A DIGITAL POTENTIOMETER FOR AN ULTRA STABLE LASER CURRENT DRIVER

by

Marshall van Zijll

A senior thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics and Astronomy

Brigham Young University

April 2007

BRIGHAM YOUNG UNIVERSITY

DEPARTMENT APPROVAL

of a senior thesis submitted by

Marshall van Zijll

This thesis has been reviewed by the research advisor, research coordinator, and department chair and has been found to be satisfactory.

Date	Dallin Durfee, Advisor
Date	Eric Hintz, Research Coordinator
Date	Scott Sommerfeldt, Chair

ABSTRACT

A DIGITAL POTENTIOMETER

FOR AN ULTRA STABLE LASER CURRENT DRIVER

Marshall van Zijll

Department of Physics and Astronomy

Bachelor of Science

I created a digital potentiometer to be used for the stable control of the current in a laser current controller. This digital potentiometer consists of a microcontroller used in conjunction with a digital to analog converter (DAC). I selected a DAC that is appropriate for our design, I programmed a microcontroller to manage the DAC, and I designed the digital circuit board for the microcontroller. Our digital potentiometer is more stable, more accurate, has better repeatability, and picks up considerably less noise than a manual potentiometer.

ACKNOWLEDGMENTS

Thanks to Dallin Durfee and Christopher Erickson.

Contents

List of Figures

Chapter 1

Introduction

I created a digital potentiometer to be used for the stable control of the current in a laser current controller. This digital potentiometer consists of a digital to analog converter (DAC) which is programmed by a microcontroller. I selected a DAC that is appropriate for our design, I programmed a microcontroller to manage the DAC, and I designed the digital circuit board for the microcontroller. I also programmed the microcontroller to perform multiple other functions including displaying information on an LCD screen. Our digital potentiometer is more stable, more accurate, has better repeatability, and picks up considerably less noise than a manual potentiometer.

This laser current driver is an integral part of a larger project; that of building an atom interferometer. The concepts associated with this project are similar to those of an optical interferometer, but with several advantages provided by the use of atoms rather than photons. The extremely small wavelengths of atoms will allow us to make more precise measurements than an optical interferometer, and their inertial structure and mass will allow measurements that are otherwise impossible.

We plan to use this interferometer for multiple measurements. We will look for variations in fundamental constants, and make measurements regarding the validity

of General and Special Relativity. Our interferometer should also provide an optical frequency standard with precision comparable to the best atomic clocks. We will also be able to use our device as a precision accelerometer and implement it as a gyroscope. Due to the precision necessary for these uses, the individual components within our project must be very accurate and stable.

A key component of the interferometer is the diode laser. In our application, diode lasers are used to split and recombine the atomic beam, as well as to measure the state of the atoms after they have passed through the interferometer. Our diode laser needs stability better than can be provided by commercial diode laser systems, so our group designed a system of our own.

The stability of the laser is directly dependant on the stability of its current driver, and for this reason we have made our current driver as stable and quiet as we can. Our driver is based on the Hall-Libbrecht [1] design, and with our improvements it performs better than any other systems that we know of. It is also relatively inexpensive which allows us to use it in our lab for every application, even those that don't require the high precision our current driver offers.

In Chapter 2 I'll explain the entire current driver, including its design and its advantages over other current drivers. In Chapter 3 I'll discuss the digital portion of our circuit, including the specific microcontroller that we chose. In chapter 4 I'll explain the preparatory necessities for using the microcontroller, with emphasis on uploading a bootloader. In Chapter 5 I'll describe specifics in the code I wrote to program the microcontroller. Finally, in Chapter 6, I'll summarize the advantages of implementing of our device.

Chapter 2

The Current Driver

Our current driver is based on the Hall-Libbrecht design [1], with a few important improvements. The schematic for our design is shown in Figure 2.1. One improvement for our board is the use of the AD8671 op-amps rather than the ones specified in the original design. The AD8671 has a higher input impedance and doesn't oscillate when driving the gate capacitance of a high current transistor, as the original op-amps did. We are also using surface mount components, which are quieter since they are closer to the board. The original Hall-Libbrecht design called for higher quality op-amps in some parts of the circuit, yet lower-quality ones in the modulation input portion of the driver. This was undesirable since the modulation input is connected directly to the driver output, and we want as little noise here as possible. To fix this we used the same high-quality op-amp throughout the circuit. As I describe the individual portions of the circuit, I will refer to them by the names shown in bold in Figure 2.1.

The central part of our circuit is the "Current Regulation" portion, specifically the NDS0605 mosfet transistor. This transistor throttles the current, fixing the amount that passes through to the laser diode. An AD8671 op-amp controls the transistor. The op-amp monitors and adjusts the current by measuring the voltage drop across

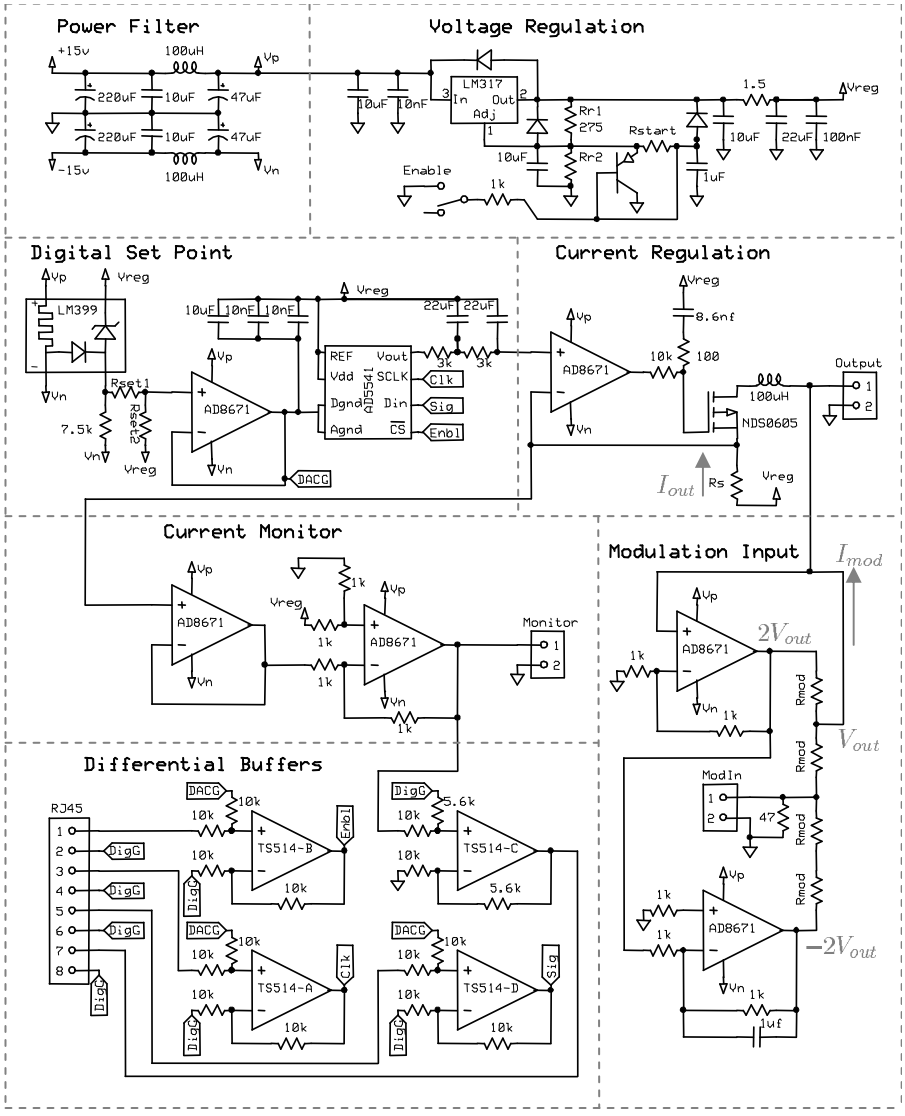

Figure 2.1 Schematic of the laser diode current driver circuit. The circuit is divided into separate functions for clarification and easy referencing.

the resistor Rs. By changing the set-point voltage on the op-amps non-inverting input, the amount of current passing through the transistor changes, and we're able to control the current that passes to the diode laser.

To keep the current passing through the transistor stable it is necessary to have a very stable voltage, labeled Vreg in Figure 2.1. To produce this voltage, we filter the power in the "Power Filter" portion of the circuit with capacitors and inductors. In this first portion of filtering we obtain a voltage stable enough to power our op-amps, but we need to regulate it even more to provide the stability necessary for the rest of the circuit. In the "Voltage Regulation" portion of the circuit we use an LM317 voltage regulator along with additional capacitors and inductors. After the power passes through this section, we have a voltage supply adequate for the rest of the circuit.

To control the current flowing through the transistor, we set the voltage at the positive end of the op-amp in the "Current Regulation" portion of the circuit. Conventionally, a design would call for a potentiometer at this point, but we have chosen instead to set this value digitally by use of a DAC. The DAC value is controlled by a microcontroller, which we have placed on a separate board. The microcontroller board and the current driver board have different grounds, so in the "Differential Buffers" section we reference the values to each other, and isolate the current driver board from external noise.

The "Modulation Input" section of the circuit allows us to modulate our laser current. The upper op-amp creates a voltage of 2Vout, while the lower op-amp creates a voltage of -2Vout. If ModIn is connected to a voltage source, then the voltage drop across Rmod from Vout to ModIn is different from the drop from 2Vout to Vout. This forces a current of $Imod = \frac{2Vout-Vout}{Rmod} - \frac{Vout-ModIn}{Rmod} = \frac{ModIn}{Rmod}$ to flow to the output. On

the other hand, if there is nothing connected to the ModIn connection, ModIn floats naturally to 0V and Imod is 0A.

Chapter 3

The Digital Potentiometer

3.1 The DAC

The DAC we use is the AD5541. This is a 16-bit serial input DAC that runs on a 5V power supply. It has a low temperature coefficient of ± 0.2 ppm/°C. It also has a 25 MHz, 3 wire serial interface, through which the microcontroller controls it. [2]

Using a DAC to control our current driver offers many advantages over using a manual potentiometer. The DAC allows for exceptional repeatability. If I am working on an experiment with a laser and wish to change my setup, it is necessary to turn off the current driver. With a manual potentiometer it proves to be very difficult to return the laser to the exact value as before the change, due to the physical inaccuracy in turning a knob. With the DAC, on the other hand, it's just a matter of setting it to the same value digitally and I return to precisely the same value every time.

The lower portion of Figure 3.1 shows the repeatability of the current driver. This repeatability is dependent on the repeatability of the DAC used to set the current. To obtain this graph we went to each value, stepped away from that value, and then returned to the value multiple times measuring the deviation from the other values

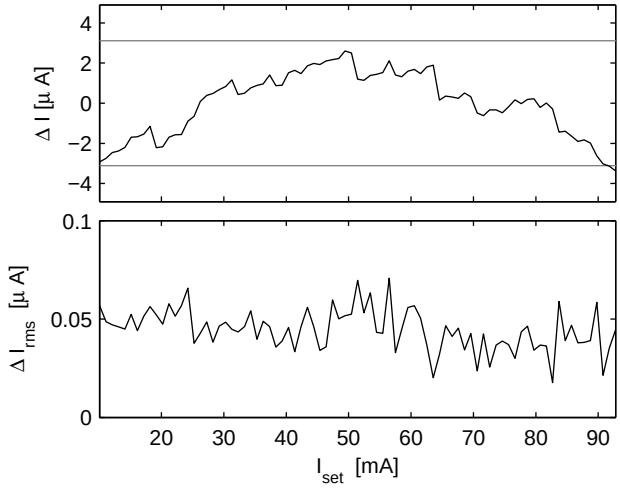

Figure 3.1 The upper graph shows the deviation from linearity over the values of the current driver. The grey bars are the specifications listed for our DAC. The lower graph shows the deviation of the output of the current driver when we stepped away from each value and returned to it multiple times.

obtained. Our current driver has repeatability accurate to within 50nA.

The top portion of Figure 3.1 shows the deviation of the current driver from linearity. We obtained this by reading the output values of the current driver for different digital programming values, and plotting the deviation of the data from a straight line. The grey lines show the limits of the current's deviation according to the DAC's specifications. As can be seen, the DAC is within spec.

Previously, in order to have the potentiometer easily accessible, we had to run long cables from the laser to wherever the potentiometer was. Through testing, we have found that even a small length of cable introduces a significant amount of noise. With the DAC, this problem doesn't exist. The DAC is placed on the same board as the rest of the driver, and the entire setup is placed next to the laser. Since the current driver board is so close to the laser, there are no long cables introducing extra noise. The lines controlling the value of the DAC from the microcontroller are unaffected by small amounts of noise since they are digital values.

3.2 The Microcontroller

The microcontroller we use is the PIC18F4550. This is a USB 2.0 programmable microcontroller. This makes it extremely simple to make software changes. All that is necessary is to connect the microcontroller to the computer through a USB port which allows updates to take place in about one second. The microcontroller has 35 pins which can be used as inputs or outputs, and three of them can be used as external interrupts. It has a 10bit Analog to Digital Converter (ADC) built in. It can run on an external clock up to 48 MHz. It also uses flash memory, which is reliable through millions of changes [3].

The microprocessor is programmed to read the signal from a digital encoder, and subsequently output a value to the DAC. [1] While many electronics have continuously running high-frequency clocks that create noise, our microprocessor is careful to only output values to the DAC when changes are made. This eliminates unnecessary noise. In fact, we can completely disconnect and reconnect the microprocessor board from the current driver board in the middle of an experiment, if we wish, and everything continues to run. (The user would only be unable to change values on the DAC while the microprocessor board is disconnected).

We have also used the ADC included in the microcontroller to read back a voltage proportional to the current output of the driver. The microprocessor displays the set current on an LCD display along with other relevant information from the microprocessor. A functional diagram of the digital circuit is shown in Figure 3.2.

The ease of updating this microcontroller, along with the many functions it can perform, make it ideal for remotely controlling the current driver. This microcon-

[1] A digital encoder is just a knob that drives a two-bit clock upon being turned.

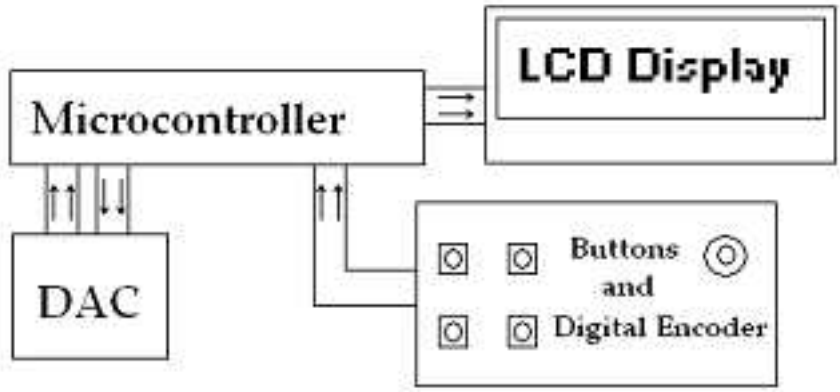

Figure 3.2 A simplified diagram of the digital circuit, showing the main components and their relations to one another

troller is truly an experimenter's microcontroller, because it not only allows for flexibility but also is simple to use.

Chapter 4

Uploading the Bootloader

The microcontroller's memory is initially empty and it does not yet know how to communicate with a computer through a USB, or do anything at all. Before a program can be written to the microcontroller using the USB capability, a bootloader must be put on the microcontroller. The bootloader is a basic program that allows the microcontroller to communicate through its USB channel.

The company that provides us with the microcontroller also provides us with code for a bootloader, along with an advertised "2-pin programming" feature for uploading the bootloader through a standard RS232 port. Three main steps are necessary for uploading a bootloader onto the microcontroller: 1. Download and install the relevant software. 2. Modify the bootloader code. 3. Upload the code to the microcontroller. I will explain each of these steps in detail, and then I will explain how to program the device after the bootloader is installed.

4.1 Download and Install Relevant Software

The first necessary step is to download the pertinent software from Microchip, the microprocessor's developer. First begin by downloading and installing MPLAB IDE from www.microchip.com/ide/. MPLAB is a free development environment specifically designed for programming microchip's microcontrollers. Next go to www.microchip.com/c18/ and download MPLAB C18. This is the compiler associated with the IDE, and allows the user to write their code in C language rather than assembly. The student edition of MPLAB C18 is the only free edition, but it offers everything needed. During installation it is important to check all the checkbox options. This automatically causes the IDE to use this compiler so you won't have to specify it every time you compile your code. Finally, download the actual bootloader program. Go to www.microchip.com/stellent/idcplus?IdeService=SS_GET_PAGE&nodeId=2124&param=en022627 to download and install "USB Bootloader Setup.EXE". This not only includes a bootloader for the microcontroller, but also provides a simple user interface for microsoft windows to help upload programs to the microcontroller through the USB port once the bootloader is installed. [4]

4.2 Modify Code

At the same time the USB bootloader is uploaded to the microcontroller, the microcontroller's configuration bits will also be set. These configuration bits determine the internal clock speed, external clock source, brown out voltage, memory protection, the programming voltage and other important information. Most of this information is defaulted to the correct value, but some values will have to be modified.

First run MPLAB. In the menu choose Project/Open, and assuming the "USB Bootloader Setup.EXE" is installed in the C directory, choose C:\MCHPFSUSB\fw\Bo

ot\MCHPUSB.mcp. Also, make sure that MPLAB is running for the correct microcontroller. In the menu, choose Configure/Select Device, and choose PIC18F4550 as the device.

In the menu choose Configure/Configuration Bits. This will open a screen allowing the user to choose the configuration bits. Set "Low Voltage Program" to "Enabled", and also check the other information for accuracy. The low voltage option allows us to program the microcontroller using just 5V rather than 13V. This allows a much simpler hardware design, since we don't have to worry about ruining the microcontroller during programming.

Once the configuration bits are set, save the bootloader code to a HEX file. In the menu choose File/Export, ensure that the configuration bits are included, and click OK. Save this file with whatever name you choose; it will be used later once we have set up the hardware for the microcontroller.

4.3 Upload and Run the Bootloader

In searching online, there are hundreds of different schematics of bootloader devices for PIC microcontrollers, but sadly there were no simple plans for our specific microcontroller. By looking at all the other circuits I noticed certain similarities in their design. Through much trial and error I was finally successful in uploading the bootloader onto our device. The schematic in Figure 4.1 shows all that is necessary to put a bootloader on the microcontroller.

The software used in conjunction with this setup is called WinPic. This can be downloaded from many sites by searching for "winpic programmer." I, personally, downloaded from the site www.qsl.new/dl4yhf/winpicpr.html . After this is installed, run the program. In the "Device, Config" tab, choose the PIC18F4550 as the mi-

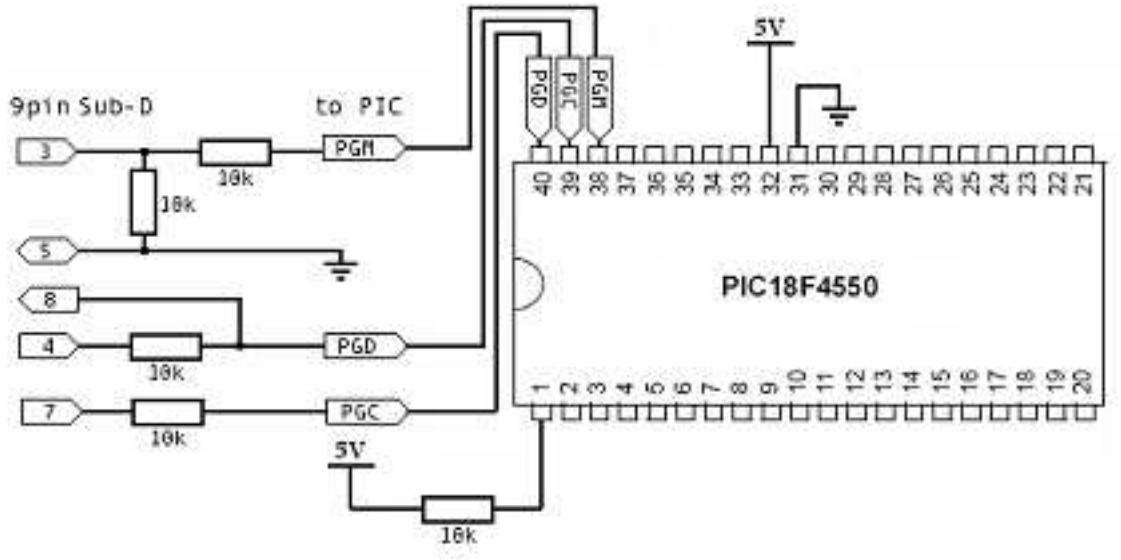

Figure 4.1 A schematic for uploading the bootloader onto the microcontroller from a 9pin Sub-D connector

crocontroller. Once the microcontroller is connected to the 9-pin COM port of the computer, go to the "Interface" tab, select "COM84 programmer for serial port" as the interface type, and press "Initialize". If the software states that initialization was successful, then the program is ready to upload the bootloader onto the microcontroller. In order to upload, select File/Load&ProgramDevice from the menu and then select the bootloader HEX file that was saved earlier. In the "Messages" tab, it will tell you information regarding the programming of the microcontroller.

Once the bootloader is installed on the microcontroller, the USB functionality is available. Figure 4.2 shows the minimal schematic necessary to use the USB connection with the microcontroller. If the capacitor value between VCC and ground is changed, the circuit will oscillate undesirably. The crystal can be chosen to be a different speed, but it must be specified in the configuration bits.

When the circuit is connected using a USB cable, hold down the button S2 (Boot),

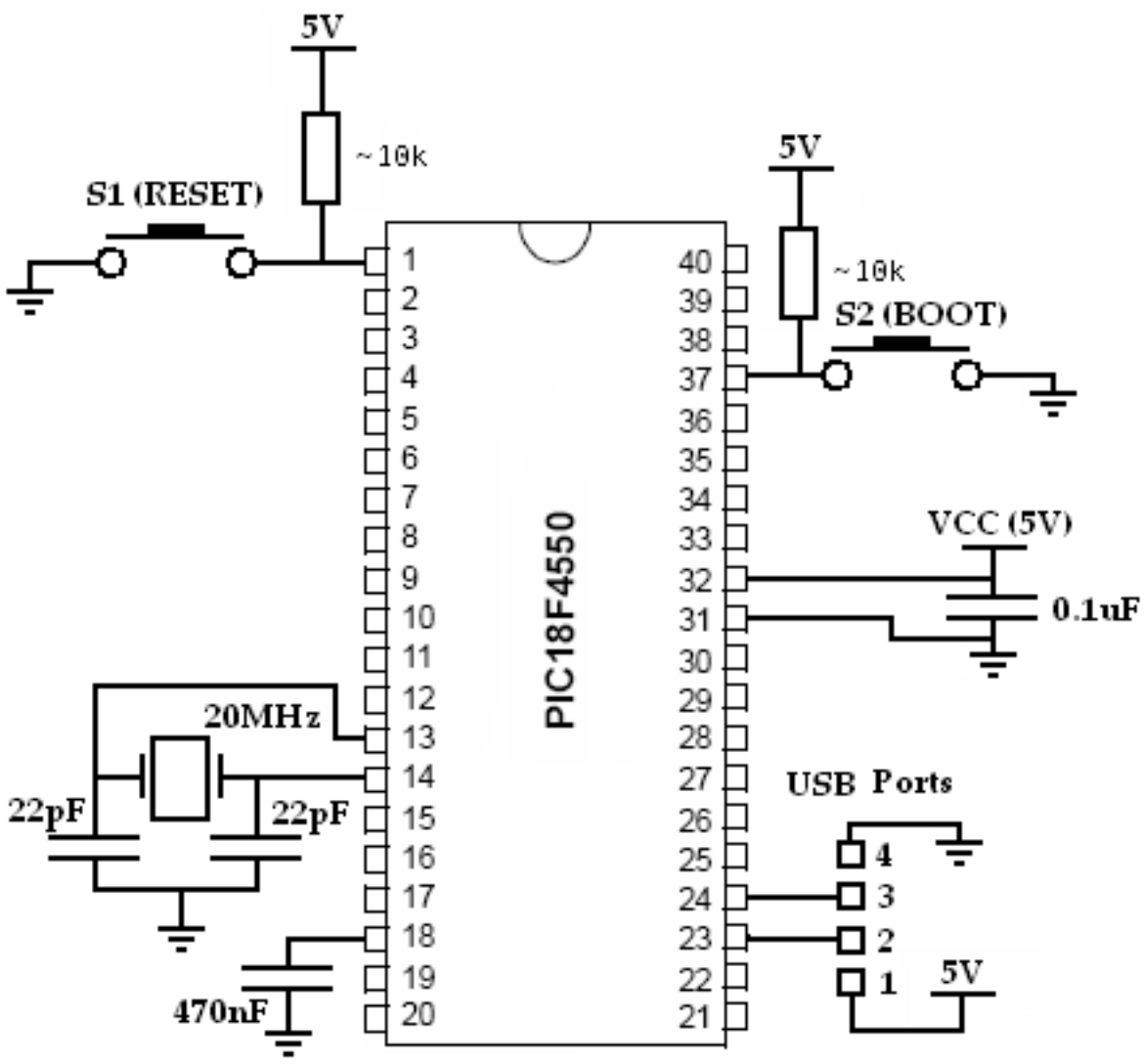

Figure 4.2 The minimal schematic for a working USB connection [4]

and press S1 (Reset) with S2 still held down. This will cause the bootloader program to run on the microcontroller, and windows should automatically detect it as a new device. When Windows asks for a driver, include the following location in its search: C:\MCHPFSUSB\Pc\MCHPUSB Driver\Release\.

At this point the microcontroller is fully able to communicate with the computer through a USB cable. The great advantages of using a USB port to upload programs are the speed of uploading, and the fact that programs can be changed while the microcontroller remains in its functioning circuit.

4.4 Upload Programs through the USB

Now that Windows is able to recognize the microcontroller, I will explain the simple process of uploading a program through the USB port. The first step is to write a program using MPLAB. The easiest method of doing this is by modifying one of the demo programs provided. Once a program is written, in the menu choose Project/Build All. This will store your program as a HEX file in the same directory as your project is kept.

After your HEX file is saved, open up an explorer to C:\MCHPFSUSB\Pc\Pdfsusb and run the PDFSUSB executable. This is an interface used to upload programs to the microcontroller. With the USB cable connected, hold down the S2 button, and press button S1. After doing this, your specific microcontroller should appear as an option in the drop down list. To upload your program onto the microcontroller, press the “Load HEX File” button, and choose the HEX file you wish to load. Next press “Program Device”. To run the program you can either press the “Execute” button on the program, or button S1 (Reset) connected to the microcontroller.

Chapter 5

Programming the Microcontroller

Programming of the microcontroller is done in the C language, with a few points that are specific to the microcontroller. Microchip provides a few samples of code for their microprocessors (see [5]), and these are good places to start in writing your own code. I will explain the code that I have written for my project, and describe the aspects of coding that are specific to the microcontroller.

Most of the pins on the microcontroller can be used as either an input or an output, and the datasheet tells the specifics regarding each pin. In the code, a few steps are necessary in initializing a pin for either input or output. Each pin has three registers: 1. the PORT register, which is basically the input to the pin. 2. the LAT register, which is the output value of the pin. 3. the TRIS register, which tells the direction I want to use the pin for. In each case I must set the TRIS value to 0 if it's an output, and 1 if it's an input. I can either set an entire collection of bits at one time, or each bit individually. See lines 56 through 103 of the code in appendix A for specific examples of how this is done.

The main purpose of the microprocessor is to update the DAC, and for this reason the code continually polls the inputs from the digital encoder to see if any values have

changed. The digital encoder has four possible states (since it acts as a 2-bit binary clock). The code I wrote checks to see what state the encoder is in, and waits for a change. Once the state changes, it checks to see what the new state is, and updates the DAC accordingly. As the code is polling the digital encoder pins, it also continually checks each of the buttons to see if one is being pressed. If a button is pressed, it enters the respective function, and then returns to the main loop.

When the digital encoder is changed, the values on the pins do not go directly from their initial value to their final value. Rather, they quickly bounce between the two values before settling on the final value. A simple fix for this is to connect a capacitor from the pin to ground. Since we had originally purchased the boards without this capacitor, I made a fix in the code that checks to make sure the pin is stable. The fix is to check the pins status multiple times in succession in order to ensure that the value has settled.

Another specific portion of the code deals with reading the ADC. See the function "ReadADC()" (line 245) in appendix A to see how to update the ADC registers. Once the registers are updated, the user can address them directly by referring to ADRESH and ADRESL, for the high bits and the low bits respectively.

A clock signal must be generated to communicate with the DAC and the LCD display. To generate this signal there must exist some sort of wait function, because this function helps determine the speed of the clock. The compiler doesn't recognize any type of pause or wait function, therefore I created my own with just a couple for loops (see wait(int time) (line 468) in App. A). For examples of clock functions see clock() (line 497) and LCDclock() (line 489) in App. A.

One of the most useful aspects of the microprocessor is the ability to use an LCD display. In order to obtain a datasheet go to www.crystalfontz.com and search for one for a 16x2 character display. [6] For examples of initializing the LCD screen and

writing values to the screen refer to the datasheet, and to the functions initLCD() (line 407) and writeLCD() (line 343) in App. A.

The LCD display has proven to be a time consuming, yet extremely valuable part of this project. Since we are able to display information related to the microprocessor, we have been granted much more functionality. Because of the LCD display I have been able to add many functions to the microcontroller. One function allows the user to set a maximum value for the current driver to reach. Other functions allow changing views between the voltage and current. The user can also use a function to increase and decrease the rate of change that takes place in turning the digital encoder. Another function allows the user to tell the program the value of the resistor Rs in Figure 2.1. With this resistor value, the microcontroller can correctly calculate the current at the diode laser from the voltage output by the DAC.

One final aspect concerning the code is the importance of parameter checking. Since the DAC can only recieve a certain range of values, it is essential to check that each value sent to the DAC falls within this range. See the function paramAdd(long value, long change) (line 442) in App. A for an example of how I check the parameters in changing the value from the digital encoder.

Chapter 6

Conclusion

Our implementation of a microcontroller and a DAC in our current driver has proven to be a significant improvement over a manual potentiometer. This digital potentiometer is more stable, more accurate, has better repeatability, and picks up considerably less noise than a manual potentiometer.

As a result of our improvements, this current driver has better specifications than any others we know of. This current driver is of high quality, and it is relatively inexpensive allowing it to be implemented as a standard throughout the lab.

The flexibility of the microcontroller also allows room for further improvements and new implementations. Currently we are working on using a digital potentiometer to drive a PID controller. In this project we plan to use the microcontroller to automatically scan and lock a laser. With such implications, the microcontroller has found a permanent place in our lab.

Bibliography

[1] K. G. Libbrecht, J. L. Hall, Rev. Sci. Instrum., **64**, 2133-2135 (1993).

[2] "AD5541 Datasheet," http://www.analog.com/UploadedFiles/Data_Sheets/AD5541_5542.pdf

[3] "PIC18F4550 Datasheet," http://ww1.microchip.com/downloads/en/DeviceDoc/39632D.pdf

[4] PIC18Fusb.online.fr, "How to use Microchip USB Bootloader," http://pic18fusb.online.fr/wiki/wikka.php?wakka=UsbBootload&show_comments=0

[5] Rawin Rojvanit, "Demo02.c", Microchip Technology, Inc. 11/15/04.

[6] "LCD Datasheet," http://www.crystalfontz.com/products/1602a-color/CFAH1602ARGHJP.pdf

Appendix A

Microprocessor Code

```
1   /** I N C L U D E S **********************************************************/
2   #include <p18cxxx.h>
3
4   /** V A R I A B L E S ********************************************************/
5   #pragma udata
6
7   /** P R I V A T E  P R O T O T Y P E S ***************************************/
8   char currentWord[9] = {'h','e','l','l','o','!',' ',' ',0};
9   char wordSetDig[9] = {'D','i','g','.',' ','V','a','l',0};
10  char wordSetMax[9] = {'S','e','t',' ','M','a','x',' ',0};
11  char wordSetCur[9] = {'S','e','t',' ','I',' ',' ',' ',0};
12  int adder = 1; // adder is a weighing factor for the inc/dec.
                   It is changed by a button
13  int currentFunc = 0, currentFuncChanger = 0, currentShowValue = 0,
    adcDiv = 0, adcDivNum = 0;
14  long digiVal = 0x0000, minVal=0x0001, prevNum = 0xFFFF, mainResistor = 1000;
15  void wait(int wait);
16  void initDAC(void);
17  void initLCD(void);
18  void test(void);
19  void clock(void);
20  void LCDclock(void);
21  void writeDAC(long value);
22  long paramAdd(long value, long change);
23  void writeToLCD(long value);
24  void writeWord(char* thisWord);
25  void writeLong(long thisLong);
26  void copyWord();
27  void changeAdderVal(long thisLong);
28  void ReadADCF(void);
29  void changeFunction(long thisLong);
30  /** V E C T O R  R E M A P P I N G *******************************************/
31
```

```
32  extern void _startup (void);
33  #pragma code _RESET_INTERRUPT_VECTOR = 0x000800
34  void _reset (void)
35  {
36      _asm goto _startup _endasm
37  }
38  #pragma code
39
40  #pragma code _HIGH_INTERRUPT_VECTOR = 0x000808
41  void _high_ISR (void)
42  {
43      ;
44  }
45
46  #pragma code _LOW_INTERRUPT_VECTOR = 0x000818
47  void _low_ISR (void)
48  {
49      ;
50  }
51  #pragma code
52
53  /** D E C L A R A T I O N S **********************************************/
54  #pragma code
55
56  /** L E D ****************************************************************/
57  /** set bin to 0 if used as output, 1 if used as input **/
58  #define mInitAllLEDs()      LATD &= 0x8C; TRISD &= 0x8C; LATA &= 0xC1; TRISA &= 0xC1;
                                LATE &= 0xF8; TRISE &= 0x 8; LATB &= 0x0E; TRISB &= 0x0E;
59
60  #define mLED_1              LATDbits.LATD0
61  #define mLED_2              LATDbits.LATD1
62
63  /** These are the LCD screen values **/
64  #define db5                 LATBbits.LATB0
65  #define db4                 LATAbits.LATA1
66  #define db3                 LATAbits.LATA2
67  #define db2                 LATAbits.LATA3
68  #define db1                 LATAbits.LATA4
69  #define db0                 LATAbits.LATA5
70  #define e                   LATEbits.LATE0
71  #define rw                  LATEbits.LATE1
72  #define rs                  LATEbits.LATE2
73  #define db7                 LATCbits.LATC1
74  #define db6                 LATCbits.LATC2
75
76  /** These are the DAC values **/
77  #define cs                  LATDbits.LATD4
78  #define sclk                LATDbits.LATD5
79  #define din                 LATDbits.LATD6
80
81
```

```
82  #define mLED_1_On()         mLED_1 = 1;
83  #define mLED_2_On()         mLED_2 = 1;
84
85  #define mLED_1_Off()        mLED_1 = 0;
86  #define mLED_2_Off()        mLED_2 = 0;
87
88  #define mLED_1_Toggle()     mLED_1 = !mLED_1;
89  #define mLED_2_Toggle()     mLED_2 = !mLED_2;
90
91
92
93  /** S W I T C H E S ***********************************************/
94  #define mInitAllSwitches()      TRISBbits.TRISB2=1;TRISBbits.TRISB1=1;
                                    TRISBbits.TRISB4=1;TRISDbits.TRISD2=1;
                                    TRISDbits.TRISD3=1;TRISCbits.TRISC0=1;
95  #define mInitSwitch2()      TRISBbits.TRISB2=1;
96  #define mInitSwitch3()      TRISBbits.TRISB4=1;
97  #define sw3                 PORTBbits.RB4
98  #define functionSw          PORTCbits.RC0
99  #define allowSwitch         PORTBbits.RB2
100 #define changeAdder         PORTBbits.RB1
101 #define dial2               PORTDbits.RD2
102 #define dial1               PORTDbits.RD3
103 #define mInitADC()          TRISAbits.TRISA0=1;ADCON0=0x01;ADCON2=0x3C;
104 #define funcSetDig      0
105 #define funcSetMax      1
106 #define funcSetRes      2
107 #define funcShowFunc    0
108 #define funcShowValue   1
109 #define funcValueVolt   0
110 #define funcValueCurr   1
111
112 void interrupt()
113 {
114     mLED_2_On();
115 }
116 void main(void)
117 {
118     int d1, d2, i=0;
119     long adcUpdate = 0;
120     prevNum = thisNum;
121     currentFunc = funcSetDig;
122     adder = 1000;
123     ADCON1 |= 0x0F;         // Default all pins to digital
124     INTCON2 |= 0xF0;
125     mInitAllSwitches();     // Initializes button and switch and Knob ports
126     mInitAllLEDs();         // initializes LED and DAC ports
127     ADCON2bits.ADFM = 1;    // ADC result right justified
128     mInitADC();
129     initDAC();
130     initLCD();
```

```
131     writeDAC(thisNum);
132 //  test();while(1);          // A good place to test functions and hardware
133     writeLong(thisNum);       // initializes Value to LCD
134 //  writeWord(currentWord);   // writes a Word to LCD (initialized above)
135
136 while(1)
137 {
138     wait(5);    // prevents increment from taking place too quickly
                        (lowers errors due to imprecise knob)
139     if(dial1 == dial2){wait(6);if(dial1 == dial2)
                        // multiple if's assure accurate read of state
140 {
141         d1 = dial1;       // mutiple while loops prevent mis-change
             due to bounce in signal (code fix rather than capacitor fix... )
142         while(dial1 == dial2){while(dial1 == dial2){while(dial1 == dial2)
143         {                       // this is most often the while state
144             adcUpdate++;
145             if(adcUpdate == 10000)
146             {
147                 adcUpdate = 0;
148                 updateADC();
149             }
150             if(allowSwitch==0)
151                 break;
152             if(changeAdder==0)
153                 changeAdderVal(thisNum);
154             if(functionSw==0)
155                 changeFunction(thisNum);
156         }wait(1);}wait(1);}
157         if(dial1 != d1)             // case increment
158             thisNum = paramAdd(thisNum, 1*adder);
159         else                        // case decrement
160             thisNum = paramAdd(thisNum, -1*adder);
161
162         if(thisNum != prevNum)
163         {
164             if(currentFunc == funcSetDig)
165                 writeDAC(thisNum);
166             writeLCD();
167             prevNum = thisNum;
168         }
169     }}
170     if(changeAdder==0)
171         changeAdderVal(thisNum);
172     if(functionSw==0)
173         changeFunction(thisNum);
174
175
176     if(dial1 != dial2){wait(6);if(dial1 != dial2)   // multiple if's
                                              assure accurate read of state
177     { 178         d1 = dial1;       // mutiple while loops prevent mis-change
```

```
            due to bounce in signal (code fix rather than capacitor fix... )
179         while(dial1 != dial2){while(dial1 != dial2){while(dial1 != dial2)
180         {                       // this is most often the while state
181             adcUpdate++;
182             if(adcUpdate == 10000)
183             {
184                 adcUpdate = 0;
185                 updateADC();
186             }
187             if(allowSwitch==0)
188                 break;
189             if(changeAdder==0)
190                 changeAdderVal(thisNum);
191             if(functionSw==0)
192                 changeFunction(thisNum);
193         }wait(1);}wait(1);}
194         if(dial1 == d1)             // case increment
195             thisNum = paramAdd(thisNum, 1*adder);
196         else                        // case decrement
197             thisNum = paramAdd(thisNum, -1*adder);
198
199         if(thisNum != prevNum)
200         {
201             if(currentFunc == funcSetDig)
202                 writeDAC(thisNum);
203             writeLong(thisNum);
204             prevNum = thisNum;
205         }
206     }}
207 }//end while
208
209
210 }//end main
211
212 void changeFunction(long thisLong)
213 {
214     if(currentFuncChanger == 0)
215     {
216         if(currentFunc == funcSetDig){
217             digiVal = thisNum;
218             currentFunc = funcSetMax;
219             thisNum = minVal;}
220         else if(currentFunc == funcSetMax){
221             minVal = thisNum;
222             currentFunc = funcSetRes;
223             thisNum = mainResistor;}
224         else
225         {
226             mainResistor = thisNum;
227             currentFunc = funcSetDig;
228             currentFuncChanger = 1;
```

```
229             thisNum = digiVal;
230         }
231     }
232     if(currentFuncChanger == 1)
233     {
234         if(currentShowValue == funcValueVolt){
235             currentShowValue = funcValueCurr;
236         }
237         else if(currentShowValue == funcValueCurr)
238         {
239             currentShowValue = funcValueVolt;
240             currentFuncChanger = 0;
241         }
242     }
243 }
244
245 void ReadADC(void)
246 {
247     ADCON0bits.GO = 1;              // Start AD conversion
248     while(ADCON0bits.NOT_DONE);     // Wait for conversion
249     return;
250 }//end ReadADC
251
252 void changeAdderVal(long thisLong)
253 {
254     if(adder = 1000)
255         adder = 1;
256     else
257         adder = adder*10;
258     while(changeAdder == 0);
259     wait(5);
260 }
261 void test()     // test funciton
262 {
263     while(1){
264         if(sw3 == 0){
265             din=1;
266         }else{
267             din=0;
268         }
269     }
270 }
271
272 void updateADC()
273 {
274     long restOfLong;
275     double thisNum;
276     int j = 0x230, thisDigit, count; // j is the offset to write to lcd
277
278     ReadADC();
279     thisNum = ADRESH; // there also exists ADRESL
```

```
280     thisNum = thisNum*2*5000/255;   // This makes it millivolts
281     if (currentShowValue == funcValueCurr)
282     {
283         thisNum = thisNum * 1000;
284         thisNum = thisNum/mainResistor;
285     }
286     adcDiv++;
287     adcDivNum += thisNum;
288     if(adcDiv == 5)
289     {
290         thisNum = adcDivNum / 5;
291         adcDiv = 0;
292         adcDivNum = 0;
293     }
294     else
295         return;
296     writeToLCD(0xC7); LCDclock(); // ddram to position f
297     writeToLCD(0x04); LCDclock(); // cursur moves in dec motion
298     restOfLong = 10000;
299     while(restOfLong > 0)
300     {
301         thisDigit = restOfLong % 10;
302         restOfLong = restOfLong - thisDigit;
303         restOfLong = restOfLong/10;
304         writeToLCD(0x2A0); LCDclock();
305     }
306     restOfLong = (int)(thisNum);
307     count=0;
308     writeToLCD(0xC7); LCDclock(); // ddram to position f
309     writeToLCD(0x04); LCDclock(); // cursur moves in dec motion
310     if (currentShowValue == funcValueCurr) // writes an A or a V
311     {
312             writeToLCD(0x241); LCDclock();
313             writeToLCD(0x2A0); LCDclock();
314     }
315     else
316     {
317             writeToLCD(0x256); LCDclock();
318             writeToLCD(0x2A0); LCDclock();
319     }
320     while(restOfLong > 0)
321     {
322         count++;
323         thisDigit = restOfLong % 10;
324         restOfLong = restOfLong - thisDigit;
325         restOfLong = restOfLong/10;
326         writeToLCD(j+thisDigit); LCDclock();
327         if(count==3)
328         {
329             writeToLCD(0x22E); LCDclock();
330         }
```

```
331     }
332     while(count < 3)
333     {
334         count++;
335         writeToLCD(j); LCDclock();
336         if(count==3)
337         {
338             writeToLCD(0x22E); LCDclock();
339         }
340     }
341 }
342
343 void writeLCD() // writes a long to the LCD in decimal form
344 {
345     long restOfLong;
346     int j = 0x230, thisDigit, count; // j is the offset to write to lcd
347     writeToLCD(0x01); LCDclock(); // clears lcd and initalizes ddram
348     if(currentFunc == funcSetDig)
349         writeWord(wordSetDig);
350     else if(currentFunc == funcSetMax)
351         writeWord(wordSetMax);
352     else if(currentFunc == funcSetRes)
353         writeWord(wordSetRes);
354     writeToLCD(0x8F); LCDclock(); // ddram to position f
355     writeToLCD(0x04); LCDclock(); // cursur moves in dec motion
356     restOfLong = thisLong;//*.3425;
357     count=0;
358     while(restOfLong > 0)
359     {
360         count++;
361         thisDigit = restOfLong % 10;
362         restOfLong = restOfLong - thisDigit;
363         restOfLong = restOfLong/10;
364         writeToLCD(j+thisDigit); LCDclock();
365         if(count==3 && currentFunc == funcSetRes)
366         {
367             writeToLCD(0x22E); LCDclock();
368         }
369     }
370     while(count < 3 && currentFunc == funcSetRes)
371     {
372         count++;
373         writeToLCD(j); LCDclock();
374         if(count==3)
375         {
376             writeToLCD(0x22E); LCDclock();
377         }
378     }
379     count=0;
380     restOfLong = adder;
381     writeToLCD(0xCF); LCDclock();
```

```
382     writeToLCD(0x04); LCDclock(); // cursur moves in dec motion
383     while(restOfLong > 0)
384     {
385         count++;
386         thisDigit = restOfLong % 10;
387         restOfLong = restOfLong - thisDigit;
388         restOfLong = restOfLong/10;
389         writeToLCD(j+thisDigit); LCDclock();
390         if(count==3 && currentFunc == funcSetRes)
391         {
392             writeToLCD(0x22E); LCDclock();
393         }
394     }
395     while(count < 3 && currentFunc == funcSetRes)
396     {
397         count++;
398         writeToLCD(j); LCDclock();
399         if(count==3)
400         {
401             writeToLCD(0x22E); LCDclock();
402         }
403     }
404     writeToLCD(0x278); LCDclock(); // writes an 'x'
405 }
406
407 void initLCD()        // inits LCD screen per instructions (datasheet model#:
                           CFAH1602A-GGB-JP   www.crystalfontz.com)
408 {
409     int i;
410     rs = 0;
411     rw = 0;
412     wait(100);
413     e = 0;
414     wait(10);
415     writeToLCD(0x030); LCDclock(); wait(50);
416     writeToLCD(0x030); LCDclock(); wait(10);
417     writeToLCD(0x030); LCDclock();
418     writeToLCD(0x038); LCDclock();
419     writeToLCD(0x008); LCDclock();
420     writeToLCD(0x001); LCDclock();
421     writeToLCD(0x006); LCDclock();
422     writeToLCD(0x00C); LCDclock();
423 }
424
425 void writeWord(char thisWord[9])    // writes characters to LCD screen,
          max length 9 (if wanted,can add more params to change starting loc and line#)
426 {
427     char thisChar;
428     int i=0;
429     int j = 0x200;
430     writeToLCD(0x001); LCDclock();
```

```
431     writeToLCD(0x080); LCDclock(); // set ddram to 0
432     writeToLCD(0x006); LCDclock(); // cursur in increment motion
433     thisChar = thisWord[0];
434     while(thisChar != 0)
435     {
436         writeToLCD(j+thisChar); LCDclock();
437         i++;
438         thisChar = *(thisWord+i);
439     }
440 }
441
442 long paramAdd(long value, long change)      // does boundary checking before
                                                              changing DAC value
443 {
444     long returnVal = 0;
445     if(change < 0)
446         if(value+change < 0 || value+change > value)
447             returnVal = 0;
448         else
449             returnVal = (value + change);
450     else
451         if(value + change < value || value + change > 0xFFFF)
452             returnVal = 0xFFFF;
453         else
454             returnVal = (value + change);
455     if(currentFunc == funcSetDig)
456     {
457         if(returnVal < minVal)
458             returnVal = minVal;
459     }
460     else if(currentFunc == funcSetMax)
461     {
462         if(returnVal > digiVal)
463             digiVal = returnVal;
464     }
465         return returnVal;
466 }
467
468 void wait(int time)     // waits approx 10ns per time... though it's not linear
                                                          approx seems sufficient.
469 {                       // took out linearity to lessen need for large numbers.
470     int i,j;
471 //time=time*3;  // slow clock
472     for(i=0; i<10*time; i++)
473     {
474         for(j=0; j<time; j++)
475         {
476             j++;
477         }
478     }
479 }
```

```
480
481
482 void initDAC()  // inits DAC (cs=1 prevents reading values)
483 {
484     cs = 1;
485     sclk = 0;
486     wait(5);
487 }
488
489 void LCDclock() // clock cycle on LCD
490 {
491     wait(3);  // 3 is the absolute minimum!!! for lcd...
492     e = 1;
493     wait(3);
494     e = 0;
495     wait(3);
496 }
497 void clock()    // clock cycle on DAC
498 {
499     wait(10);
500     sclk = 0;
501     wait(10);
502     sclk = 1;
503     wait(10);
504 }
505
506 void writeDAC(long value)
507 {
508     int x = 0;
509 //mLED_1_On();
510     if(value%2 == 1)    // prevents odd numbers, fix would be good
511     {//mLED_1_Off();
512         value = value-1;  // plus i have no idea what's causing this problem!
513 }
514     sclk = 1;
515     wait(5);
516     cs = 0;
517     for(x=15; x>-1; x--) // reads values serially into DAC
518     {
519         din = ((value>>x) & 0x01);  clock();
520     }
521     cs = 1;
522     clock();
523     sclk = 0;
524 //mLED_1_Off();
525 }
526 /** EOF Demo02.c ***************************************************************/
527
528 void writeToLCD(long value) // used by other functions to write to LCD.
                                                codes found on LCD datasheet.
529 {
```

```
530     rs = ((0xFFFF&value)>>9);
531     rw = ((0xFFFF&value)>>8);
532     db7 = ((0xFFFF&value)>>7);
533     db6 = ((0xFFFF&value)>>6);
534     db5 = ((0xFFFF&value)>>5);
535     db4 = ((0xFFFF&value)>>4);
536     db3 = ((0xFFFF&value)>>3);
537     db2 = ((0xFFFF&value)>>2);
538     db1 = ((0xFFFF&value)>>1);
539     db0 = ((0xFFFF&value)>>0);
540 }
```

[5]