Charles Leung – 400049422 – leungt11

ENGPHYS 3BB4 ELECTRONICS II PROJECT REPORT

CANDU Liquid Zone PID Control
using TI MSP-430

ABSTRACT

The objective of the 5-week Design Project is to design and implement a PID control system to the Liquid Zone Control (LZC) system for a simulated CANDU nuclear reactor
using digital and analog components available in the electronics laboratory. The objective of the PID controller is to accept a pulse-width-modulated (PWM) signal from an Arduino representing the liquid level of the LZC into the TI MSP-430 microcontroller where it is digitized, sampled and operated on by the PID control system. The output of the MSP-430 is fed back into the Arduino as a form of feedback control while the sampled input of the MSP-430 is simultaneously transferred and visually displayed in MATLAB software.

The final implementation was able to successfully perform the sampling, PWM read and PID functions. A power level was observed to approach the desired set point; however, it would not settle there consistently and would occasionally display wild fluctuations. Nonetheless, it can be concluded that much of the design objectives were successful in implementation.

INTRODUCTION AND THEORY

Introduction

The Design Project aims to consolidate and make use of the electronic theories and laboratory work with the Texas Instruments MSP430 microcontroller over the duration of the course so far to create a PID controller for the Liquid Zone Control (LZC) of a CANDU nuclear reactor. The aim is to feed an Pulse-Width Modulated (PWM) electronic signal generated by an Arduino Nano – which simulated the level of cooling fluid in the storage tanks of the reactor – receiving port of the TI MSP430 microcontroller, where it is sampled by the Analogue-to-Digital-Coverter (ADC) module and then operated on by a software PID feedback loop. The output of the PID loop is also a PWM waveform generated by the MSP-430 that is fed back into the Arduino as an analogue signal.

After the input signal is sampled by the MSP-430, the microcontroller then transmits the digital data to a computer which outputs a digital waveform through a MATLAB plotting tool, allowing the user to visualize the approximate shape of the signal, which represents the fill level of the liquid tanks. Many of the fundamental techniques used in this lab are heavily based upon those implemented in Lab 5 when creating a simple oscilloscope. Furthermore, there are several new techniques that perform critical functions that allow the PID controller system to function. These techniques include:

- Configuration of the onboard clock and baud rate settings
- Setting input and output ports in anticipation of establishing UART serial communication
- Writing a C program to transmit data from the MSP430
- Writing a MATLAB program to receive data on the PC
- Outputting results on a digital plot via MATLAB
- Converting a PWM signal into a stable analogue signal
- Converting an analogue waveform into discrete digital data through sampling
- Implementing a PID control loop

Theory

This section explains some of the techniques listed in the Introduction in greater detail.

Pulse Width Modulation (PWM)

Pulse width modulation is a method of reducing the average power delivered by a discretized electrical signal by varying its duty cycle. Effectively, any desired voltage level between a maximum voltage and 0 can be delivered these two discrete voltage levels chopping a signal up determining how long a signal should be on ‘high’ and how long it should be on ‘low’.

PWM is particularly suited for running inertial loads such as motors, which are not as easily affected by this discrete switching, because their inertia causes them to react slowly. This is especially true in the case of a Liquid Zone Control as fluid flow is highly inertial and takes time to respond to any input changes.

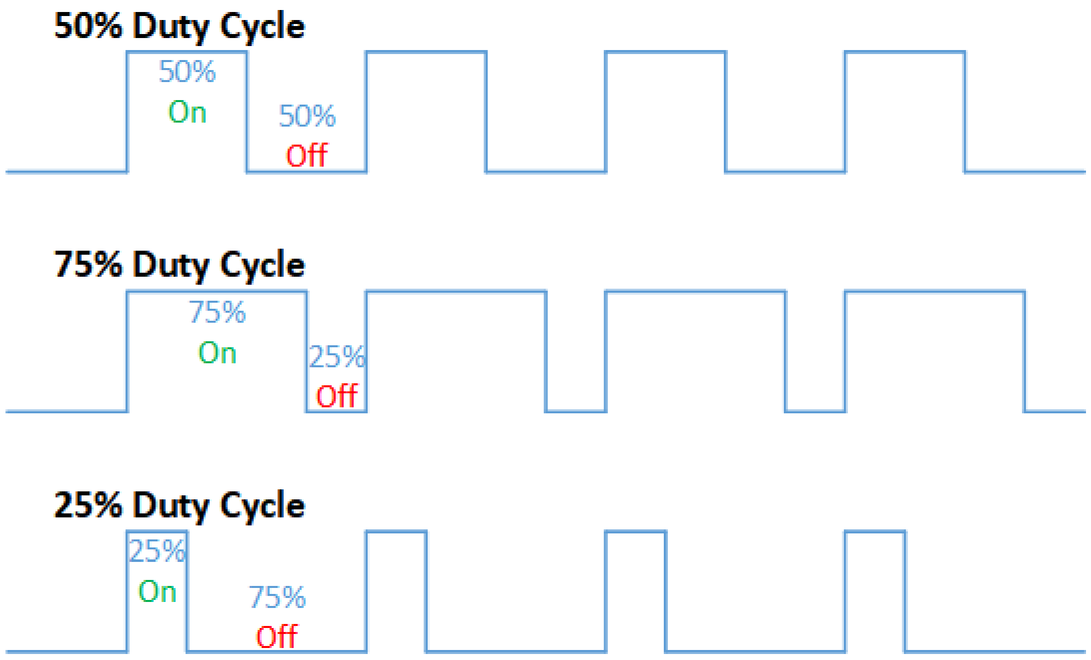


Figure 1: *Effects of varying duty cycle of a discrete DC voltage signal*

Comparator & Analogue-to-Digital-Converter Module

The MSP430 contains an onboard 10-bit Analogue-to-Digital-Converter (ADC) Module. As the name implies, this module provides means of digitizing an analogue input signal so that it can be turned into discrete data that can be operated on by the microcontroller. This functionality is crucial for the microcontroller to be able to output a sampled waveform for display in MATLAB.

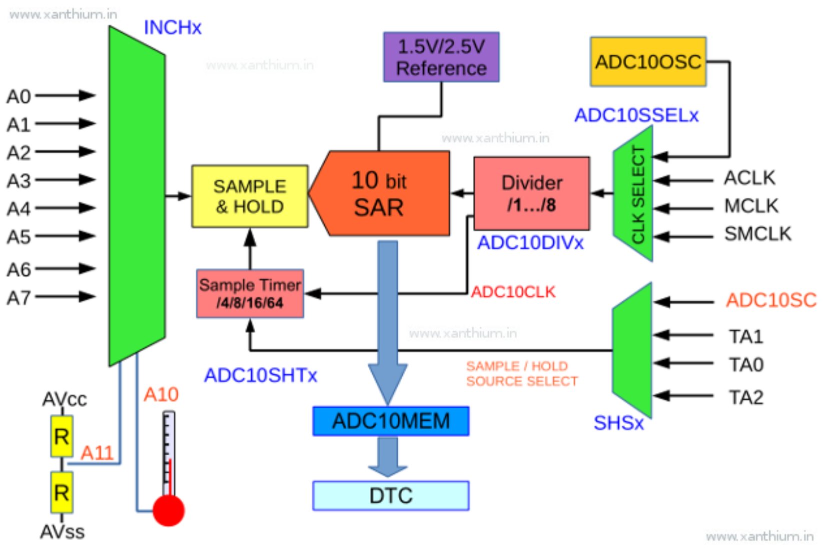


Figure 2: *Block diagram of TI MSP-430 ADC module*

Lowpass filter and op-amp setup

As the analogue inputs of both the MSP-430 microcontroller and Arduino Nano both require analogue DC voltage signals to function, it is necessary that any PWM output signals have to be converted into this form.

There are several ways that this can be achieved. One such way is to do so using a lowpass filter in conjunction with an op-amp.

A low-pass filter is an analogue filter that blocks the passage of frequencies higher than a specified threshold frequency. This threshold frequency is determined by the values of the resistor and capacitor used in combination.

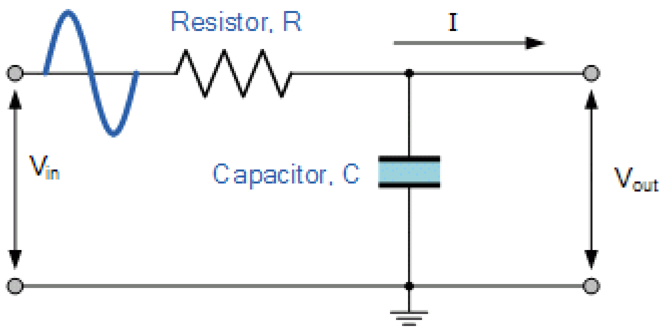


Figure 3: *Example a of first-order low-pass filter.*

Low-pass filters generally lead to voltage drops. The analogue inputs of the devices used in this project have a maximum input voltage of 3.3V. Thus, an op-amp in non-inverting mode can be used to boost this voltage up to 3.3V to avoid dealing with software scaling (this offloads some of the operations that would have otherwise been performed by the CPU.)

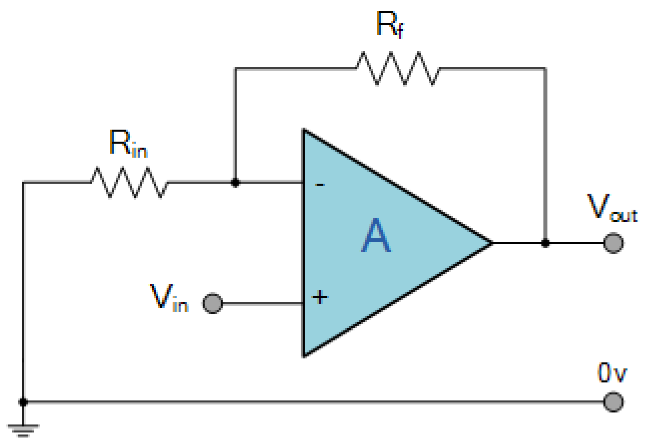


Figure 4: *Example a of operational amplifier in non-inverting mode.*

Nyquist-Shannon Sampling Theorem

Sampling is a process of converting a continuous signal into a sequence of values taken across a regular interval. The frequency of which the samples are taken is known as the sampling rate.

The Nyquist theorem describes a curious phenomenon: the samples of sine waves can be identical when at least one of them is at a frequency above half the sample rate. This establishes a lower bound for a reliable sampling frequency based on what the frequency of the input signal is known to be, and any sampled input signal that is above this sampling frequency may not be an accurate representation of what the actual input waveform is supposed to look like in reality.

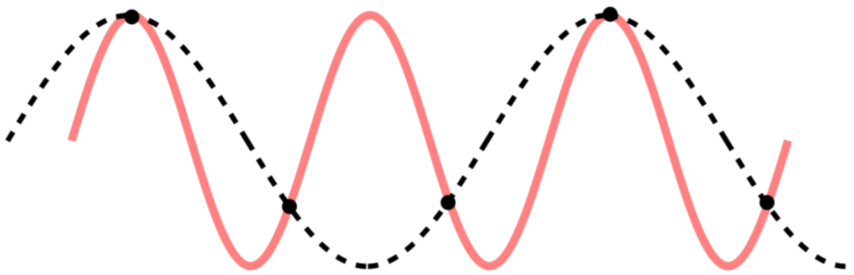


Figure 5: *The pink waveform has a frequency double that of the sampling rate. Its image will be an alias of a waveform with equal frequency to the sampling rate.*

Serial Communication

The MSP-430 establishes communication with other devices, in this case the user's personal computer, through the USB (Universal Serial Bus) interface using the UART (Universal Asynchronous Receiver Transmitter) protocol. As the name implies, this data is transmitted serially i.e. individual bits are transmitted sequentially through a single bus. There are 10-bits being transmitted between the MSP-430 and the MATLAB GUI. The data is encoded through the use of the ASCII protocol which converts information – letters, numbers, etc. – into a string of bits that is similarly decoded and interpreted by the receiving computer.

Due to the asynchronous nature of the UART protocol, the baud rate (the number of signaling events across the transmission medium per unit of time) of the transmitter and receiver must be identical. The output baud rate is set the microcontroller using the UART clock, which is a function of the CPU clock that defines the rate of data transfer via software configuration.

After the transmitting and receiving pins are initialized in the MSP-430 C code, the information transfer can then be initialized via an internal function call. In the context of this project, the transmitted information is sent to a corresponding MATLAB GUI on the user's computer, which is used to visually display the waveform in the manner of an oscilloscope.

PID Control

Proportional-Integral-Derivative (PID) is a time of feedback loop control mechanism. It functions by accepting an input and continuously calculating an error value which is the numerical deviation between a set point and the input value, then applying a correction that is based on the Ki, Kp and Kd terms that can be defined by the user. These user-defined values have an effect on how quickly a system can settle as its desired set point, its oscillatory/damping behaviour, as well as its response to variation in input. Common uses of PID control include cooling systems and vehicle cruise control.

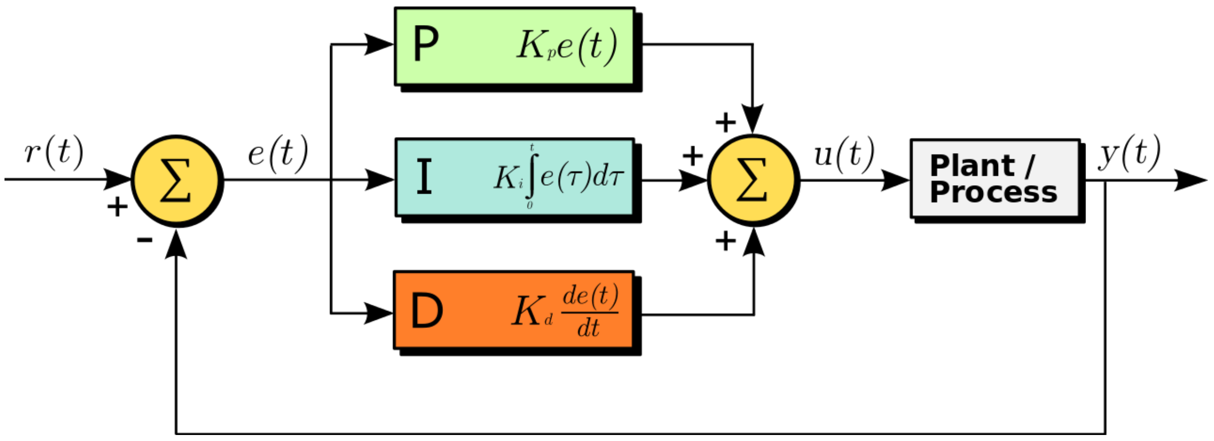


Figure 6: *PID control flowchart*

Method

Arduino Nano setup

The provided Arduino and C code are relatively straightforward to set up. The C code, provided via Avenue, can be compiled and run directly without modifications, as long as the Arduino IDE is configured correctly. The D9 pin on the Arduino serves an output that provides a PWM signal and the A0 pin serves as an input that accept an analogue signal.

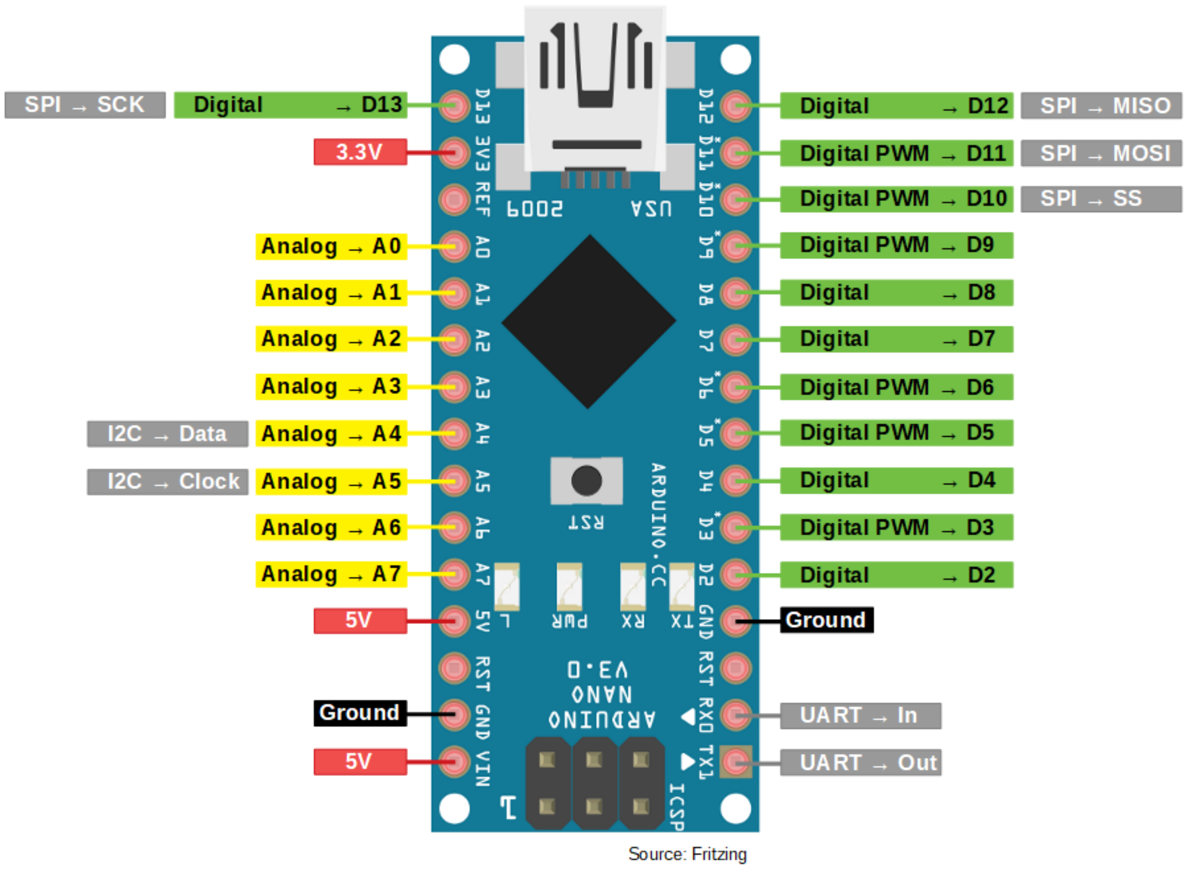


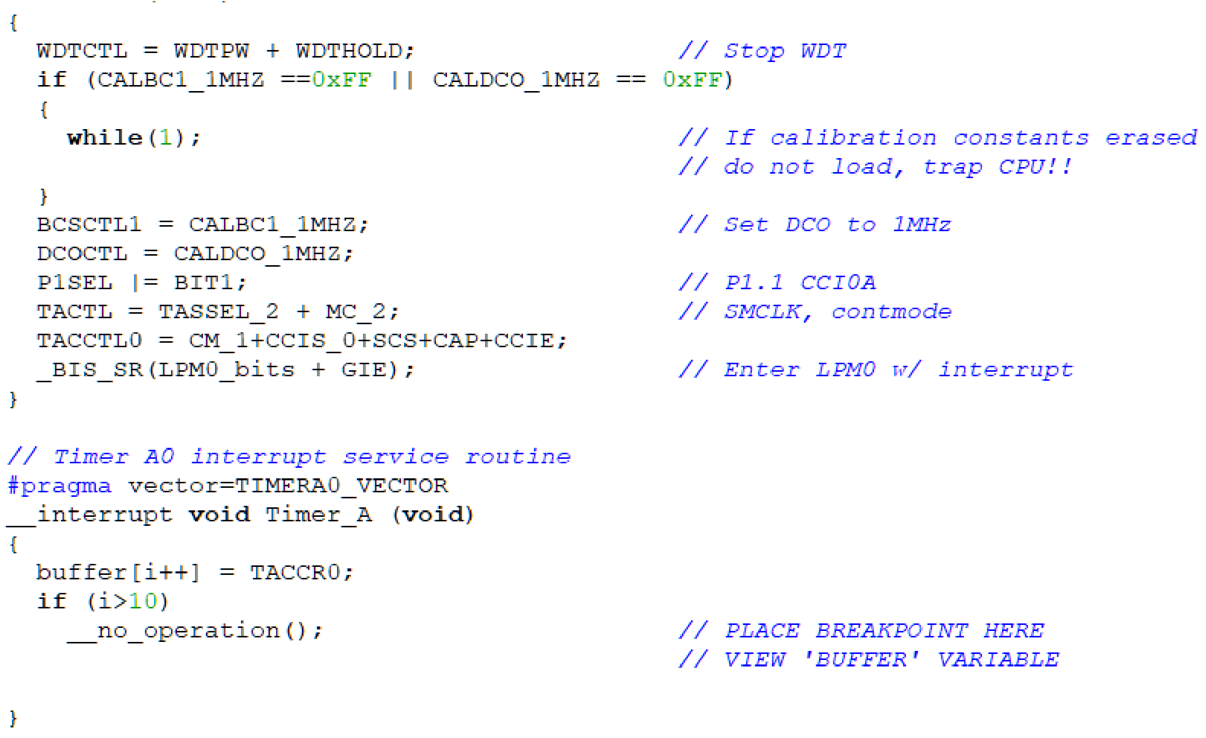
Figure 7: *Arduino Nano pinout*

Lowpass filter and op-amp setup

Initially, both hardware and software approaches to the PWM-analogue conversion were considered. In terms of a software implementation, a possible approach would be to note the dependence of the voltage in terms of the duty cycle:

$$V_{RMS} = A\sqrt{\frac{T_{on}}{T}}$$

where A is the amplitude and T_{on} is the 'high' period of the PWM signal. The amplitude is defined and the period is known, while T_{on} can be measured using the timer capture mode. This function essentially saves the current timer value into the CCP registers when a specific event occurs (usually a transition on an input pin.) A timer capture can be set up in the C code as follows:



```
{
  WDTCTL = WDTPW + WDTHOLD;                   // Stop WDT
  if (CALBC1_1MHZ ==0xFF || CALDCO_1MHZ == 0xFF)
  {
    while(1);                                 // If calibration constants erased
                                              // do not load, trap CPU!!
  }
  BCSCTL1 = CALBC1_1MHZ;                      // Set DCO to 1MHz
  DCOCTL = CALDCO_1MHZ;
  P1SEL |= BIT1;                              // P1.1 CCI0A
  TACTL = TASSEL_2 + MC_2;                    // SMCLK, contmode
  TACCTL0 = CM_1+CCIS_0+SCS+CAP+CCIE;
  _BIS_SR(LPM0_bits + GIE);                   // Enter LPM0 w/ interrupt
}

// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  buffer[i++] = TACCR0;
  if (i>10)
    __no_operation();                         // PLACE BREAKPOINT HERE
                                              // VIEW 'BUFFER' VARIABLE

}
```

Figure 8: *MSP-430 timer configuration C code*

Once the timer value is obtained and saved, the MSP-430 can be used to perform the numerical computation. Since the MSP-430 does not have more complex mathematical functions such as square roots built in, a way to implement it could be to use Newton's method to provide an iterative technique to estimate a square root. An interrupt could be used to resume the PID task once the computation is completed.

The reason this method was not used in the actual implementation was because it was found that the MSP-430 CPU running at 16 MHz would be far too slow to perform such computations, handle subsequent interrupts and compute PID output values all in real-time. This is particularly relevant because the MSP-430 is a RISC (Reduced Instruction Set Computer) device where each clock cycle executes a simple instruction, thus increasing the amount of clock cycles needed to execute different functions.

Ultimately, a hardware filter was chosen instead. The lowpass RC filter is used as an integrator circuit to convert from a PWM signal to an analogue voltage. The filter in the top and bottom circuits have a cutoff frequency of 0.0159 Hz and 0.159Hz, respectively.

The op-amp is used to compensate for the voltage loss following the lowpass filter. The necessary gain is calculated by

$$\text{Gain} = \frac{3.3\text{ V}}{V_{out}}$$

where V_{out} is the voltage output of the lowpass filter. The op-amp in the top circuit in Figure 8 has a gain of 2.2 and the op-amp in the bottom circuit has a gain of 1.5.

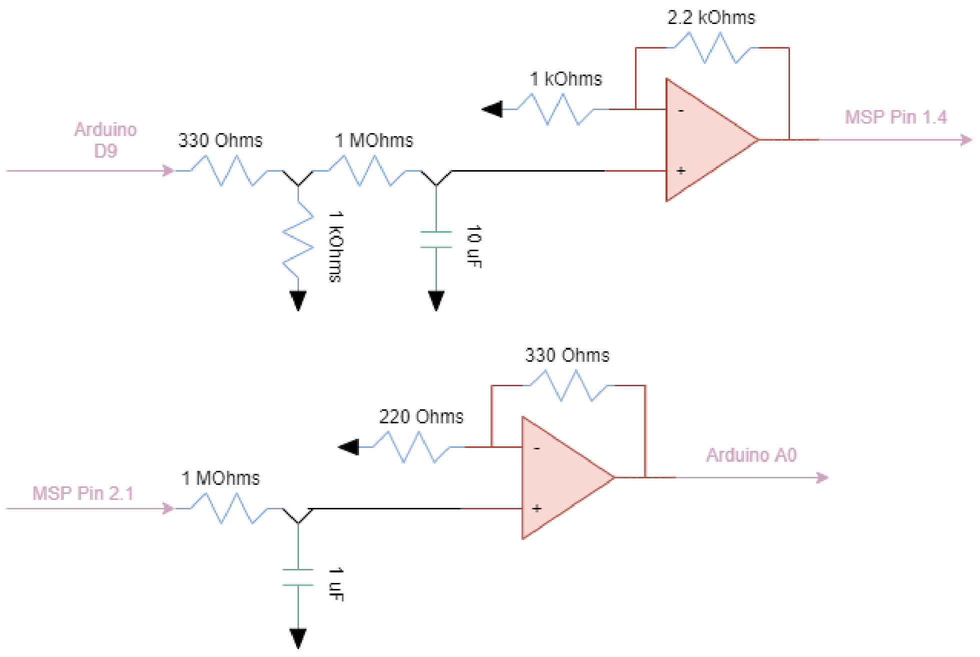


Figure 9: *Low-pass filter and operational amplifier circuit diagram*

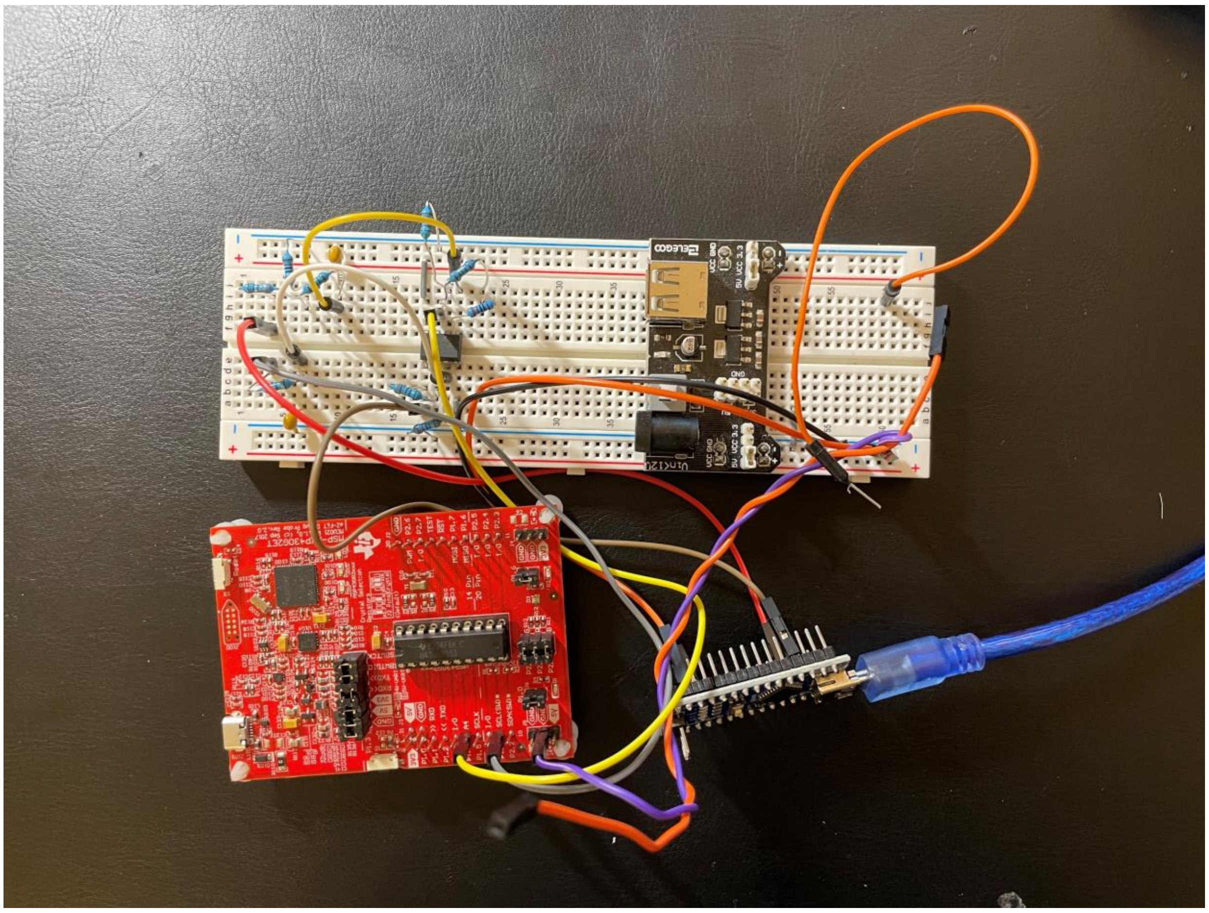


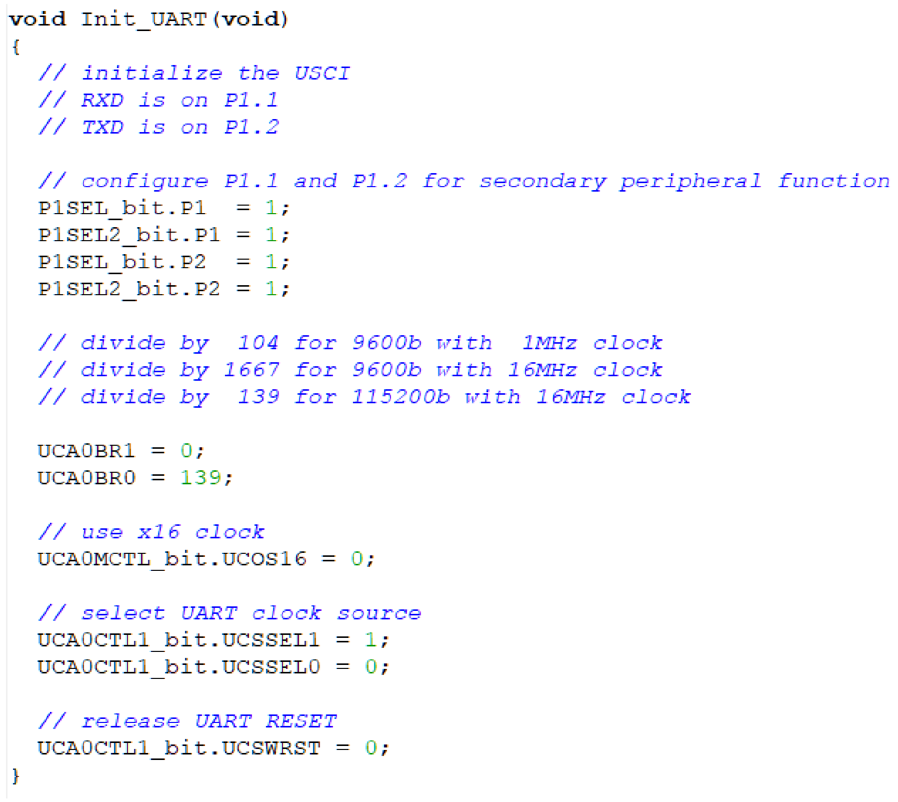
Figure 10: *Low-pass filter and operational amplifier circuit diagram*

Serial communication setup

The baud rate used in this project is chosen to be 115200. First, in the user PC, the baud rate must be set in the Device Manager as '115200'. Then, the assigned COM port is noted and configured in MATLAB via the command

```
comport = serial('COM10','BaudRate',115200);
```

The function call required to initialize the serial communication is provided in the Lab 5 documentation. The modifications required are to set the UCA0BR0 divider register to 139 in order to configure the MSP-430 for a 115200 baud rate using the 16 MHz clock, set the control bit UCA0MCTL_bit.UCOS16 to 0 to select the 16 MHz clock, and selecting it as the UART clock source:



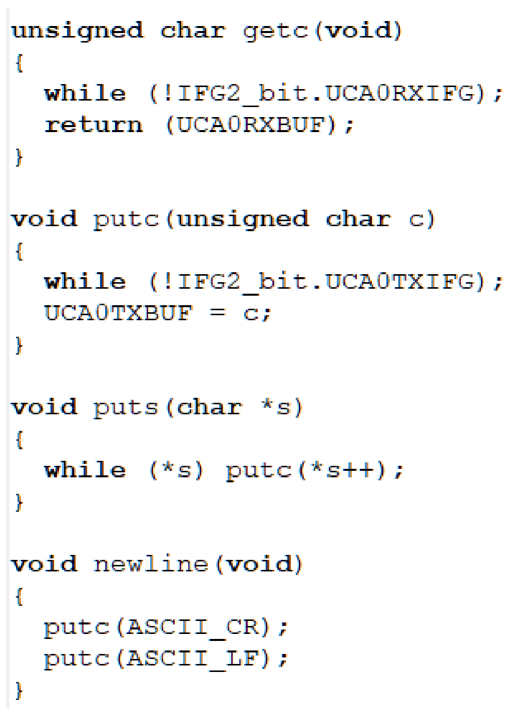
```
void Init_UART(void)
{
  // initialize the USCI
  // RXD is on P1.1
  // TXD is on P1.2

  // configure P1.1 and P1.2 for secondary peripheral function
  P1SEL_bit.P1  = 1;
  P1SEL2_bit.P1 = 1;
  P1SEL_bit.P2  = 1;
  P1SEL2_bit.P2 = 1;

  // divide by  104 for 9600b with  1MHz clock
  // divide by 1667 for 9600b with 16MHz clock
  // divide by  139 for 115200b with 16MHz clock

  UCA0BR1 = 0;
  UCA0BR0 = 139;

  // use x16 clock
  UCA0MCTL_bit.UCOS16 = 0;

  // select UART clock source
  UCA0CTL1_bit.UCSSEL1 = 1;
  UCA0CTL1_bit.UCSSEL0 = 0;

  // release UART RESET
  UCA0CTL1_bit.UCSWRST = 0;
}
```

Figure 11: *MSP-430 clock configuration C code*

For this project, P1.4 and P2.1 are used as input and output pins, respectively. They are defined by the commands

```
P1SEL_bit.P1  = 1;
P1SEL2_bit.P1 = 1;
P1SEL_bit.P2  = 1;
P1SEL2_bit.P2 = 1;
```

Subsequently, putc and getc functions are defined. These functions are used to ‘push’ and ‘pull’ specific characters to and from a data stream between devices.



```c
unsigned char getc(void)
{
  while (!IFG2_bit.UCA0RXIFG);
  return (UCA0RXBUF);
}

void putc(unsigned char c)
{
  while (!IFG2_bit.UCA0TXIFG);
  UCA0TXBUF = c;
}

void puts(char *s)
{
  while (*s) putc(*s++);
}

void newline(void)
{
  putc(ASCII_CR);
  putc(ASCII_LF);
}
```

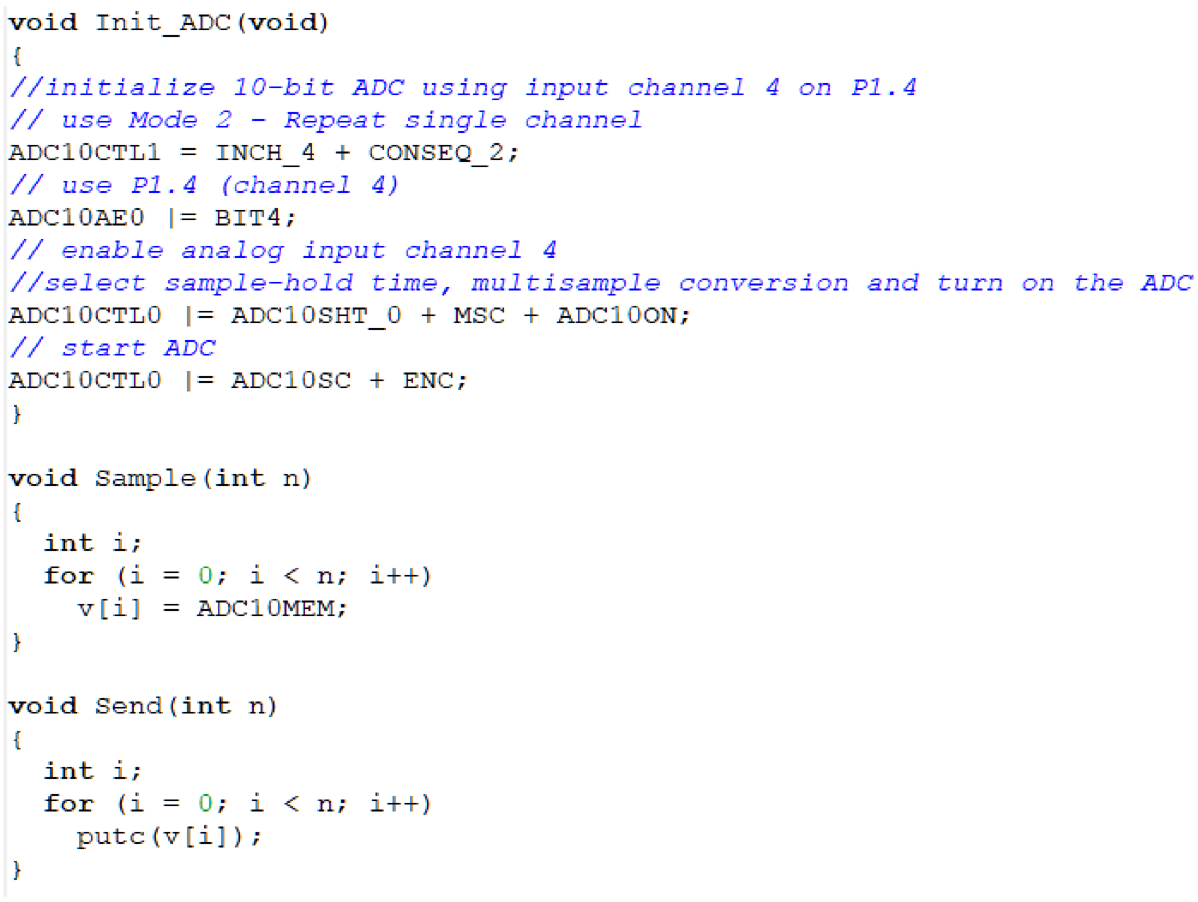
Figure 12: *getc & putc configuration C code*

ADC Module

The ADC module is set up as shown in Fig. 11 below. It is set up in Mode 2 using the

```
ADC10CTL1 = INCH_4 + CONSEQ_2;
```

command. Mode 2 of the MSP-430 ADC is known as the ‘repeat single channel’ which repeatedly reads and updates data (overriding the previous stored value) on a single channel.



```
void Init_ADC(void)
{
//initialize 10-bit ADC using input channel 4 on P1.4
// use Mode 2 - Repeat single channel
ADC10CTL1 = INCH_4 + CONSEQ_2;
// use P1.4 (channel 4)
ADC10AE0 |= BIT4;
// enable analog input channel 4
//select sample-hold time, multisample conversion and turn on the ADC
ADC10CTL0 |= ADC10SHT_0 + MSC + ADC10ON;
// start ADC
ADC10CTL0 |= ADC10SC + ENC;
}

void Sample(int n)
{
  int i;
  for (i = 0; i < n; i++)
    v[i] = ADC10MEM;
}

void Send(int n)
{
  int i;
  for (i = 0; i < n; i++)
    putc(v[i]);
}
```

Figure 13: *MSP-430 ADC configuration C code*

PID Control

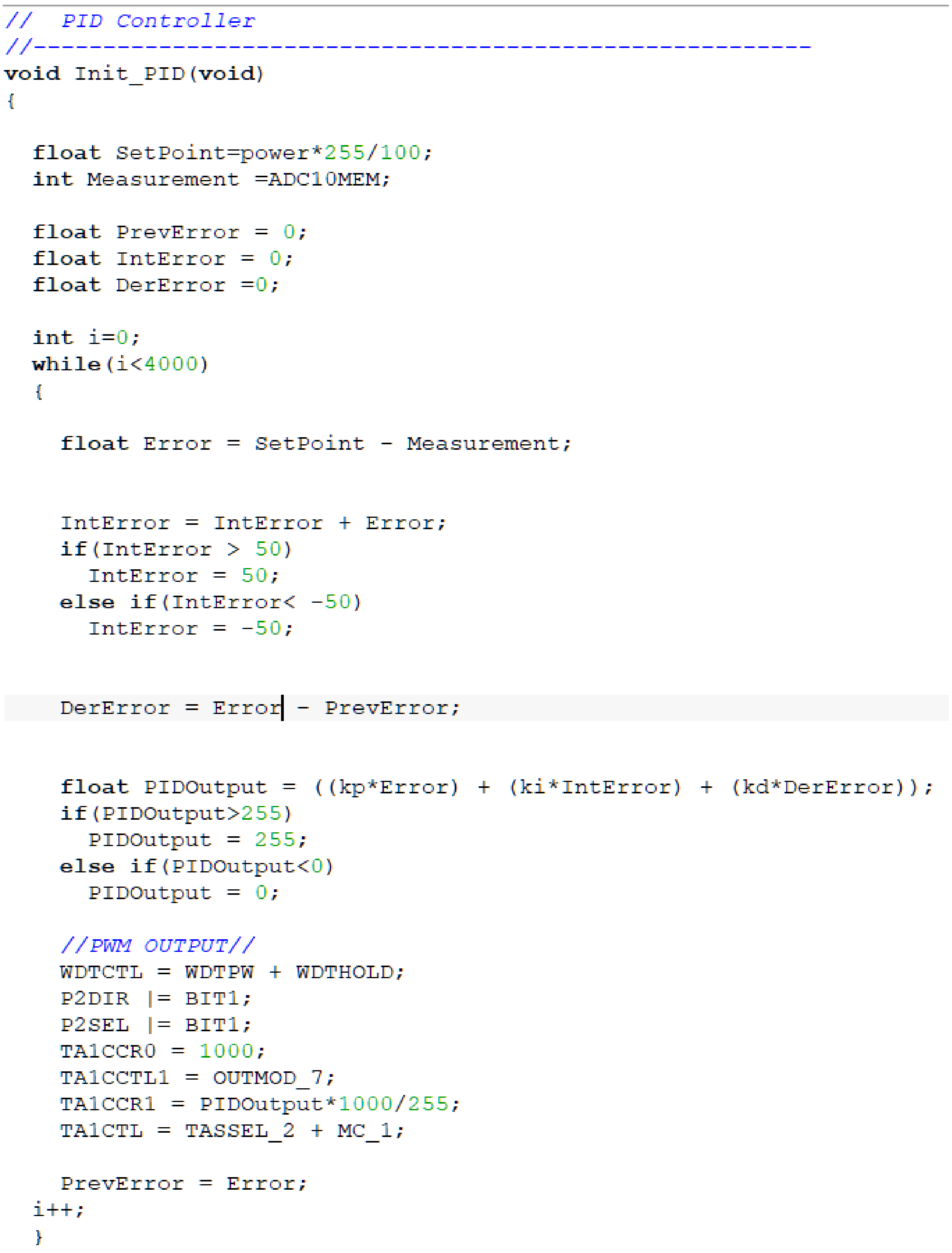
The PID control is set up as shown in Fig. 13. First, the float variables for the error terms are defined. Then the numerical computations to compute the output can be performed:

```
float PIDOutput = ((kp*Error) + (ki*IntError) + (kd*DerError));
```

Finally, the PWM output is configured using the PWM output PIN2.1 on the MSP-430. The line

```
TA1CCR1 = PIDOutput*1000/255;
```

defines the ‘ON’ time of the signal and thus the duty cycle.



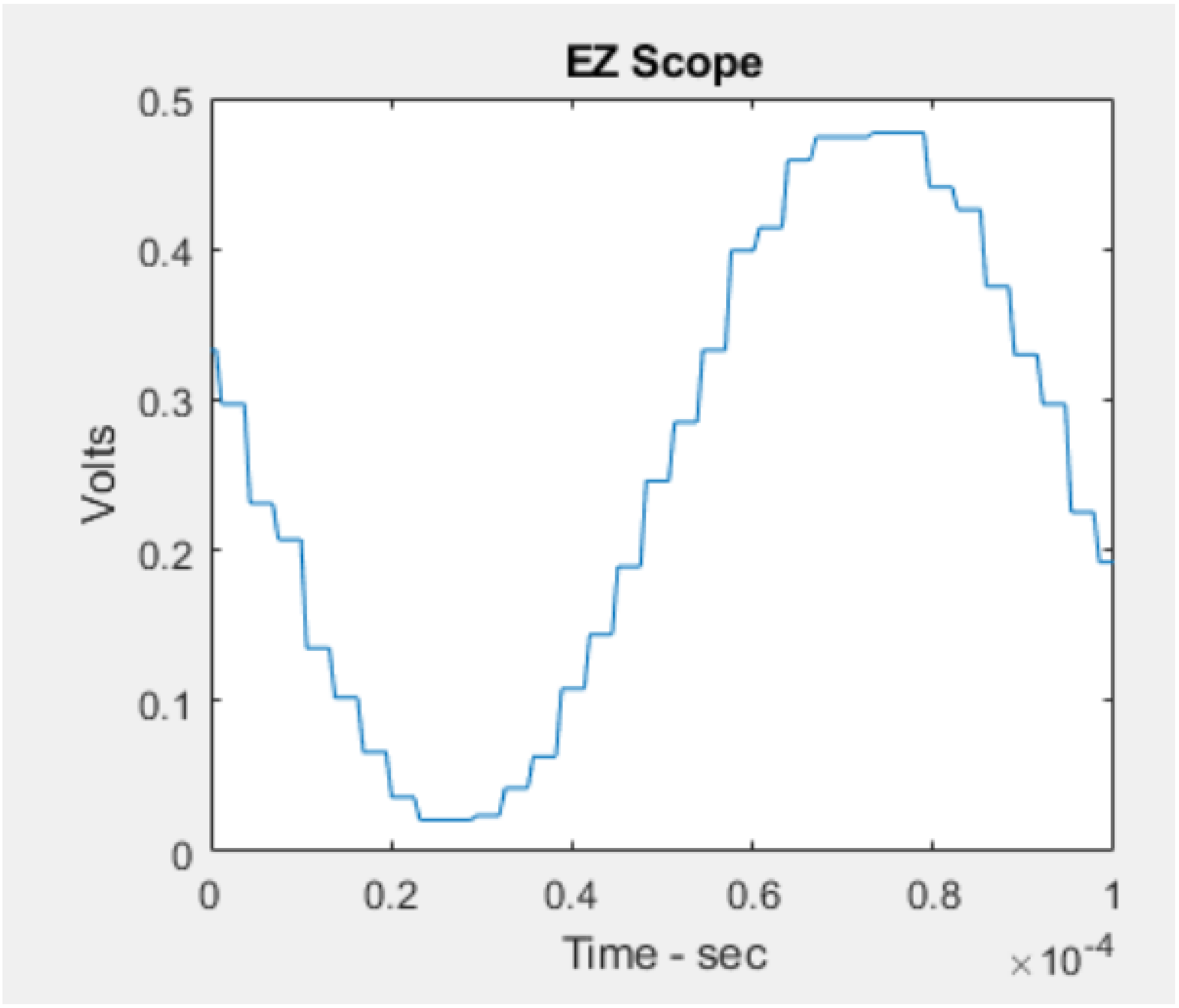
```c
//  PID Controller
//------------------------------------------------------------
void Init_PID(void)
{

  float SetPoint=power*255/100;
  int Measurement =ADC10MEM;

  float PrevError = 0;
  float IntError = 0;
  float DerError =0;

  int i=0;
  while(i<4000)
  {

    float Error = SetPoint - Measurement;


    IntError = IntError + Error;
    if(IntError > 50)
      IntError = 50;
    else if(IntError< -50)
      IntError = -50;


    DerError = Error - PrevError;


    float PIDOutput = ((kp*Error) + (ki*IntError) + (kd*DerError));
    if(PIDOutput>255)
      PIDOutput = 255;
    else if(PIDOutput<0)
      PIDOutput = 0;

    //PWM OUTPUT//
    WDTCTL = WDTPW + WDTHOLD;
    P2DIR |= BIT1;
    P2SEL |= BIT1;
    TA1CCR0 = 1000;
    TA1CCTL1 = OUTMOD_7;
    TA1CCR1 = PIDOutput*1000/255;
    TA1CTL = TASSEL_2 + MC_1;

    PrevError = Error;
  i++;
  }
```

Figure 14: *PID control and PWM configuration C code*

INTRODUCTION AND THEORY

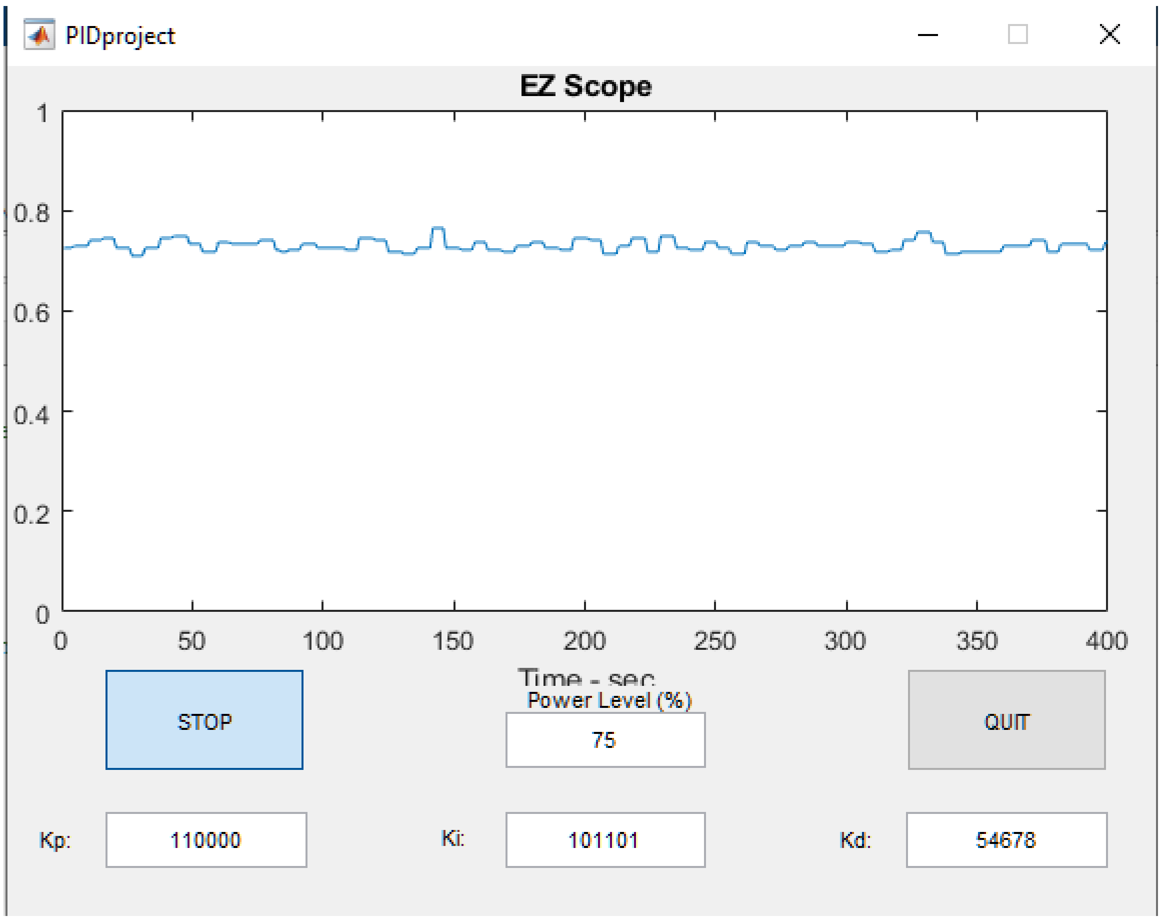
Results

In order to verify that the ADC sampling function and MATLAB GUI both work as intended, a 44KHz sine wave was passed into the MSP-430.



The waveform displayed is indeed close to that of a sine wave, showing that the aforementioned components were functional.

Then, the Arduino was configured and the PID control was run. The output shown below was configured for a set point of 75% and Kp, Ki and Kd values of 10, 10, 10, respectively.



Power Level (%) 75

Kp: 110000

Ki: 101101

Kd: 54678

Although from a solitary screenshot the power level appears to reach the set point, the transient behavior of the graph is much more peculiar. While the power level quickly settles at the set point after running the relevant programs, after a while it starts to display wild fluctuations between 0 and 1, and then settling back to 75%. The graph repeats this behavior on a consistent basis.

Modifying the Ki, Kp and Kd values does not appear to have a reliable effect on the oscillatory and damping behavior of the system, rather, it appeared that arbitrary sets of numbers would cause the output to not work at all without reason. While it is understood that there are heuristics methodology such as *Ziegler-Nichols* developed to quickly narrow down a best set of values, it seemed that in this context a trial-and-error approach was the best in terms of producing a result.

DISCUSSION & CONCLUSION

The individual component testing and the screenshots indicate that the individual components specified in the Theory and Methods section were successful in terms of implementation; but they did not work together as intended,

However, when testing the PID controller in MATLAB, it was found that it behaved abnormally and could not produce a reliable power level, although it would generally fluctuate around the user-define set point. The main observation was that when the circuit is powered on, the power level would slowly increase to the set point, settle there are a brief period of time, then shoot up to 1, drop rapidly to 0, then slowly increase until it reached the set point again. This behavior was consistently replicated.

Despite the fact that PID controller was not able to reliably produce the desired waveform during the presentaion, the testing demonstrates that in a vacuum, each of the target techniques specified in the 'Introduction' were satisfied: initial testing of the ADC Module showed that it functioned as aspect with a clean waveform comparable to that seen by the HANTEK oscilloscope; the filter and op-amp would produce clean analogue outputs up to 3.3V corresponding to the equivalent PWM input; and the PID control showed *some* tendency to gravitate towards a desired set point. As a result, it can be concluded chosen hardware filter in the circuit schematic, as well as the execution of the C and MATLAB code were all largely successful towards objectives of this Design Project.

Conclusion

The Design Project aimed to consolidate and make use of the electronic theories and laboratory work with the Texas Instruments MSP430 microcontroller over the duration of the course so far to create a PID controller for the Liquid Zone Control (LZC) of a CANDU nuclear reactor. The implementation showed that the PWM reading, ADC sampling were successfully, and PID feedback implementation was large successfully as well though it did not perform consistently. This lab was the culmination of critical fundamental techniques learned and practiced through the course, including embedded C programming, clock and timer configuration, ADC module usage, serial communication and sampling, and they were all drawn upon in the implementation of this PID controller to great effect.

Appendix

1. https://en.wikipedia.org/wiki/Pulse-width_modulation
2. https://en.wikipedia.org/wiki/Symbol_rate
3. https://en.wikipedia.org/wiki/PID_controller
4. http://www.ocfreaks.com/msp430-timer-programming-tutorial/