ECE 4760

Lab 4: 1 DOF Helicopter

Author:	*NetID:*
Jonah Okike-Hephzibah	jo356
Sebastian Roubert	sr949
Rahul Desai	rd542

Section: Thursday 4:30pm
Performed on October 26th, 2017

Contents

List of Figures

1 Introduction

In this lab, we sought to build a one degree of freedom helicopter. The helicopter was composed of a wooden beam with a small motor on the end. On the first end of the beam was a potentiometer that could read the angle of the helicopter. The motor speed was controlled by pulse width modulation via dedicated circuit with a voltage supply of 4.5V. The helicopter reaches the desired angle via a PID controller that dictates the duty cycle of the pulse width modulation.

There is a startup sequence where the helicopter moves from -90 degrees to 0 to 30 to -30 then back 0 over a period of 20 seconds. Afterwards, the user can control the desired angle, proportional gain, derivative gain, and integral gain via two buttons and 10k trim potentiometer. The first button cycles through the user selections and second is for entering the value. The user alters the value via the trim potentiometer.

We succeeded in making a one degree of freedom helicopter. It worked fully and was critically damped when moving from -90 to 0 degrees. We displayed stable behavior for all angles between -90 and 90 degrees from the horizontal. All specs were met with success.

2 Design and Testing

2.1 Hardware

2.1.1 Motor Driver

The motor is controlled by the PWM signal generated by the PIC32. The current still flows from the circuit during the negative cycle of the signal, which can severely damage the circuit transistors. So a diode (1N4001) is placed in parallel with the motor to protect it and to avoid inductive loading of the system.

Diode working: The current doesn't completely stop flowing through the motor even in the 'OFF' period but still flows for a short amount of time. This results into a back EMF, which can severely damage the sensitive circuit transistors and gravely affect the system response. Also, the DC motor adds voltage spikes which contribute to the damage. So as a safety measure, the diode 1N4001 is used in parallel with the motor. This diode acts as a short circuit and the currents passes through this path instead of the motor thereby avoiding any damage to it. The optoisolator 4N35 as seen in the Figure 2 (Courtesy of Bruce Land), which consists of a BJT, isolates the MCU from the circuitry for protection.

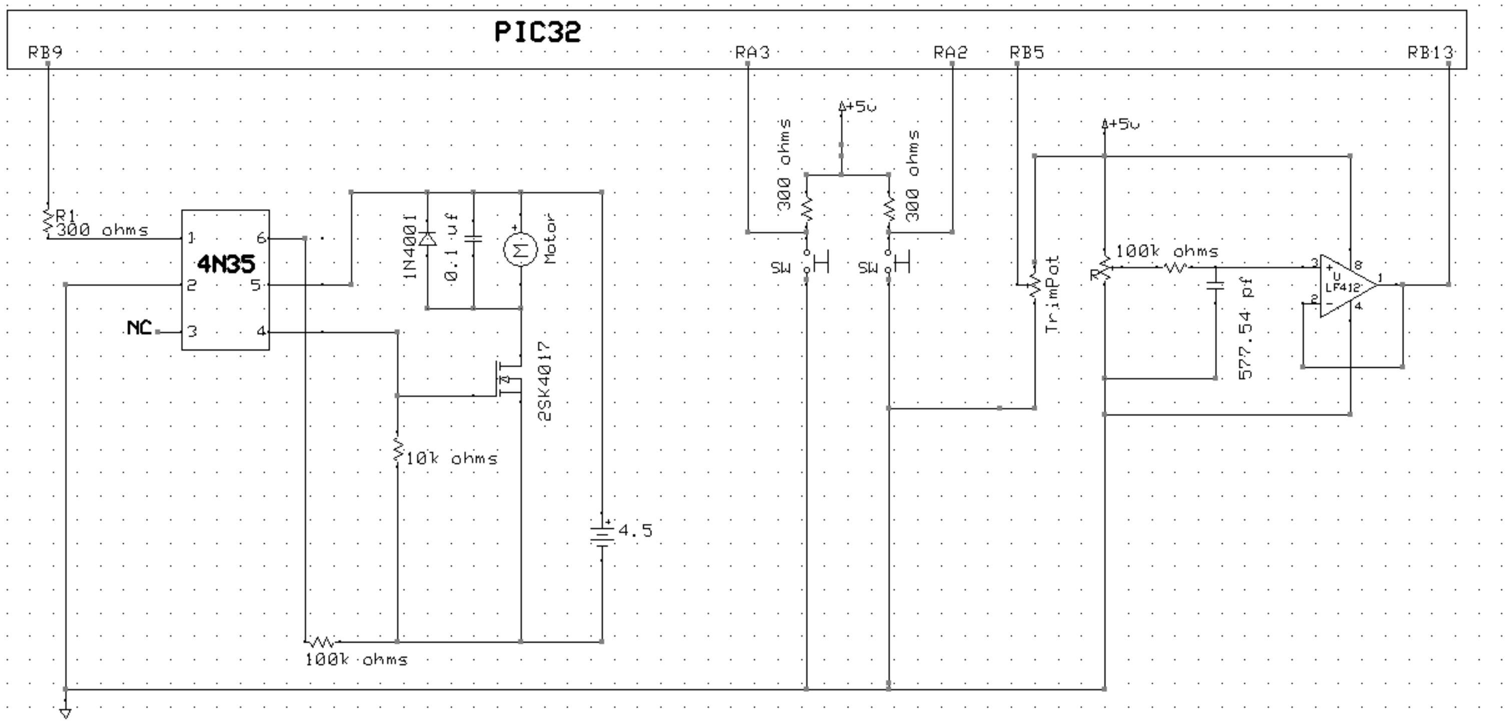


Figure 1: Overall schematic.

2.1.2 Potentiometer Filter

The potentiometer is an Angle Sensor which outputs the corresponding resistance associated with the particular angle and is in charge of the angular movement of the 1-DOF Helicopter. This sensor is connected to a current limiting resistor (100k, pin2) to ensure the power ratings don't go beyond the specified values.

The Opamp MCP6242 as seen in the Figure 3 (courtesy of Bruce Land) acts as a unity gain buffer for the anti-aliasing RC filter whose Capacitance is calculated by the following formula(with the frequency set to 2500 Hz)

F = 1/(2*Pi*R*C)

The calculated value of C was: 636.943 pF

The hardware setup is as follows:

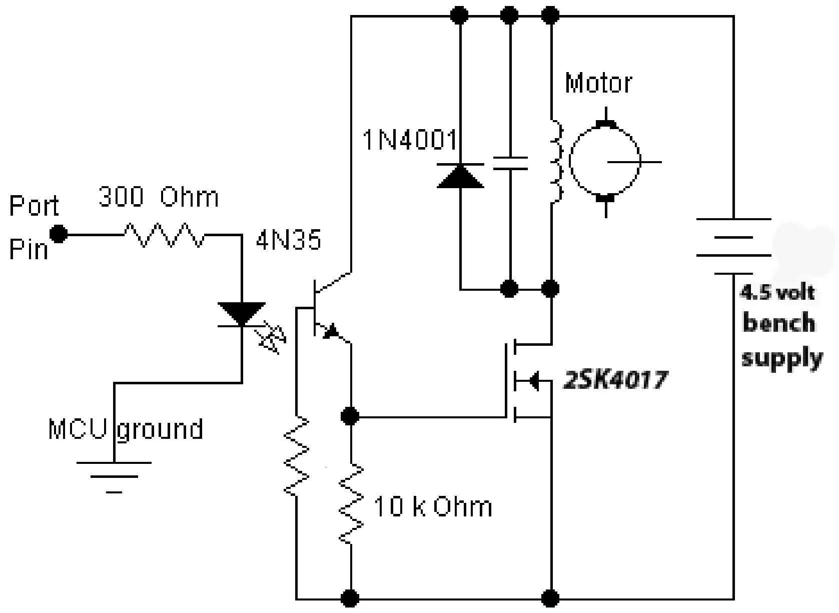


Figure 2: Hardware diagram for the Motor control circuit

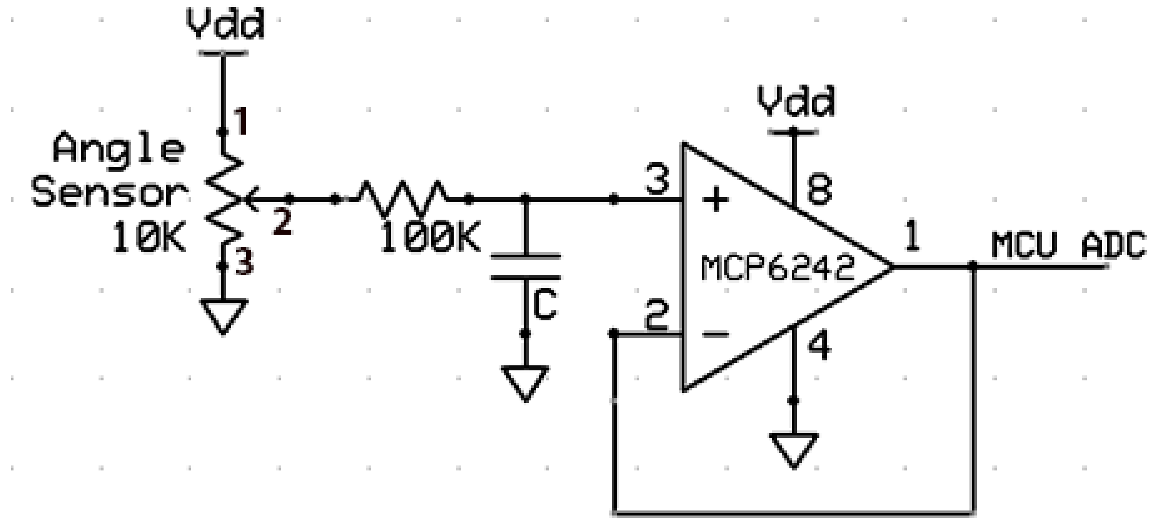


Figure 3: Hardware diagram for the Angle Sensor

2.1.3 Trim Pot. & Buttons

The buttons and the trim Pot were used for the user interface; One push button was used to prompt the user to enter the values with the help of the Trim Pot and the other one was used to change the

parameters. The trim Pot was then used to change these values with the help of a screwdriver. The trim Pot was connected to pin RB5 whose value was then read by an ADC channel.
The analog pins RA2 and RA3 are directly connected to VCC via resistors, so when the buttons are pushed, the pins are connected to ground as seen in Figure 1.

2.1.4 Building the helicopter

The 1 DOF Helicopter had to hover in one dimension so an angle sensor, which is a potentiometer was joined at one end of the helicopter shaft with the motor hot glued to the other end. The motor kept coming off from the shaft maybe because of the heat generated and had to be glued again to it.

The wires, which were soldered to the motor added weight and torque of their own, so care was taken to route them as close as possible to the potentiometer.

The shaft, which was attached to the motor and the beam, was adjusted to a 1/2 full resistance of 4.5k ohms for the ease of calculations later. While building the setup, the some filing was needed for the potentiometer hole on the bracket as it was too small for the potentiometer sleeve.

As the helicopter could go past full vertical, there was a need to put a rotation stop as well for safety purposes and also so that the setup wouldn't get damaged.

The circuit was made and was connected to the mechanical setup for the input and control signals.

2.2 Software

2.2.1 ADC and DAC

We used two instances of analog to digital (ADC) conversion for this. Specifically, we converted the trim potentiometer angle that the user to interact with the software and the helicopter potentiometer angle. These ADC units varied from 0 to 1023 and were the values used for comparison.

To monitor the motor angle and the motor displacement, we used the digital to analog (DAC) pins on the MCU. Specifically, we used DAC channel A and B to out the motor potentiometer ADC value and motor displacement, respectively.

We defined the motor displacement as the previous motor displacement plus the difference from the current duty cycle and motor displacement divided by 16. The code for this was

```
motor_disp = motor_disp + ((pwm_on_time - motor_disp)>>4);
```

We found that the division by 16 decreased motor noise most.

We transmitted to the DAC channels via the serial peripheral interface (SPI). See the appendix section 7.1 for detailed listing.

2.2.2 Button Pressing

There were two buttons used in our system. The first button allows the user to cycle through which parameter (desiredAngle, K_p, K_d, and K_i) is selected by user to alter. In the software, the current selection is represented by a variable, inputVal.

The second button allows the user to select the parameter for alteration and then a second press of the second button loads the new parameter for computation.

The button pressing functionality was fully handled by buttonCheck protothread. The protothread utilized a rather crude debouncing method that worked functionally well.

The protothread, buttonCheck, yields for 250ms. It checks if the buttons are pressed. It then increments how many times the button has been pressed. For example, for the first button:

```
        //adding how many times the first button has been pressed
        firstButton += (int) (mPORTAReadBits ( BIT_3)==0);
```

If the button has been detected as pressed 3 times, it is then pushed. For example, for the first button:

```
        //if first button has been read as pressed twice, declare as push
        if ( (mPORTAReadBits ( BIT_3)==0) && (firstButton > 2)) {
           firstPush = 1;}
```

An else if statment was utilized to reset the variables once there was no longer a press. For example, for the first button:

```
    //if first button is not pressed but a push was previously record then
        //reset values
    }else if ( (mPORTAReadBits ( BIT_3)==8) && (firstButton>2)) {
        firstButton = 0; firstPush = 0;
    }
```

As mentioned earlier, a push for the first button is used to cycle between parameter selections and the second button was used to enter them into the system for computation. A second button pushed altered a variable, enter, that signaled whether to alter the value or enter the value for computation. When it wasn't pushed, the trim pot value was read and altered the proposed value, represented by an array propVal[]. This can be seen for the angle parameter below:

```
if (enter == 0 ) {
        trimPot = ReadADC10(1); //read trimPot value
        if (inputVal == 0) {
            // Do conversion for angle
            propVal[0] = 0.00307*(float)trimPot - 1.57; //radians
            propVal[4] = (0.5181 * (float) trimPot) + 200; //degrees
```

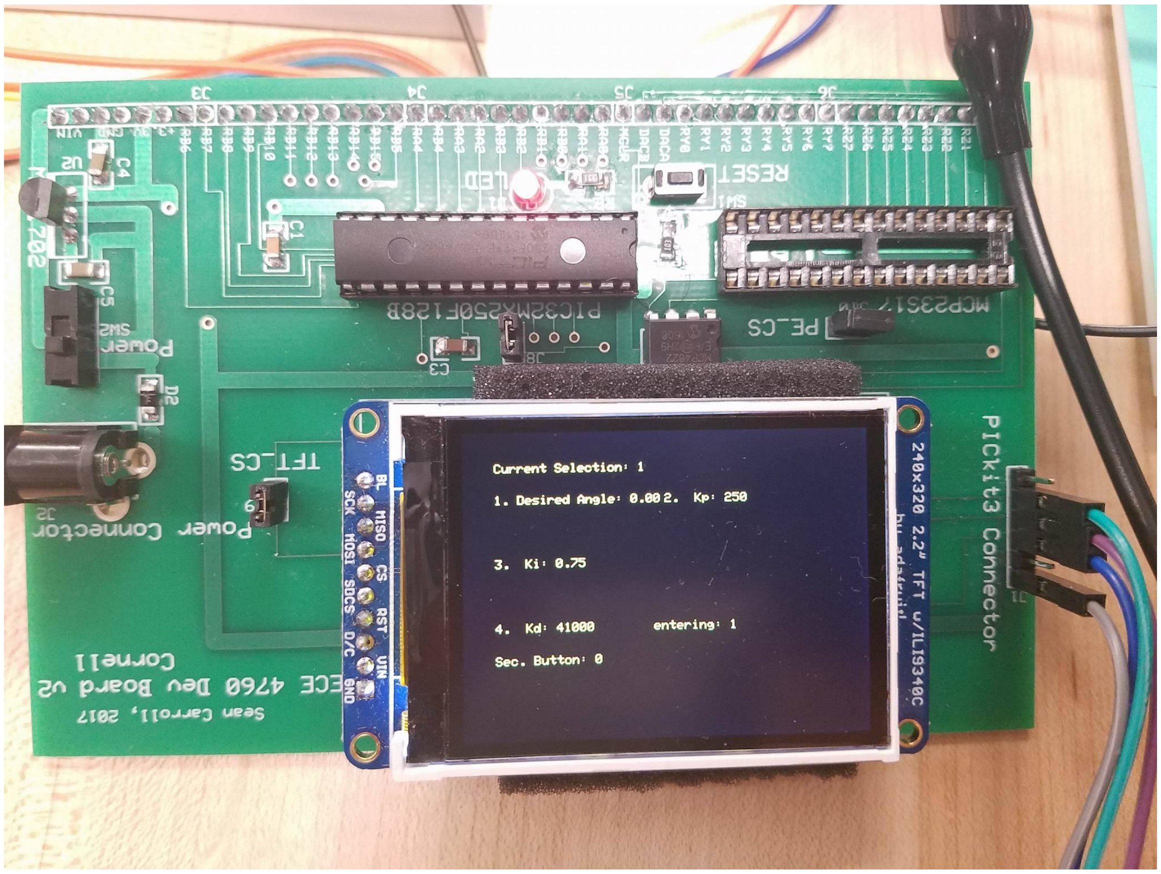
Once the user entered the value for computer, it was directly altered in software. This can be seen for the desired angle and Kp:

```
    //if user has entered values then update the chosen value
    } else if (enter == 1) {
        if (inputVal == 0) {
            if (enter == 1){
            desiredAngle = (int) propVal[4] ;} else desiredAngle = desiredAngle;
        } else if (inputVal == 1) {
            Kp = (int)propVal[1] ;
```

2.2.3 GUI

The GUI was rather straightforward. On the screen, we showed the parameter values of interest (desired angle, Kp, Ki, and Kd) along with the current selection of the user and whether they were currently entering values or if they had just entered. See **figure 4** for an example. This was all a simple implementation of the TFT graphics library. The one part of interest is the updating the display based on the enter parameter. We displayed on the screen whether or not the user was entering or just entered with the following:

```
if (enter == 1) {
    tft_setCursor(150, 150);
    tft_fillRoundRect(150,150, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
    tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
    sprintf(buffer,"entered: %d", inputVal+1);
    tft_writeString(buffer);
}
if (enter == 0) {
    tft_setCursor(150, 150);
    tft_fillRoundRect(150,150, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
    tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
    sprintf(buffer,"entering: %d", inputVal+1);
    tft_writeString(buffer);
}
```



```
Current Selection: 1

1. Desired Angle: 0.002.  Kp: 250


3.  Ki: 0.75


4.  Kd: 41000          entering: 1

Sec. Button: 0
```

Figure 4: Example GUI.

2.2.4 Motor Control and PID

The speed of the motor was controlled by pulse width modulation (PWM) to control the effective voltage supplied to the motor without directly altering the voltage supply. See the hardware section for full description of the circuitry.

The pin that controlled the PWM was pin 18. This was done by tying configuring the pin to output compare on PWM mode. The output compare used was tied to timer 2, which had an overflow period of 40,000, represented as the variable generate_period. Therefore, the duty cycle of the motor was the ratio of the on time of the output compare and timer period, the on time being represented by the variable pwm_on_time. See below for code configuration:

```
// === Config timer and output compares to make pulses ========
// set up timer2 to generate period
```

```
OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_1, generate_period);

// set up compare3 for PWM mode
OpenOC3(OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE , pwm_on_time, pwm_on_time); //
// OC3 is PPS group 4, map to RPB9 (pin 18)
PPSOutput(4, RPB9, OC3);
```

We controlled the angle by altering the motor speed via directly altering the duty cycle by the pwm_on_time.

The pwm_on_time was controlled via a PID controller. The PID was completely interrupt-driven. We configured the interrupt to fire at 1kHz.

PID stands for proportional-integral-derivate controller. This means that the controller takes into account the current error, the derivative of the error, and integral of the error to decide the output of the system to correct to zero error.

The parameters for error were defined as follows:

```
  //calculate error
  error = trimConv - potenVal; //trim pot value minus motor potentiometer
  //parameters for control
  integral += error;
  derivative = error - prevError;
```

The controller sets the output as the sum of error, derivate, and integral times their respective gains (proportional gain, derivate gain, and integral gain). See below:

```
  //alter duty cycle based on PID algorithm
  pwm_on_time= Kp*error + Kd*derivative + Ki*integral;
```

With PID, the gains are tuned such that system is stable, has zero steady state error, and is critically damped. The proportional gain, K_p determines the response to a step input, K_i determines reaching steady state error, and K_d determines the damping. To ensure that our system met all three criteria we followed a three step process:
1) With $K_d = K_i$=0, we tuned K_p such that the response for -90 degrees to 0 had little overshoot.
2) With a small K_d and the K_p from step 1, we increased K_i until the system reached a steady state error near zero.
3) With keeping K_p and K_i the same as step 2, we increased K_d until the system was critically damped, i.e. reached steady state error of 0 with no overshoot in the quickest time possible.

We included some other measures to ensure our system behaved accordingly for a large range of angles. We zeroed the integral value below angles of about -165 so that it did not have too much influence on our response early on. We set hard boundaries of 0 and 40000 for our pwm_on_time so that we always had valid duty cycle values. Perhaps the most important measure was that if the error ever crossed 0, we changed the integral value to 99% of its value so that helicopter would always hover around zero steady error and the pwm_on_time would never become negligibly small once it reached zero error. This was implemented with the following:

```
//if the error crossed 0, change integral so that
//controller stays active
if (sign(error) != sign(prevError)){
    integral=(integral*999)/(1000);
}
```

2.2.5 Start Up Sequence

To generate the start up sequence we had to measure the system time and check where we were in the 20 second start up time. Every 5 seconds we altered the desiredAngle. This was done with the following:

```
//if cascade is to check if still in start up sequence and
//if so change desired angle. If out of startup sequence, the
//if-cascade defaults to final else statement, where the desired angle
//is set by user input
if (sys_time_seconds-timeVal > 5) {
   j += 1 ;
    if ( j < 4 ) {
        timeVal = sys_time_seconds ;
        trimConv = refAngle[j] ;
        // angle
    } else {
        trimConv = desiredAngle ;
    }
}
```

To ensure that the user input would not alter the start up sequence, we had the buttonCheck protothread only run if the system time was above seconds. See the following:

```
// schedule the threads
while(1) {
  PT_SCHEDULE(protothread_time(&pt_time));
  PT_SCHEDULE(protothread_tftDisplay(&pt_tftDisplay)) ;
  if (sys_time_seconds > 20) PT_SCHEDULE(protothread_buttonCheck(&pt_buttonCheck));
}
```

3 Documentation

static PT_THREAD (protothread_buttonCheck(struct pt *pt))

The purpose of this thread was to check the user button pressures through a crude debouncing method. Basically, it yields for 250ms and if after two yielding instances the button has been read as pressed then it is considered a push. The first button is used to cycle through user inputs. A push of the first button iterates the variable $inputVal$. The second button is used to check if the user is entering the current value or if the value is entered/should be updated based on the current value.

static PT_THREAD (protothread_tftDisplay(struct pt *pt))

The purpose of this thread was to update the TFT display. When the user is enter values, the value does not change in the software but does update on the screen to allow the user to see up to where they are changing.
static PT_THREAD (protothread_time(struct pt *pt))

The sole purpose of this thread is to update the system time in seconds for use in the startup sequence and for user awareness. It uses a protothread that yields for a second and increments the variable sys_time_seconds.

4 Results and Discussion

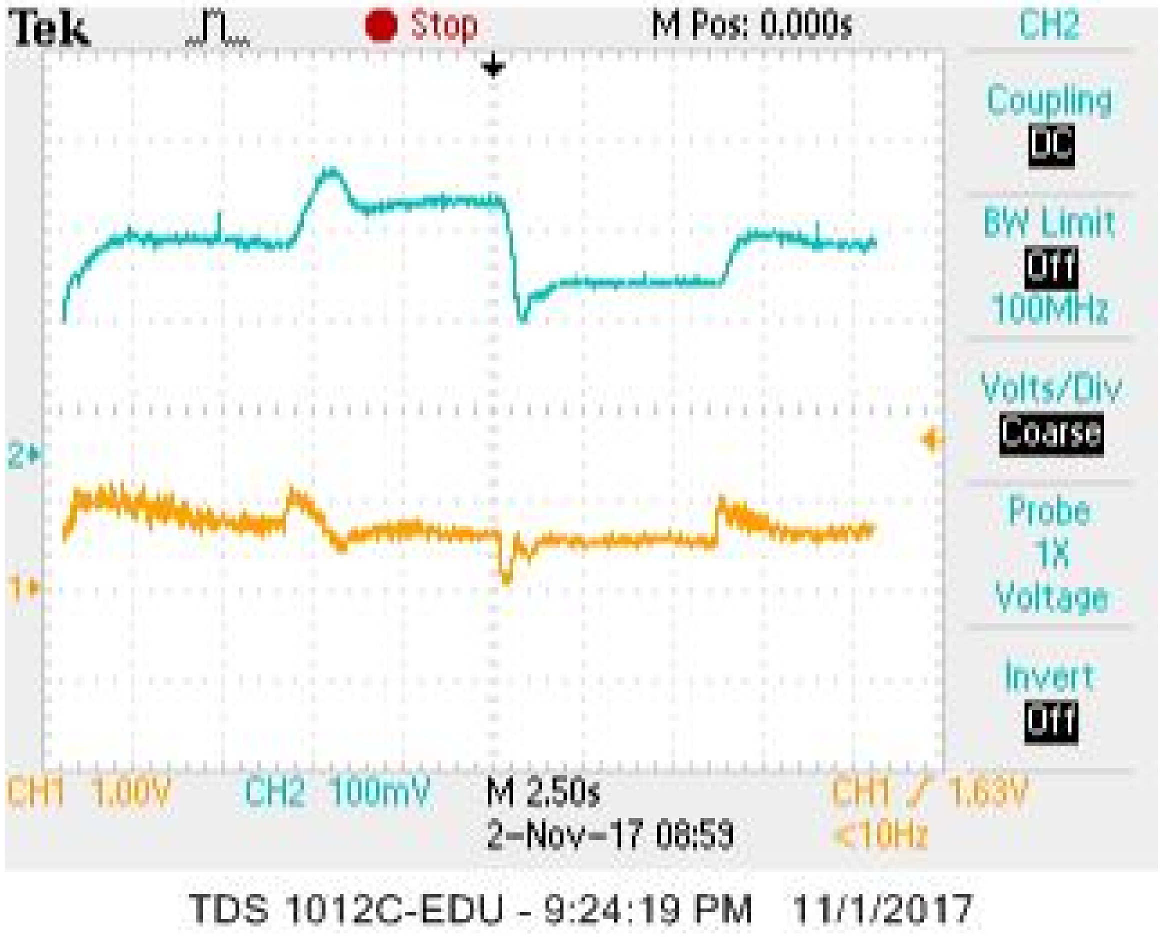


Figure 5: Full Plot of Startup Sequence. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

Our controller performed reasonably well during the startup sequence (figure 5). The step from the horizontal to positive 0.79 rad (45 degrees) had a 10-90% rise time of 400 milliseconds (figure 6) and a 2% settling time of 1.5 seconds (figure 7). This is because there was a slight overshoot in the step. The step from the positive 0.79 rad mark to the negative 0.79 rad mark had a rise time of 240 milliseconds (figure

8) and a settling time of 1.2 seconds (figure 9). For this step, the controller turned the motors off allowing gravity to drive it down and turned the motors back on when the beams end approached the desired angle. This explains why the rise time in this case is faster than that of the step from the horizontal. Finally, the rise time for a step input from the negative 0.79 rad mark to the horizontal (zero degrees) was 520 milliseconds (figure 11) with a settling time of 1.52 seconds (figure 10). This is pretty consistent with the step from the horizontal to 45 degrees above the horizontal so we are very satisfied with our controllers performance during this startup sequence.

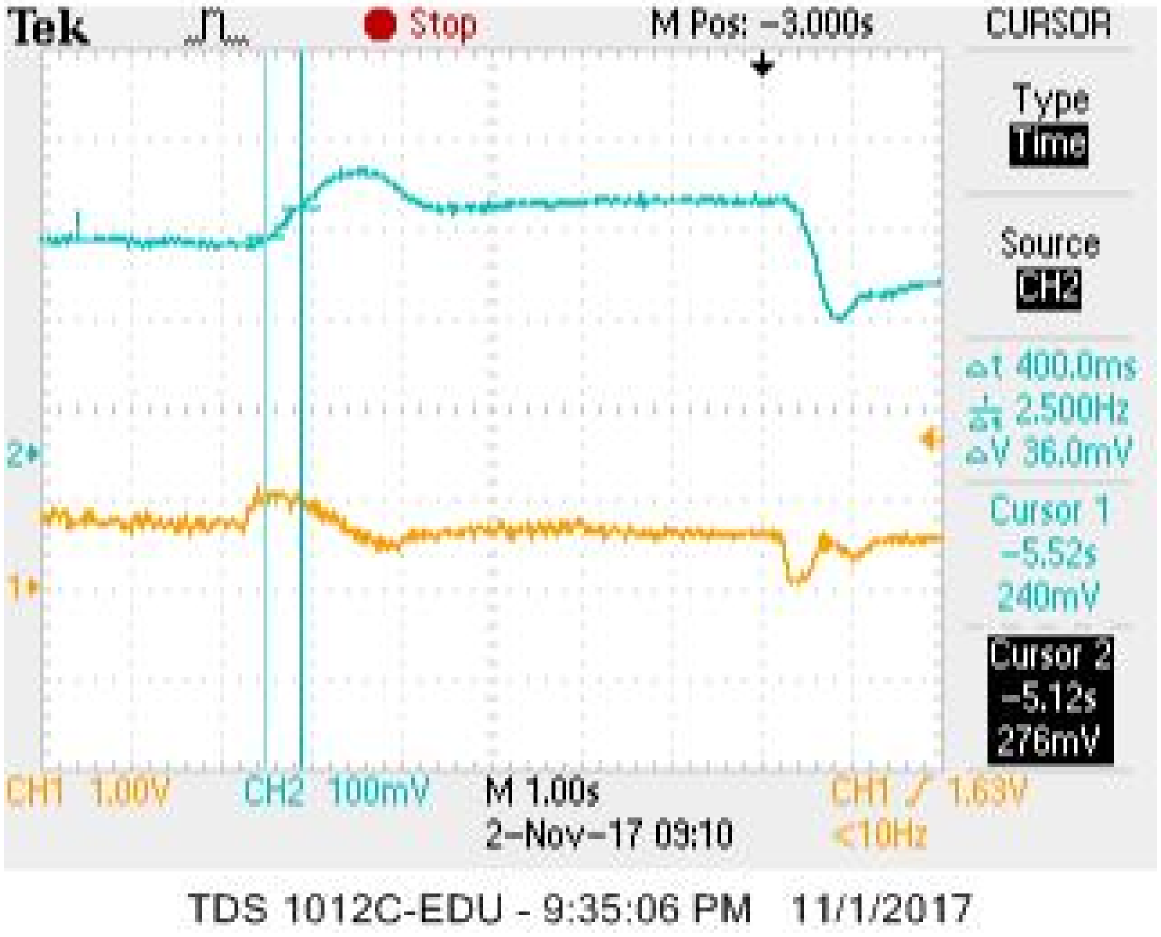


Figure 6: Rise Time for the Second Transition in the Startup Sequence. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

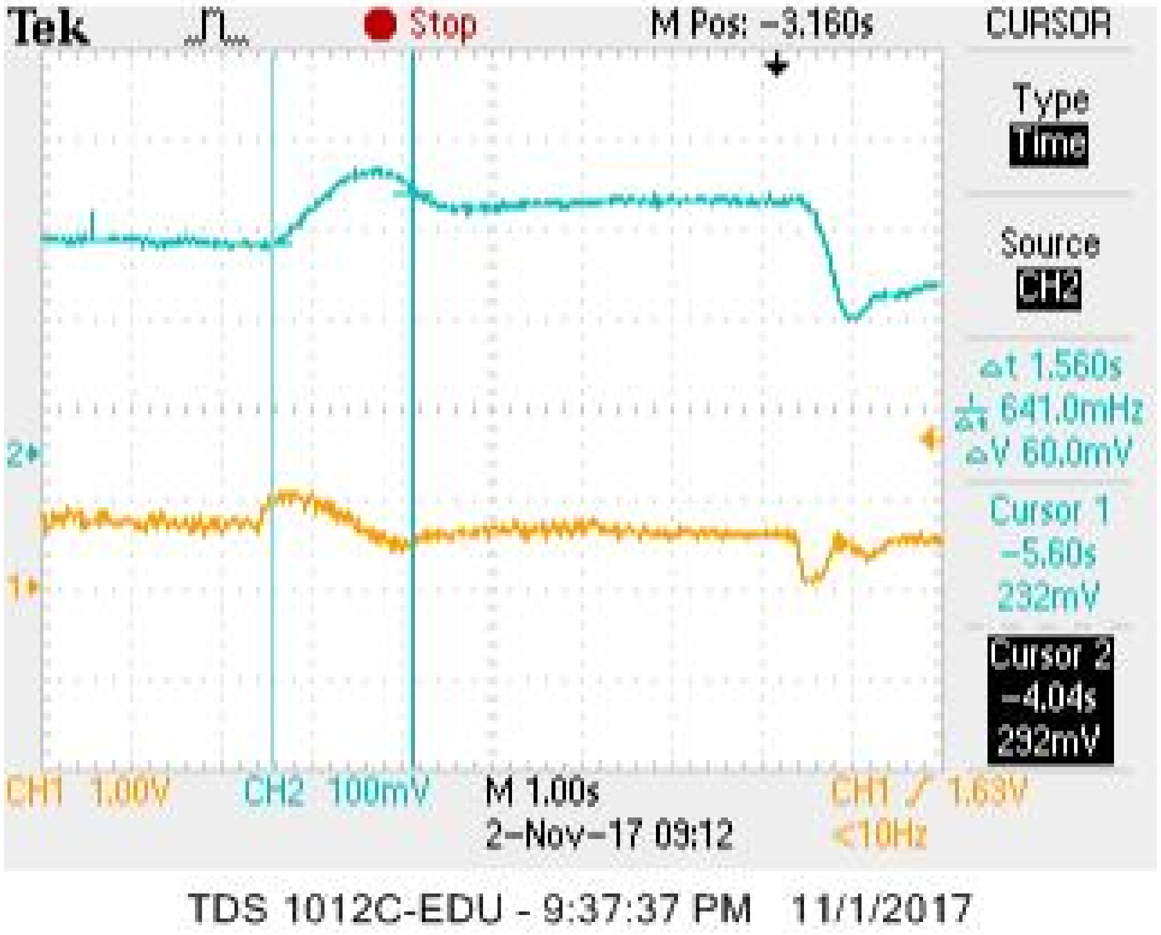


Figure 7: Settling Time for the Second Transition in the Startup Sequence. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

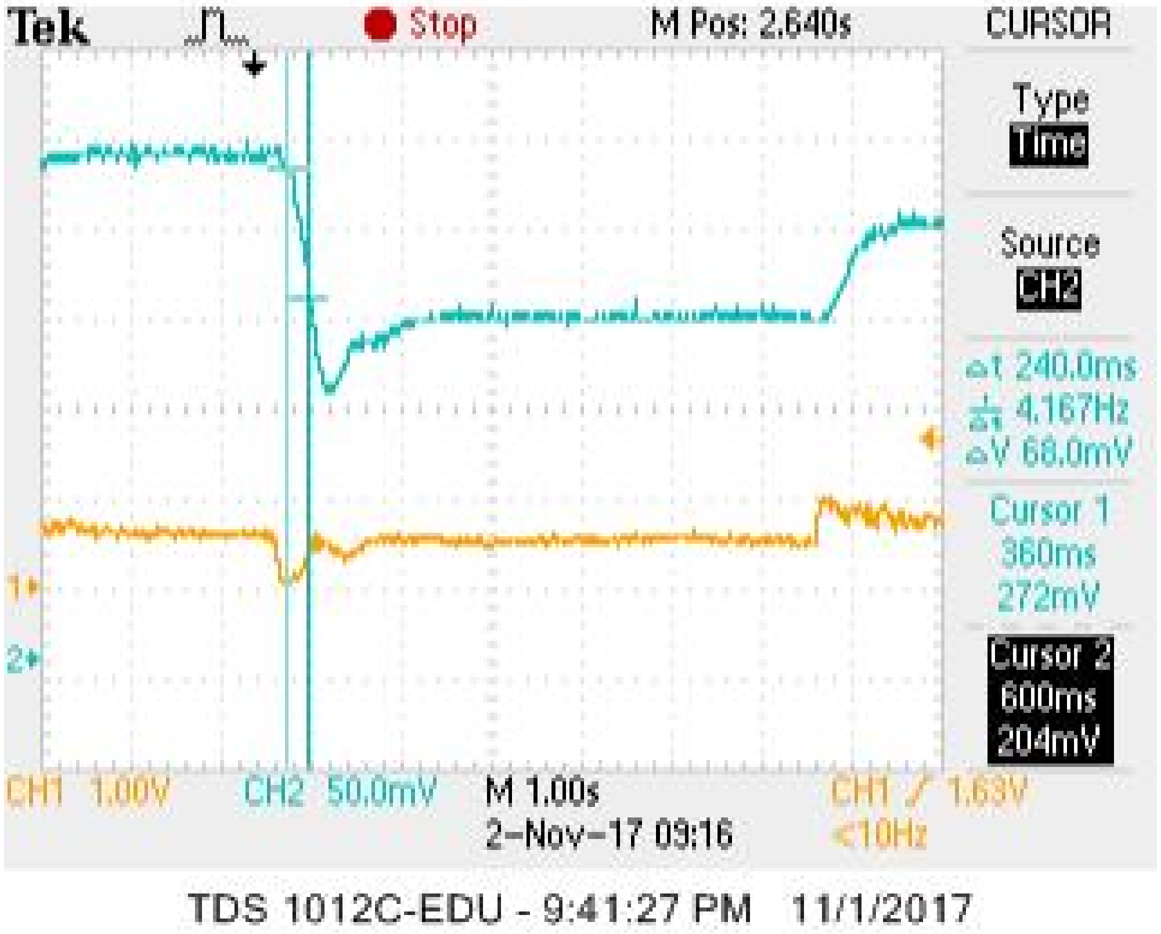


Figure 8: Rise Time for the Third Transition in the Startup Sequence. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

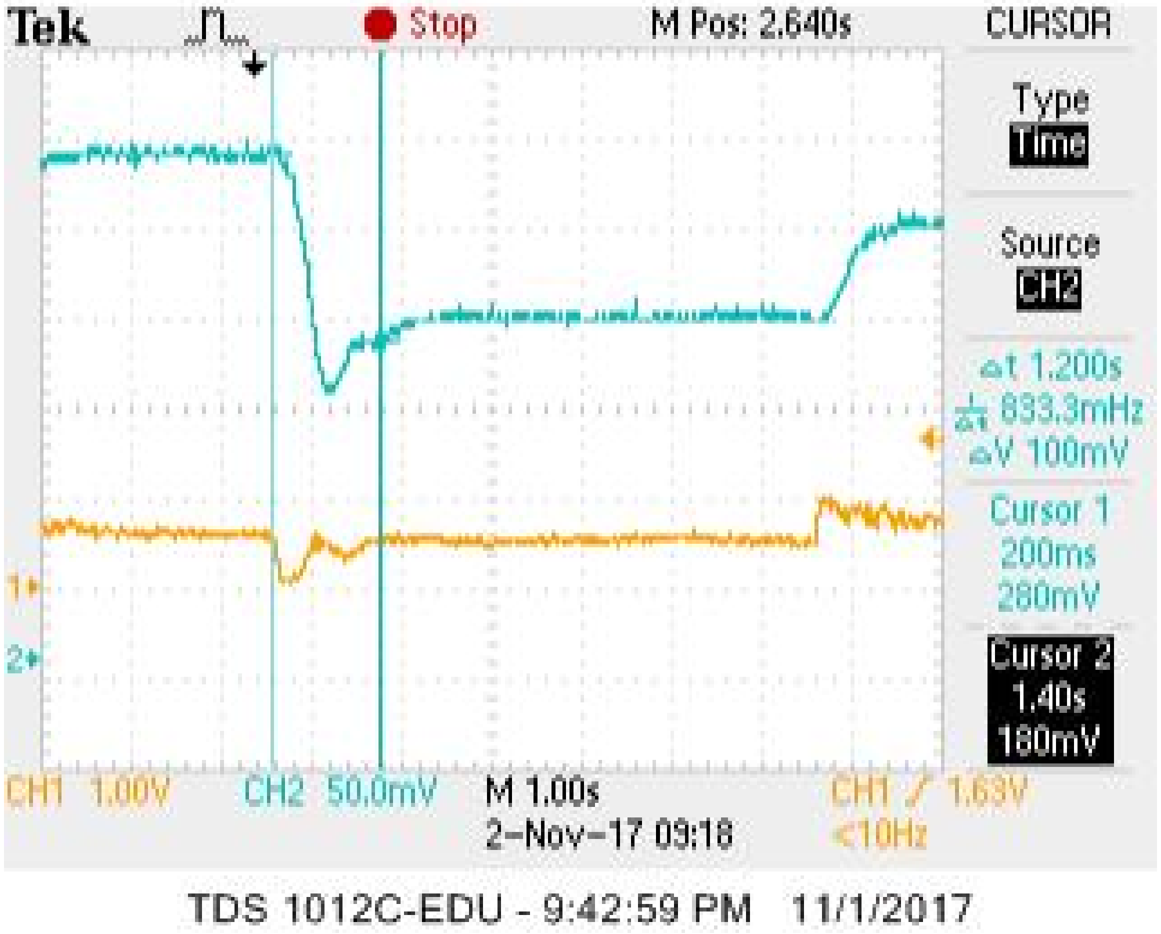


Figure 9: Settling Time for the Third Transition in the Startup Sequence. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

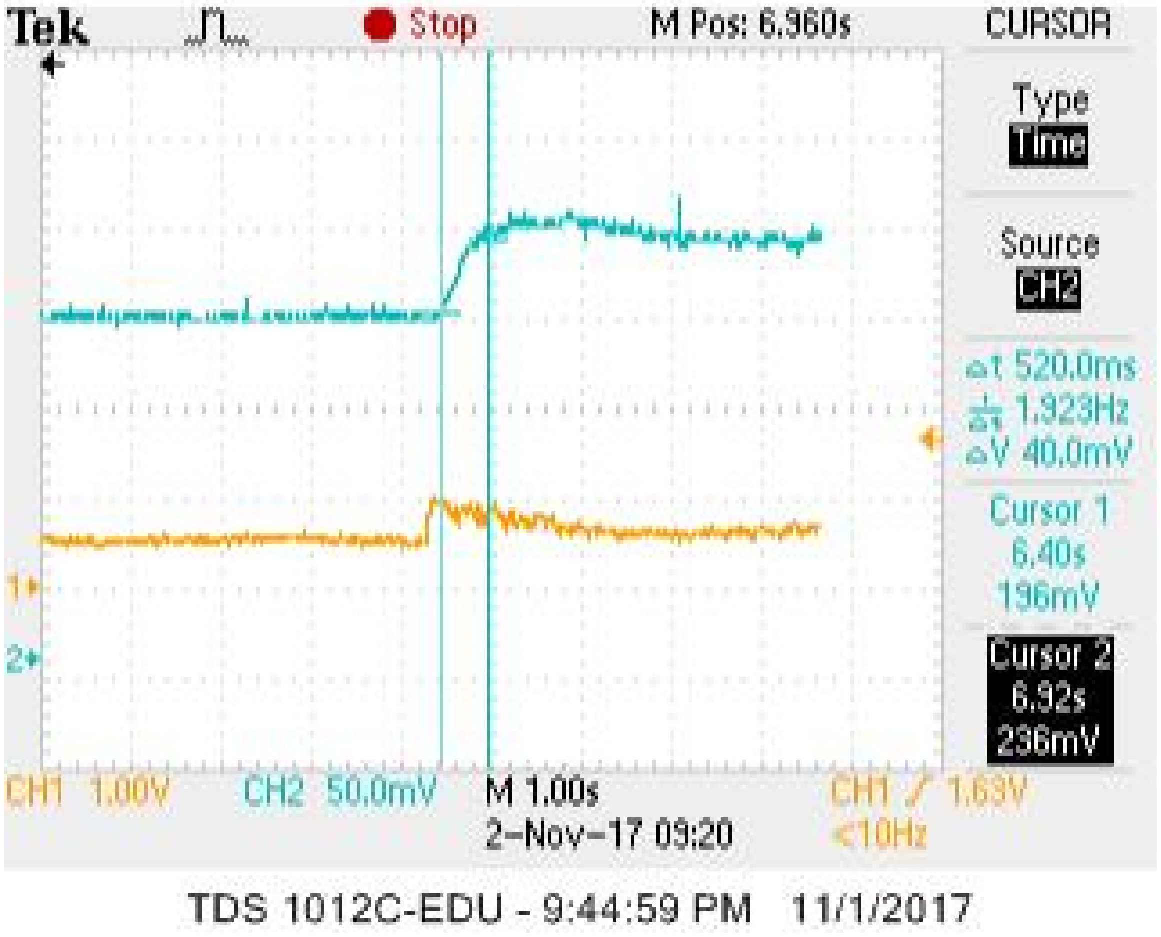


Figure 10: Rise Time for the Fourth Transition in the Startup Sequence. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

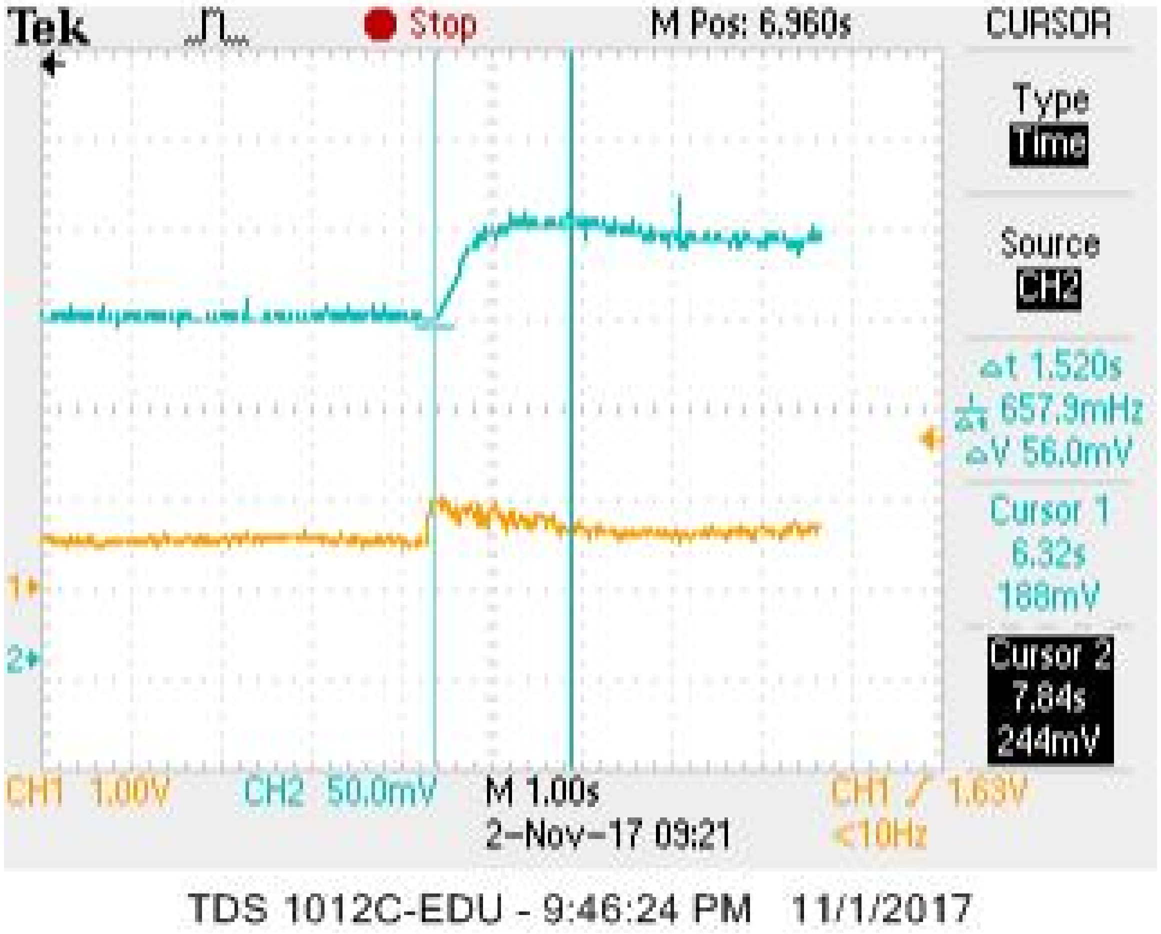


Figure 11: Settling Time for the Fourth Transition in the Startup Sequence. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

To evaluate the controllers instability, we applied various large steps and analyzed the response of the controller. For a step input from the horizontal to 1.01 rad (57.9 degrees), the rise time 360 milliseconds (figure 12) with a settling time of 3.04 seconds (figure 13). We believe that this settling time is rather high considering the somewhat stable behavior that we observed. Referring to figure 13, the beams angle was somewhat level for just over 2 seconds before the controller further corrected the angle. This may imply that there potentially could have been some random fluctuation in the potentiometer reading. However, when we applied a step of the same magnitude below the horizontal, we saw a rise time of 320 milliseconds (figure #) with a settling time of 600 milliseconds (figure #). This behavior alleviated these concerns and hinted at the possibility of this being an instable point.

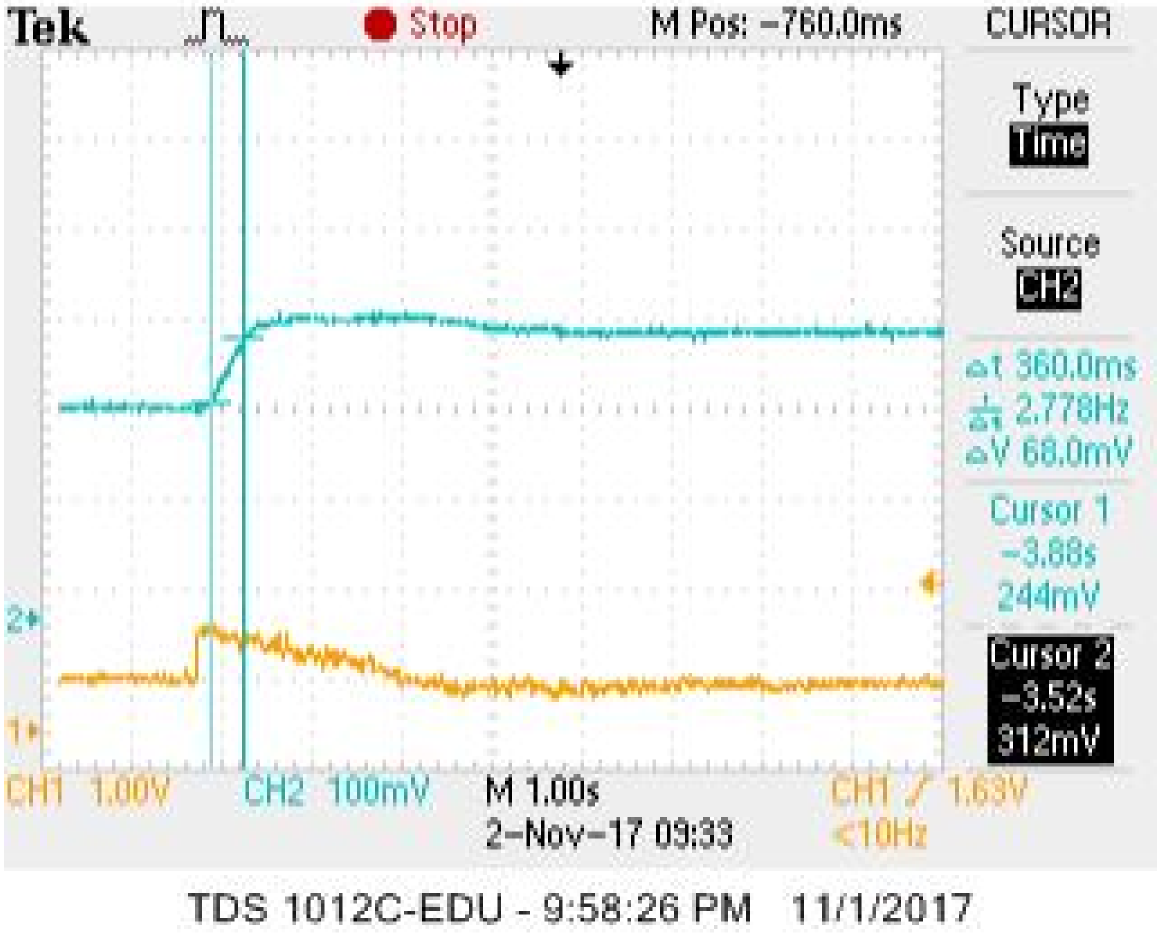


Figure 12: Rise Time for a Step Input From the Horizontal to 1.01 rad. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

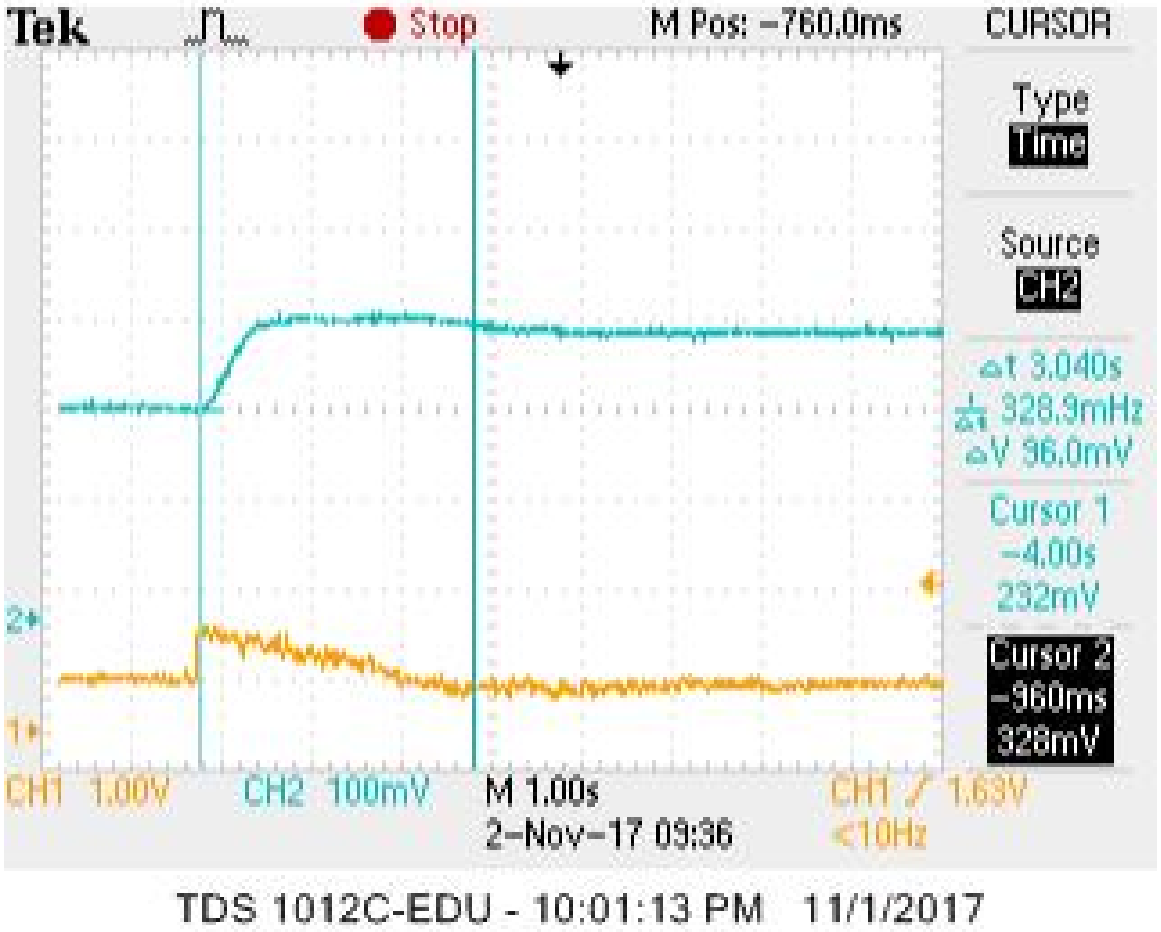


Figure 13: Settling Time for a Step Input From the Horizontal to 1.01 rad. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

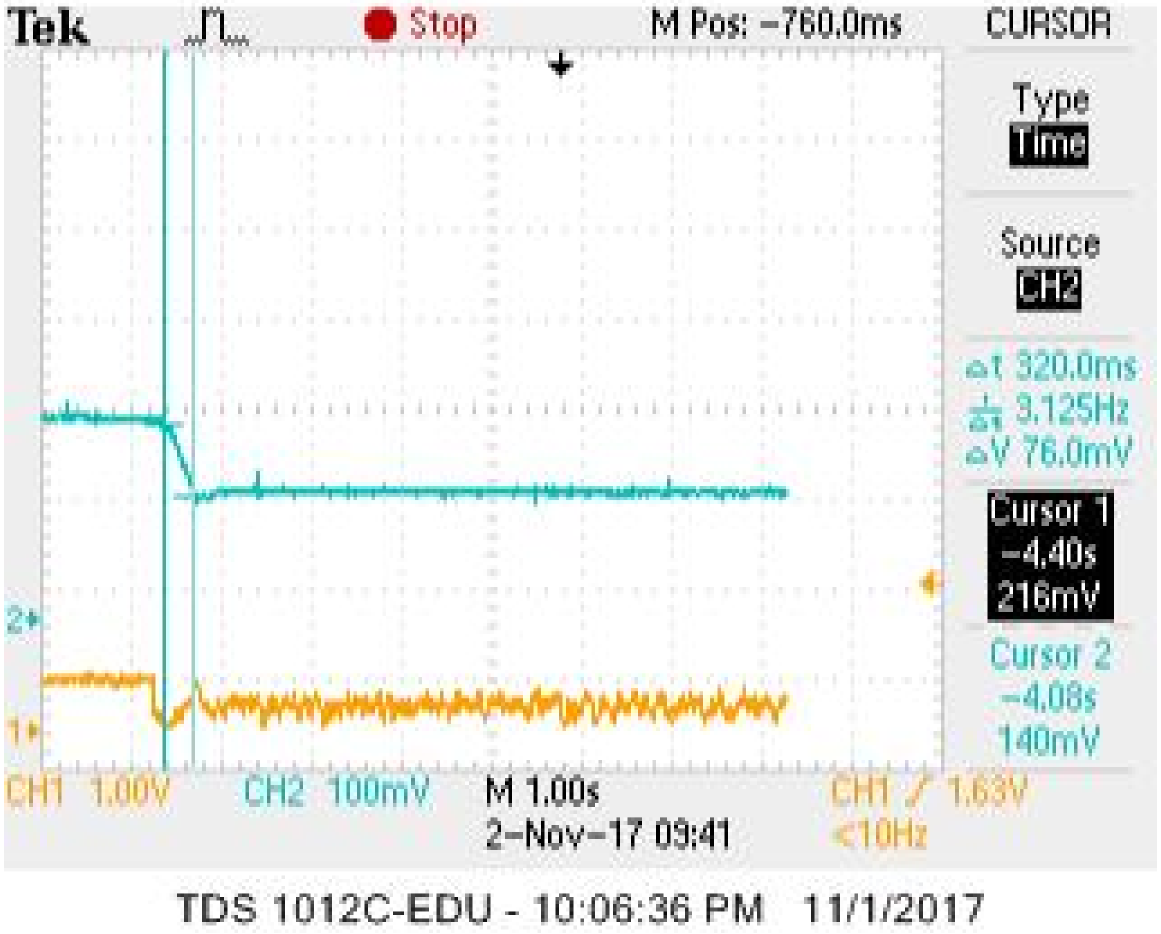


Figure 14: Rise Time for a Step Input From the Horizontal to -1.01 rad. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

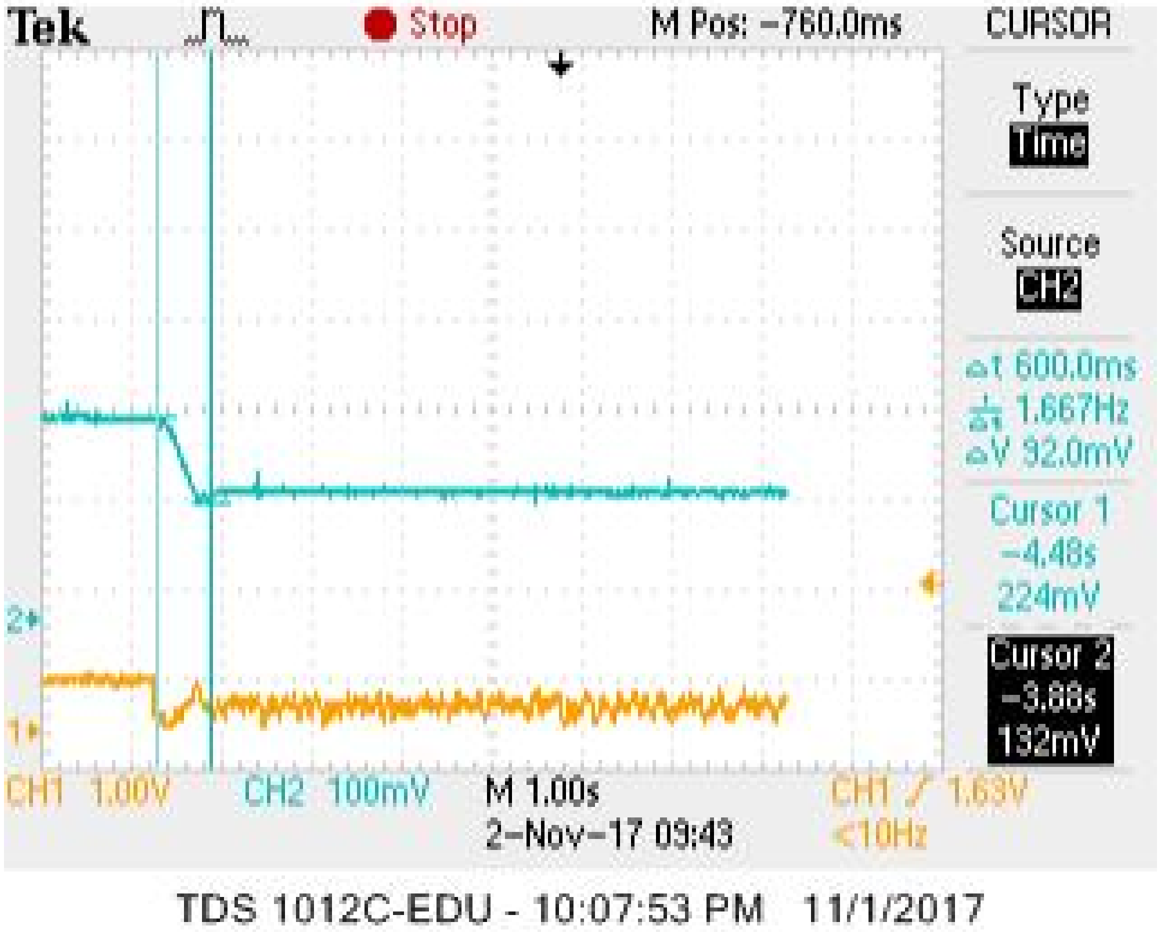


Figure 15: Settling Time for a Step Input From the Horizontal to -1.01 rad. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

To further test the controllers capabilities, we applied a step input from -1.57 rad (-90 degrees) to 1.06 rad (60.7 degrees). The resulting rise time was 480 milliseconds with a settling time just over 4 seconds. Taking a closer look at figure 16, we see that the controller tried to send a voltage input greater than its maximum output. This is apparent by observing the flat portion of the angle output. This behavior was a result of the saturation limit we placed in the controller that forced the maximum voltage output to be that of the motors. Applying a step of similar magnitude from the positive 1.57 rad (90 degrees) mark to below the horizontal resulted in a rise time of 480 milliseconds (figure 17) and a settling time of 1.48 seconds (figure 18). The key difference in the response of these two steps is that the step from positive 90 degrees to -64.2 degrees resulted in a slight oscillation before the angle reached the desired value. All of these results lead us to conclude that our controller was stable for all angle inputs. If we had not added the saturation, and had the stopper at the 90 degree mark, the controller may have driven the beam past the 90 degree mark causing it to loop around.

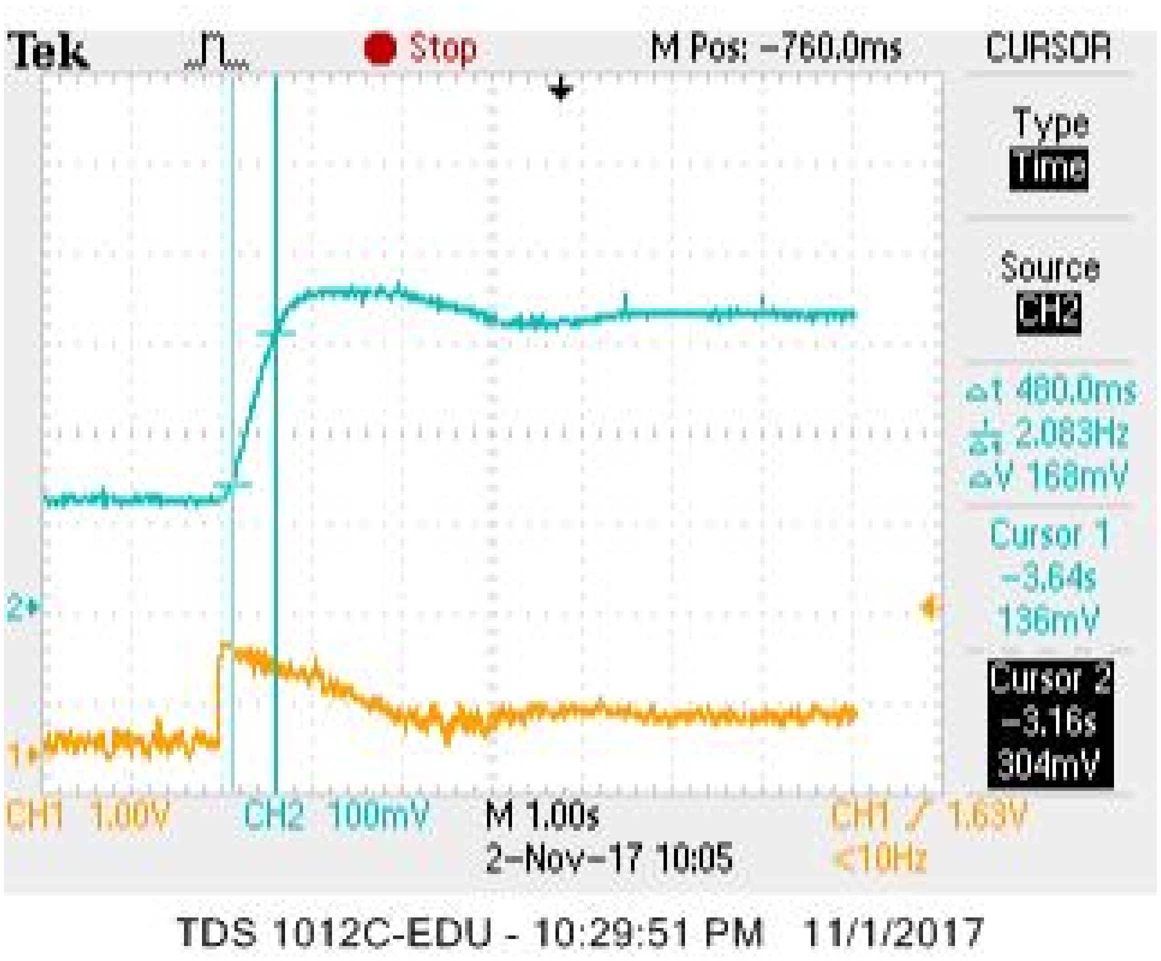


Figure 16: Rise Time for a Step Input From the -1.57rad to 1.06 rad. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

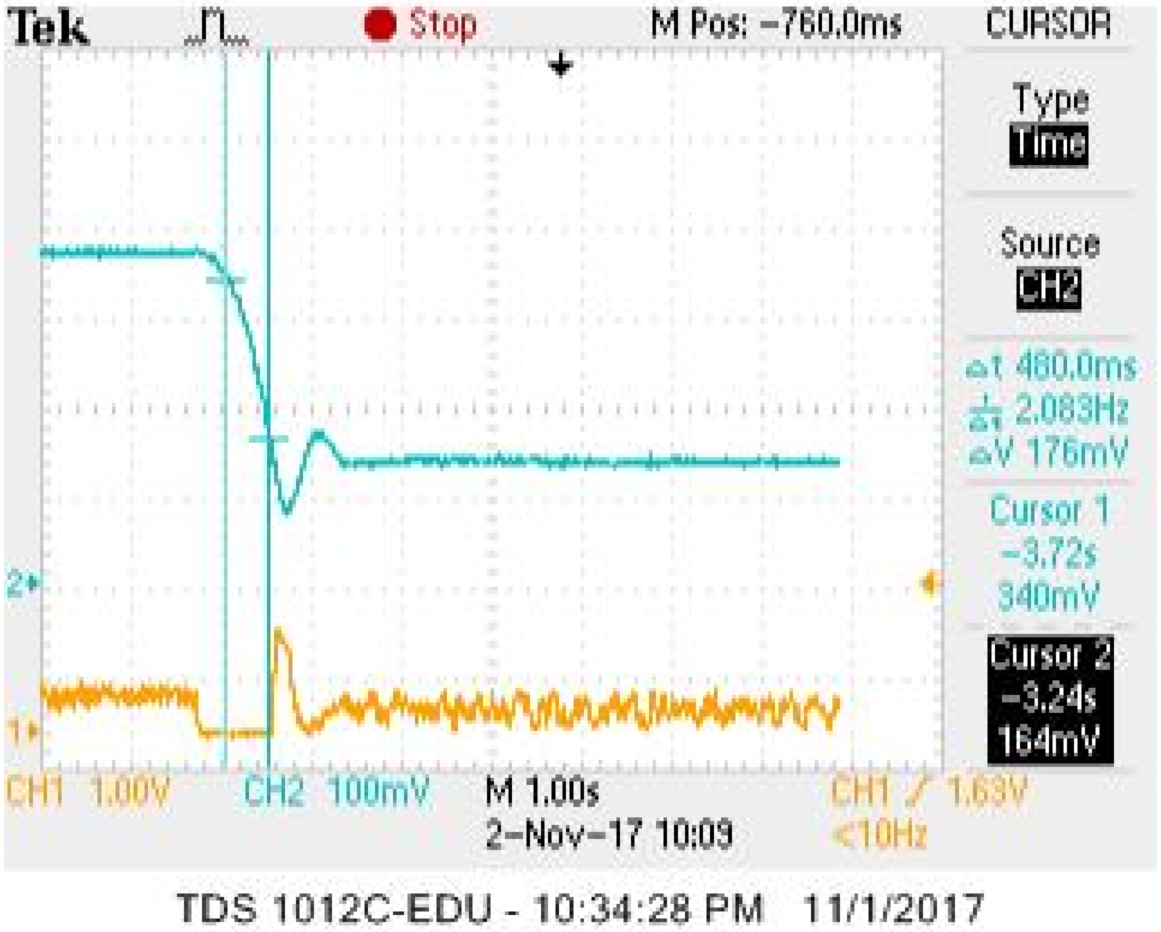


Figure 17: Rise Time for a Step Input From 1.57 rad to -1.12 rad. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

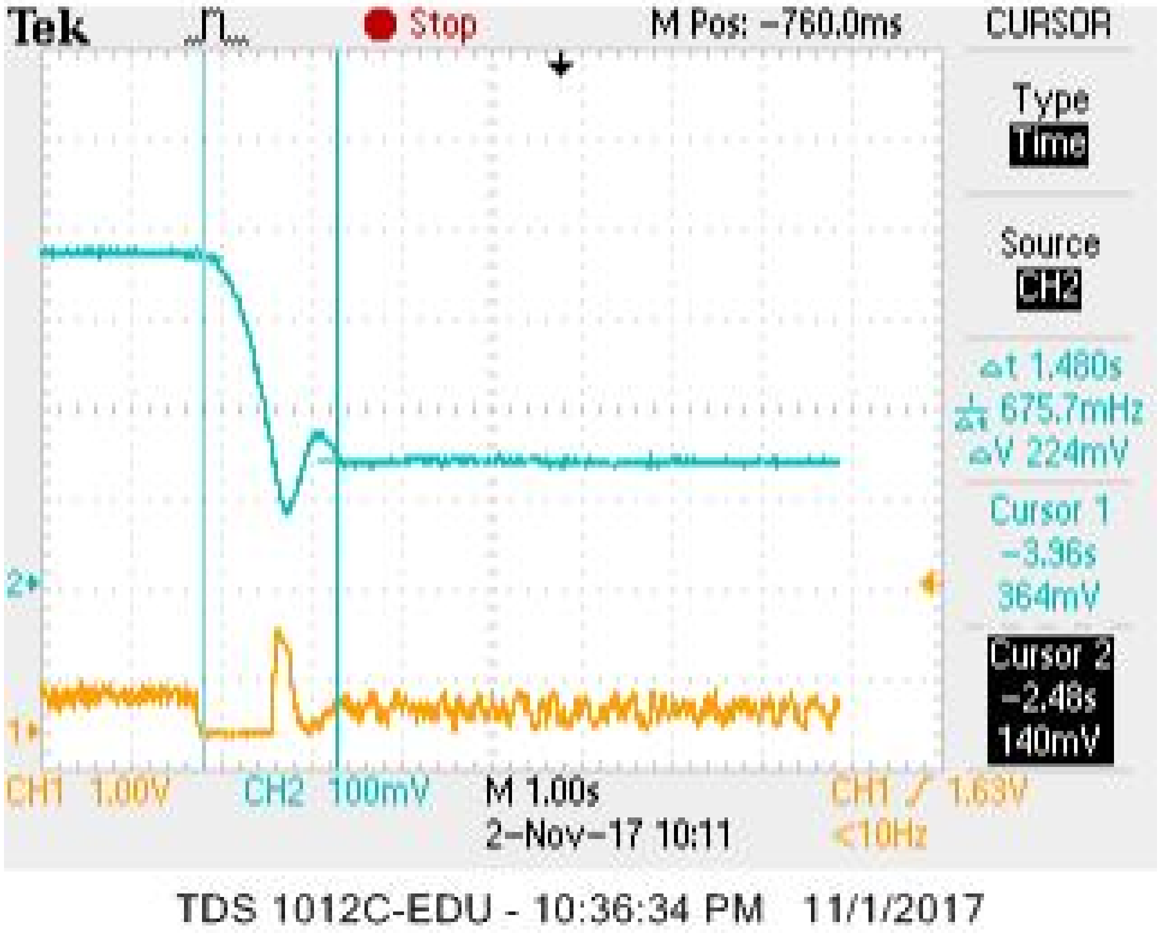


Figure 18: Settling Time for a Step Input From 1.57 rad to -1.12 rad. Blue Corresponds to the Angle and Yellow corresponds to the Reference Input.

The final test that we performed was to evaluate the time the controller took to complete the calculations as well as the DAC transfers. During peak load (a large step input), the ISR would run for 372 counts out of the 400 that we delegated for the timer. This means that are ISR, during peak operation, would take up 93% of the CPUs time. To lower the time spent in the ISR, we could have chosen to use the DMA to handle the transfer of the angle measurement and reference input. Fortunately, during steady state operation, the ISR only used 75% of the CPUs time.

5 Conclusion

In this lab, we successfully built a 1-DOF Helicopter which took user inputs for different parameters.

We exercised our skills with building hardware based systems with intricate circuitry. We explored PID control and pulse width modulation, and built a system that was stable across a large range of angles. We utilized principles from previous labs such digital-to-analog conversion, analog-to-digital conversion, SPI communication, interrupt driven procedures, and button debouncing. We successfully built a critically damped system with a -90 to 0 degrees step input.

The system was fine tuned for its hovering abilities. Noise reduction from the system was an important aspect which was overcome by reducing the Motor noise by using a diode, software and hardware filtering.

There are a couple ways to make this lab more interesting. For instance, we could allow the helicopter to go above the 90 degree marker and upon that switch engage another motor. This would have to utilize robust logical reasoning and carrying over of the motor displacement. You could somehow make it a 2 DOF helicopter where you could allow the beam to rotate towards you and so you would have two errors and two inputs/outputs, respectively. You apply these same PID principles to a variety of systems such as a 1 DOF car that moves in one direction along a track.

There were many problems which were faced during setting up the ADC channel which was resolved by setting up the correct parameters in the code. We accidentally set up the ADC pin for the trim pot to the same pin as the SPI clock pin. This took a massive amount of troubleshooting and taught to triple check which pins are in use and feasible for your application. This results in a small range for the ADC value of the trim pot that we originally tuned for and set us back a few hours. However, we learned probably the most valuable lesson of the semester from this lab: with microcontrollers, anything can go wrong.

Work Distribution

Sebastian Minor hardware development, PID development, PID tuning, general troubleshooting, start up sequence, documentation, button debouncing

Jonah Hardware development, circuity, PID development, PID tuning, GUI, button debouncing, general troubleshooting, startup sequence, ADC and DAC, results analysis

Rahul Hardware development, circuity, ADC and DAC, general troubleshooting

6 References

Figures 2 and 3 were taken from Bruces' website.

7 Appendix

7.1 C Code

```
/*
 * File: lab4.c
 * Author: Sebastian Roubert, Jonah Okike-Hephzibah, Rahul Desai
 * Adapted from:
 *  tahmidsDACAB
 * Author:  Syed Tahmid Mahbub
 * Target PIC: PIC32MX250F128B
 */

////////////////////////////////////
// clock AND protoThreads configure!
// You MUST check this file!
#include "config.h"
// threading library
#include "pt_cornell_1_2_1.h"
#include "tft_master.h"
#include "tft_gfx.h"
#include <stdio.h>
#include <math.h>

/*******************************************************************
The following code was developed by us to fulfill the requirements
set forth by Lab 4 of ECE 4760: Digital Systems Design using
Microcontrollers.

The goal was to operate a 1 degree-of-freedom helicopter with angle
control implement via a PID controller.

There is a startup sequence where the helicopter moves from -180 degress to
0 to 30 to -30 then back 0 over a period of 20 seconds.

After 20 seconds, the user can controller the desired angle, proportional
gain, derivative gain, and integral gain.

This is achieved via using button to cycle between input choices and
button 2 to start editing an input choice and clicking button 2 again
```

```
to confirm the choice. You edit the value of your input choice by rotating
a trim pot.

We hope you enjoy this lab more than we enjoyed making it.

*******************************************************************/


#define DAC_config_chan_A 0b0011000000000000
#define DAC_config_chan_B 0b1011000000000000

//pull down resistors for the DAC
// PORT B
#define EnablePullDownB(bits) CNPUBCLR=bits; CNPDBSET=bits;
#define DisablePullDownB(bits) CNPDBCLR=bits;
#define EnablePullUpB(bits) CNPDBCLR=bits; CNPUBSET=bits;
#define DisablePullUpB(bits) CNPUBCLR=bits;
//PORT A
#define EnablePullDownA(bits) CNPUACLR=bits; CNPDASET=bits;
#define DisablePullDownA(bits) CNPDACLR=bits;
#define EnablePullUpA(bits) CNPDACLR=bits; CNPUASET=bits;
#define DisablePullUpA(bits) CNPUACLR=bits;

volatile SpiChannel spiChn = SPI_CHANNEL2 ; // the SPI channel to use
// for 60 MHz PB clock use divide-by-3
volatile int spiClkDiv = 2 ; // 20 MHz DAC clock
// === thread structures ============================================
// thread control structs
static struct pt pt_cmd, pt_time, pt_input, pt_output, pt_DMA_output ;
static struct pt pt_tftDisplay, pt_buttonCheck;

//the gains that produced a critically damped response from vertically
//down, -180 degrees, to horizontal, 0 degrees.
#define stableKP 250
#define stableKI 0.75
#define stableKD 41000

// system 1 second interval tick
int sys_time_seconds ;


//Period to be used for timer interrupt
int generate_period = 40000 ;

//variable used for duty cycle. pwm_on_time/generate_period is duty cycle of motor
int pwm_on_time = 0 ;
int currentAngle = 0 ;  //helicopter angle
int trimPot = 0 ; //potentiometer for trimpot input
int potenVal = 0 ; //potentiometer for motor angle

// string buffer
char buffer[60];


//== Timer 2 interrupt handler ===========================================
// actual scaled DAC
volatile int DAC_data;
// profiling of ISR
volatile int isr_time;

//defining values for PID control calculation
volatile int error, prevError, desiredVolt, currentVolt, integral=0;
volatile int prevVolt=0, derivative, output ;

//desired angle in ADC units
static int desiredAngle = 200 ;

//control gains
int Kp = 250, Kd = 41000 ;
float Ki=0.75;

//variable to display motor displacement on oscilloscope
volatile int motor_disp = 0;

//trimpot value converted to ADC. Initially at 465 for 0 degrees.
volatile int trimConv = 465;

//storing reference angle for startup sequence
static int refAngle[4] = {465,553, 377,465};

//initializing variables to be altered by user input. All start out at
//start up sequence values.
static float propVal[5] = {0,250,0.75,41000,465} ;

//used for time calculations to see if still in start up sequence
volatile int timeVal = 0, j = 0 ;
static int inputVal = 0 ;

//function to figure out sign of number
int sign(int x){
    return (x>0) - (x<0);
}

// == Timer 2 ISR =====================================================
// does entire control algorithm calculation
void __ISR(_TIMER_2_VECTOR, ipl2) Timer2Handler(void) {
    //trimPot = ReadADC10(1) ;
    mT2ClearIntFlag();

    //read motor potentiometer
    potenVal = ReadADC10(0) ; //ranges from 200 to 730

    //enter if cascade is to check if still in start up sequence and
    //if so change desired angle. If out of startup sequence, the
    //if-cascade defaults to final else statement, where the desired angle
    //is set by user input
    if (sys_time_seconds-timeVal > 5) {
       j += 1 ;
       //mPORTAToggleBits(BIT_0);
        if ( j < 4 ) {
            timeVal = sys_time_seconds ;
            trimConv = refAngle[j] ;
            // angle
        } else {
            trimConv = desiredAngle ;
        }

    }


    //calc error
    error = trimConv - potenVal;

    //parameters for control
    integral += error;
    prevError = trimConv - prevVolt;
    derivative = error - prevError;

    //neglecting integral term if below certain angle
    if (potenVal < 288) integral =0 ;

    //if the error crossed 0, change integral so that
    //controller stays active
    if (sign(error) != sign(prevError)){
        integral=(integral*999)/(1000);
    }
    //alter duty cycle based on PID algorithm
    pwm_on_time= Kp*error + Kd*derivative + Ki*integral;

    //setting duty cycle to 0 if less than 0
    if (pwm_on_time < 0) pwm_on_time = 0;

    //setting duty cycle to 100 if calculated to be over 100
    if (pwm_on_time > generate_period) pwm_on_time = generate_period;

    //changing PWM
    SetDCOC3PWM(pwm_on_time);

    prevVolt= potenVal;

    // === Channel A =============
    // CS low to start transaction

     mPORTBClearBits(BIT_4); // start transaction
     //DAC A transfaction, the motor angle
     WriteSPI2( DAC_config_chan_A | potenVal);
    while (SPI2STATbits.SPIBUSY); // wait for end of transaction
     // CS high
    mPORTBSetBits(BIT_4); // end transaction

    motor_disp = motor_disp + ((pwm_on_time - motor_disp)>>4);

    mPORTBClearBits(BIT_4); // start transaction

    //DAC B transfaction, total motor displacement
    WriteSPI2( DAC_config_chan_B | motor_disp>>4 );
    while (SPI2STATbits.SPIBUSY);

    mPORTBSetBits(BIT_4); // end transaction

    isr_time = ReadTimer2() ; // - isr_time;





}

// === Button Thread ======================================================
//following thread deals with button pushes by the user, very simple debounce
```

```
//over the 250 msec yield time

static int firstButton=0, secondButton=0, firstPush=0, secondPush=0;

static int enter = 0;

static PT_THREAD (protothread_buttonCheck(struct pt *pt))
{

   PT_BEGIN(pt);
   while(1) {

      PT_YIELD_TIME_msec(250) ;

       //adding how many times the first button has been pressed
       firstButton += (int) (mPORTAReadBits ( BIT_3)==0);

       //if first button has been read as pressed twice, declare as push
       if ( (mPORTAReadBits ( BIT_3)==0) && (firstButton > 2)) {
          firstPush = 1;

          //checking which value user is currently inputting, rangles from 0-3
          if (inputVal > 3) {inputVal = 0 ;}
          else {inputVal += 1 ;}
          if (inputVal == 4) inputVal = 0;

       //if first button is not pressed but a push was previously record then
       //reset values
       }else if ( (mPORTAReadBits ( BIT_3)==8) && (firstButton>2)) {
          firstButton = 0; firstPush = 0;
       }

       //adding how many times the first button has been pressed
       secondButton += (int) (mPORTAReadBits ( BIT_2)==0);

       //if first button has been read as pressed twice, declare as push
       if ( (mPORTAReadBits ( BIT_2)==0) && (secondButton > 2)) {

          secondPush = 1;

       //if second button is not pressed but a push was previously record then
       //reset values
       }else if ( (mPORTAReadBits ( BIT_2)==4) && (secondButton>2)) {
          secondButton = 0; secondPush = 0;
       }
```

```
    //if second button has been pushed and currently being pushed then in enter
    //state and user is inputting value
    if ( (mPORTAReadBits ( BIT_2)==0) && (secondButton > 2) && (secondPush == 1)){
        if (enter == 0) {enter = 1;} else {enter = 0;}
    }

    // inputVal = 0: angle
    // inputVal = 1: Kp for PID
    // inputVal = 2: Ki
    // inputVal = 3: Kd

    //if user is entering values then read values of current selection without actually
    //altering values
    if (enter == 0 ) {

        trimPot = ReadADC10(1);

        if (inputVal == 0) {
            // Do conversion for angle
            propVal[0] = 0.00307*(float)trimPot - 1.57;
            propVal[4] = (0.5181 * (float) trimPot) + 200;

        } else if (inputVal == 1) {
            //convert for Kp
            propVal[1] = 0.48875*(float)trimPot;
        } else if (inputVal == 2) {
            //convert for Ki
            propVal[2] = 0.00098*(float)trimPot;
        } else if (inputVal == 3 ) {
            //convert for Kd
            propVal[3] = 48.876*(float)trimPot;
        }

    //if user has entered values then update the chosen value
    } else if (enter == 1) {



        if (inputVal == 0) {
            if (enter == 1){
                mPORTAToggleBits(BIT_0);
            desiredAngle = (int) propVal[4] ;} else desiredAngle = desiredAngle;
        } else if (inputVal == 1) {
            Kp = (int)propVal[1] ;
        } else if (inputVal == 2) {
            Ki = propVal[2] ;
            } else if (inputVal == 3) {
                Kd = (int)propVal[3] ;
            }
        }

    }
    PT_END(pt) ;

}

// === Tft Thread =======================================================
// The following thread is to update the TFT for the current values of the
// PID parameters, helicopter angle, and system time.

static PT_THREAD (protothread_tftDisplay(struct pt *pt))
{

    PT_BEGIN(pt);
    while(1) {

        PT_YIELD_TIME_msec(1000) ;



        currentAngle = (int) (0.3214 * (float) potenVal - 144.6380) ; // angle



        tft_setCursor(25, 25);
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        tft_fillRoundRect(25,25, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
        //sprintf(buffer,"Current Angle: %d", currentAngle);
        sprintf(buffer,"Current Selection: %d",inputVal+1);
        tft_writeString(buffer);



        tft_setCursor(25, 50);
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        tft_fillRoundRect(25,50, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
        sprintf(buffer,"1. Desired Angle: %.2f",propVal[0]);
        tft_writeString(buffer);

        tft_setCursor(160, 50);
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        tft_fillRoundRect(160,50, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
        sprintf(buffer,"2. Kp: %d", (int)propVal[1]);
        //sprintf(buffer,"Pot Value: %d", potenVal);
        tft_writeString(buffer);

        tft_setCursor(25, 100);
        tft_fillRoundRect(25,100, 120, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        sprintf(buffer,"3. Ki: %.2f", propVal[2]);
        tft_writeString(buffer);

        tft_setCursor(25, 150);
        tft_fillRoundRect(25,150, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        sprintf(buffer,"4. Kd: %d", (int) propVal[3]);
        tft_writeString(buffer);

        if (enter == 1) {
        tft_setCursor(150, 150);
        tft_fillRoundRect(150,150, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        sprintf(buffer,"entered: %d", inputVal+1);
        tft_writeString(buffer);
        }
        if (enter == 0) {
        tft_setCursor(150, 150);
        tft_fillRoundRect(150,150, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        sprintf(buffer,"entering: %d", inputVal+1);
        tft_writeString(buffer);
        }

        tft_setCursor(25, 175);
        tft_fillRoundRect(25,175, 150, 20, 1, ILI9340_BLACK);// x,y,w,h,radius,color
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        //sprintf(buffer,"Desired Angle: %d", desiredAngle);
        sprintf(buffer,"Sec. Button: %d", secondPush);
        tft_writeString(buffer);

    }
    PT_END(pt) ;

}
```

```
// === One second Thread ======================================================
// update a 1 second tick counter
static PT_THREAD (protothread_time(struct pt *pt))
{
    PT_BEGIN(pt);

      while(1) {
            // yield time 1 second
            PT_YIELD_TIME_msec(1000) ;
            sys_time_seconds++ ;
            // NEVER exit while
      } // END WHILE(1)

  PT_END(pt);
} // thread 4

// === Main ======================================================

int main(void)
{
     mPORTASetPinsDigitalIn ( BIT_3 | BIT_2) ; // setting buttons as input
    SYSTEMConfigPerformance(PBCLK);

    ANSELA = 0; ANSELB = 0; CM1CON = 0; CM2CON = 0;


  // === Config timer and output compares to make pulses ========
  // set up timer2 to generate period
  OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_1, generate_period);
  ConfigIntTimer2(T2_INT_ON | T2_INT_PRIOR_2);
  mT2ClearIntFlag(); // and clear the interrupt flag

  // set up compare3 for PWM mode
  OpenOC3(OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE , pwm_on_time, pwm_on_time); //
  // OC3 is PPS group 4, map to RPB9 (pin 18)
  PPSOutput(4, RPB9, OC3);


  // POTENTIOMETERS ARE WORKING:
  // ANGLE POTENTIOMETER CONNECTED TO RB13
  // TRIM POT TO RB14

  #define PARAM1 ADC_FORMAT_INTG16 | ADC_CLK_AUTO | ADC_AUTO_SAMPLING_ON
  #define PARAM2 ADC_VREF_AVDD_AVSS | ADC_OFFSET_CAL_DISABLE | ADC_SCAN_OFF | ADC_SAMPLES_PER_INT_2 |
      ADC_ALT_BUF_OFF | ADC_ALT_INPUT_ON
  #define PARAM3 ADC_CONV_CLK_PB | ADC_SAMPLE_TIME_15 | ADC_CONV_CLK_Tcy
```

```
#define PARAM4 ENABLE_AN1_ANA | ENABLE_AN5_ANA
#define PARAM5 SKIP_SCAN_ALL

SetChanADC10( ADC_CH0_NEG_SAMPLEA_NVREF | ADC_CH0_POS_SAMPLEA_AN1 | ADC_CH0_NEG_SAMPLEB_NVREF |
    ADC_CH0_POS_SAMPLEB_AN5 );
OpenADC10( PARAM1, PARAM2, PARAM3, PARAM4, PARAM5 ); // configure ADC using the parameters defined
    above

EnableADC10(); // Enable the ADC

// === config the uart, DMA, vref, timer5 ISR ===========

// === set up i/o port pin ===============================
mPORTBSetPinsDigitalOut(BIT_0 ); //Set port as output

 /// SPI setup ////////////////////////////////////////////
  // SCK2 is pin 26
  // SDO2 is in PPS output group 2, could be connected to RB5 which is pin 14
  PPSOutput(2, RPB5, SDO2);
  // control CS for DAC
  mPORTBSetPinsDigitalOut(BIT_4);
  mPORTBSetBits(BIT_4);
  // divide Fpb by 2, configure the I/O ports. Not using SS in this example
  // 16 bit transfer CKP=1 CKE=1
  // possibles SPI_OPEN_CKP_HIGH; SPI_OPEN_SMP_END; SPI_OPEN_CKE_REV
  // For any given peripherial, you will need to match these
  SpiChnOpen(spiChn, SPI_OPEN_ON | SPI_OPEN_MODE16 | SPI_OPEN_MSTEN | SPI_OPEN_CKE_REV , spiClkDiv);

  mPORTASetPinsDigitalOut(BIT_0);
  mPORTASetBits(BIT_0);

// === now the threads ===================================

  PT_setup();

  // === setup system wide interrupts ====================
INTEnableSystemMultiVectoredInt();

// init the threads
//PT_INIT(&pt_cmd);
PT_INIT(&pt_time);
PT_INIT(&pt_tftDisplay);
 PT_INIT(&pt_buttonCheck);
```

```
  tft_init_hw();
  tft_begin();
  // erase
  tft_fillScreen(ILI9340_BLACK);
  //240x320 vertical display
  tft_setRotation(3); // Use tft_setRotation(1) for 320x240
  // seed random color
  srand(1);


  // schedule the threads
  while(1) {
    PT_SCHEDULE(protothread_time(&pt_time));
    PT_SCHEDULE(protothread_tftDisplay(&pt_tftDisplay)) ;
    if (sys_time_seconds > 20) PT_SCHEDULE(protothread_buttonCheck(&pt_buttonCheck));
  }
} // main
```