DEVELOPMENT OF A DIGITAL CONTROLLER FOR MOTOR CONTROL EXPERIMENTS IN THE EE472 LAB

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Jeffrey David D'Amelio

December 2014

COMMITTEE MEMBERSHIP

TITLE:	Development of a Digital Controller for Motor Control Experiments in the EE472 Lab
AUTHOR:	Jeffrey David D'Amelio
DATE SUBMITTED:	December 2014
COMMITTEE CHAIR:	Arthur C. MacCarley, Ph. D, Professor, Electrical Engineering
COMMITTEE MEMBER:	Taufik, Ph. D, Professor, Electrical Engineering
COMMITTEE MEMBER:	Xiao-Hua (Helen) Yu, Ph. D, Professor, Electrical Engineering

ABSTRACT

Development of a Digital Controller for Motor Control Experiments in the EE472 Lab

Jeffrey David D'Amelio

A digital motor controller for student lab use is designed, built, and tested. The controller uses an encoder for position measurement, and an H-bridge to drive the electromechanical plant.

A user interface is created to enhance usability of the device. The user interface is able to display key parameters of the control algorithm as well as the state of the system. It is also used to modify the gains and sample rate of the control algorithm.

The design of the system is refined, and 10 units are built for the EE472 Digital Controls Lab. The lab manuals for the first 4 experiments are revised to match and support the new system.

The possible future for the project is described with some suggestions for improving the system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The current Cal Poly classical controls lab hardware, the Electro Craft Motomatic™, has been in use for nearly 30 years. It is used to demonstrate classical control theory to the students, and allow them to experiment with controller design [1]. Through a series of experiments they are familiarized with the equipment, taught to characterize the system, and finally learn how to implement a controller for their plant in hardware.

More recently, a digital controller was designed to work with the Motomatic, replacing some of the analog hardware and allowing a digital controls lab, EE472, to be offered. In this course students learn first about the digital controller, then how to characterize their plant and digital system, and finally how to implement a digital control algorithm for control of the plant [2].

These digital controllers had reached the end of their useful lives. Some of their features had become obsolete or insufficient, and many of the controllers had become unreliable or broken. This thesis details the

design of the replacement controller, as well as it's fabrication, testing, and integration into the lab environment.

The remainder of this document is organized as follows. The next chapter explains the requirements for the device. Chapter 3 details the hardware design. Chapter 4 details the software design. Chapter 5 reports the verification and testing done on the system. Chapter 6 describes the enclosure fabrication. Chapter 7 explores the possibilities for future work on this system.

1.1 List of Terms

Quantization - The process of constraining a continuous value to a discrete set of values.

Discretization - The process of transforming a continuous time signal into a discrete time signal.

H-Bridge - Circuit enabling a unipolar voltage to be applied across a load in either direction.

Sample Rate - The interval at which control inputs are sampled.

UI - User Interface, a computer application for interacting with the digital controller.

Encoder - A digital sensor that converts a linear or rotary motion into a series of pulses.

Quadrature - Separation of the two encoder phases by 90 degrees.

PWM - Pulse width modulation, a modulation scheme for encoding the amplitude of a signal into the width of a pulse.

ISR - Interrupt subroutine, a particular section of code that runs only as a result of the interrupt being triggered.

DAC - Digital to analog converter, converts a digital value to an analog voltage.

ADC - Analog to Digital Converter, converts an analog voltage to a digital value.

1.2 Previous Digital Controller

The following is a brief description of the operation of the previous digital controller, shown in Figure 1, in a typical lab [2]. This configuration uses the same electromechanical apparatus as would be used with the redesigned lab hardware. The previous controller was intended to be a digital, drop in replacement for the summing node and amplifier built in to the Motomatic console, effectively replacing the amplifier with a digital equivalent. Figure 2 shows the closed loop system block diagram of this system with a lead compensator for the control algorithm [5].

Figure 1: Original Digital Controller

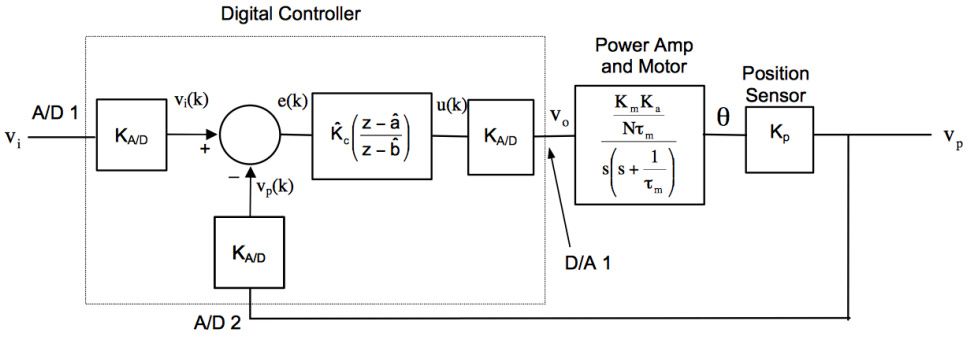


Figure 2: Closed loop block diagram for original digital controller with a lead compensator

This controller has 3 potential input sources, a step input voltage from the Motomatic console, a position voltage from the feedback potentiometer, and a digital position from the encoder on the motor shaft [1]. The step input signal, V_i, generated from the Motomatic console ranges between ±15V. It is generated from a linear potentiometer and three-position switch. The switch selects between +15V, ground, and -15V, while the potentiometer acts as a voltage divider between the selected voltage and ground. In this way the student is able to set the magnitude of the step input signal using the potentiometer, and use the switch to determine the sign of the step.

Similarly, a continuous rotation potentiometer is used as position feedback. It acts as a voltage divider between the ±15V supplies, with the position of the electromechanical apparatus' output shaft determining the output voltages.

When either of these signals is connected to the controller, they must first be scaled to a 0V to 5V range. This is done with the amplifier circuit in Figure 3. The clamping diodes are included to ensure that the signal cannot exceed 0-5V even if the input exceeds the intended ±15V range.

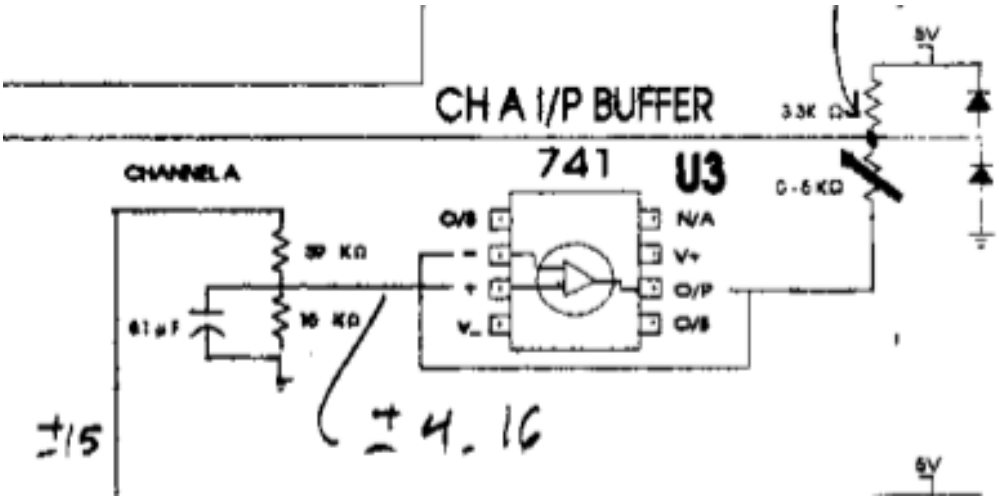


Figure 3: Analog input scaling circuit from original digital controller

The 0-5V analog signal is then sampled with the ADC built in to the MC68HC11. When the conversion has completed, the digital value is presented to the student as an 8 bit two's complement integer.

As an alternative to the feedback potentiometer, the digital shaft encoder can be used. This is a 2048 count per revolution quadrature encoder connected to the motor shaft of the electromechanical apparatus. The encoder is read with an HCTL-2020 encoder counter IC. From the A and B channels of the encoder, this IC generates a 16-bit position value. The encoder counter IC is interfaced to the microcontroller with a parallel bus, clock signal, and other control lines. The proper scaling and manipulation of this 16-bit position value for use as position feedback in the control algorithm is left to the student.

In assembly, the student is responsible for generating the error signal from the step input and the selected position feedback source. The

student then implements a digital control algorithm to generate an output value from the error signal. These output values are also 8 bit two's complement integers.

The microcontroller uses an external DAC, the MAX 508, to convert the output value to a voltage. This conversion is done such that the full range of the two's complement value the voltage is converted from corresponds to ±10V. The output voltage is buffered before being made available to the student. This scaling gives the controller hardware an inherent gain of approximately 2/3, which must be accounted for by the student in the design of their algorithm. The two DAC channels included in the controller are interfaced to the microcontroller using the same parallel bus as the encoder counter.

Next the output voltage is returned to the Motomatic console where it is connected to the input of the power amplifier. This amplifier generates the voltage that actually drives the motor in the electromechanical apparatus. It is current limited to approximately 4 Amps, and will light a red status LED when the limit is reached or the output is saturated. It is the responsibility of the student to avoid creating a controller that is overly affected by the current limit.

In a typical lab, the student will copy the exp1.asm file to their home directory, along with the AS11.exe assembler and the executable DOS file as.bat. They will modify the exp1.asm file with their control algorithm,

written in assembly. Then they will run the as.bat file from the command line to assemble their code, generating a machine code .s19 file and an .lst list file. The list file must be examined for errors to ensure that the code assembled correctly.

Now the student must load the assembled code onto the microcontroller. First, the box is connected to the computer using the RS-232 port built into the HC11 evaluation board. The student must open HyperTerminal on the PC, and select the appropriate COM port, baud rate, and other settings. Next the student will reset the control box causing it to send a header message, verifying proper communication with the PC. The student must send a carriage return to get a prompt from the serial monitor. Now the student types and sends "load t" to prepare the serial monitor to receive the new file. Finally, the student can use HyperTerminal's download feature to send the assembled .s19 file over the serial port. The serial monitor should recognize and properly store this file in program memory.

Before testing their control code, the student must remember to initialize the sample period, or the controller will not function. This too is done through the serial connection to HyperTerminal. It requires that the student know the memory location of the variable SPERIOD, and cannot be performed while the control code is running.

Now the student can test their control code. They start execution of their code by sending the command "g d000". This runs their control algorithm, but also prevents them from modifying any of the control constants or other variables. They must reset the controller to make these changes. Once the control code is running, HyperTerminal™ is used to view the control variables such as input, output, and error. This display starts automatically when the control code is run, and is shown in Figure 4.

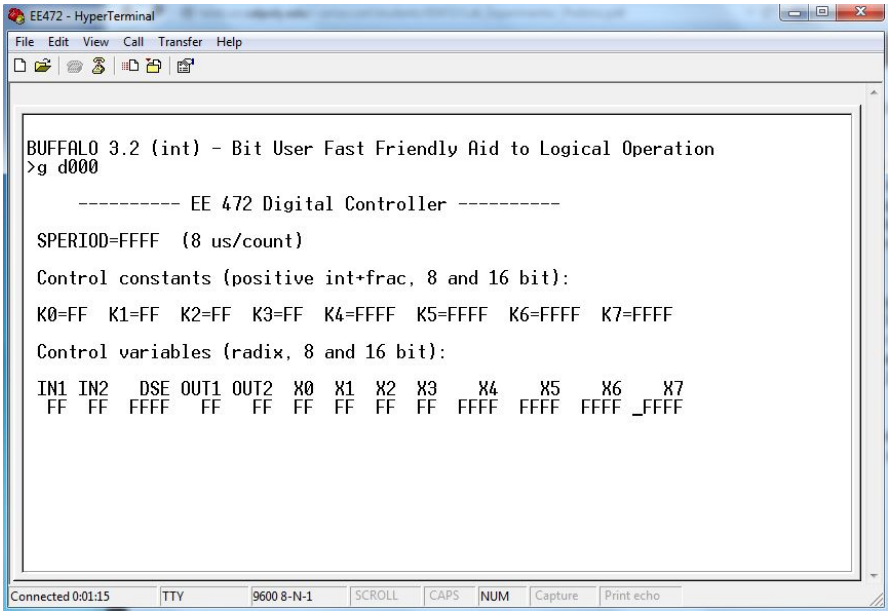


Figure 4: HyperTerminal serial display

Chapter 2

Design Requirements

The replacement controller was required to match or exceed the functionality of the previous system in all ways, while increasing ease of use to allow a student to easily function in the new lab environment. The following sections will detail both specific hardware requirements as well as softer goals for the project, and will cover aspects such as size and shape, ease of use, suitability for future experiments and labs, ease of repair, and educational value.

2.1 Student Usability

To facilitate student use in the lab, operation of the Control Box must be simple and easy for the student. This should include being easy to understand, simple to setup, and fast to reconfigure.

All external connections should be clearly labeled with an intuitive and consistent naming scheme. While the student is expected to familiarize themselves with the hardware before attending lab, it should be readily apparent what the boxes external connections are and how to use them. In addition to being labeled, each connection should be color-coded

to indicate whether it is a supply, a ground, or a signal. The connectors should be compatible with the cables available for lab work.

Additionally, the box must be able to easily interface with the existing Motomatic Electromechanical apparatus. It should not require any adapter cables, or additional hardware to perform the current set of experiments.

Programming the Control Box should be as simple as possible. The student should not need to be familiar with how code is compiled and loaded into a microcontroller to program the device. The procedure for loading code should be consistent between all experiments. Additionally, the number of times the device must be programmed in normal lab use should be minimized to allow the student to focus on other aspects. The Control Box should be programmable from any standard computer without additional hardware.

The student must be able to modify constants and variables to tune their control algorithms without reprogramming the device. This process should be extremely simple and able to be performed while the control algorithm is running. This will allow the students to immediately see the effects of their actions on the response of the system and rapidly iterate on their algorithm design.

The Control Box should not be overly large, heavy, or cumbersome. It should be able to comfortably fit on a small portion of the lab bench,

without creating too much cable clutter or being physically awkward to work with.

2.2 Educational Value

The manner in which the required controller functionality is implemented must done with the educational value of the device in mind.

The hardware and software should be balanced between presenting things, such as data or features, to the student clearly and not obfuscating the operation of the hardware at it's most basic level. That is, while the student should not need to concern themselves with how the control hardware operates at it's most basic level, it should still be apparent to them what the Control Box is doing and how it does it. For example, if the controller performs and analog to digital conversion with a 10 bit ADC and truncates the digital value to 8 bits to simplify the student's implementation of the control algorithm, the student doesn't need to be aware that the reading was taken with a 10 bit ADC, but it should be clear to them that the 8 bit value they are presented with was acquired with an ADC, is representative of an analog value, and will show the effects of quantization.

The Control Box should be capable of paralleling the experiments of the EE342 Controls Lab. This will help to demonstrate to the student the differences between continuous time and discrete time control by allowing

them to observe the similarities and differences between controllers implemented with analog and digital methods. Students will be able to observe the exact same controllers they implemented previously operate differently when implemented digitally.

Key to the Control Box's educational value is the instructor's ability to use the box. It therefore should be designed in such a way that an instructor can quickly familiarize themselves with the hardware and the lab experiments. This means that the hardware design should be intuitive where possible, and the software should be kept simple and well documented.

2.3 Ease of Repair

The apparatus is expected to be in use for at least 10 years. It therefore must be easily serviceable. Tune-up and repair should be not only possible, but also simple and cheap.

The box must be serviceable by a relatively untrained individual, and should not require an instructor or technician. Ideally, a TA should be able to understand the hardware well enough to perform maintenance or repairs. Surface mount components are to be avoided where possible in favor of through hole components. This should allow any necessary rework or repair to be performed by a student or assistant with basic soldering skills.

This also means the hardware should be cheap where possible. Use of expensive or specialty components is to be avoided. Components selection should be done with robustness, and longevity as high priority to minimize the required maintenance.

It is expected that in the course of regular student use the Control Boxes will see some amount of wear and misuse. Whenever possible, the hardware and software should be designed to withstand improper operation, improper connection, and other forms of unintended use.

2.4 Connectivity

The controller is not intended for standalone operation. It should be designed to operate while connected to a computer at all times. However, the controller, while it will lose some functionality, should be able to maintain control of the electromechanical apparatus if communication to the computer fails. The computer connection serves multiple functions.

First, the computer connection allows the student to load new code to the microcontroller. This must be done each time the control algorithm is changed, and will likely be required at least a few times each lab period. The process should be relatively fast and simple, and should be extremely reliable. The student should not have to deal with most of the issues related to the previous hardware, such as file location, operation of the assembler, or misnamed file types.

Next, the connection to the computer should allow the student to monitor and control the operation of the controller hardware and the control algorithm. This is closely related to the functionality of the user interface, which will be discussed in the next section.

Finally, if possible, the computer connection should support connectivity with with Matlab Simulink. This is not a feature that is required by the current realization of the EE472 Digital Controls Lab, but could be useful to other courses interested in using the same hardware. This feature should significantly expand the capability of the Control Box, allowing to be more easily integrated into a variety of future courses.

2.5 User Interface

The user interface will be the student's main point of interaction with the Control Box [10]. It must be simple to connect to the Control Box, even for a new user.

The user interface should allow students to view critical control variables, such as error, position, and output, in real time. The display of these variables should be intuitive and responsive, with relatively low latency. This easy access to control variables in real time should assist the student in verifying that their control algorithm is functioning properly.

Additionally, the user interface should allow the student to manipulate certain constants or variables in the controller, without

requiring the controller to be reprogrammed or reset. This will enable the student to rapidly iterate on their controller design by adjusting their control constants. This could potentially let the student modify control constants even while the controller is running, making the control loop tuning procedure significantly faster than it was with the previous hardware.

Chapter 3

Hardware Design

This section will explain the functionality and design of the final hardware configuration found in the new Control Box. The intent of this design was to meet all the requirements listed in Chapter 2, while doing everything possible to provide the best possible digital controls learning experience to the students. Care was also taken to design a controller flexible enough for changing lab conditions and expectations. Many additional features were included to improve on the capabilities of the previous digital controller.

The following sections describe the functionality and implementation of each hardware element of the Control Box. Section 1 describes the overall system configuration and how it interacts with the electromechanical apparatus. Section 2 covers the microcontroller selection. The following sections cover the H-bridge, analog inputs, analog outputs, step input generator, current sensors, power supplies, and encoder interface respectively.

3.1 System Configuration

The new Control Box was designed to completely eliminate the need for the Motomatic console. It includes modern hardware capable of replacing that found in the Motomatic console. It can perform all the functions of both the previous digital controller and the Motomatic console; its hardware layout is shown in Figure 5. Figure 6 shows the new closed loop system block diagram for a typical experiment with an undetermined control algorithm.

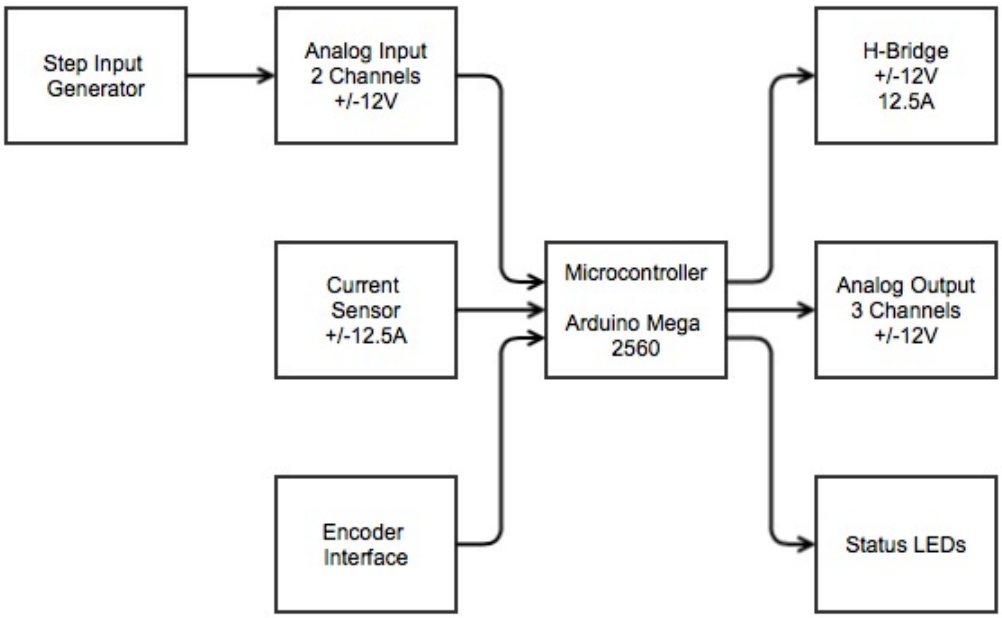


Figure 5: Hardware Layout

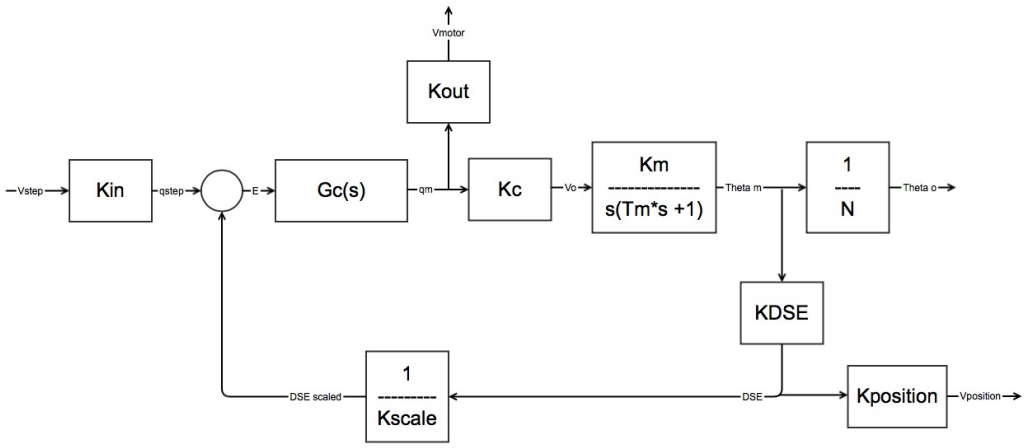


Figure 6: Closed Loop Block Diagram for new Control Box

The step input generator is built into the new Control Box, and like the previous version, is read by the microcontroller's built in ADC. An auxiliary analog input is also included. This could be used for reading the voltage from an analog position sensor such as the position sensing potentiometer on the electromechanical apparatus, or from another source. However, it is intended that the digital shaft encoder will now be the only source of position feedback for the EE472 experiments, so in most cases this analog input will not be used.

The new Control Box must also replace the power amplifier and DAC used to drive the electromechanical apparatus in the old configuration. This was done with a PWM H-bridge, interfaced directly to the microcontroller. The H-bridge replaces both the DAC and the power amplifier by removing the need to first convert the output value to an analog voltage, instead generating the motor voltage directly from the

output value. This implementation allows the motor driver to be built into the Control Box without requiring excessive board space.

Despite the DAC no longer being needed for control of the motor, 3 DAC channels are included in the new Control Box. Each channel allows the student to convert one of the internal variables of the Control Box into an output voltage for display on an oscilloscope or digital multi-meter. Each channel is an analog representation of the digital value it is generated from. The first channel represents the motor output voltage, the second represents the output shaft position, and the third can display either the auxiliary analog input voltage, or the motor current.

The new Control Boxes each include three current sensors built into the H Bridge. There is one sensor in each half of the bridge, and one hall effect sensor directly in series with the motor. These are included to enable the student to measure power in the controller itself, so that this information can be used in the control loop if an experiment ever wishes to use this functionality.

3.2 Microcontroller

An Arduino Mega 2560 was selected for the microcontroller of the new Control Box. While it was possible to build the microcontroller directly onto the circuit board, it was decided that it would be cheaper and easier to simply build a circuit board that plugged into the Arduino. It was also

very difficult to find a through hole microcontroller that met the requirements for this project.

The Arduino Mega 2560 includes a 16MHz clock, a UART to USB Converter, and breaks out the majority of the microcontroller pins to female headers. The UART to USB converter allows the Control Box to be directly connected to any PC without any sort of external converter or adapter. The Arduino mounts directly onto the Control Board, and interacts with all the peripheral devices and supporting hardware through this board, pictured in Figure 7. Conveniently, the Arduino also includes a 5V linear regulator, which is used to power the 5V devices on the Control Board.

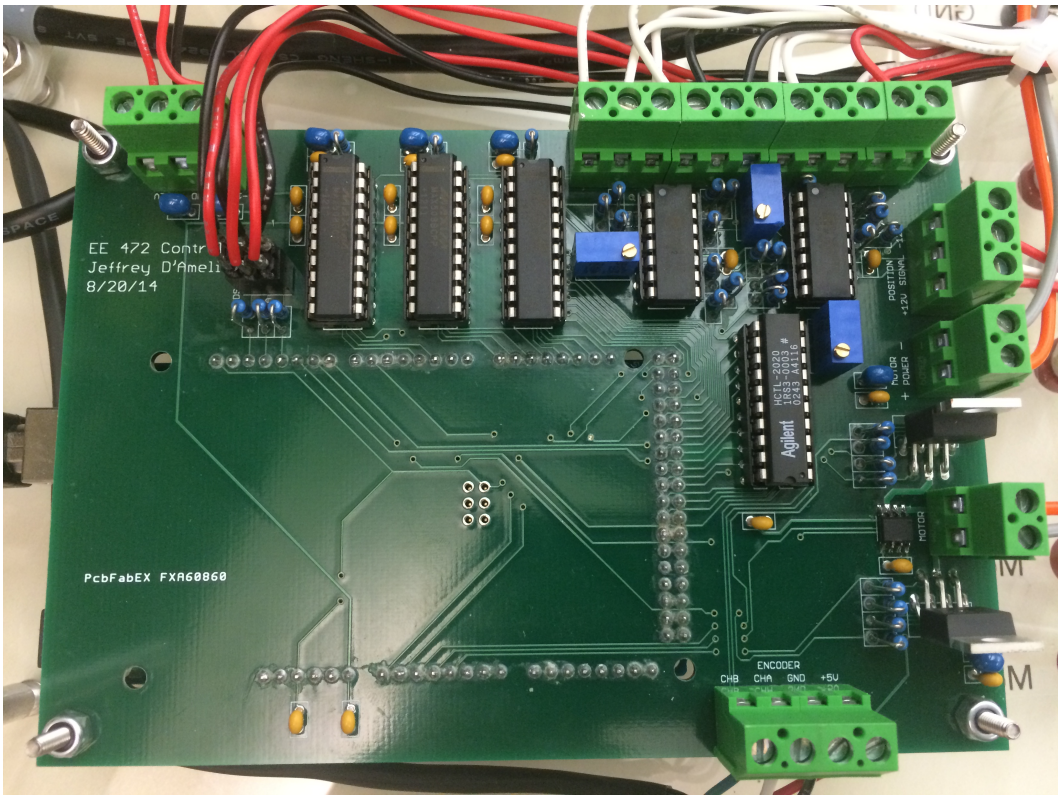


Figure 7: EE472 Control Box Control Board

The Arduino also satisfies the requirement for compatibility with Matlab Simulink. Simulink has built in support for Arduino that will not require any hardware modifications to the Control Box.

3.3 H-bridge

The H-bridge was one of the biggest improvements made from the old system. Specifically, the current limit on the power amplifier was obscuring the step response of the motor by clamping the available torque. The new motor driver had to avoid this problem. The power amplifier had to current limit because it used transistors in their linear

region to drive the motor. This meant the device had to dissipate significant power, and would heat very quickly.

The H-bridge avoids this by not operating its mosfets in the linear region. They are instead switched at a high frequency between their fully on and fully off states. This makes the H-bridge extremely efficient, if it requires a current limit it will be at significantly higher currents than the power amplifier.

This H-bridge uses two half H-bridges, the BTN7960. Its schematic can be seen in Figure 8. In a full bridge configuration the BTN7960 will have a resistance of 30.5 mOhm and can drive up to 33A. This is far more current than the electromechanical apparatus requires. Additionally, the BTN7960 has built in over temperature, over current, over voltage, and under voltage protection. As shown in Figure 8, the three lines, IN1, IN2, and INH control the H-bridge. IN1 and IN2 are both PWM signals driving the bridge halves, while INH enables and disables both halves of the bridge. M+ and M- connect directly to the motor leads.

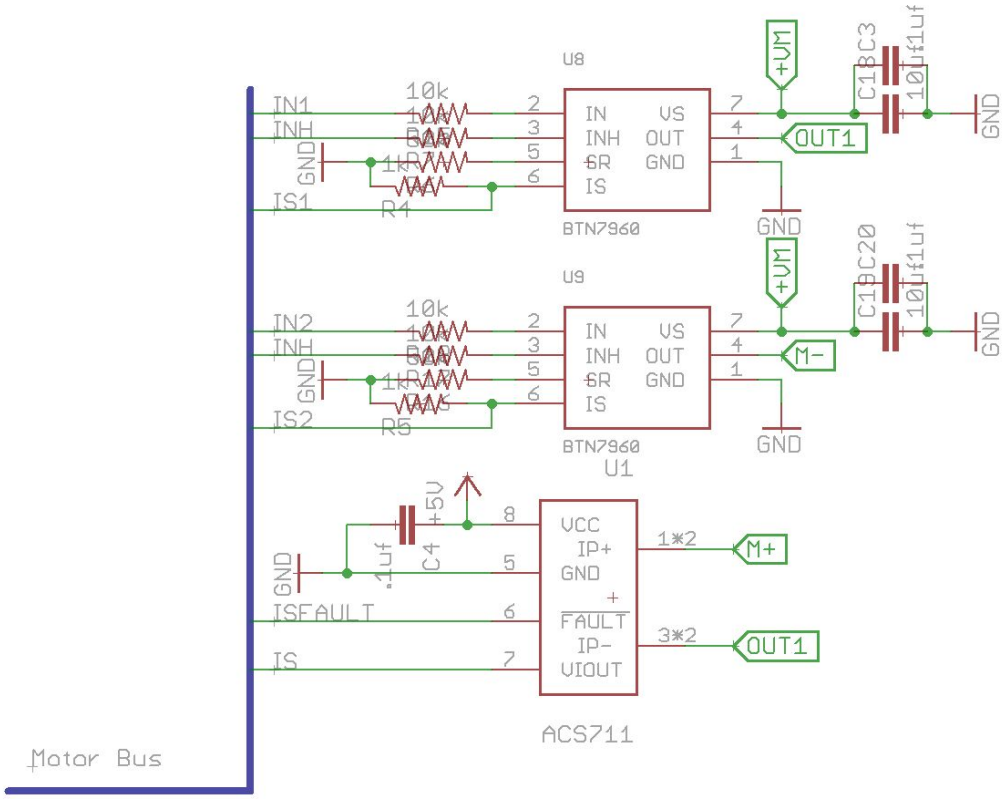


Figure 8: H-bridge schematic

3.4 Analog Inputs

The analog inputs are very similar to their previous implementation. However, their intended range is ±12V instead of ±15V. First the signal is buffered with the OP484. Next the signal is passed through a voltage divider that shifts the signal range to 0V to 12V. The signal is buffered again with the OP484. Last the signal passes through another voltage divider, scaling the signal range down to 0V to 5V. Clamping diodes are not necessary, as the microcontroller pins already have them built in. After being scaled to the 0v to 5V range the signal is converted by the

microcontroller's built in ADC. Figure 9 shows this input range adjustment circuit for both the step input and auxiliary analog input.

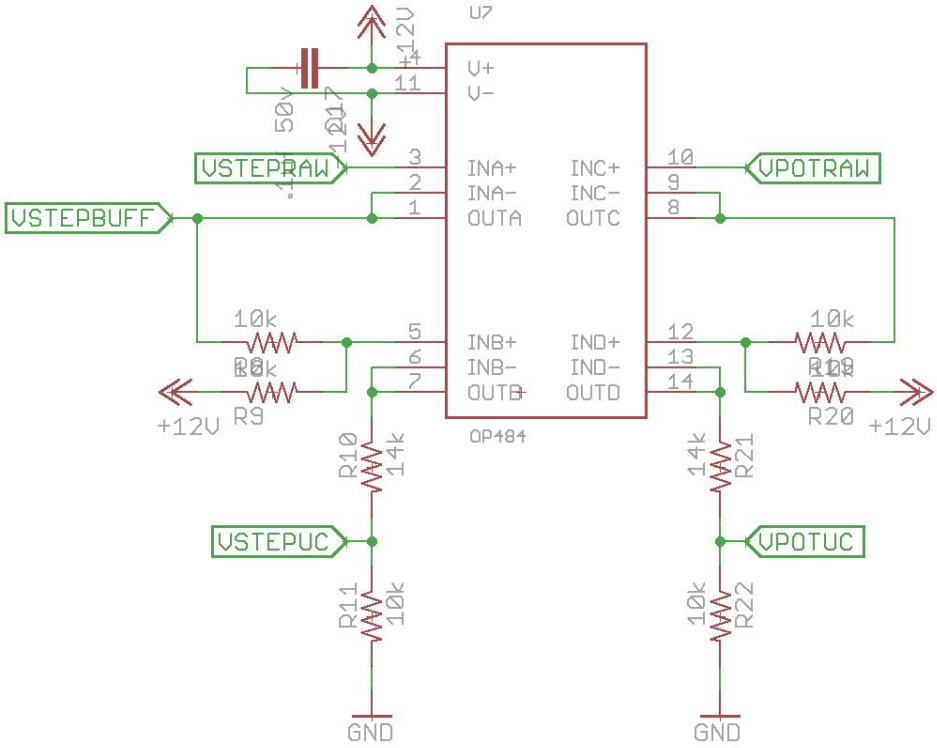


Figure 9: ADC schematic

Because both analog input signal sources were resistive, they were able to generate signals equal to the supplies powering the sensors and the op-amps. Since the op-amp was configured as a buffer, and expected input and output voltages very close to the supply voltages, it was critical that it be a rail-to-rail amplifier on both the input and output. For this application, other attributes of the amplifier were not critical. The OP484

was selected largely because it was available in a four amplifier through hole package, while meeting the requirement for rail-to-rail operation.

3.5 Analog Outputs

The three analog output channels included in the new Control Box differ somewhat from the previous hardware. They were required to match the scaling of the analog inputs to give the Control Box an intrinsic gain of 1. This meant their output range must be ±12V. As in the previous controller, the MAX508 was used to generate a bipolar output voltage. These DACs are 12 bits, interfaced to the Arduino with an 8 bit parallel bus and four control signals. However, this time the output range of the MAX508 is configured to ±5V instead of ±10V. This signal is then amplified to ±12V with an OP484 configured as a non-inverting amplifier. A potentiometer is included in series with one of the resistors in the non-inverting amplifier circuit to ensure the signal range is maintained, as we expect the reference voltage and therefore the output voltage range of the DAC to degrade as the component ages. This ±12V signal is not immediately made available to the student, but is first passed through a 10k Ohm resistor to protect the op-amps output from misuse. After the resistor, the analog output is ready for measurement by an oscilloscope or digital multi-meter. The schematic for these three DACs as well as their amplifiers is shown in Figure 10.

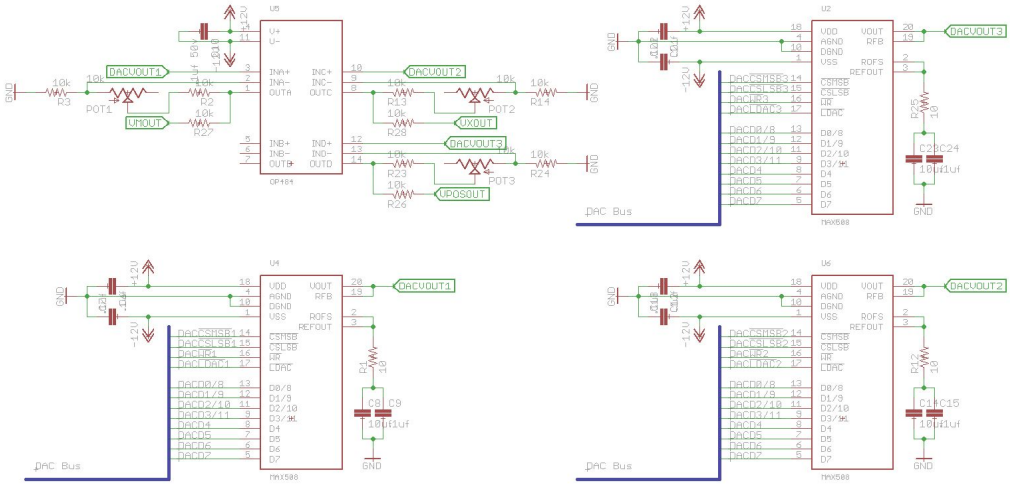


Figure 10: DAC schematic

3.6 Step Input Generator

The step input generation circuit is nearly identical to what was included in the previous system. The first exception is that this time the circuit is built into the Control Box instead of being part of the Motomatic console. The second is that the signal is buffered before the student has the opportunity to access it. As with the analog output circuit, the output of the buffer is protected with a 10k resistor in series. This should prevent the student from damaging the op-amp output. The step input switch and potentiometer are mounted to the Control Box case, not the board, so their schematic isn't shown. They function exactly as they did in the previous system, with the potentiometer controlling magnitude and the switch controlling the sign of the step signal.

3.7 Current Sensing

There are three current sensors in the motor driver circuit of the Control Box. Two are built in to the BTN7960 half H-bridges. While these do have drivers written for them, they are not part of the current implementation and only the third sensor is used. This third sensor is a hall effect current sensor directly in series with the motor. The ACS711 was selected for this. It is a ratio-metric sensor, scaling its output voltage range to match its supply voltage. It has an input range of ±12.5A and an output range of 0V to 5V, with 2.5V corresponding to 0A. While the allowable range for current measurement is significantly lower than what the H-bridge is capable of supplying, the current sensor cannot be damaged by over current, it will simply stop making accurate current measurements and set the fault pin low. The sensor's fault pin, ISFAULT, is connected to the Arduino but currently has no software support. The schematic for this sensor is included in Figure 11.

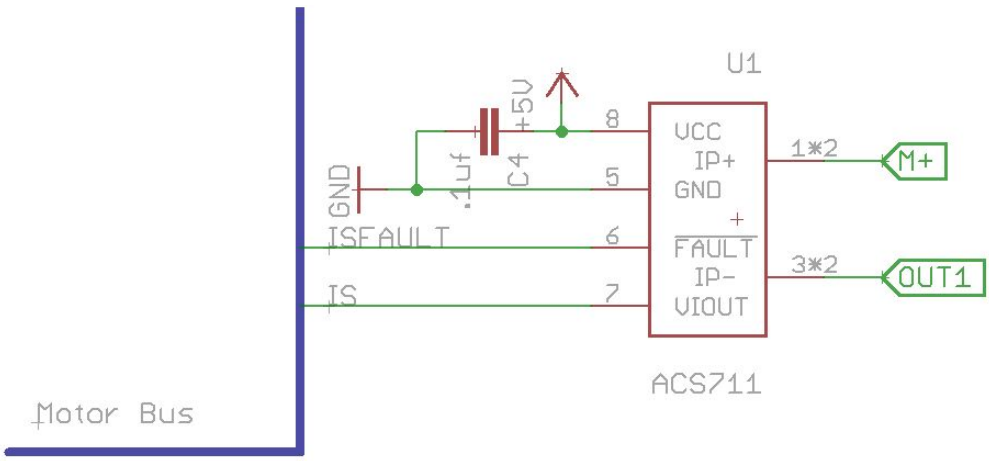


Figure 11: Motor hall effect current sensor schematic

3.8 Power Supplies

The Motomatic console has severe issues with power supply noise coupling into the analog signals. To avoid this, the redesigned controller uses two separate supplies. One supply powers the motor, and the second powers the microcontroller and analog circuitry. This keeps the analog signals very clean; there is no obvious noise from either the motor or the 60Hz AC present in the analog signals.

For the microcontroller and instrumentation the ECL15UD01, a 650mA, ±12V supply was selected. This current rating was higher than necessary, but was still the lowest available. This powers the step input generator, the DAC, and all the op-amps. The 12V rail supplies the 5V and 3.3V regulators on the Arduino, which in turn supply the encoder and other circuitry. Control of the negative rail is slaved to the positive side of the

regulator; so imbalanced loads will cause the magnitude of the rails to diverge from one another slightly.

The Motomatic console was unable to fully drive the motor on the electromechanical apparatus when the motor was stalled and would instead current limit. In many cases this behavior was a dominant effect, causing the measured response of the system to deviate significantly from what the students would expect to see. It was critical that a supply capable of powering a stalled motor at 12V without current limiting or overheating be selected. The motor would be driven at 12V to match the signal ranges of the rest of the system. While another voltage range could have been selected, it was decided it would be most intuitive for the students if all signals in the system shared the same range.

To determine the current rating for the supply, the motor stall current at 12V was required. To find this, the rotor was locked, and the voltage across the motor was measured at various currents. Additionally, the motor current was measured at various voltages. This data is presented in Figure 12. The slope of the line is the resistance of the motor. The current drawn by the motor at 12V was extrapolated from this data, and found to be 6.91A. The PMV12V150W from Delta Electronics was selected. It is capable of 12V at 12.5A, with over current, over temperature, and over voltage protection.

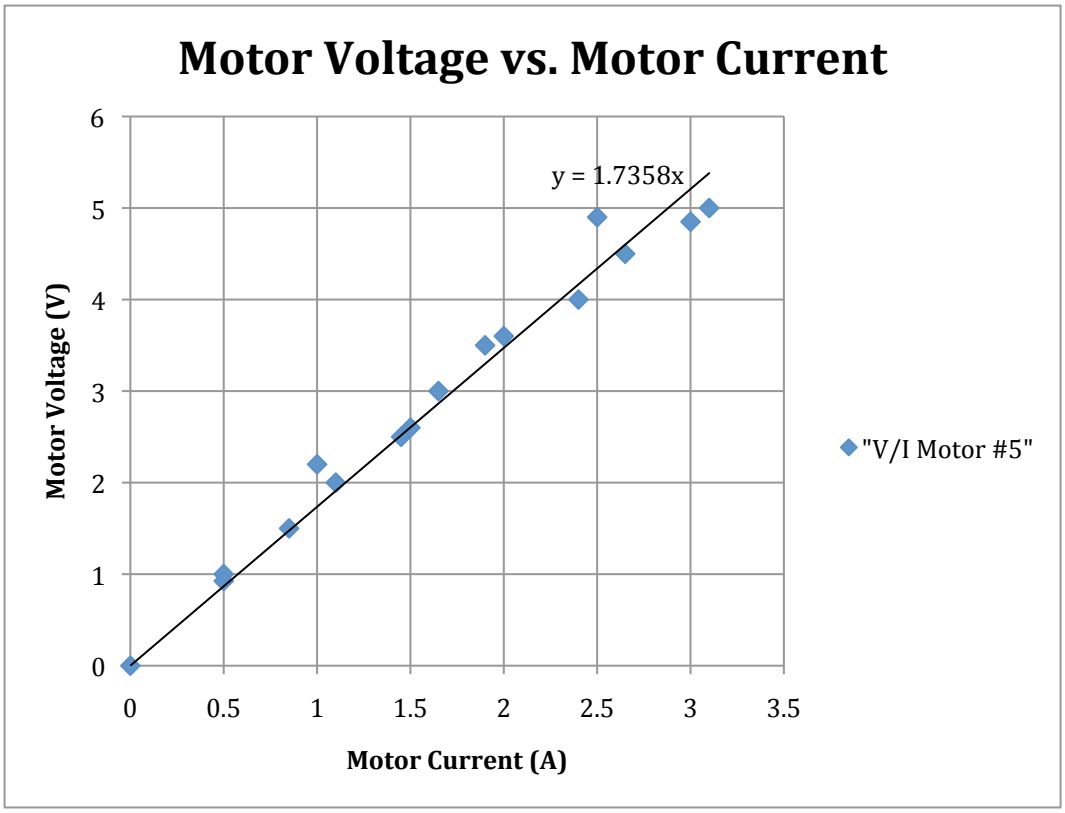


Figure 12: Motor Voltage vs. Motor Current at Stall

3.9 Encoder Interface

While the A and B channels of the encoder could have been attached directly to interrupts on the Arduino, it was decided it would be better to use an external encoder counter and let the Arduino read the encoder position from the counter. Using interrupts on the Arduino could have been problematic for the already rather long ISR that handled the bulk of the sampling and control calculations. The HCTL-2020 was selected for this. It is connected to the Arduino via an 8 bit parallel bus, clock, and various other control lines as shown in Figure 13. While a

discrete oscillator could have been used to generate the clock signal, it was cheaper and easier to just use one of the Arduino's timers.

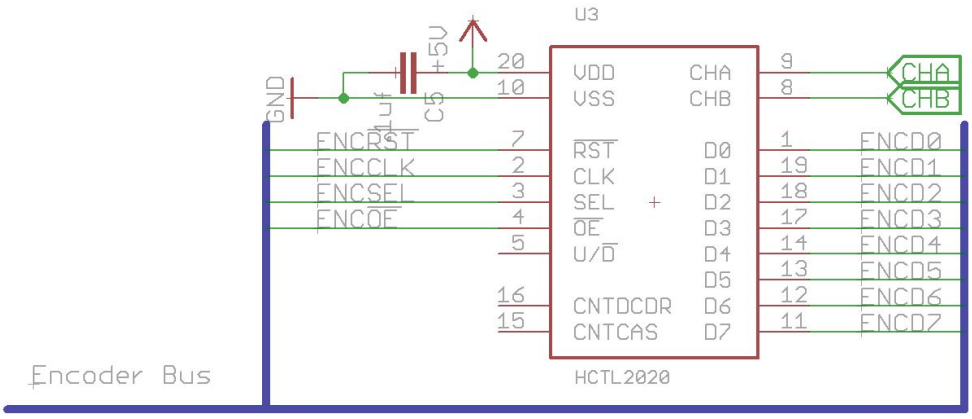


Figure 13: Encoder counter schematic

Chapter 4

Software Design

This section explains the code running on the Control Box, the design of the user interface, and the link between them.

4.1 Code Structure

The Control Box is programmed through the Arduino IDE, using the Arduino libraries. The majority of the code that interacts with the microcontroller peripherals is written in C, and does not make use of Arduino functions. This is done to guarantee that the controller functions as intended, and to allow the students to inspect the driver software to see how the Control Box functions. These functions are explained in Section 4.2.

Students are not expected to concern themselves with the software at the microcontroller level, they are only expected to code the control algorithm, this is explained in greater detail in Section 4.5. They will build the control algorithm from the saturated math functions described in Section 4.3. These functions are intended to use the variables x0-x7 and k0-k7, which are already declared for student use. Students are not

expected to have to declare any of their own variables, or write any of their own functions. Nearly all of the variables the students interact with will be 8 bit. When the hardware supports a higher resolution, for example the 10 bit ADC, the value is either truncated or padded with zeroes so that it appears to the student as an 8-bit value.

The bulk of the controller code is executed inside an ISR, and executes each sample period. The ISR first samples the inputs, then runs the student written control algorithm, and finally updates the H-bridge and DAC outputs before exiting. Code outside the ISR is only responsible for servicing the UI. It writes and reads data from the serial port, and responds to commands from the UI.

4.2 Hardware Drivers

This section will explain the functionality and design of the driver software controlling the sample period, H-bridge, encoder counter, DAC channels, and ADC channels. Students are not intended to have to use or understand these functions; they are called outside of the student written code.

4.2.1 Timer Interrupt

The sampling and control algorithm are run from with a timer interrupt. This timer interrupt is enabled with the Timer1ISRenable function

when the controller is initially powered on or reset. This function first configures the timer by setting the prescaler to 256, which given the Arduino's 16MHz base clock causes each count to be 16us. Next it sets the waveform generation mode to normal, that is, the timer counts up until a compare match at which point the count resets. Next the function sets the default output compare value to 625, or 10ms. Finally, the interrupt source is unmasked, and interrupts are enabled globally with the sei() function. Interrupts will now occur every 10ms until the output compare value is modified by the UI. This ISR calls all the sampling functions, runs the control code, and then updates the outputs of the Control Box.

4.2.2 H-bridge Control

The H-bridge has two functions to control it. The MotorEnable function runs once when the controller is initially powered on or reset. This function sets the INH, IN1, and IN2 pins which control the two half H-bridges to outputs. Next it configures Timer2 to use fast pwm in non-inverting mode. The prescaler is set to 8. Since this is an 8-bit timer in fast pwm mode, a prescaler of 8 will create a frequency of approximately 8kHz. Next the duty cycles of both IN1 and IN2 are set to 0, and INH is set high enabling the H-bridge.

The Motor function is used to change the output voltage of the bridge by modifying the duty cycle of IN1 and IN2. It takes as its argument

an 8 bit signed number. If this number is positive, it will be multiplied by two, and then used to set the duty cycle of IN1 while the duty cycle of IN2 is left at 0. This creates a positive output voltage. The input is doubled because the H-bridge has a range of ±255 bits while the input only has a range of -128 to 127. If the input is negative the opposite happens, and the duty cycle of IN2 is modified while the duty cycle of IN1 is set to 0. This creates a negative output voltage. If the input is 0, both duty cycles are set to 0, and the output voltage will be 0. This function is called once every sample period at the end of the ISR.

4.2.3 Encoder Interface

The encoder has three functions, one to enable it, another to read it, and one more to reset the count. The enable function, DSEEnable, configures all the pins connected to the encoder counter IC as outputs and then sets them high since they are all active low. Next it configures Timer4 to generate a high frequency clock signal for the encoder IC. This is done by setting the timer to CTC mode so the timer value clears on a compare match. The output compare value is set to 1, the prescaler is set to 1, and the waveform generation mode is set to toggle. This causes the timer to trigger an output compare match and toggle the pin every cycle, creating an 8MHz clock source for the encoder IC.

The getDSE function is used to read the encoder position. When called, it sets ENCOE low, then sets ENCSEL low to select the high byte and copies the data on the parallel bus into an intermediate variable, DSEhigh. Next it sets ENCSEL high to select the low byte, and copies the data from the parallel bus into an intermediate variable DSElow. Next ENCOE is set back to high, and ENChigh and ENClow are combined to form the 16 bit position variable that is returned by the function.

The reset function, resetDSE, is used to clear the encoder count. This makes the current motor shaft position the new zero position. The function simply sets the ENCRST pin low, then high again immediately after, resetting the encoder count.

4.2.4 DAC Control

The DAC channels share one function that enables all three channels, but have individual functions for writing to each DAC channel. The enable function, DACenable, sets the 8 bit parallel bus and the control signals for each channel as output. Then it sets the active low control lines high. The levels of the pins on the parallel bus are not set in the enable function.

The three functions for writing to a DAC channel are qmout, Vposout, and Vxout. Vposout writes to the position voltage channel, qmout writes to the motor voltage channel, and Vxout writes to the auxiliary

channel. Even though the DACs are 12 bit, they should appear to the students as 8 bit devices, and take an 8-bit value as their argument. This 8 bit signed number is added to 128 to convert it to an unsigned number. Then it is shifted left by four to convert it to a 12 bit unsigned number, and split into two bytes so it can be sent over the 8 bit parallel bus to the DAC. First the MSB is put onto the parallel bus. CSMSB and WR are both set low, then high to latch in the MSB. Next the LSB is put onto the parallel bus. CSLSB and WR are both set low, then high to latch in the LSB. Now LDAC is set low, then high to latch the MSB and LSB into the DAC, updating the output voltage.

4.2.5 ADC Control

Each ADC channel has its own read function. The channels do not require an enable function, as the ADC is built in to the microcontroller and is already initialized by the Arduino libraries. There are three main ADC read functions, getqstep which returns the step input voltage, getqpot which returns the auxiliary analog input voltage, and getVcurrent which returns the voltage output by the hall effect current sensor. These functions all use the 10-bit ADC. As with the DAC channels, these functions should appear to the students to be 8 bit. To do this, the number read from the ADC is first shifted right by two to convert from 10 bit to 8

bit. Then it has 128 subtracted from it to convert from an unsigned number to a signed number. This signed 8-bit result is returned by the function.

There are two additional ADC functions, getVcurrent1 and getVcurrent2, for reading the half H-bridge current sensors. These functions are implemented but unused.

4.3 Functions for Control Algorithm

This section will explain the various functions created to assist the students in writing their control algorithms. Subsection 4.3.1 explains the saturated math functions, while 4.3.2 explains the 16-bit integrator.

4.3.1 Saturated Math Functions

There are four saturated math functions available for student use. They all have 8 bit signed numbers as their arguments, and return 8 bit numbers. This is done to maintain the ability to use only 8 bit numbers to implement the control algorithm. One of the functions performs 8-bit addition, and the other three perform multiplication. Each function performs its specified mathematical operation in addition to a saturation check. This prevents wrap around and overflow for the students. While the multiplication functions can multiply any two signed 8 bit numbers, they are specifically intended to multiply a control variable by a control constant. There are three different multiplication functions to allow the

student to implement constants with different numerical ranges, including fractional numbers.

The saturated addition function, sadd, takes two signed 8-bit numbers as its arguments. They are added together and stored temporarily in a 16-bit variable, temp, to prevent overflow. If temp is greater than 127, 127 is returned. If temp is less than -128, -128 is returned. If neither of these conditions is met, the addition did not overflow and temp is returned as an 8-bit signed number. This function is also used for subtraction by first negating one of the arguments.

The first multiplication function is smuli, which stands for saturated multiplication by an integer. It is nearly identical to sadd, but implements multiplication instead of addition. This function is intended to multiply one of the control variables by an integer constant, and is able to represent constants in the range of -128 to 127. It does this by multiplying the two input arguments and storing the result in the temporary variable temp. As with sadd, the result is checked and the appropriate value is returned.

The next multiplication function is smulf, which stands for saturated multiplication by a fraction. It is intended to multiply one of the control variables by a fractional constant, and is able to represent constants in the range of -128/128 to 127/128. It does this by multiplying the two input arguments, then dividing by 128 to implement the fraction, and returning the result. This function does not perform a saturation check, as the result

cannot overflow. The largest magnitude constant it can represent is 1, which when multiplied with the control variable is guaranteed to be in the allowable range for an 8 bit signed number.

The last multiplication function is smula. It implements multiplication by a combination integer and fractional number. That is, the constant will have both integer and fractional components. It can represent a constant ranging from -128/8 to 127/8.This is done by multiplying the two input arguments together, and then dividing by 8 before storing the result in the temporary variable temp. The result of this operation can overflow, so just as with smuli a saturation check is performed and the appropriate result is returned.

4.3.2 16-bit Integrator

The last function for student use, isadd, is specifically for experiment 4. Its purpose is to maintain the 16-bit integrator sum for PID control. It differs from the other functions in that one of its arguments is a 16-bit signed number. The other two arguments are the 8-bit signed number being added to the 16-bit sum, and the 16-bit signed number representing the integrator clamps, ilimit. The integrator clamps are included to allow the student to limit the integrator windup. The function casts both the 16-bit sum and the 8-bit input, usually the control error, to 32 bit numbers before adding them together and storing the result in the

temporary variable temp. Instead of being checked against the limits of a 16-bit signed number, temp is checked against the integrator clamp ilimit which is itself guaranteed to be within the limits of a 16-bit signed number. If temp is greater then ilimit, ilimit is returned. If temp is less than ilimit, negative ilimit is returned. If neither condition is met, the magnitude of temp must be less than ilimit, so temp is returned as a 16-bit number. This allows the student's algorithm to maintain a 16-bit sum that is clamped by the integrator limits.

4.4 User Interface

The user interface was developed as a replacement for the serial connection the old controller used. The old serial connection worked fine for displaying the control variables, but it only worked in one direction. The controller could only display values on the computer; the computer couldn't send anything to the controller. In order to change a constant or modify the sample period, the student had to first reset the controller. They could then set the constant and then start to run the control code. This made tuning a controller a very slow process. The user interface attempts to address this issue by creating a program that supports bidirectional communication with the computer. It can then be used for anything from setting constants and displaying variables, to disabling the motor.

The UI was designed with Qt Creator, and is coded in C++. The UI is based on a simple terminal demo program included with Qt Creator [11]. However, instead of the standard terminal text field the interface has been customized to control the hardware in this specific controller. It is capable of displaying all the variables and constants declared for the student, modifying the constants and sample period while the control algorithm is running, and many other functions. Figure 14 shows the current version of the UI, updated for experiment 4.

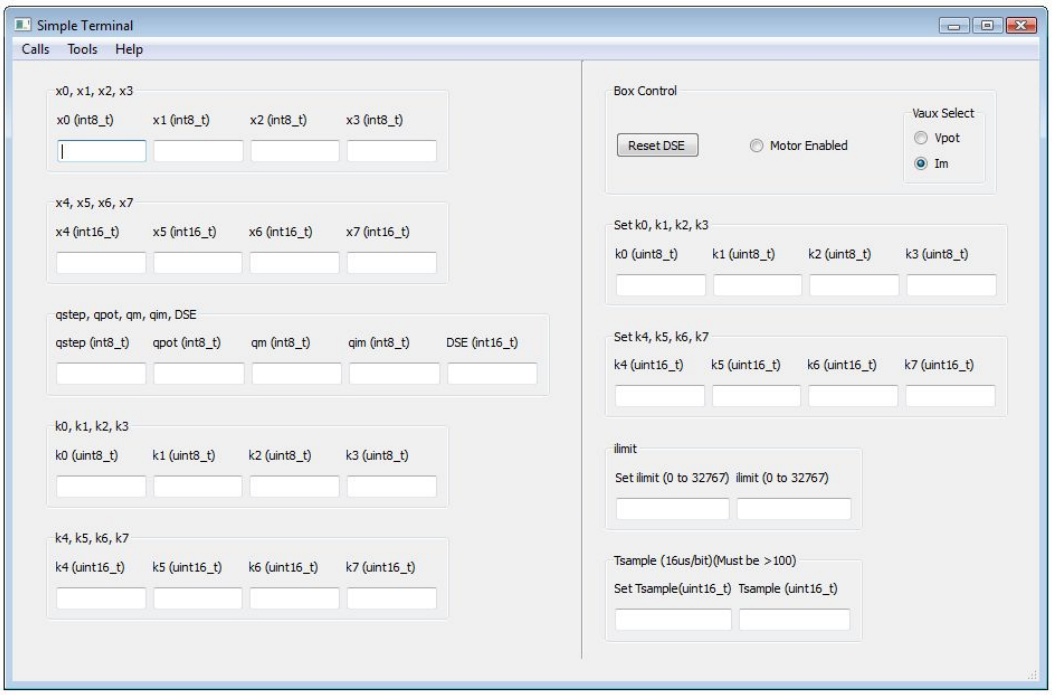


Figure 14: EE472 digital Control Box UI

The UI is divided into two parts, the left side displays variables sent from the controller to the computer, while the right side allows the students

to interact with the Control Box. The top left of the UI displays x0-x7, these are the pre-declared variables students will use for intermediate values in their control algorithms. Constant access to these variables makes debugging the control algorithm much simpler for the students. The middle line of the left half of the UI displays all the inputs and outputs for the Control Box; each value in this line has physical significance. These include the step input voltage, auxiliary analog input voltage, motor voltage, motor current, and motor shaft position. The bottom left of the UI displays the current values of the control constants k0-k7.

The right side of the UI is dedicated to controlling the Control Box. The top right includes two key new features. First is the “Reset DSE” button. When clicked this causes the Control Box to call the resetDSE function. This will make the current motor shaft position the new zero position. With the old hardware, this could only be done by completely resetting the controller. Now it can be done even while the control algorithm is running. Next is the “Motor Enabled” radio button. While selected, the H-bridge will be enabled. While deselected the H-bridge is disabled, this lets the students test their algorithm without risk of losing control of the electromechanical apparatus. With the old system, the student would have to physically unplug the motor if they wanted to test their algorithm.

The next section of the right side of the UI is for setting the value of the control constants k0-k7. This lets the students tune their algorithm even while the controller is running and the motor is enabled. This allows the students to immediately see the effects of modifying these constants. Previously, the controller had to be completely reset to change even one of the gains. Tuning a controller can now be completed in minutes instead of hours. The last part of the UI is used to set and monitor the sample rate and integrator clamps. These are set the same way as the control constants, but shouldn't need to be modified as frequently. Easy access to the sample rate makes it simple for the students to see the effects of modifying the sample rate, both on the response of the controller and the scaling of their time dependant gains.

The last part of the UI, shown in Figure 15, consists of the drop down menu and secondary window used to configure the serial port and connect to the controller. This window can modify most of the parameters of the serial connection such as COM port, baud rate, and data bits. However, the UI will typically recognize the controller on initialization and default to the right settings.

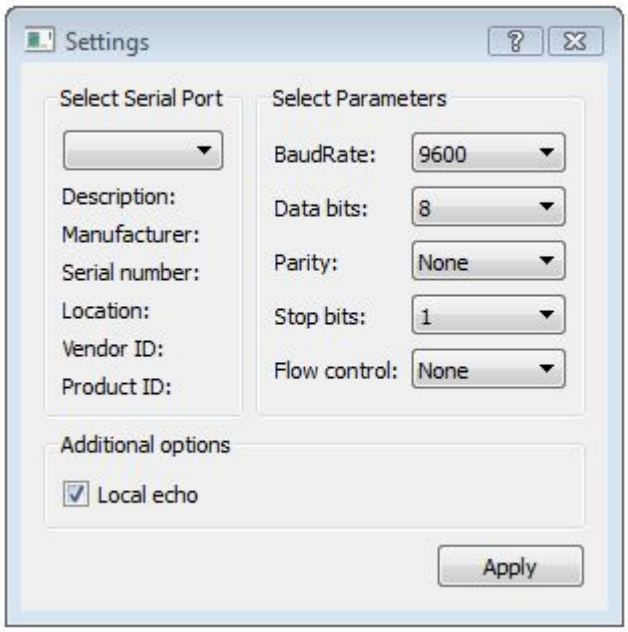


Figure 15: Serial port configuration window

4.5 Communication Between UI and Microcontroller

Implementation of the communication between the microcontroller and computer was difficult. The Qt serial library had many peculiarities, and was very difficult to work with. Among these are the apparent inability to transmit any variables other than strings, and the inability to view received data without actually reading that data out of the received bytes buffer.

Communication from the microcontroller to the computer is completely separate from communication from the computer to the microcontroller. There is no handshake, no error checking, no retransmission of bad data, and no other interaction between sending and receiving data. Communication in each direction uses a simple packet

structure consisting of three start characters, and then the data being sent. This method does occasionally allow errors to be transmitted and displayed, but they are typically quickly overwritten. For lab use, the current method has proven plenty robust.

4.5.1 Microcontroller to UI Communication

To transmit data to the computer, the microcontroller first makes a copy of every variable being sent. This step ensures that all the data being displayed by the computer is from the correct sample period. Without this step, it is possible that the timer ISR could run in the middle of a transmission, modifying some of the variables being sent. This could make it appear to the student that their algorithm was not working properly, as the values would be incorrect assuming they had come from the same sample period. Next the microcontroller sends the three start characters. Theses are arbitrary but fixed, the microcontroller will always send the same three characters, and the UI will always expect the same three characters. After the start characters, the microcontroller will send all the data to be displayed by the UI. This includes k0-k7, x0-x7, Tsample, ilimit, qstep, qpot, Imotor, Vmotor, and DSE. All the values are sent in every packet, so the packet will always have the same structure, and the UI can always tell which variable is which. This packet contains 3 start bytes, and

34 data bytes, for a total of 37 bytes. This transmission is repeated every 50ms.

When receiving data from the microcontroller, the UI first checks the number of bytes in the received bytes buffer. If this number is less than 37, a whole packet has not been sent and the UI does nothing. If the number of received bytes is greater than or equal to 37, then the UI knows a whole packet was received. Next, the UI reads the first 3 bytes from the buffer and checks to see if they match the start bytes. If they do match, the UI continues to read the next 34 bytes of data. These bytes are then cast to the proper variable type, converted to character strings, and displayed in the appropriate field on the UI. However, if the start bytes do not match, then the UI must be reading the middle of a packet. In this case the UI has become misaligned from the packets, and must regain alignment to read the proper data. It does this by reading an additional 2 bytes of data each time the start bytes have a mismatch. In this way, the position in the packet in which the UI expects to find the start bytes is incremented by 5 bytes each time a new packet is received. After 37 iterations of this, the UI will have checked all possible combinations of start bytes and regained alignment to the packet. This works because the product of 5 and 37 is also the LCM of 5 and 37. If this were not true the UI would never be able to check certain sequences of start bytes, instead it would cycle through the same series of start bytes while always missing the actual start bytes.

4.5.2 UI to Microcontroller Communication

Data is sent from the UI to the microcontroller in 8 byte packets. The first three bytes, bytes 0-2, are the start bytes. The fourth byte, byte 3, is an action code telling the microcontroller what to do; these action codes are listed in Table 1. The last four bytes, bytes 4-7, contain the data being sent. Bytes 4 and 5 represent the high byte of the data being sent, and bytes 6 and 7 represent the low byte of the data being sent. Many of the action codes do not require data to be sent, in this case the last four bytes will contain four zero characters.

Action Code (ASCII)	Action
0	Reset DSE
1	Enable Motor
2	Disable Motor
3	Connect Aux DAC to Aux Analog Input
4	Connect Aux DAC to Motor Current
5	Set ilimit
E	Set Tsample
6	Set k4
7-9, A, B-D	Set k0-k2, k3, k5-k7

Table 1: Serial Communication Action Codes

When the user clicks a button on the UI, or presses return to enter data into a field, a function specific to that action is called. This function first transmits the start bytes and the action code. If there is no data to be sent for that action code, four zero characters are sent. If there is data to be sent for that action code, the data is copied out of its field as a QString. This QString is converted first to a quint, and then to an array of characters that can be sent by Qt's serial write function. However, since the serial write function can only send arrays of characters, it cannot send a null character, as this is already used to mark the end of a character array. This means that nonzero values can be sent in this manner, but zero values will be mistaken for null characters. To circumvent this issue, a single byte of data is sent as two characters. The first character determines whether or not the data being sent is zero, and the second character is the actual data. If the data being sent is zero, the first byte will be a zero character, and the second byte will be a zero character. If the data being sent is nonzero, then the first byte will be a 1 character, and the second byte will contain the actual data converted to a character as described above. In this way, the UI is able to transmit both zero and nonzero values to the microcontroller, while only being able to send character strings. For a two-byte number, this process is repeated for each byte.

The microcontroller checks the received bytes buffer for packets every 50 ms. When the microcontroller receives a packet from the UI, it first checks to make sure that the proper number of bytes has been received. If it has not, the microcontroller simply empties the buffer. If the buffer does contain at least one packet of bytes, it copies all 8 bytes from the received bytes buffer into a second buffer, serialbuffer. Next the three start bytes are checked. If they don't match, the packet is discarded. If they do match, then the action code is checked and the appropriate action is taken. If data was sent with the action code, the data is reassembled from characters into the proper variable type for that action code. Then according to the action code, one of the variables in the controller is set to the transmitted data.

4.6 Student Controller Design

The only part of the code that must be written by the student in lab is the control algorithm. The place for this code is very clearly marked in the Arduino file. They will start by deciding how to use x0-x7 and k0-k7. Next they will formulate the control error from the sampled values of the input and feedback signals. Next they use the saturated math functions to calculate the motor output voltage, and store that result in qm. This is all that is required of the student. The provided code already performs sampling of all the inputs and updating of all the outputs. After writing their

algorithm, the students should use the UI to verify that their algorithm works properly before enabling the motor and actually tuning their controller for the electromechanical apparatus. The students will occasionally be responsible for converting between 8 and 16 bit numbers. For example, the encoder value is natively a 16-bit value, which must be converted to 8 bits before the error can be formed. The exact process for writing the algorithm depends greatly on the experiment. Both of these issues are discussed in great detail in the lab manuals.

Chapter 5

Controller Verification

Initially, a prototype Control Box was constructed. This iteration was completed during the summer of 2014. Its overall performance was briefly tested and found to be sufficient for the current lab experiments. Immediately following this, it was decided that the new Control Box should be ready for integration into the EE472 lab being offered in Fall Quarter of 2014. The fabrication of these boxes is described in Chapter 6, and their performance in lab is described in this chapter.

The new batch of Control Boxes was completed in time for the first lab experiments of the quarter. They were able to perform all of their intended functions even under student operation with no faults or failures. The controllers survived multiple instances of misuse including bugs in student written code and improper connection. Each lab group was able to successfully complete every lab with the new Control Boxes. However, over the course of the quarter it was found that the Control Boxes did have some undesirable characteristics. Most of these issues were expected, and were not serious enough to impair student use. These issues are explored in the following sections.

5.1 A/D and D/A Performance

The ADC and DAC circuits were intended to convert perfectly between an 8 bit signed number and a ±12V signal range. Neither circuit was actually able to implement this conversion perfectly. Power Supply voltage error, component tolerances, and inaccurate tuning all contribute to the conversion error.

The input voltage to output value relationship for the ADC circuit is shown in Figure 16, along with the ideal relationship. This graph shows two main types of error present in the conversion, slope and offset. The presence of offset error means that the conversion function does not intersect the origin. So a 0V signal is not converted to a zero value. While component tolerances make a small contribution to this source of error, it is primarily attributable to the fact that the +12V rail of the instrumentation power supply is actually slightly below +12V when loaded. This voltage is used to add a positive offset to the initial ±12V signal range, shifting it to a 0V to 12V range. However, if the actual supply is less than +12V, slightly less offset will be added to the signal. This problem can largely be corrected by adjusting the supply voltage with the voltage adjust potentiometer on the power supply. The error in the slope of the conversion function is primarily due to the tolerances of the resistors in the voltage divider that divides the 0V to 12V signal down to the 0V to 5V

range of the ADC. This source of error can only be corrected by modifying the circuit to add a potentiometer in series with one of these resistors. This would allow the user to tune the slope of the conversion function to better match the ideal relationship.

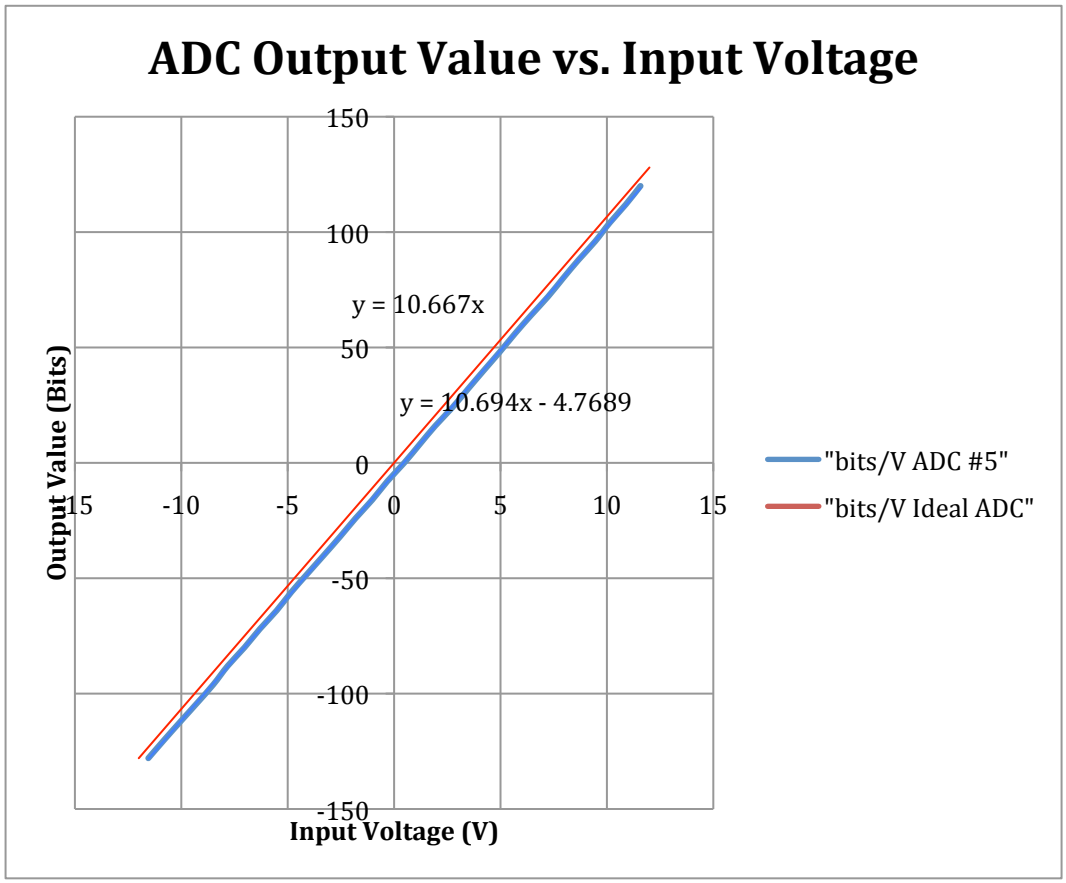


Figure 16: Relationship between ADC input voltage and output value

The input value to output voltage relationship for the DAC circuit is shown in Figure 17, along with the ideal relationship. The difference between actual function and ideal function is much smaller for the DAC circuit. The DAC circuit should have no offset error, this happens because

the actual DAC output before being scaled to the ±12V range is already bipolar. It only has to be amplified from a ±5V range to a ±12V range. There is no offset added to the output of the DAC so no offset error is present. Additionally, it was expected that the internal voltage reference of the DAC would drift with time, as it did in the old controllers. To correct for this effect, potentiometers were included in series with the amplifiers feedback resistors to adjust the gain. This allows the user to tune the gain of the DAC circuit to very closely match the slope of the ideal function.

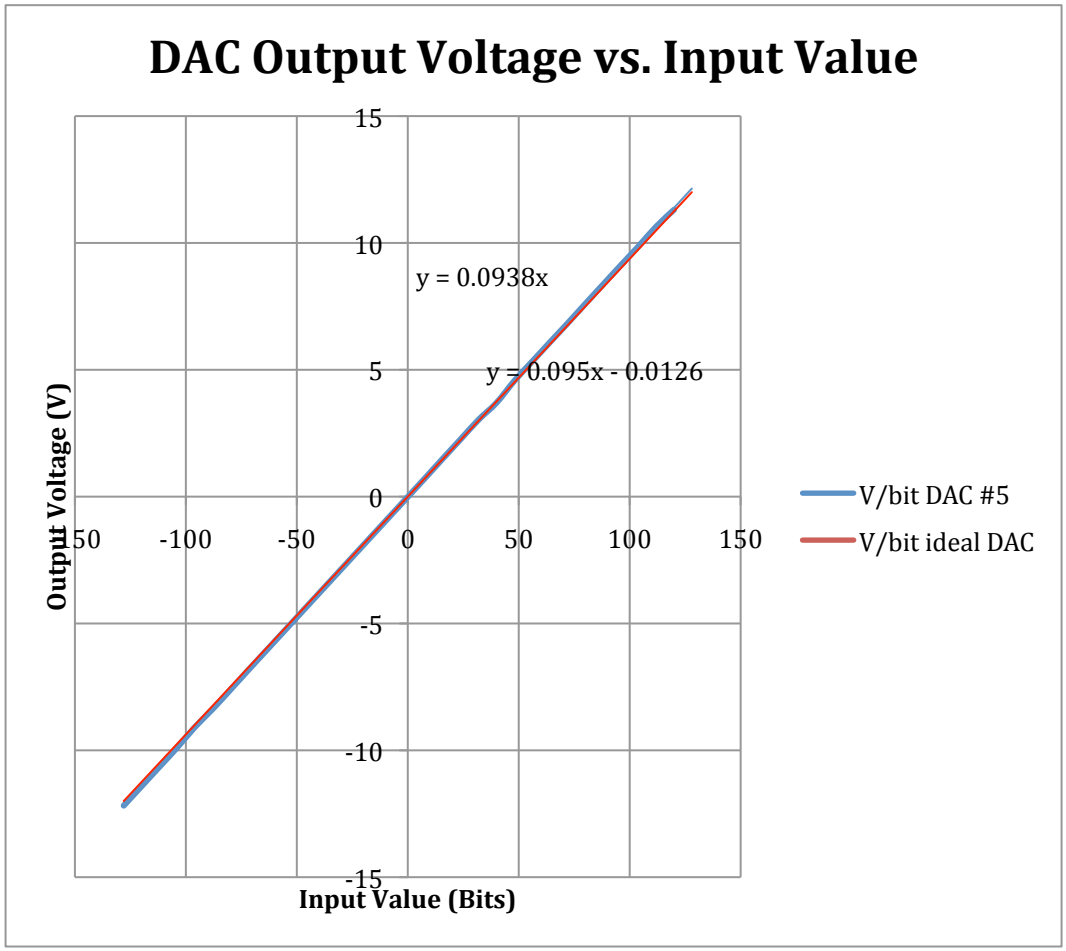


Figure 17: Relationship between DAC output voltage and input value

While both circuits will always show some divergence from their ideal conversion functions, they can both be tuned to be reasonably close to ideal. The current implementation for each circuit has proven sufficient for the EE472 laboratory experiments. However, the omission of potentiometers to tune the slope and offset of the ADC conversion function is an oversight that will likely need to be corrected.

5.2 Sample Rate Limitations

While the timer interrupt controlling the sample rate can be configured to trigger at any sample rate in increments of 16us, the ISR cannot actually execute that quickly. Setting the sample rate too low has the potential to break two different parts of the controller's functionality. As the sample period is decreased, the ISR is executed more and more frequently. The first feature to break down is communication with the UI. Once the sample rate has been increased sufficiently it will begin to interrupt communication to the UI, causing packets to be dropped. When this happens, the UI becomes unresponsive and the sample period cannot be increased, as it is set through the UI. The box must be reset to reset the sample period. Fortunately, even in this case the code within the ISR executes properly and control of the electromechanical apparatus is maintained. However, continuing to decrease the sample period will

eventually cause the timer interrupt to trigger during the ISR. In this case, the first ISR will execute, return to the main program, and then immediately enter another ISR. Eventually some of the interrupts will be overwritten and ignored, causing the sample period to become unstable. This doesn't completely lose the ability to control the motor, but the intended control algorithm will not be faithfully implemented.

These situations have been avoided by setting a lower bound on the sample period in the microcontroller. The sample period is limited to approximately 1.6ms and cannot be set lower. This completely prevents students from encountering either of the two potential issues discussed above. Fortunately, 1.6ms is a far lower sample period than is required by the EE472 experiments.

5.3 Step Input Quality

The quality of the step input signal is critical to obtaining a useful step response for the lab experiments. If the step input signal contains too much noise, the step response will be corrupted and the students will be left without an acceptable way to verify their controller designs. The largest source of step input noise is mechanical switch bounce from the step input switch. This causes the step input voltage to swing rapidly between 0V and the step input voltage. A worst-case example of step input signal switch bounce is shown in Figure 18.

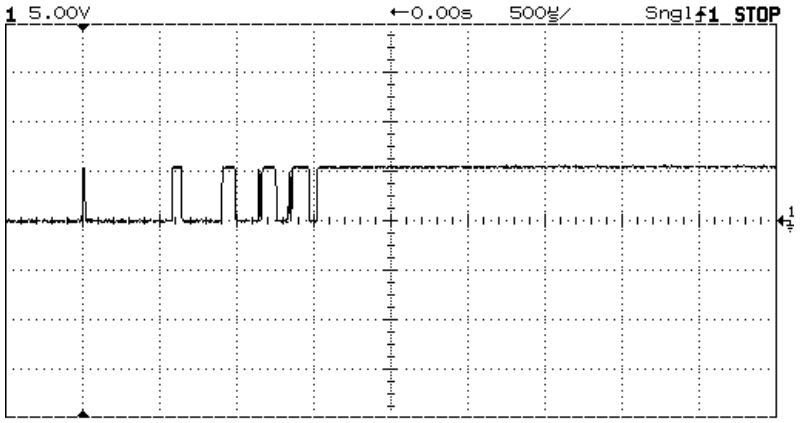


Figure 18: Worst-case step input signal switch bounce

These signals do contain significant switching noise. However, in all cases the noise is limited to within less than 1.6 ms of the initial voltage step. Because this time is lower than the minimum controller sample period, the sampled step input signal will not show this noise. At most the sampled step input signal can only contain one of the transitions generated by the noise in the analog step input signal. This noise cannot affect the step response of the controller.

5.4 Motor Power Supply Limitations

The motor power supply has no issues powering the motor during position control. However, it was observed that when running the motor open loop or with a velocity controller that the power supply would occasionally shut down. Upon further inspection it was determined that the power supply's overvoltage protection was being triggered causing it to temporarily disable the output voltage.

This happens because the power supply is unable to dissipate the motor's stored rotational kinetic energy during a sharp change in velocity. Attempting a sharp change in motor velocity essentially implements regenerative braking. This causes the motor to temporarily behave as a generator, driving power back into the supply and triggering it's over voltage protection.

This is not an issue for the current EE472 experiments. The motor is never able to achieve high enough velocity that dissipating the energy is problematic. However, if the Control Box were ever to be used for a velocity control experiment, this issue would need to be addressed. One possible solution is the addition of a shunt regulator between the power supply and the H-bridge to limit the supply voltage below the over voltage protection limit.

Chapter 6

Enclosure Fabrication

The enclosure for the Control Box consists of two parts, both made from acrylic, the top plate to which all of the components are mounted and the bottom shell. The bottom shell mounts to the top plate with 4 aluminum standoffs. The bottom shell is easily removable, allowing easy access to service the Control Boxes. Both the top plate and bottom shell were machined and assembled by hand. Figure 19 shows a complete Control Box, Figure 20 shows the top plate with components installed, and Figure 21 shows the bottom shell.

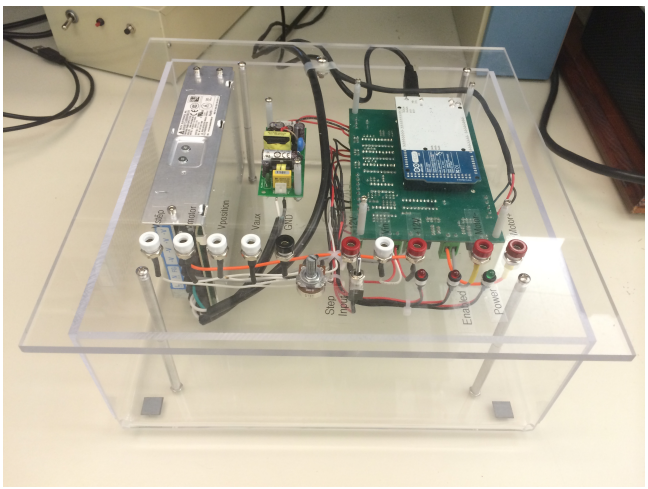


Figure 19: Complete Control Box

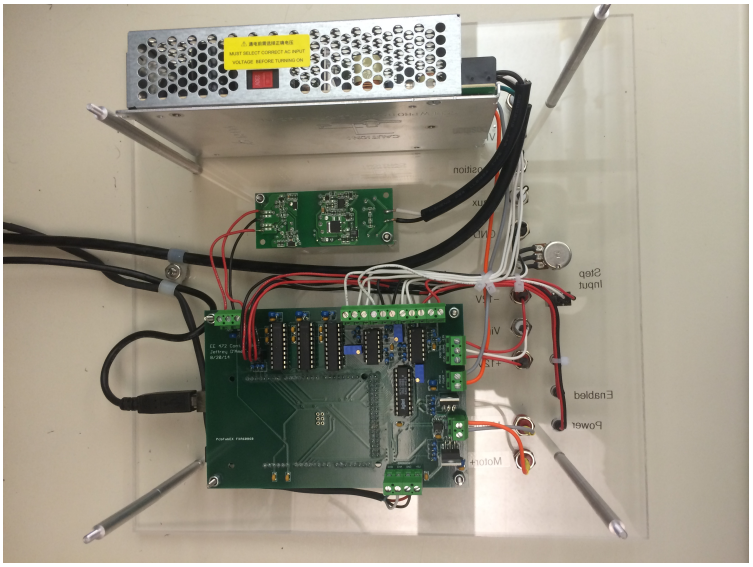


Figure 20: Control Box top plate

Figure 21: Control Box bottom shell

Chapter 7

Future Work

While the current versions of the digital controller hardware and software are sufficient for labs 1-4, there are a number of improvements that could be made. These improvements and additions improve the functionality of the controller, add features, and improve the user experience.

7.1 Motor Velocity Estimation

Currently, the control box does not support velocity feedback. The electromechanical apparatus does have a tachometer that produces an analog voltage. However, its signal range is not compatible with the auxiliary analog input on the Control Box. Velocity can also be estimated digitally from the encoder position with various methods. The simplest of these is the backwards difference approximation, though this is certainly not the most accurate or effective.

To support such a feature, the Arduino code would need to be modified to declare a velocity variable, implement the estimation algorithm, and add the velocity variable to the packet being sent to the UI. The UI

would need to be modified to accept the new packet structure, and to display the velocity data once it had been received.

Estimation of velocity from discrete time position data can actually be a fairly involved process. This has the potential to be part of a lab, or even its own lab. To support this, the velocity estimator would be included in the section of the code that the students are expected to modify. The additional DAC channel could be modified to display the velocity estimate on the oscilloscope at the same time as the position from the encoder. The velocity signal from the DAC and the tachometer voltage could both be analyzed by the spectrum analyzer for frequency content to show how the differentiator amplifies noise.

7.2 Experiment 5, Sampling Effects

EE472 Lab Experiment 5 [7] is not currently supported. This would likely require only small modifications to the student code section of the Arduino code, to pass variables through from input to output with no additional calculation or modification. The lab manual for experiment 5 would also need to be rewritten to support the new procedure for the new hardware.

7.3 User Interface Improvements

Various modifications could be made to improve the user interface. It currently only supports the basic features required to support labs 1-4, and could be modified to include expanded options for controlling and observing the operation of the Control Box. Additionally, the look and feel of the user interface could be modified or improved to streamline the user experience and simplify use for the students.

7.4 ADC Gain Adjustment

As explained in Section 5.1, the ADC transfer function differs somewhat from the intended function, and does not include potentiometers for adjustment. Both the gain and the offset portions of the circuit could be modified to be adjustable via potentiometers.

This would require a slight redesign of the circuit to include the potentiometers, and selection of the resistor and potentiometer values. The boards would then need to be reworked to include the new components, and each ADC channel would need to be tuned to have the correct transfer function.

7.5 Support for Matlab

Section 2.4 claimed that compatibility with Matlab was a requirement for the hardware. Because an Arduino was selected as the

microcontroller, the microcontroller is compatible with Matlab. However, the supporting hardware is not compatible. Matlab has no way to drive the DAC channels and H-bridge, or read the encoder counter. For true connectivity with Matlab, it would need to be investigated whether the current hardware drivers can be modified to support Matlab connectivity, or new drivers need to be written to allow Matlab access to the hardware.

Chapter 8

Summary and Lessons Learned

Design requirements for a new digital controller for the EE472 Digital Controls Lab were selected. A controller was then built to those specifications, and duplicated for each lab bench. Software was written for the Control Boxes with the Arduino IDE. A UI was developed to communicate with the Control Boxes using Qt Creator. The Control Boxes were then tested by a whole quarter of student use in the EE472 lab and proved to be robust and effective. Students were able to quickly and easily complete the labs.

The Control Boxes had some limitations during the first quarter, including power supply overvoltage shutdown and non-ideal ADC performance. Neither of these issues disrupted the student's ability to complete the labs.

There are a number of improvements that could be made to the hardware and software to support current and future lab activities. Some of these options were explored in Chapter 8. Without any additions or improvements, the new Control Boxes are ready to continue being used for Experiments 1-4 of the EE472 Digital Controls Lab.

Bibliography

[1] MacCarley, Arthur C. *EE342 Control Systems Laboratory, Experiments and Supporting Information*. San Luis Obispo: California Polytechnic State U, n.d. PDF.

[2] MacCarley, Arthur C. *EE472 Digital Control Systems Laboratory Preliminaries*. San Luis Obispo: California Polytechnic State U, 2012. PDF. Version 12.

[3] MacCarley, Arthur C. *EE472 Digital Control Systems Laboratory, Experiment 1 System Modeling*. San Luis Obispo: California Polytechnic State U, n.d. PDF.

[4] MacCarley, Arthur C. *EE472 Digital Control Systems Laboratory, Experiment 2 Digital Proportional Control*. San Luis Obispo: California Polytechnic State U, n.d. PDF.

[5] MacCarley, Arthur C. *EE472 Digital Control Systems Laboratory, Experiment 3 Digital Lead Compensation*. San Luis Obispo: California Polytechnic State U, n.d. PDF.

[6] MacCarley, Arthur C. *EE472 Digital Control Systems Laboratory, Experiment 4 Digital PID Control*. San Luis Obispo: California Polytechnic State U, n.d. PDF.

[7] MacCarley, Arthur C. *EE472 Digital Control Systems Laboratory, Experiment 5 Sampling Phenomena*. San Luis Obispo: California Polytechnic State U, n.d. PDF.

[8] Costas, Lucia, Jose Farina, and Juan J. Rodriguez-Andina. *A Configurable Framework for the Education of Digital Electronic Control Systems*. Tech. Pontevedra: U of Vigo, 2009. *IEEE Xplore*. Web. 25 Nov. 2014.

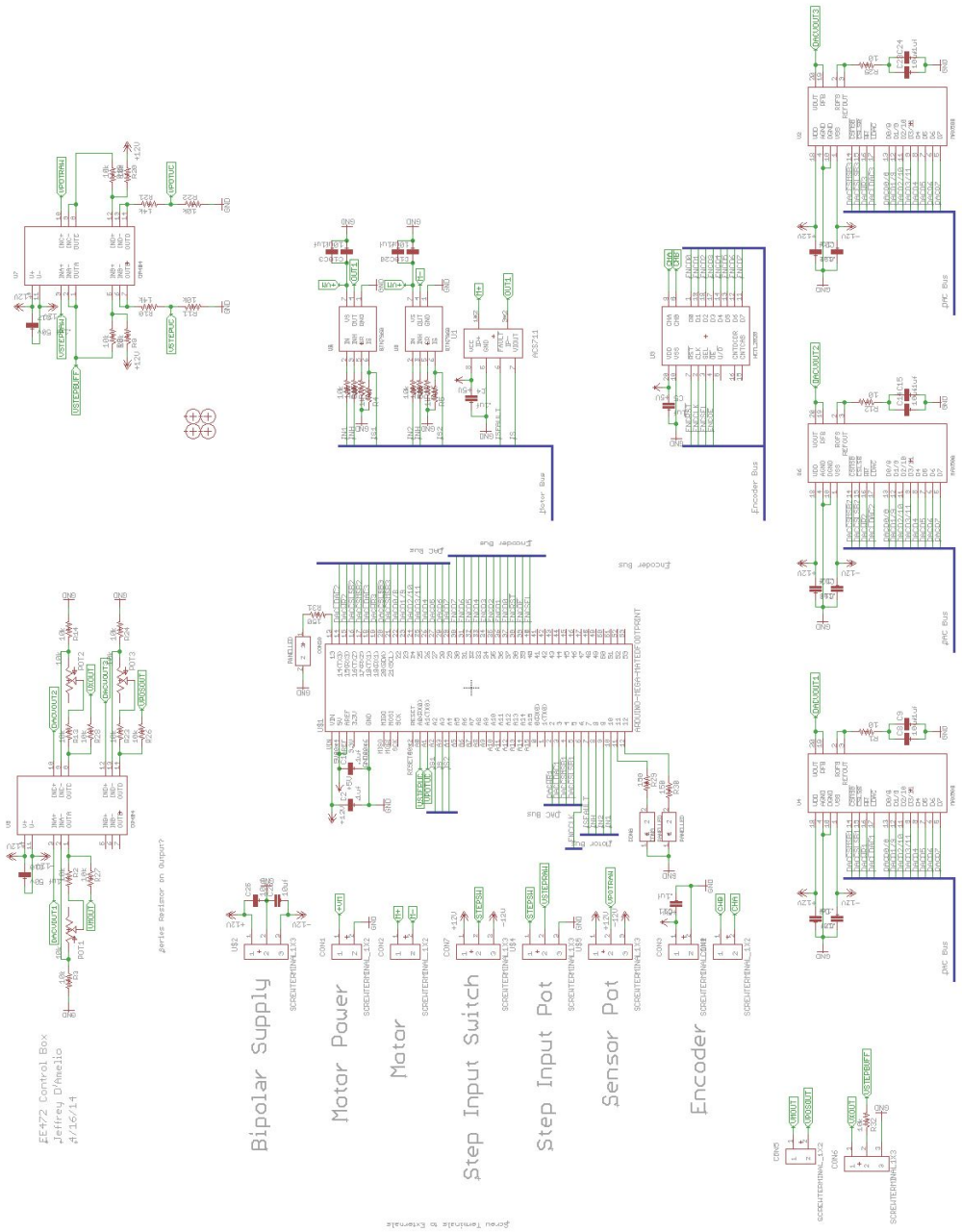
[9] Zilouchian, Ali. *The Development of a Real-time Digital Computer Control Laboratory for Electrical Engineering Education*. Tech. Boca Raton: Florida Atlantic U, 1992. *IEEE Xplore*. Web. 25 Nov. 2014.

[10] Shoults, Raymond R., and Edmundo Barrera-Cardiel. *Use of a Graphical User Interface Approach for Digital and Physical Simulation in Power Systems Control Education: Application to an HVDC Transmission System Model*. Tech. 4th ed. Vol. 7. Arlington: U of Texas at Arlington, 1992. *IEEE Xplore*. Web. 25 Nov. 2014.

[11] *Qt Creator*. Computer software. *Qt Project*. Vers. 5.3.1. Digia PLC, 2014. Web. 1 Aug. 2014. <http://qt-project.org>.

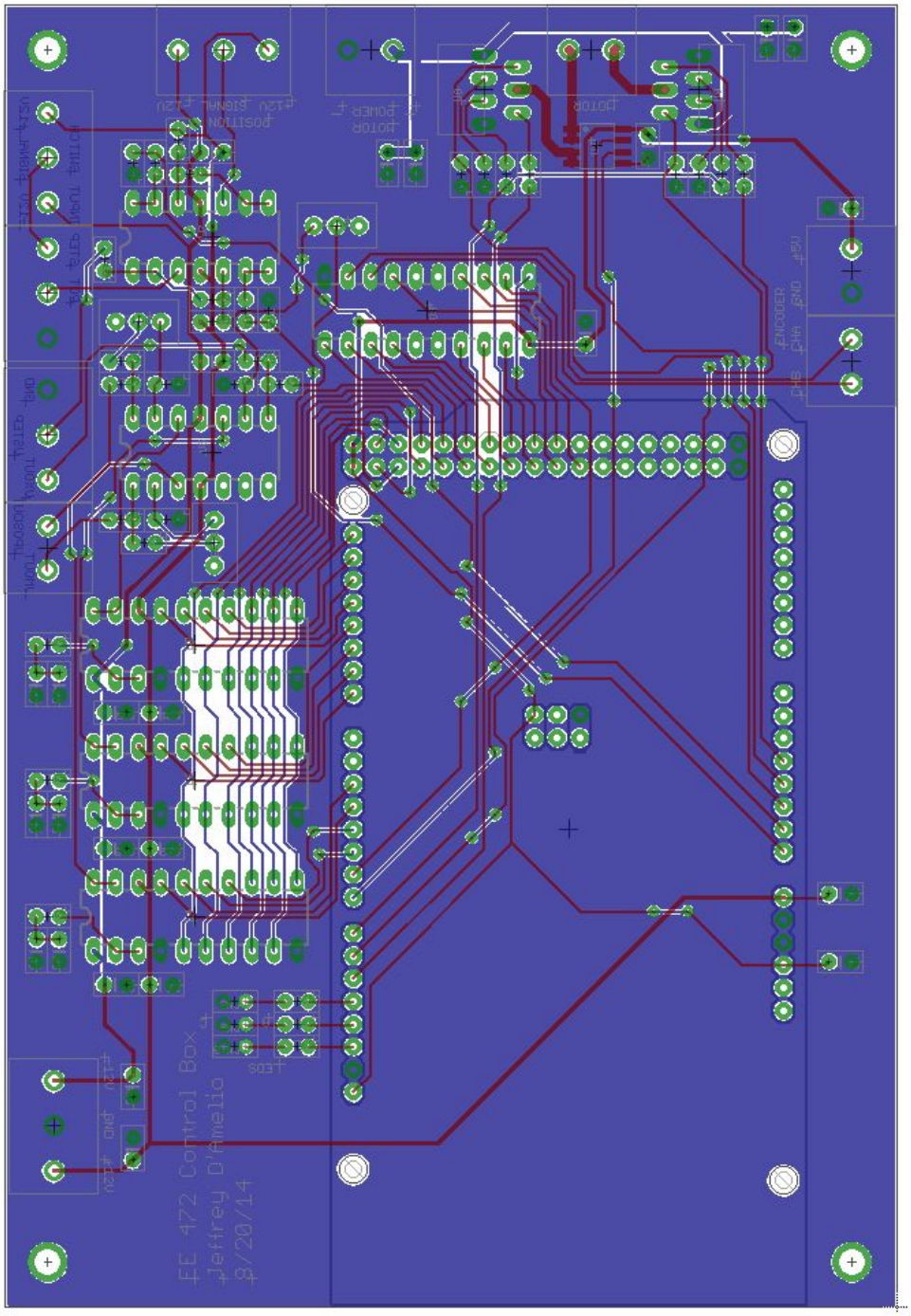
Appendix A

Control Board Circuit Schematic



Appendix B

Control Board Layout



Appendix C

Control Board Bill of Materials

Qty	Value	Package	Description
16	.1uf	CAP-PTH-SMA	Digikey Part Number: 399-9776-ND
2	.1uf 50v	CAP-PTH-SMA	Digikey Part Number: 399-9776-ND
3	10	AXIAL-0.4	Digikey Part Number: 10.0XBK-ND
3	10k	TRIM_POT	Digikey Part Number: T93YA-10K-ND
20	10k	AXIAL-0.4	Digikey Part Number: 10.0KXBK-ND
7	10uf	CAP-PTH-SMA	Digikey Part Number: 445-8465-ND
2	14k	AXIAL-0.4	Digikey Part Number: 14.0KXBK-ND
3	150	AXIAL-0.4	Digikey Part Number: 150XBK-ND
2	1k	AXIAL-0.4	Digikey Part Number: 1.00KXBK-ND
2	50k	AXIAL-0.4	Digikey Part Number: 49.9KXBK-ND
1	ACS711	8SOIC	Digikey Part Number: 620-1373-1-ND
2	BTN7960	TO220-7	Digikey Part Number: BTN7960P-ND
1	HCTL2020	DIP20	
3	MAX508	DIP20	Digikey Part Number: MAX508BCPP+-ND
2	OP484	DIP14	Digikey Part Number: OP484FPZ-ND
3	PANELLED	PINHEADER(1	Digikey Part Number: S7071-ND
5	SCREWTERMINAL1X3	SCREWTERM	Digikey Part Number: ED1632-ND
5	SCREWTERMINAL_1X2	SCREWTERM	Digikey Part Number: ED1631-ND
4	IC Socket	20pin DIP	Digikey Part Number: 3M5477-ND
2	IC Socket	14pin DIP	Digikey Part Number: 3M5474-ND
5	Male Pins .100"	Male Pin Head	Digikey Part Number: S1012EC-40-ND
1	Board	PCB	pcbfabexpress

Appendix D

Control Box Bill of Materials

Qty	Value	Device	Description
1	+/-12V	Instrumentation Supply	Digikey Part Number: 1470-1163-ND
1	+12V, 12.5A	Motor Supply	Digikey Part Number: 1145-1071-ND
1		Grounded Power Cable	Digikey Part Number: AE9893-ND
1		USB A to B	Digikey Part Number: Q364-ND
5	white	Banana Jack	Digikey Part Number: J150-ND
4	red	Banana Jack	Digikey Part Number: J151-ND
1	black	Banana Jack	Digikey Part Number: J152-ND
8		Nylon Lock Nut, 4-40	McMaster Part Number: 90631A005
2		Screw, 4-40, 2-1/2”	McMaster Part Number: 92196A122
4		Screw, 4-40, 1-3/4”	McMaster Part Number: 92196A034
2		Screw, 4-40, 1”	McMaster Part Number: 92196A031
4		Spacer, nylon 1”	McMaster Part Number: 94639A020
6		Spacer, nylon 1/4”	McMaster Part Number: 94639A620
4		Screw, M3x8	McMaster Part Number: 91292A112
1	On-off-on	Toggle Switch	Digikey Part Number: 360-1814-ND
1	10k	Potentiometer	Digikey Part Number: PDB181-K420K-103B-ND
2		Acrylic sheet	
2	red	LED	Digikey Part Number: 67-1188-ND
1	green	LED	Digikey Part Number: 67-1189-ND
1		Arduino Mega 2560	Sparkfun Part Number: DEV-11061
1		Encoder Wire Harness	Digikey Part Number: 516-2027-ND
1		Misc Wire	