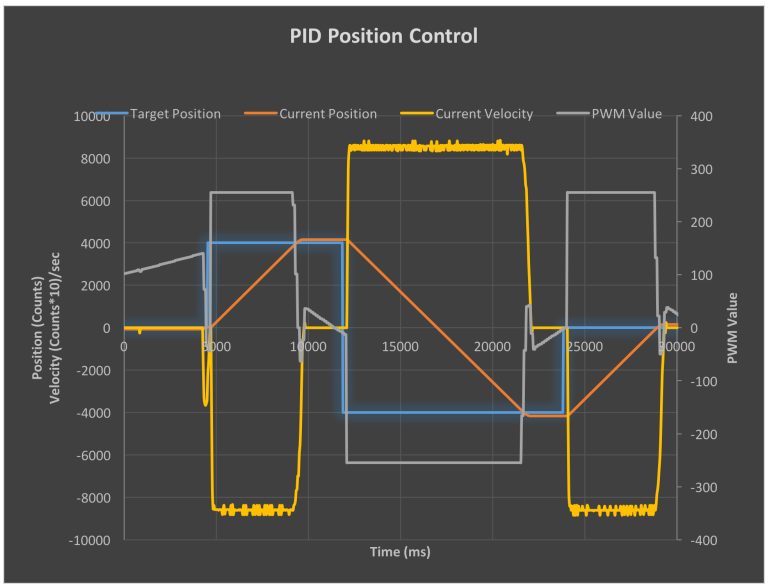
Dan Berman

Team F: ADD_IN

Teammates: Nikhil Baheti, Astha Prasad, Ihsane Debbache



ILR01

10/16/2015

Table of Contents

1. Individual Progress

Overview

For the Sensors and Motors lab, my primary role was development of the Graphical User Interface (GUI). Since the GUI forms the central interface to all parts of the system, it was also fitting that I played a large role in the overall systems integration. In our initial team meeting we determined the overall dataflow for the system; specifically, how each input (sensor, button, GUI setting) controls each output (Motor, GUI display, file logging). From this the following functional requirements were specified for the GUI.

1. Provide graphical control for setting the gain between each sensor/motor combination (proportional control)
2. Provide graphical control for commanding a position and velocity for the DC brushed motor
3. Allow selection and connection to an available COM port
4. Display all sensor readings, motor positions, button status, and DC motor speed in real time
5. Log all state variables to a time stamped *.csv* file for plotting

Implementation

I chose to implement the GUI using the Qt Designer since I wanted to improve my skills with this framework. Previously I had written programs and C++, but never specifically with the Qt framework. I also considered using LabVIEW, which I have significant experience with, but preferred the opportunity to learn a new software, especially one that can operate well across different OS platforms.

The first step to implementing the system was to identify the system's state variables that need to be controlled and/or displayed. Based on the above requirements, I identified following system state variables shown in Table 1.

System State Variables								
Stepper			**Servo**			**Brushed DC**		
Variable	**Units**	**Data Type**	**Variable**	**Units**	**Data Type**	**Variable**	**Units**	**Data Type**
Gain	-	uint_16	Gain	-	uint_16	Gain	-	uint_16
Current Position	Steps	uint_16	Current Position	Degree	uint_16	Current Position	Counts	int_32
						Current Velocity	(Counts*100)/s	int_32
Potentiometer			**Distance**			**Force Sensor**		
Variable	**Units**	**Data Type**	**Variable**	**Units**	**Data Type**	**Variable**	**Units**	**Data Type**
ADC Reading	-	uint_16	Distance	cm	uint_16	ADC Reading	-	uint_16
Push Button								
Variable	**Units**	**Data Type**						
State	-	bool						

Table 1: System State Variables

With the state variables identified, a communication protocol for the GUI to control the system was needed. After discussing with Ihsane, who was responsible for writing the firmware side of the communication interface, we decided upon a simple protocol whereby each instruction was represented by a command letter, an ASCII represented numeric value, and a newline.

For example, the following command sets the servo motor gain to 500

A500\n

So that we could work in parallel, we developed the following header file which defines all commands and is shared between both the GUI code and firmware.

```
 //Commands (Computer -> RAMBO)
#define SET_SERVO_GAIN 'A'
#define SET_STEPPER_GAIN 'B'
#define HOME_STEPPER 'C'
#define SET_DC_CONTROL_SENSOR 'D'
#define SET_DC_GAIN 'E'
#define SET_DC_PID_POS 'F'
#define SET_DC_VEL 'G'
#define SET_DC_POS 'H'
#define ENABLE_STEPPER 'I'
#define ENABLE_SERVO 'J'
#define ENABLE_DC 'K'
#define STOP_ALL 'X'
```

```
//Responses  (RAMBO -> Computer)
#define RESP_STEPPER_POSITION 'a'
#define RESP_SERVO_POSITION 'b'
#define RESP_STEPPER_SENSOR 'c'
#define RESP_SERVO_SENSOR 'd'
#define RESP_DC_SENSOR 'e'
#define RESP_BUTTON_STATUS 'f'
#define RESP_SERVO_GAIN 'g'
#define RESP_STEPPER_GAIN 'h'
#define RESP_DC_POS 'i'
#define RESP_DC_VEL 'j'
#define RESP_DC_GAIN 'k'
#define RESP_DC_CONTROL_SENSOR 'l'
#define RESP_DC_PID_POS 'm'
#define RESP_MILLIS 'n'
#define RESP_DEBUG 'o'
#define RESP_PWM 'p'
```

This header file was modified slightly over the course of the project implementation, most notably the “RESP_DEBUG” command was added which allowed the firmware to send formatted strings to the Qt debug terminal, thus allowing firmware debugging while simultaneously running the GUI.

Based on the state variables and the GUI functional requirements, I then used Qt Designer to layout the following user interface shown in Figure 1.

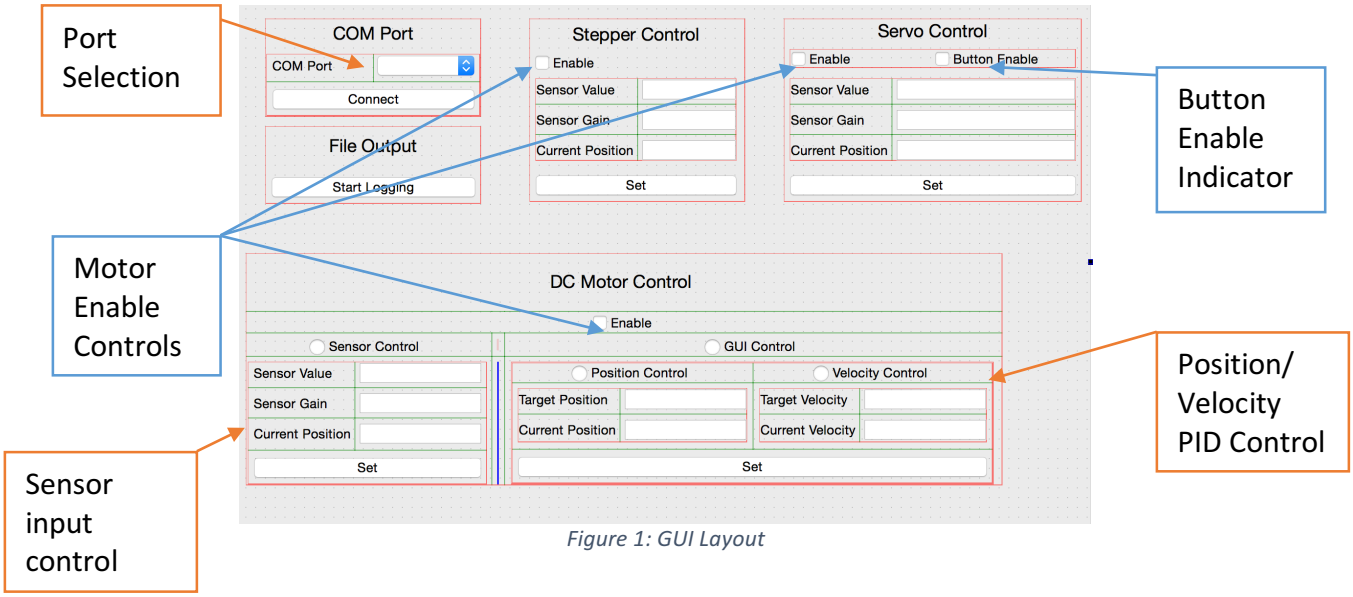


Figure 1: GUI Layout

Qt designer creates an XML file describing the layout of the GUI, but all of the functional implementation (including all slots/callbacks) was still implemented manually in C++. I wanted the user interface to be robust, so all inputs and controls have input validators and error handling. The GUI also provides an enable check box for each motor. In the case of the DC and stepper motors, the enable line of the motor driver is deactivated when the box is deselected. For the servo motor, the input from the enable check box is ANDed with the status of a physical button, and the motor is only controlled when both are selected. The status of the physical button is shown in the 'button enable' check box, which is only an indicator and cannot be modified by the user. Ultimately, the manually written portion of the GUI code, including all functions for USB communication and *.csv* file output, resulted in over 650 lines of code.

In my view the project's results were a complete success. All system components are robustly operating. Graphs for the inputs and outputs of the position and velocity PIDs were produced using the *.csv* logged data files and are shown in Figure 2 and Figure 3. The data for every state variable is logged to the computer and time stamped at approximately 50ms intervals. The results clearly show that the PID is actively controlling the motor's position/velocity.

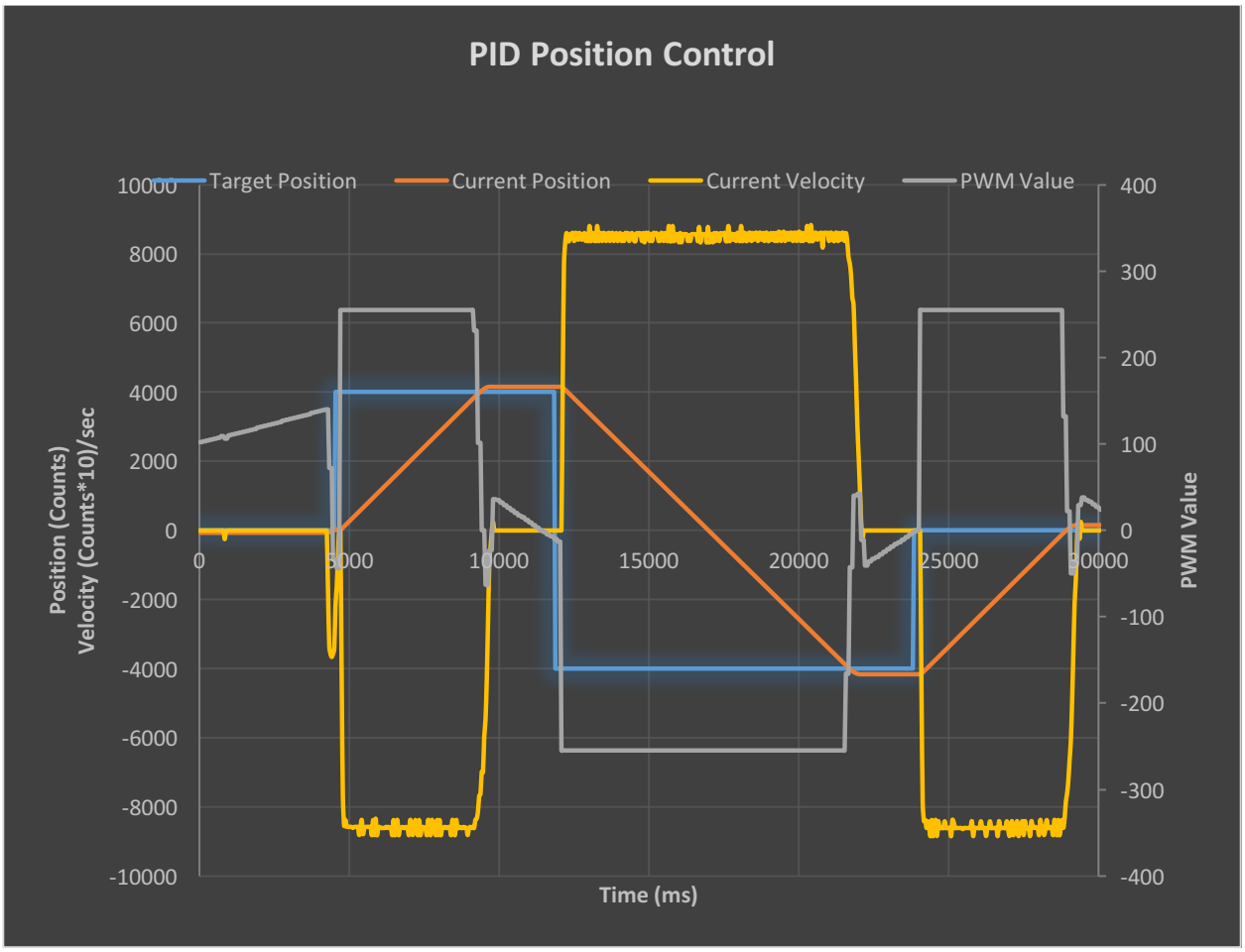


Figure 2: PID Position control of brushed DC motor

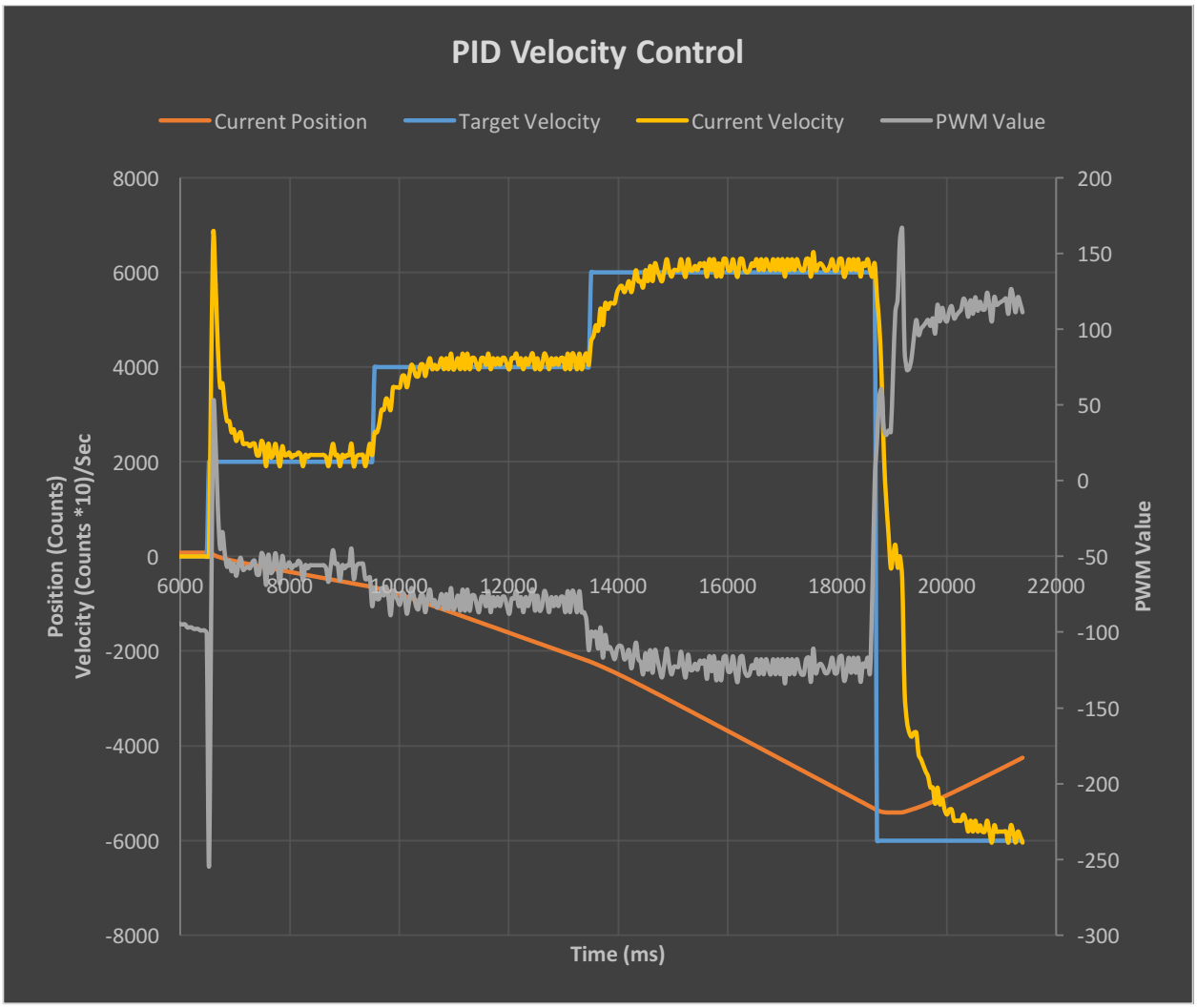


Figure 3: PID velocity control of brushed DC motor

6. Challenges

The primary challenge with this project was 'getting up the learning curve' of Qt. Although experienced in C++ programming, becoming familiar with the Qt specific classes, methodologies, and design tools was time consuming. Originally I began to naively design the GUI entirely from code, which, although educational, was much more time consuming than using Qt Designer, Qt's drag-and-drop layout editor.

Unlike other GUI creation programs I have used (i.e. MATLAB's GUIde) which use callback functions, Qt uses a 'signals and slots' framework whereby 'signal' generating GUI elements are 'connected' to 'slot' functions. Although flexible (signals can easily be reassigned to different slots), the signals and slots framework can at times make passing function parameters more difficult and thus necessitated the use of more class variables.

Finally, the Qt framework overloads many traditional C++ classes with their own 'Q' versions of the class (i.e. 'QString'), which although powerful, requires adapting to the change in nomenclature and understanding the available functions. However, after becoming familiar with the framework, I

found that it significantly extends the capabilities of many standard C/C++ libraries and would definitely use it again.

7. Teamwork

This project necessitated a well planned distribution of work and integration phase. After a team meeting to discuss the project, we broke the work down as follows:

Astha Prasad– Develop hardware and firmware for reading the distance sensor and controlling the firmware.
Ihsane Debbache– Develop the hardware and firmware for reading the potentiometer and controlling the stepper. Develop the firmware to send, read, and parse serial data.
Nikhil Baheti– Develop the hardware and firmware for reading the force sensor, and controlling the brushed DC motor using either speed or velocity PID.
Dan Berman – GUI development and systems integration.

The execution of this plan taught us many practical lessons. Specifically, not enough time was given for integration, and not all systems had been functionally tested before integration. Despite this, the team effectively worked together and got all systems fully working and implemented by the project deadline. A major difficulty was the team's lack of prior programming experience, so common interface standards (such as function prototypes) were not developed until late in the project. The most valuable aspect, was that after the ILR presentation the team met and discussed what went well and what didn't, and devised strategies to facilitate project execution in the future.

8. Future Plans

From now until the next ILR my primary focus will be understanding and implementing the necessary modifications to the 3D printer slicer software to allow the user to select the correct 'insertion layer' at which a COTS part to be encapsulated by 3D print material wil be placed. Recent work has focused on understanding a specific slicer software, 'slic3r', and shown that there is a built in function for separating *.stl* part files at a specific layer. Current plans are to use this feature, in combination with additional post processing, to select the insertion layer, command the printer the move to it's 'insertion configuration', and afterwards return to printing. Current obstacles to overcome are designing the user interface (not necessarily graphical!) to select this insertion layer, and then actually invoking the functions of the 'slic3r' software. The 'slic3r' software is written predominately in Perl which none of the teammates have experience with, so more research is needed to understand the basics of this programming language, the exact architecture of the 'slic3r' software, and for determination of the file location and programming language which should be used for our modifications.

Although my primary work will be with Astha on the slicer portion of the project, I will also be continuing to assist Nikhil and Ihsane with the mechanical design portion of the project. I expect that in the coming two weeks, as they design the first iteration of the nozzle design, they will rapidly become more familiar with mechanical design, machining, and CAD, thus enabling me to focus more of my time on software development.

Appendix 1: GUI Source Code

1. GUMainWindow.h

```
#ifndef GUIMAINWINDOW_H
#define GUIMAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class GUIMainWindow;
}

class GUIMainWindow : public QMainWindow
{
Q_OBJECT
public:
explicit GUIMainWindow(QWidget *parent = 0);
~GUIMainWindow();
private slots:

//***********INITIALIZATION****************//
void init();
void initStateVariables();
void enableAll(bool);
void slotConnectDisconnectSerial();


//***********COM PORT****************//
void initSerial();
void slotUpdatePortInfo(QString);
void slotReadSerial();
void sendSerial(char,long);

//***********FILE LOGGING****************//
bool createOutputFile();
void slotEnableFileLogging();
void writeFilesv(uint32_t);


//***********STEPPER CONTROL****************//
void slotEnableStepper(int);
void slotSetStepperState();

void initStepperState();
void updateStepperState();

//***********SERVO CONTROL****************//
void slotEnableServo(int);
void slotSetServoState();
void servoButtonEnable(bool);
void initServoState();
void updateServoState();


//***********DC MOTOR CONTROL****************//
//Slots
void slotEnableDC(int);
void slotGUIControlMode();
void slotSensorControlMode();
void slotPIDPositionMode();
void slotPIDVelocityMode();

void slotSetDCStateSensor(); //sets fields for Sensor-Mode DC Control
void slotSetDCStateGUI(); //sets fields for GUI Control

//Functions
void initDCState();

void updateDCState();
void updateDCStateSensor(); //Updates fields for Sensor-Mode DC Control
void updateDCStateGUI(); //Updates fields for GUI Control
void updateDCStatePosition(); //Updates fields for Position PID
```

```
void updateDCStateVelocity(); //Updates fields for Velocity PID


void setDCStatePosition(); //sets fields for Position PID
void setDCStateVelocity(); //sets fields for Velocity PID

void enableDCControlGUI(bool);
void enableDCControlSensor(bool);

void enableDCPIDPosition(bool);
void enableDCPIDVelocity(bool);

void enableDCControlRadios(bool);
void enablePIDControlRadios(bool);


private:
Ui::GUIMainWindow *ui;

};


#endif // GUIMAINWINDOW_H
```

2. GUIMainWindow.c

```
#include "guimainwindow.h"
#include "protocoldefinitions.h"
#include "ui_guimainwindow.h"
#include <QtSerialPort/QSerialPort>
#include <QtSerialPort/QSerialPortInfo>
#include <QDebug>
#include <QPalette>
#include <QValidator>
#include <QMessageBox>
#include <QFile>
#include <QFileDialog>


//DEFINES
#define SERIAL_BAUD QSerialPort::Baud19200
#define SERIAL_DATA_BITS QSerialPort::Data8
#define SERIAL_PARITY QSerialPort::NoParity
#define SERIAL_FLOW_CONTROL QSerialPort::NoFlowControl
#define SERIAL_STOP_BITS QSerialPort::OneStop
#define SERIAL_MAX_READ_LENGTH 1000

#define STEPPER_MIN_GAIN 0
#define STEPPER_MAX_GAIN 65535

#define SERVO_MIN_GAIN 0
#define SERVO_MAX_GAIN 65535

#define DC_MIN_GAIN 0
#define DC_MAX_GAIN 65535

#define DC_MIN_POSITION -65535
#define DC_MAX_POSITION 65535
#define DC_MIN_VELOCITY -65535
#define DC_MAX_VELOCITY 65535


//Palettes
QPalette disabledLineEditPalette;
QPalette enabledLineEditPalette;

//State Variables
struct stateVariables{
//Control Variables
    //Stepper
    uint16_t stepperGain;
    int32_t stepperCurrentPos;
    uint16_t stepperSensorVal;
    bool stepperEnabled;

    //Servo
    uint16_t servoGain;
    uint16_t servoCurrentPos;
    uint16_t servoSensorVal;
    bool servoEnabled;

    //DC Motor
    uint16_t dcGain;
    uint16_t dcSensorVal;
    int32_t dcCurrentPos;
    int32_t dcCurrentVelocity;
    int32_t dcTargetPos;
    int32_t dcTargetVelocity;
    int16_t dcPWM;
    bool dcSensorControlMode; //true = sensor control mode, false = GUI control mode
    bool dcPositionPIDMode; //true = position PID control, false = velocity PID control
    bool dcEnabled; //true = motor enabled, false = motor disabled

    bool buttonStatus;
} sv;

//File Logging
bool logging;
```

```
QString filename;

//Communications
QSerialPort *serial;
QList<QSerialPortInfo> COMPorts;
QByteArray serialData;


GUIMainWindow::GUIMainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::GUIMainWindow)
{
    ui->setupUi(this);
    setWindowTitle(tr("Team F Motor Interface"));
    serial = new QSerialPort(this);
    serialData = QByteArray();
    slotUpdatePortInfo(QString(""));

    //***********GLOBAL INITIALIZATION****************//
    disabledLineEditPalette.setColor(QPalette::Base, QColor(190,190,190)); //Grey
    enabledLineEditPalette.setColor(QPalette::Base, QColor(255,255,255)); //White

    //***********GUI INITIALIZATION****************//
    init();
    //***********FILE OUTPUT***************//
    connect(ui->LogFileButton,SIGNAL(clicked(bool)),this,SLOT(slotEnableFileLogging()));
    //***********COM PORT CONNECTS***************//
    connect(ui->COMComboBox,SIGNAL(activated(QString)),this,SLOT(slotUpdatePortInfo(QString)));
    connect(ui->COMConnectButton,SIGNAL(clicked(bool)),this,SLOT(slotConnectDisconnectSerial()));
    connect(serial,SIGNAL(readyRead()),this,SLOT(slotReadSerial()));
    //***********STEPPER CONNECTS***************//
    connect(ui->StepperEnableCheck,SIGNAL(stateChanged(int)),this,SLOT(slotEnableStepper(int)));
    connect(ui->StepperSetButton,SIGNAL(clicked(bool)),this,SLOT(slotSetStepperState()));
    //***********SERVO CONNECTS***************//
    connect(ui->ServoEnableCheck,SIGNAL(stateChanged(int)),this,SLOT(slotEnableServo(int)));
    connect(ui->ServoSetButton,SIGNAL(clicked(bool)),this,SLOT(slotSetServoState()));
    //***********DC MOTOR CONNECTS***************//
    //Enable
    connect(ui->DCEnableCheck,SIGNAL(stateChanged(int)),this,SLOT(slotEnableDC(int)));
    //Radio Buttons
    connect(ui->DCSensorControlRadio,SIGNAL(pressed()),this,SLOT(slotSensorControlMode()));
    connect(ui->DCGUIControlRadio,SIGNAL(pressed()),this,SLOT(slotGUIControlMode()));
    connect(ui->DCPositionControlRadio,SIGNAL(pressed()),this,SLOT(slotPIDPositionMode()));
    connect(ui->DCVelocityControlRadio,SIGNAL(pressed()),this,SLOT(slotPIDVelocityMode()));
    //Set Buttons
    connect(ui->DCSetGUIButton,SIGNAL(clicked(bool)),this,SLOT(slotSetDCStateGUI()));
    connect(ui->DCSetSensorButton,SIGNAL(clicked(bool)),this,SLOT(slotSetDCStateSensor()));
}


//***********INITIALIZATION****************//
void GUIMainWindow::init()
{
    initStateVariables();
    initDCState();
    initServoState();
    initStepperState();
    initSerial();
    enableAll(false);
}

void GUIMainWindow::initStateVariables(){
    sv.stepperEnabled = false;
    sv.servoEnabled = false;
    sv.dcEnabled = false;
    sv.dcPositionPIDMode = true;    //Position PID
    sv.dcSensorControlMode = true; //Sensor Control Mode

    sv.servoEnabled = 0;
    sv.stepperEnabled = 0;
    sv.dcEnabled = 0;

    sv.servoGain = 5;
    sv.stepperGain = 5;
    sv.dcGain = 5;
```

```
    sv.dcGain = 10;
    sv.dcTargetPos = 0;
    sv.dcTargetVelocity = 0;
}

//***********COM PORT****************//
void GUIMainWindow::slotUpdatePortInfo(QString text)
{
    int cmp = QString::compare(text,"Refresh",Qt::CaseInsensitive);
    if(cmp == 0){
        ui->COMComboBox->clear();
        QStringList list;
        COMPorts = QSerialPortInfo::availablePorts();
        Q_FOREACH(QSerialPortInfo port, COMPorts) {
                ui->COMComboBox->addItem(port.portName());
            }
        ui->COMComboBox->insertSeparator(ui->COMComboBox->maxCount() - 1);
        ui->COMComboBox->addItem(QString("Refresh"));
        ui->COMConnectButton->setDisabled(false); //enable connect button
    }
}
void GUIMainWindow::initSerial(){
    ui->COMConnectButton->setDisabled(true);
    ui->COMComboBox->addItem(QString("Refresh"));
    ui->COMComboBox->insertSeparator(0);
}
void GUIMainWindow::slotConnectDisconnectSerial(){
    if (serial->isOpen()) //If serial is open
    {
        serial->close();
        ui->COMConnectButton->setText("Connect");
        enableAll(false);
    }
    else
    {
        serial->setPort(COMPorts.value((ui->COMComboBox->currentIndex())));
        serial->setBaudRate(SERIAL_BAUD);
        serial->setDataBits(SERIAL_DATA_BITS);
        serial->setParity(SERIAL_PARITY);
        serial->setStopBits(SERIAL_STOP_BITS);
        serial->setFlowControl(SERIAL_FLOW_CONTROL);
        if (serial->open(QIODevice::ReadWrite)) {
            ui->COMConnectButton->setText("Disconnect");
            enableAll(true);
        } else {
            QMessageBox::critical(this, tr("Error"), serial->errorString());
        }
    }
}
void GUIMainWindow::sendSerial(char command, long value){
    if(serial->isOpen()){
        QByteArray data = QByteArray(QString(command).toLocal8Bit());
        QString valString = QString::number(value);
        data.append(valString.toLocal8Bit());
        data.append('\n');
        qDebug() << "Send: " << QString(data);
        serial->write(data);
    }
}

//***********FILE OUTPUT****************//
bool GUIMainWindow::createOutputFile(){
    filename = QFileDialog::getSaveFileName(this,tr("Create Logging File"));
    filename.append(".csv");
    QFile f(filename);
    bool success = f.open(QIODevice::WriteOnly);
    if(success){
        logging = true;
        QTextStream fileStream(&f);
        fileStream << "Time" << ','
                           << "Target Position" << ','
                           << "Current Position" << ','
                           << "Target Velocity" << ','
                           << "Current Velocity" << ','
                           << "PWM Value" << ','
```

```
                            << "Stepper Gain" << ','
                            << "Stepper Pos" << ','
                            << "Pot" << ','
                            << "Stepper Enabled" << ','
                            << "Servo Gain" << ','
                            << "Servo Pos" << ','
                            << "Distance" << ','
                            << "Servo Enabled" << ','
                            << "DC Gain" << ','
                            << "Force Sensor" << ','
                            << "DC Sens Control Mode" << ','
                            << "DC Pos PID Mode" << ','
                            << "DC Enabled" << ','
                            << "Button Status" << '\n';


        f.close();
    }
    return success;
}

void GUIMainWindow::slotEnableFileLogging(){
    if(logging){ //Logging enabled, stop logging
        ui->LogFileButton->setText("Start Logging");
        logging = false;
    }
    else{ //Logging stopped, start logging
        createOutputFile();
        ui->LogFileButton->setText("Stop Logging");
    }
}

void GUIMainWindow::writeFilesv(uint32_t millis){
    QFile f(filename);
    bool success = f.open(QIODevice::WriteOnly | QIODevice::Append );
    if(success){
        QTextStream fileStream(&f);
        fileStream     << millis << ','
                       << sv.dcTargetPos << ','
                       << sv.dcCurrentPos << ','
                       << sv.dcTargetPos << ','
                       << sv.dcCurrentVelocity << ','
                       << sv.dcPWM << ','
                       << sv.stepperGain << ','
                       << sv.stepperCurrentPos << ','
                       << sv.stepperSensorVal << ','
                       << sv.stepperEnabled << ','
                       << sv.servoGain << ','
                       << sv.servoCurrentPos << ','
                       << sv.servoSensorVal << ','
                       << sv.servoEnabled << ','
                       << sv.dcGain << ','
                       << sv.dcSensorVal << ','
                       << sv.dcSensorControlMode << ','
                       << sv.dcPositionPIDMode << ','
                       << sv.dcEnabled << ','
                       << sv.buttonStatus << '\n';

        f.close();
    }
}

void GUIMainWindow::slotReadSerial(){
        char command, byte;
        int num;
        uint32_t millis;
        QString value;
        serialData+=serial->readAll();
        if(serialData.indexOf('\n') == -1) //if no newline
        {
            return;
        }
        if(serialData.length() > SERIAL_MAX_READ_LENGTH)
        {
            //qDebug() << "serialData " << serialData;
            serialData.clear();
```

```
            qDebug() << "Serial Read Buffer Cleared";
        }
 //       qDebug() << "d: " << QString(serialData);
        //Parse
        while(serialData.indexOf('\n') > -1)
        {
//            qDebug() << "serialData " << serialData.data();
            command = serialData[0]; //get command
            serialData.remove(0,1);
            if(command < 'a' || command > 'p')
                continue;
            byte = serialData[0]; //get value
            bool failed = false;
            do{
                if(byte < '0' && byte > '9')
                {
                    failed = true;
                    break;
                }
                value+=(QString(byte));
                serialData.remove(0,1); //get next value
                byte = serialData[0];
            }while(byte != '\r' && serialData.length() > 0);
            if(failed)
                break;
            serialData.remove(0,1); //remove \r
 //           qDebug() << "Command: " << command << " Value: " << value << '\n';
            switch(command){
                case RESP_STEPPER_POSITION:
                    num = value.toLong();
                    //qDebug() << "Recd: " << QString(value);
                    sv.stepperCurrentPos = num;
                    ui->StepperCurrentPositionLineEdit->setText(QString::number(num));
                    break;
                case RESP_SERVO_POSITION:
                    num = value.toInt();
                    sv.servoCurrentPos = num;
                    ui->ServoCurrentPositionLineEdit->setText(QString::number(num));
                    break;
                case RESP_STEPPER_SENSOR:
                    num = value.toInt();
                    sv.stepperSensorVal = num;
                    ui->StepperSensorLineEdit->setText(QString::number(num));
                    break;
                case RESP_SERVO_SENSOR:
                    num = value.toInt();
                    sv.servoSensorVal = num;
                    ui->ServoSensorLineEdit->setText(QString::number(num));
                    break;
                case RESP_DC_SENSOR:
                    num = value.toInt();
                    sv.dcSensorVal = num;
                    ui->DCSensorLineEdit->setText(QString::number(num));
                    break;
                case RESP_BUTTON_STATUS:
                    num = value.toInt();
                    servoButtonEnable((bool)num);
                    break;
                case RESP_SERVO_GAIN:
                    //num = value.toInt();
                    //sv.servoGain = num;
                    break;
                case RESP_STEPPER_GAIN:
                    //num = value.toInt();
                    //sv.stepperGain = num;
                    //ui->StepperGainLineEdit->setText(QString::number(num));
                    break;
                case RESP_DC_POS:
                    num = value.toLong();
                    sv.dcCurrentPos = num;
                    ui->DCCurrentPositionGUILineEdit->setText(QString::number(num));
                    ui->DCCurrentPositionSensorLineEdit->setText(QString::number(num));
                    break;
                case RESP_DC_VEL:
                    num = value.toLong();
                    sv.dcCurrentVelocity = num;
```

```
                    ui->DCCurrentVelocityLineEdit->setText(QString::number(num));
                    break;
                case RESP_DC_GAIN:
                    //num = value.toInt();
                    //sv.dcGain = num;
                    //ui->DCGainLineEdit->setText(QString::number(num));
                    break;
                case RESP_DC_CONTROL_SENSOR:
                    num = value.toInt();
                    sv.dcSensorControlMode = num;
                    break;
                case RESP_DC_PID_POS:
                    num = value.toInt();
                    sv.dcPositionPIDMode = num;
                    break;
                case RESP_MILLIS:
                    //Update file only after all parameters have been received
                    millis = value.toLong();
                    if(logging)
                    {
                        writeFilesv(millis);
                    }
                    break;
                case RESP_DEBUG:
                    qDebug() << "Recd: " << value;
                    break;
                case RESP_PWM:
                    num = value.toInt();
                    sv.dcPWM = num;
                    qDebug() << "PWM = " << num;
                    break;
                default:
                    qDebug() << "Invalid Serial Response";
                        break;
            } //switch
            value.clear();
        }
}

//***********ENABLE CONTROLS****************//
void GUIMainWindow::enableAll(bool enable){
    ui->StepperEnableCheck->setDisabled(not enable);
    ui->ServoEnableCheck->setDisabled(not enable);
    ui->DCEnableCheck->setDisabled(not enable);
    if(not enable){ //Disable all controllers
        slotEnableDC(false);
        slotEnableServo(false);
        slotEnableStepper(false);
    }
}

//***********STEPPER CONTROL****************//
void GUIMainWindow::initStepperState(){
    //Line Edit Validators
    QValidator *stepperGainValidator = new QIntValidator(STEPPER_MIN_GAIN,STEPPER_MAX_GAIN,this);
    ui->StepperGainLineEdit->setValidator(stepperGainValidator);
    //Read Only line edits
    ui->StepperCurrentPositionLineEdit->setReadOnly(true);
    ui->StepperSensorLineEdit->setReadOnly(true);

    slotEnableStepper(false);
    updateStepperState();
}
void GUIMainWindow::updateStepperState(){
    QString gainStr = QString::number(sv.stepperGain);
    ui->StepperGainLineEdit->setText(gainStr);
}
void GUIMainWindow::slotSetStepperState(){
    QString gainStr = ui->StepperGainLineEdit->text();
    sv.stepperGain = gainStr.toInt();
    sendSerial(SET_STEPPER_GAIN,sv.stepperGain);
}
void GUIMainWindow::slotEnableStepper(int ienabled){
    QPalette palette;
    bool enabled = (bool)ienabled;
    sv.stepperEnabled = enabled;
```

```
    ui->StepperEnableCheck->setChecked(enabled);
    ui->StepperSensorLineEdit->setEnabled(enabled);
    ui->StepperGainLineEdit->setEnabled(enabled);
    ui->StepperCurrentPositionLineEdit->setEnabled(enabled);
    ui->StepperSetButton->setEnabled((int)enabled);

    if(enabled == 0)
        palette = disabledLineEditPalette;
    else
        palette = enabledLineEditPalette;

    ui->StepperSensorLineEdit->setPalette(palette);
    ui->StepperGainLineEdit->setPalette(palette);
    ui->StepperCurrentPositionLineEdit->setPalette(palette);

    updateStepperState();
    sendSerial(ENABLE_STEPPER,(int)sv.stepperEnabled);
}


//***********SERVO CONTROL****************//
void GUIMainWindow::initServoState(){
    //Line Edit Validators
    QValidator *servoGainValidator = new QIntValidator(SERVO_MIN_GAIN,SERVO_MAX_GAIN,this);
    ui->ServoGainLineEdit->setValidator(servoGainValidator);
    //Read Only line edits
    ui->ServoCurrentPositionLineEdit->setReadOnly(true);
    ui->ServoSensorLineEdit->setReadOnly(true);
    ui->servoButtonEnable->setDisabled(true);
    slotEnableServo(0);
}
void GUIMainWindow::updateServoState(){
    QString gainStr = QString::number(sv.servoGain);
    ui->ServoGainLineEdit->setText(gainStr);
}
void GUIMainWindow::slotSetServoState(){
    QString gainStr = ui->ServoGainLineEdit->text();
    sv.servoGain = gainStr.toInt();
    sv.stepperGain = gainStr.toInt();
    sendSerial(SET_SERVO_GAIN,sv.servoGain);
}
void GUIMainWindow::servoButtonEnable(bool enable){
    ui->servoButtonEnable->setChecked(enable);
    sv.buttonStatus = enable;
    if(sv.servoEnabled && !ui->ServoEnableCheck->isChecked())
        slotEnableServo(true);
}
void GUIMainWindow::slotEnableServo(int ienabled)
{
    QPalette palette;
    ui->ServoEnableCheck->setChecked((bool)ienabled);
    sv.servoEnabled = (bool)ienabled;
    bool enabled = ((bool)ienabled & sv.buttonStatus);
    ui->ServoSensorLineEdit->setEnabled((bool)enabled);
    ui->ServoGainLineEdit->setEnabled((bool)enabled);
    ui->ServoCurrentPositionLineEdit->setEnabled((bool)enabled);
    ui->ServoSetButton->setEnabled(enabled);

    if(enabled == 0)
        palette =  disabledLineEditPalette;
    else
        palette = enabledLineEditPalette;

    ui->ServoSensorLineEdit->setPalette(palette);
    ui->ServoGainLineEdit->setPalette(palette);
    ui->ServoCurrentPositionLineEdit->setPalette(palette);

    updateServoState();
    sendSerial(ENABLE_SERVO,(int)sv.servoEnabled);
}


//***********DC MOTOR CONTROL****************//
void GUIMainWindow::slotEnableDC(int ienabled)
{
    bool enabled = (bool)ienabled;
```

```
    sv.dcEnabled = enabled;
    ui->DCEnableCheck->setChecked(enabled);
    updateDCState();
    if(enabled){ //enable only selected section
        enableDCControlSensor(sv.dcSensorControlMode);
        enableDCControlGUI(not sv.dcSensorControlMode);
    }
   else{ //Disable all sections
        enableDCControlGUI(enabled);
        enableDCControlSensor(enabled);
    }
    enableDCControlRadios(enabled);
    sendSerial(ENABLE_DC,(int)sv.dcEnabled);
}

//Initialization
void GUIMainWindow::initDCState(){
    //Set up radio buttons
    ui->DCGUIControlRadio->setAutoExclusive(false);
    ui->DCSensorControlRadio->setAutoExclusive(false);
    ui->DCPositionControlRadio->setAutoExclusive(false);
    ui->DCVelocityControlRadio->setAutoExclusive(false);

    //Set lineEdit masks
    //ui->DCTargetPositionLineEdit->setMask(numericMasks32);
    //ui->DCTargetVelocityLineEdit->setMask(numericMaskss16);
    //ui->DCGainLineEdit->setMask(numericMasku16);

    QValidator *dcGainValidator = new QIntValidator(DC_MIN_GAIN,DC_MAX_GAIN,this);
    ui->DCGainLineEdit->setValidator(dcGainValidator);
    QValidator *dcTargetPosValidator = new QIntValidator(DC_MIN_POSITION,DC_MAX_POSITION,this);
    ui->DCTargetPositionLineEdit->setValidator(dcTargetPosValidator);
    QValidator *dcTargetVelValidator = new QIntValidator(DC_MIN_VELOCITY,DC_MAX_VELOCITY,this);
    ui->DCTargetVelocityLineEdit->setValidator(dcTargetVelValidator);

    //set read only line edits
    ui->DCCurrentPositionGUILineEdit->setReadOnly(true);
    ui->DCCurrentVelocityLineEdit->setReadOnly(true);
    ui->DCSensorLineEdit->setReadOnly(true);

    updateDCState();
    slotEnableDC(0);
}

//STATE VARIABLE UPDATES - update values shown in GUI from sv

//update all DC motor fields
void GUIMainWindow::updateDCState(){
    updateDCStateSensor();
    updateDCStateGUI();
}

void GUIMainWindow::updateDCStateSensor(){
    QString gainStr = QString::number(sv.dcGain);
    ui->DCGainLineEdit->setText(gainStr);
}
void GUIMainWindow::updateDCStateGUI(){
    updateDCStatePosition();
    updateDCStateVelocity();
}

void GUIMainWindow::updateDCStatePosition(){
    QString posStr = QString::number(sv.dcTargetPos);
    ui->DCTargetPositionLineEdit->setText(posStr);
}
void GUIMainWindow::updateDCStateVelocity(){
    QString velStr = QString::number(sv.dcTargetVelocity);
    ui->DCTargetVelocityLineEdit->setText(velStr);
}

void GUIMainWindow::slotSetDCStateSensor(){
    sv.dcSensorControlMode = true;
    QString gainStr = ui->DCGainLineEdit->text();
    sv.dcGain = gainStr.toInt();
    sendSerial(SET_DC_CONTROL_SENSOR,(int)sv.dcSensorControlMode);
    sendSerial(SET_DC_GAIN,sv.dcGain);
```

```
}
void GUIMainWindow::slotSetDCStateGUI(){
    QString str;
    sv.dcSensorControlMode = false;
    if(ui->DCPositionControlRadio->isChecked()){ //Position Control Mode
        setDCStatePosition();
        sv.dcPositionPIDMode = true;
        str = ui->DCTargetPositionLineEdit->text();
        sv.dcTargetPos = str.toInt();
        sendSerial(SET_DC_POS,sv.dcTargetPos);
    }
    else{
        setDCStateVelocity();
        sv.dcPositionPIDMode = false;
        str = ui->DCTargetVelocityLineEdit->text();
        sv.dcTargetVelocity = str.toInt();
        sendSerial(SET_DC_VEL,sv.dcTargetVelocity);
    }
    sendSerial(SET_DC_CONTROL_SENSOR,(int)sv.dcSensorControlMode);
    sendSerial(SET_DC_PID_POS,(int)sv.dcPositionPIDMode);
}


void GUIMainWindow::slotPIDPositionMode(){
    enablePIDControlRadios(false); //For some reason can only set radio states when buttons are
disabled
    ui->DCPositionControlRadio->setChecked(true);
    ui->DCVelocityControlRadio->setChecked(false);
    enablePIDControlRadios(true);
    enableDCPIDPosition(true);
    enableDCPIDVelocity(false);
}

void GUIMainWindow::slotPIDVelocityMode(){
    enablePIDControlRadios(false); //For some reason can only set radio states when buttons are
disabled
    ui->DCPositionControlRadio->setChecked(false);
    ui->DCVelocityControlRadio->setChecked(true);
    enablePIDControlRadios(true);
    enableDCPIDPosition(false);
    enableDCPIDVelocity(true);
}

void GUIMainWindow::setDCStatePosition(){
    QString posStr = ui->DCTargetPositionLineEdit->text();
    sv.dcTargetPos = posStr.toInt();
}
void GUIMainWindow::setDCStateVelocity(){
    QString velStr = ui->DCTargetVelocityLineEdit->text();
    sv.dcTargetVelocity = velStr.toInt();
}


//Widget Enables
//Enable widgets for GUI control mode
void GUIMainWindow::enableDCControlGUI(bool enabled){
    //GUI Control Visibility
    updateDCStateGUI();
    ui->DCSetGUIButton->setEnabled((bool)enabled);

    //Set Radio Buttons
    enableDCControlRadios(false); //For some reason can only set radio states when buttons are disabled
    ui->DCSensorControlRadio->setChecked(not enabled);
    ui->DCGUIControlRadio->setChecked(enabled);
    enableDCControlRadios(true);

    enablePIDControlRadios(false); //For some reason can only set radio states when buttons are
disabled
    ui->DCPositionControlRadio->setChecked(sv.dcPositionPIDMode);
    ui->DCVelocityControlRadio->setChecked(not sv.dcPositionPIDMode);
    enablePIDControlRadios(enabled);

    if(enabled){
        //Enable only selected PID mode
        enableDCPIDPosition((int)sv.dcPositionPIDMode);
        enableDCPIDVelocity((int)not sv.dcPositionPIDMode);
```

```
    }
    else{
        //Disable Both
        enableDCPIDPosition(0);
        enableDCPIDVelocity(0);
    }
}
void GUIMainWindow::enableDCPIDPosition(bool enabled){
    QPalette palette;
    updateDCStatePosition();
    ui->DCCurrentPositionGUILineEdit->setEnabled((bool)enabled);
    ui->DCTargetPositionLineEdit->setEnabled((bool)enabled);
    if(enabled == 0)
        palette =  disabledLineEditPalette;
    else
        palette = enabledLineEditPalette;

    ui->DCCurrentPositionGUILineEdit->setPalette(palette);
    ui->DCTargetPositionLineEdit->setPalette(palette);
}
void GUIMainWindow::enableDCPIDVelocity(bool enabled){
    QPalette palette;
    updateDCStateVelocity();
    ui->DCCurrentVelocityLineEdit->setEnabled((bool)enabled);
    ui->DCTargetVelocityLineEdit->setEnabled((bool)enabled);
    if(enabled == 0)
        palette =  disabledLineEditPalette;
    else
        palette = enabledLineEditPalette;
    ui->DCCurrentVelocityLineEdit->setPalette(palette);
    ui->DCTargetVelocityLineEdit->setPalette(palette);
}
//Enable widgets for sensor control mode
void GUIMainWindow::enableDCControlSensor(bool enabled){
    QPalette palette;
    updateDCStateSensor();

    ui->DCSensorLineEdit->setEnabled(enabled);
    ui->DCGainLineEdit->setEnabled(enabled);
    ui->DCCurrentPositionSensorLineEdit->setEnabled(enabled);
    ui->DCSetSensorButton->setEnabled(enabled);


    if(enabled)
        palette =  enabledLineEditPalette;
    else
        palette = disabledLineEditPalette;

    ui->DCSensorLineEdit->setPalette(palette);
    ui->DCGainLineEdit->setPalette(palette);
    ui->DCCurrentPositionSensorLineEdit->setPalette(palette);

    //Set Radio Buttons
    enablePIDControlRadios(false);
    enableDCControlRadios(false); //For some reason can only set radio states when buttons are disabled
    ui->DCSensorControlRadio->setChecked(enabled);
    ui->DCGUIControlRadio->setChecked(not enabled);
    enableDCControlRadios(true);

}
//Enable control mode radio buttons
void GUIMainWindow::enableDCControlRadios(bool enabled){
    ui->DCSensorControlRadio->setEnabled(enabled);
    ui->DCGUIControlRadio->setEnabled(enabled);
}
void GUIMainWindow::enablePIDControlRadios(bool enabled){
    ui->DCPositionControlRadio->setEnabled(enabled);
    ui->DCVelocityControlRadio->setEnabled(enabled);
}

//Radio Buttons Callbacks

//Sets GUI Control Mode
void GUIMainWindow::slotGUIControlMode(){
    enableDCControlSensor(false);
    enableDCControlGUI(true);
```

```
}
//Sets Sensor Control Mode
void GUIMainWindow::slotSensorControlMode(){
    enableDCControlSensor(true);
    enableDCControlGUI(false);
}

GUIMainWindow::~GUIMainWindow()
{
    delete ui;
}
```

Appendix 2: RAMBO Firmware

Main.c

```
#include "guimainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    GUIMainWindow w;
    w.show();

    return a.exec();
}
```

Firmware

Main.c

```
#include <stdint.h>
#include "main.h"
#include "stepper.h"
#include "serial.h"
#include "servo.h"

stateVariables sv;

#define SERIAL_COUNTDOWN_LOAD 1  //Number of loop iterations per call to serialSend()

unsigned long serialSendCountdown;

void setupSV(){
 sv.servoEnabled = 0;
 sv.stepperEnabled = 0;
 sv.dcEnabled = 0;

 sv.servoGain = 5;
 sv.stepperGain = 5;
 sv.dcGain = 5;

 sv.dcTargetPos = 0;
 sv.dcTargetVelocity = 0;
}

//ADD SETUP FUNCT
void setup() {
 setupSV();
 setupSerial();
 setupStepper();
 setupPot();
 setupDCmotor();
 setupForce();
 setupServo();

 serialSendCountdown = SERIAL_COUNTDOWN_LOAD;
}
void loop(){
 parseSerial();

 readPot();
 readProx();
 readForce();

 controlStepper();
 controlServo();
 controlDCmotor();
 serialSendCountdown --;
 if(serialSendCountdown == 0){
  sendSerial();
  serialSendCountdown = SERIAL_COUNTDOWN_LOAD;
 }
}
```

Main.h

```
#ifndef mainh
#define mainh

//State Variables
struct stateVariables{
//Control Variables
   //Stepper
   uint16_t stepperGain;
   int32_t stepperCurrentPos;
   uint16_t stepperSensorVal;
   bool stepperEnabled;

   //Servo
   uint16_t servoGain;
   int32_t servoCurrentPos;
   uint16_t servoSensorVal;
   bool servoEnabled;

   //DC Motor
   uint16_t dcGain;
   uint16_t dcSensorVal;
   double dcCurrentPos;
   double dcCurrentVelocity;
   double dcTargetPos;
   double dcTargetVelocity;
   int dcPWM;
   bool dcSensorControlMode; //true = sensor control mode, false = GUI control mode
   bool dcPositionPIDMode; //true = position PID control, false = velocity PID control
   bool dcEnabled; //true = motor enabled, false = motor disabled

   bool buttonStatus;
};
```

Serial.c

```
#include "main.h"
#include "stepper.h"
#include "protocoldefinitions.h"

extern stateVariables sv;

void setupSerial(){
   //Confiugre serial
 Serial.begin(BAUD);
}

void sendSerial(){
 Serial.print(RESP_STEPPER_POSITION);
 Serial.println(sv.stepperCurrentPos);

 Serial.print(RESP_SERVO_POSITION);
 Serial.println(sv.servoCurrentPos);

 Serial.print(RESP_STEPPER_SENSOR);
 Serial.println(sv.stepperSensorVal);

 Serial.print(RESP_SERVO_SENSOR);
 Serial.println(sv.servoSensorVal);

 Serial.print(RESP_DC_SENSOR);
 Serial.println(sv.dcSensorVal);

 Serial.print(RESP_BUTTON_STATUS);
 Serial.println(sv.buttonStatus);

 Serial.print(RESP_SERVO_GAIN);
 Serial.println(sv.servoGain);

 Serial.print(RESP_STEPPER_GAIN);
 Serial.println(sv.stepperGain);

 Serial.print(RESP_DC_POS);
 Serial.println((int)sv.dcCurrentPos);

 Serial.print(RESP_DC_VEL);
 Serial.println((int)sv.dcCurrentVelocity);

 Serial.print(RESP_DC_GAIN);
 Serial.println(sv.dcGain);

 Serial.print(RESP_DC_CONTROL_SENSOR);
 Serial.println(sv.dcSensorControlMode);

 Serial.print(RESP_DC_PID_POS);
 Serial.println(sv.dcPositionPIDMode);

 Serial.print(RESP_PWM);
 Serial.println((int)sv.dcPWM);

 Serial.print(RESP_MILLIS);
 Serial.println(millis());

 }

void parseSerial(){
 unsigned char nextByte;
 while(Serial.available() > 0){
      nextByte = Serial.read();
      switch(nextByte){
        case SET_SERVO_GAIN:
           sv.servoGain = (uint16_t)Serial.parseInt();      //Set gain between servo and distance sensor
         break;
        case SET_STEPPER_GAIN:
```

```
          sv.stepperGain = (uint16_t)Serial.parseInt();     //Set gain between stepper position and pot
        break;
      case HOME_STEPPER:
        stepperhome();                    ///Home stepper motor, Update stepper position when done
        break;
      case SET_DC_CONTROL_SENSOR:
          sv.dcSensorControlMode = (bool)Serial.parseInt();      //0 = sensor control, 1 = GUI controlled
        break;
      case SET_DC_GAIN:
          sv.dcGain = (uint16_t)Serial.parseInt();;            //Set gain between force sensor and DC motor
        break;
      case SET_DC_PID_POS:
          sv.dcPositionPIDMode = (bool)Serial.parseInt();           //0 = position control, 1 = velocity control
        break;
      case SET_DC_VEL:
          sv.dcTargetVelocity = (double)Serial.parseInt();   //Set position of DC motor (in encoder counts)
        break;
      case SET_DC_POS:
          sv.dcTargetPos = (double)Serial.parseInt();        //Set velocity of DC motor (encoder counts/second)
        break;
      case ENABLE_STEPPER:
          stepperEnable((bool)Serial.parseInt());      //0 = disable, 1 = enable
        break;
      case ENABLE_SERVO:
          sv.servoEnabled = (bool)Serial.parseInt();;        //0 = disable, 1 = enable
        break;
      case ENABLE_DC:
          sv.dcEnabled = (bool)Serial.parseInt();          //0 = disable, 1 = enable
        break;
      case STOP_ALL:
          sv.dcEnabled = false;
          sv.servoEnabled = false; //Disable all motors
          stepperEnable(false);
        break;
      default:
          //Serial Read Error
          break;
    }//switch
  }//while
}//parseSerial
```

Serial.h

```
#ifndef serialh
#define serialh

#define BAUD 19200

void setupSerial();
void sendSerial();
void parseSerial();

#endif serialh
```

DCMotor.c

```
#include "pindefinitions.h"
#include "protocoldefinitions.h"
#include "DCmotor.h"
#include "main.h"
#include <Encoder.h>
#include <PID_v1.h>


extern stateVariables sv;

int previousForce = 0;
double Output_p, Output_v, targetDistance = 0;
unsigned long lastTime;
signed long lastBase = 0;
signed int v_pwm;

Encoder base(CHANNELA, CHANNELB);
PID position_PID(&sv.dcCurrentPos, &Output_p, &sv.dcTargetPos, Kc_p, Ki_p, Kd_p, DIRECT);
PID velocity_PID(&sv.dcCurrentVelocity, &Output_v, &sv.dcTargetVelocity, Kc_v, Ki_v, Kd_v, DIRECT);
//int stepperSpeed = 10;

void setupDCmotor() {
 //Set up pints
 pinMode(DC_ENABLE_PIN, OUTPUT);
 pinMode(MOTOR_SUPPLY, OUTPUT);
 pinMode(DIR1_PIN, OUTPUT);
 pinMode(DIR2_PIN, OUTPUT);
 pinMode(CHANNELA, INPUT);
 pinMode(CHANNELB, INPUT);
 digitalWrite(DC_ENABLE_PIN, LOW);
 digitalWrite(MOTOR_SUPPLY, HIGH);
 digitalWrite(DIR1_PIN, LOW);
 digitalWrite(DIR2_PIN, LOW);
 //set up pull up resistors for encoder
 digitalWrite(CHANNELA, HIGH);
 digitalWrite(CHANNELB, HIGH);

 //initialize variables
 sv.dcGain = DEFAULT_DCMOTOR_GAIN;
 sv.dcCurrentPos = 0;
 sv.dcCurrentVelocity = 0;
 sv.dcTargetPos = DEFAULT_TARGET_POSITION;
 sv.dcTargetVelocity = DEFAULT_TARGET_VELOCITY;
 sv.dcSensorControlMode = DEFAULT_SENSOR_CONTROL_MODE; //true = sensor control mode, false = GUI control mode
 sv.dcPositionPIDMode = DEFAULT_PID_CONTROL_MODE; //true = position PID control, false = velocity PID control
 sv.dcEnabled = LOW; //true = motor enabled, false = motor disabled

 //Position PID Settings
 position_PID.SetOutputLimits(-255,255);
 //Velocity PID Settings
 velocity_PID.SetOutputLimits(-255,255);

 sv.buttonStatus = LOW;
 position_PID.SetMode(AUTOMATIC);
```

```
 velocity_PID.SetMode(AUTOMATIC);
 velocity_PID.SetSampleTime(20);
}


void enableDCmotor(bool enable) {
 sv.dcEnabled = enable;
}

//255 = full forward
// 0 = stop
//-255 = full reverse
void runDCmotor(int pwm) {
 sv.dcPWM = pwm;
 if(pwm > 0)
 {
  analogWrite(DC_ENABLE_PIN, pwm);
  digitalWrite(DIR1_PIN, HIGH);
  digitalWrite(DIR2_PIN, LOW);
 }
 else{
  analogWrite(DC_ENABLE_PIN, pwm*-1);
  digitalWrite(DIR1_PIN, LOW);
  digitalWrite(DIR2_PIN, HIGH);
 }
}


void controlDCmotor() {
 int force = sv.dcSensorVal;
 if (sv.dcEnabled)
 {
  unsigned long now = millis(); //Get current time
  signed long newbase = base.read(); //get current encoder pos
  unsigned long timeChange = (unsigned long)(now - lastTime);
  sv.dcCurrentPos = (double)base.read() + 10000.0; //update current position state variable
  sv.dcCurrentVelocity = (double)(lastBase - newbase)/(double)timeChange; //update current velocity state variable
  sv.dcCurrentVelocity *= 10000.0;
  //sv.dcCurrentVelocity += 10000.0;
  lastTime = now;
  lastBase = newbase;

  if (sv.dcSensorControlMode)
  {
   float gain = (float)sv.dcGain / 100.0;
   int16_t pwm = ((int16_t)sv.dcSensorVal) * gain;
   pwm = constrain(pwm, 0, 255);
   runDCmotor(pwm);
  }
  else {
   if (sv.dcPositionPIDMode) {
      controlMotorPosition();
   }
   else{
    controlMotorVelocity();
   }
  }
 }
 else{
  analogWrite(DC_ENABLE_PIN, 0); //disable motor
 }

}

void controlMotorPosition() {
 //position pid control code
 /*
 Serial.print(RESP_DEBUG);
 Serial.print("Output_p = ");
 Serial.println(Output_p);
```

```
 */
 position_PID.Compute();
 runDCmotor(Output_p);
}

void controlMotorVelocity() {
 //velocity pid control code
 /*Serial.print(RESP_DEBUG);
 Serial.print("error = ");
 Serial.println(sv.dcCurrentVelocity - sv.dcTargetVelocity);*/

 velocity_PID.Compute();

 v_pwm -= Output_v;
 v_pwm = constrain(v_pwm,-255,255);
 runDCmotor(v_pwm);
}

//////////**********FORCE**********////////////////////////

void readForce() {
 sv.dcSensorVal = analogRead(FORCE_PIN);
}

void setupForce() {
 pinMode(FORCE_PIN, INPUT);
 sv.dcSensorVal = analogRead(FORCE_PIN);
}
```

DCMotor.h

```
#ifndef DCmotor
#define DCmotor

#define DEFAULT_DCMOTOR_GAIN 10
#define DEFAULT_TARGET_POSITION 10000
#define DEFAULT_TARGET_VELOCITY 10000
#define DEFAULT_SENSOR_CONTROL_MODE LOW
#define DEFAULT_PID_CONTROL_MODE LOW
#define Kc_p 1.3
#define Ki_p .18
#define Kd_p .2
#define Kc_v .005
#define Ki_v .0004
#define Kd_v .0008
#define MAX_PWM 255

void setupDCmotor();
void controlDCmotor();
void controlMotorPosition();
void controlMotorVelocity();

void setupForce();
void readForce();

#endif
```

Stepper.c

```
#include "pindefinitions.h"
#include "protocoldefinitions.h"
#include "stepper.h"
#include "main.h"

extern stateVariables sv;

int stepperSpeed = 10;

void setupStepper(){
 //Set up pints
 pinMode(STEP_PIN, OUTPUT);
 pinMode(DIR_PIN, OUTPUT);
 digitalWrite(STEP_PIN, LOW);
 digitalWrite(DIR_PIN, LOW);
 pinMode(STEPPER_ENABLE_PIN, OUTPUT);
 pinMode(XMS1, OUTPUT);
 pinMode(XMS2, OUTPUT);
 digitalWrite(XMS1, LOW);
 digitalWrite(XMS2, HIGH);

 sv.stepperCurrentPos = 32767;
 sv.stepperGain = DEFAULT_STEPPER_GAIN;
 stepperEnable(false);
}

void stepperhome(){
 sv.stepperCurrentPos = 0;
}

// A custom delay function used in the run()-method
void holdHalfCycle(double speedRPS) {
 long holdTime_us = (long)(1.0 / (double) stepsInFullRound / speedRPS / 2.0 * 1E6);
 int overflowCount = holdTime_us / 65535;
 for (int i = 0; i < overflowCount; i++) {
  delayMicroseconds(65535);
 }
 delayMicroseconds((unsigned int) holdTime_us);
}

// Runs the motor according to a chosen direction, speed (rounds per seconds) and the number of steps
void runStepper(boolean runForward, double speedRPS, int stepCount) {
 digitalWrite(DIR_PIN, runForward);
 for (int i = 0; i < stepCount; i++) {
  digitalWrite(STEP_PIN, HIGH);
  holdHalfCycle(speedRPS);
  digitalWrite(STEP_PIN, LOW);
  holdHalfCycle(speedRPS);
 }
 if(runForward)
  sv.stepperCurrentPos -= stepCount;
 else
  sv.stepperCurrentPos += stepCount;
 }

void controlStepper(){
 int pot = sv.stepperSensorVal;
 if(sv.stepperEnabled)
 {
    float gain = (float)sv.stepperGain/100.0;
    int16_t targetPos = ((int16_t)pot-512)*gain + 32767;
    int16_t error = sv.stepperCurrentPos - targetPos;
  if(error > 0)
    runStepper(true, stepperSpeed, error);
  else
    runStepper(false, stepperSpeed, error*-1);
 }
}
```

```
void stepperEnable(bool enable){
   digitalWrite(STEPPER_ENABLE_PIN, !enable);
   sv.stepperEnabled = enable;
}

//////////**********POT**********////////////////////////////

void readPot(){
   sv.stepperSensorVal = analogRead(POT_PIN);
}

void setupPot(){
  pinMode(POT_PIN, INPUT);
   sv.stepperSensorVal =0;
}
```

Stepper.h

```
#ifndef stepper
#define stepper

#define stepsInFullRound 200
#define DEFAULT_STEPPER_GAIN 10

void setupStepper();
void controlStepper(int);

void setupPot();
void readPot();

#endif
```

Servo.c

```
#include "pindefinitions.h"
#include "protocoldefinitions.h"
#include "servo.h"
#include "main.h"

#include <Servo.h>
#include <SharpIR.h>

SharpIR sharp(SharpPin, 20, 50, model);   //(ReadPin, #readings library will take before calculating mean distance, #min difference between 2 consecutive measurements to be valid, range of sensor)
Servo MyServo;
bool B1_old = LOW; /////////////////
bool B1_debounced = LOW;/////////////////
bool B1_new = LOW; /////////////////
bool buttonEnable = LOW; ///////////////
unsigned long timeOfLastButton1 = 0; ///////////////////
int debounceInterval = 20; ////////////////
#define Button1 5 ////////////////////////////
#define ButtonLED 2

extern stateVariables sv;

void setupServo(){

  MyServo.attach(ServoPin);
  pinMode(ServoPin, OUTPUT);
  pinMode (SharpPin, INPUT);
  pinMode(Button1, INPUT); /////////////////
  pinMode(ButtonLED, OUTPUT); ///////////////////
}
```

```
void controlServo()
{
 ////////////////////////////////////////////////
 B1_new = digitalRead(Button1);
 if(B1_new == HIGH && B1_old == LOW && millis() - timeOfLastButton1 > debounceInterval){
     timeOfLastButton1 = millis();
     buttonEnable = !buttonEnable;
     sv.buttonStatus = buttonEnable;
 }

 ////////////////////////////////////////////////

 if (sv.servoEnabled && buttonEnable){ ////////////////////
   sv.servoCurrentPos = map(sv.servoSensorVal, 7,50, 0,180*sv.servoGain);
   sv.servoCurrentPos = constrain(sv.servoCurrentPos,0,180);
   MyServo.write(sv.servoCurrentPos);
 }

 B1_old = B1_new; ////////////////

}

void enableServo(bool enable){
 sv.servoEnabled = enable;
}

void readProx(){
   sv.servoSensorVal = sharp.distance();
   sv.servoSensorVal = constrain(sv.servoSensorVal,7,50);
}
```

Servo.h

```
#ifndef servo
#define servo


#define model 1080

void setupServo();
void controlServo();
void readProx();

#endif
```

Protocoldefinitions.h (note: this file is shared between the GUI and Firmware)

```
#ifndef PROTOCOLDEFINITIONS
#define PROTOCOLDEFINITIONS

//Commands
#define SET_SERVO_GAIN 'A'
#define SET_STEPPER_GAIN 'B'
#define HOME_STEPPER 'C'
#define SET_DC_CONTROL_SENSOR 'D'
#define SET_DC_GAIN 'E'
#define SET_DC_PID_POS 'F'
#define SET_DC_VEL 'G'
#define SET_DC_POS 'H'
#define ENABLE_STEPPER 'I'
#define ENABLE_SERVO 'J'
#define ENABLE_DC 'K'
#define STOP_ALL 'X'

//Responses
#define RESP_STEPPER_POSITION 'a'
#define RESP_SERVO_POSITION 'b'
#define RESP_STEPPER_SENSOR 'c'
```

```
#define RESP_SERVO_SENSOR 'd'
#define RESP_DC_SENSOR 'e'
#define RESP_BUTTON_STATUS 'f'
#define RESP_SERVO_GAIN 'g'
#define RESP_STEPPER_GAIN 'h'
#define RESP_DC_POS 'i'
#define RESP_DC_VEL 'j'
#define RESP_DC_GAIN 'k'
#define RESP_DC_CONTROL_SENSOR 'l'
#define RESP_DC_PID_POS 'm'
#define RESP_MILLIS 'n'
#define RESP_DEBUG 'o'
#define RESP_PWM 'p'

#endif // PROTOCOLDEFINITIONS
```

pindefintions.h

```
#ifndef pindefinitions
#define pindefinitions


#define RAMBO

#ifdef RAMBO
 //Stepper Motor
 #define STEP_PIN 37
 #define DIR_PIN 48
 #define STEPPER_ENABLE_PIN 29
 #define XMS1 40
 #define XMS2 41

 //Pot
 #define POT_PIN 60

 //DC Motor
 #define DC_ENABLE_PIN 4
 #define DIR1_PIN 32 //70
 #define DIR2_PIN 44 //71
 #define MOTOR_SUPPLY 3

 //Encoder Channel
 #define CHANNELA 18
 #define CHANNELB 19

 //Force
 #define FORCE_PIN 59

 //Prox
 #define SharpPin 62
 #define ServoPin 13
#endif

#ifdef UNO
 //Stepper Motor
 #define STEP_PIN 37
 #define DIR_PIN 48
 #define STEPPER_ENABLE_PIN 29
 #define XMS1 40
 #define XMS2 41

 //Pot
 #define POT_PIN 0

 //DC Motor
 #define DC_ENABLE_PIN 13
 #define DIR1_PIN 4 //70
```

```
 #define DIR2_PIN 5 //71
 #define MOTOR_SUPPLY 3

 //Encoder Channel
 #define CHANNELA 79
 #define CHANNELB 80

 //Force
 #define FORCE_PIN 59

 //Prox
 #define SharpPin 5
 #define ServoPin 8
#endif
```