
float DC_actual_speed = 0.0;
float DC_last_speed = 0.0;
float dt = 0.0;
float DC_pos_error_sum = 0;
int entry = 0;
long IR_reading = 0;

/* CONFIGURATION PARAMTERS */
unsigned long DC_speed_control_task_rate = 50; //ms // okay 50
int           DCstate = 1; // 0 for speed control, 1 for position(angle)
control,
float DC_kp = 1.25; // okay 1.25, 1.25
float DC_ki = .7; // okay .6, .7
float DC_integral_burn_off = .98; //okay .95, .95

float DC_kp_pos = 3; // okay 1.25, 1.25
float DC_kd_pos = .7; // okay .6, .7
float DC_ki_pos = 1;
float DC_integral_burn_off_pos = .98;

/* FUNCTION PROTOTYPES  */
void Motor_command(float);

void loop() {
  int button_value = digitalRead(button_pin);
  if (button_value - button_value_old > 0) {
    button_state++;
  }
  button_value_old = button_value;
  IR_reading = .96*IR_reading + .04*get_IR_reading();

  if (Serial.available() > 0) {
    integerValue = 0;
    delay(10);
    while(1) {
      incomingByte = Serial.read();
      if (incomingByte == '\n') {
        control_value_pct = integerValue;
        break;
      }
      if (incomingByte == -1) break;
      if (incomingByte == ','){
        control_mode = integerValue;
        integerValue = 0;
      }else{
      integerValue *= 10;  // shift left 1 decimal place
      // convert ASCII to integer, add, and shift left 1 decimal place
      integerValue = ((incomingByte - 48) + integerValue);
      }
    }
  }
  if (control_mode == 1) {
    motor_control_value = control_value_pct-100;
  } else {
    switch (button_state%4) {
      case 0: //DC POSITION CONTROL
        motor_control_value = map(analogRead(potpin),0,1023,-100,+100);

      break;
      case 1: //DC SPEED CONTROL
        motor_control_value = map(analogRead(A0),0,1023,-100,+100);


      break;
      case 2: //SERVO CONTROL
          digitalWrite(trigPin, LOW);
        delayMicroseconds(2);

        digitalWrite(trigPin, HIGH);
        delayMicroseconds(10);

        digitalWrite(trigPin, LOW);
        duration = pulseIn(echoPin, HIGH);

        //Calculate the distance (in cm) based on the speed of sound.
        distance = duration/58.2;
        motor_control_value_new = map(distance,0,100,-100,+100);
        motor_control_value = motor_control_value_new*.1 +
motor_control_value*.9;
      break;
      case 3: //STEPPER CONTROL
         motor_control_value = map(get_IR_reading(),0,800,-100,+100);
      break;
    }
  }

  switch (button_state%4) {
    case 0: //DC POSITION CONTROL
      // Go To Position
      if(entry || control_mode_old!=control_mode){
        myEnc.write(0);
        entry = 0;
      }
      control_mode_old = control_mode;
      DC_actual_pos = myEnc.read();
      dt = millis() - last_millis;
      // Run loop every 30ms
      if(dt > 30){
          last_millis = millis();
          DC_motor_command_pos = map(motor_control_value,-100,100,0,1023);
//Dropped off the multiplication by 4 factor.
          // PID CONTROLLER
          DC_motor_command_value = DC_kp_pos * (DC_motor_command_pos -
DC_actual_pos);
          DC_actual_speed = (DC_actual_pos - DC_last_pos) / dt;
          DC_motor_command_value = DC_motor_command_value + DC_kd_pos *
(DC_last_speed - DC_actual_speed);
          DC_pos_error_sum = (DC_pos_error_sum + (DC_motor_command_pos -
DC_actual_pos)) * DC_integral_burn_off_pos;
          DC_motor_command_value = DC_motor_command_value +
DC_ki_pos*DC_speed_error_sum;

          // Save Last States
          DC_last_speed = DC_actual_speed;
          DC_last_pos = DC_actual_pos;

          // Command Motor
          DC_Motor_command( DC_motor_command_value);
          int DC_actual_pos_sent_value = map(DC_actual_pos, 0, 1023, 540, 0);
          serial_op(button_state%4,DC_motor_command_pos,
DC_actual_pos_sent_value);
      }
      break;
    case 1: //DC SPEED CONTROL
      // SPEED CONTROL
      // speeds estimates not very accurate over small periods,
      // only update estimate and speed controller periodically
      entry = 1;
      if(millis() - last_millis > DC_speed_control_task_rate){

        // Update position estimate of DC Motor
        DC_actual_pos_revs = ((float)myEnc.read())/180.0/4.0; // 180
counts/rev * 4 for quadature encoder

        // Update speed estimates of DC Motor
        DC_actual_speed_rpm = ( (DC_actual_pos_revs -
last_DC_actual_pos_revs) / ((float)millis() - last_millis) ) * 1000.0 * 60.0;
        // Reference: Speed at 255 ~= 60 rpm     Speed at -255 ~= -60 rpm

        // Get desired speed from sensor reading
        // DC_motor_command_value = map(analogRead(pin_pot), 0, 1023, -255,
255);
        DC_desired_speed_rpm = map(motor_control_value, -100, 100, -80, 80);
        dc_speed_val = DC_actual_speed_rpm;
        if(abs(DC_desired_speed_rpm) < 18){
          DC_desired_speed_rpm = 0;
        }

        // update speed command from estimated error
        float error = DC_desired_speed_rpm - DC_actual_speed_rpm;
        DC_speed_error_sum = DC_speed_error_sum*DC_integral_burn_off + error;
        DC_motor_command_value = DC_kp*error + DC_ki*DC_speed_error_sum;

        // Update motor command and difference variables
        DC_Motor_command( DC_motor_command_value);
        last_DC_actual_pos_revs = DC_actual_pos_revs;
        last_millis = millis();
        sensor_value = map(dc_speed_val, -80, 80, 1023, 0);
        serial_op(button_state%4,sensor_value, dc_speed_val);
      }
      break;
    case 2: //SERVO CONTROL
      DC_Motor_command(0); // Don't forget to set motor command to 0!!
      val = map(motor_control_value, -100, 100, 0, 180);     // scale it to
use it with the servo (value between 0 and 180)
      myservo.write(val);                  // sets the servo position
according to the scaled value
      servo_val = val;
      sensor_value = map(val, 0, 180, 0, 100);
      serial_op(button_state%4,sensor_value, servo_val);
      delay(15);
      break;
    case 3: //STEPPER CONTROL
      DC_Motor_command(0); // Don't forget to set motor command to 0!!

      int val_old = val;
      val = analogRead(potpin);            // reads the value of the
potentiometer (value between 0 and 1023)
      val = map(motor_control_value, -100, 100, 0, 200);     // scale it to
use it with the stepper (value between 0 and 200, stepper rotates by 1.8
degree per step)
      stepper_val = val;

      if (val>val_old) {
        digitalWrite(4,HIGH); // Set Dir high
      } else {
        digitalWrite(4,LOW); // Set Dir low
      }
      // Number of steps to rotate is based on the change in the pot value
      int steps;
      if (val > val_old)
        steps = val-val_old;
      else
        steps = val_old-val;

      for(int x = 0; x < steps; x++) // Loop step times
      {
        digitalWrite(5,HIGH); // Output high
        delayMicroseconds(1000); // Wait
        digitalWrite(5,LOW); // Output low
        delayMicroseconds(1000); // Wait
      }
      delay(10); // pause one second
      sensor_value = map(val, 0, 200, 0, 1023);
      serial_op(button_state%4,sensor_value, stepper_val);
      break;

  }
  delay(10);//ms
}

 /* FUNCTION DEFINITIONS */

void DC_Motor_command(float cmd){
  if(cmd > 0){
      digitalWrite(pin_DC_H1, LOW);
      digitalWrite(pin_DC_H2, HIGH);
  }else if( cmd < 0){
      digitalWrite(pin_DC_H1, HIGH);
      digitalWrite(pin_DC_H2, LOW);
  }else{
      digitalWrite(pin_DC_H1,LOW);
      digitalWrite(pin_DC_H2,LOW);
  }

  if(abs(cmd) > 255){
    analogWrite(pin_DC_mod, 255 );
  }else{
    analogWrite(pin_DC_mod, abs(cmd) );
  }
}

// Read IR Sensor
long get_IR_reading() {
  long sensorReading = analogRead(IR_pin);
  // Cut off noise at low values
  if (sensorReading < 100) sensorReading = 0;
  return sensorReading;
}

void serial_op(int button_state, int sensor_val, int motor_val) {
  String str_out;
  str_out= String(button_state%4);
  str_out+=",";
  str_out+=String(sensor_val);
  str_out+=",";
  str_out+=String(motor_val);
  str_out+="\n";
  Serial.print(str_out);
}
```

Python code for GUI controls:
Note: ‘state_control_gui_form.ui’ is the user interface file created from Qt Developer also depicted in Figure 1.

```
#!/usr/bin/python
import sys, time, threading, random, Queue
from PyQt4 import QtGui, QtCore as qt
import serial
from PyQt4 import QtGui, QtCore, uic

state_list = ['DC POSITION','DC SPEED','SERVO','STEPPER']
motor_list = ['DC motor position(Deg)', 'DC motor speed (rpm)','Servo (Deg)',
'Stepper (Steps)']
sensor_list= ['Potentiometer (Volts)', 'Pressure (%)', 'Sonar (cm)', 'IR
(cm)']
control_mode_global = 0

SERIALPORT = '/dev/ttyACM0'
BAUD = 115200

class GuiPart(QtGui.QMainWindow):
    def __init__(self, queue, endcommand, *args):
        QtGui.QMainWindow.__init__(self, *args)
        self.setWindowTitle('Arduino Serial Motor Controller')
        self.queue = queue

        self.ui = uic.loadUi('state_control_gui_form.ui', self)

        self.connect(self.control_mode,
QtCore.SIGNAL("currentIndexChanged(int)"), self.control_mode_cb)
        self.connect(self.control_value_slider,
QtCore.SIGNAL("valueChanged(int)"),self.control_value_slider_cb)
        self.connect(self.control_value_text,
QtCore.SIGNAL("valueChanged(int)"), self.control_value_text_cb)
        self.endcommand = endcommand

    def control_mode_cb(self, value):
        #self.ui.actual_servo_value.setText("motor select = " + str(value))
        global control_mode_global
        control_mode_global = value
        ser = serial.Serial(SERIALPORT, BAUD)
        ser.write(str(control_mode_global)+",0\n")
        ser.close()
        self.control_value_text.setValue(0)
        self.control_value_slider.setValue(0)

    def control_value_text_cb(self, value):
        self.control_value_slider.setValue(value)
        global control_mode_global
        ser = serial.Serial(SERIALPORT, BAUD)
        ser.write(str(control_mode_global)+"," +str(value+100) +"\n")
        ser.close()

    def control_value_slider_cb(self, value):
        self.control_value_text.setValue(value)
        global control_mode_global
        ser = serial.Serial(SERIALPORT, BAUD)
        ser.write(str(control_mode_global)+"," +str(value+100) +"\n")
        ser.close()

    def closeEvent(self, ev):
        self.endcommand()

    def processIncoming(self):
        """
        Handle all the messages currently in the queue (if any).
        """
        #while self.queue.qsize(): #seems to cause problems not sure why...
        global control_mode_global
        try:
            msg = self.queue.get(0)
                # Check contents of message and do what it says
                # As a test, we simply print it
                #if str(msg)[0] == '1':
            msg_str = str(msg)
            [state, sensor_value, motor_value] = msg_str.split(",")
            self.motor_select_text.setText(state_list[int(state)])

            self.motor_label.setText(motor_list[int(state)])
            self.actual_motor_value.setText(motor_value)
            if (control_mode_global):
                self.sensor_label.setText("GUI Control Value")
                self.actual_sensor_value.setText((sensor_value))
            else:
                self.sensor_label.setText(sensor_list[int(state)])

                my_value = int(sensor_value)
                if(int(state)==0):
                    my_input = 5.0*my_value/1023.0
                    my_flt = float("{0:.2f}".format(my_input))
                    self.actual_sensor_value.setText(str(my_flt))
                elif(int(state)==1):
                    my_input = 100.0/1023.0*my_value
                    my_flt = float("{0:.2f}".format(my_input))
                    self.actual_sensor_value.setText(str(my_flt))
                elif(int(state)==2):
                    self.actual_sensor_value.setText(sensor_value)
                else:
                    if(my_value == 0):
                        my_input = 0
                    else:
                        my_input = 15327.0*my_value**-1.195
                    my_flt = float("{0:.2f}".format(my_input))
                    self.actual_sensor_value.setText(str(my_flt))




        except Queue.Empty:
            pass
            #self.actual_dcspeed_value.setText("incoming: not found")
```

```
class ThreadedClient:
    """
    Launch the main part of the GUI and the worker thread. periodicCall and
    endApplication could reside in the GUI part, but putting them here
    means that you have all the thread controls in a single place.
    """
    def __init__(self):
        # Create the queue
        self.queue = Queue.Queue(0)

        # Set up the GUI part
        self.gui=GuiPart(self.queue, self.endApplication)
        self.gui.show()

        # A timer to periodically call periodicCall :-)
        self.timer = qt.QTimer()
        qt.QObject.connect(self.timer,
                           qt.SIGNAL("timeout()"),
                           self.periodicCall)
        # Start the timer -- this replaces the initial call to periodicCall
        self.timer.start(25)

        # Set up the thread to do asynchronous I/O
        # More can be made if necessary
        self.running = 1
        self.thread1 = threading.Thread(target=self.workerThread1)
        self.thread1.start()

    def periodicCall(self):
        """
        Check every 100 ms if there is something new in the queue.
        """
        self.gui.processIncoming()
        if not self.running:
            root.quit()

    def endApplication(self):
        self.running = 0

    def workerThread1(self):
        """
        This is where we handle the asynchronous I/O.
        Put your stuff here.
        """
        ser = serial.Serial(SERIALPORT, BAUD)
        while self.running:

            msg = ser.readline();
            if (msg):
                self.queue.put(msg)
            else:
                pass
        ser.close()

root = QtGui.QApplication(sys.argv)
client = ThreadedClient()
```

```
sys.exit(root.exec_())
```