ME 333: Introduction to Mechatronics

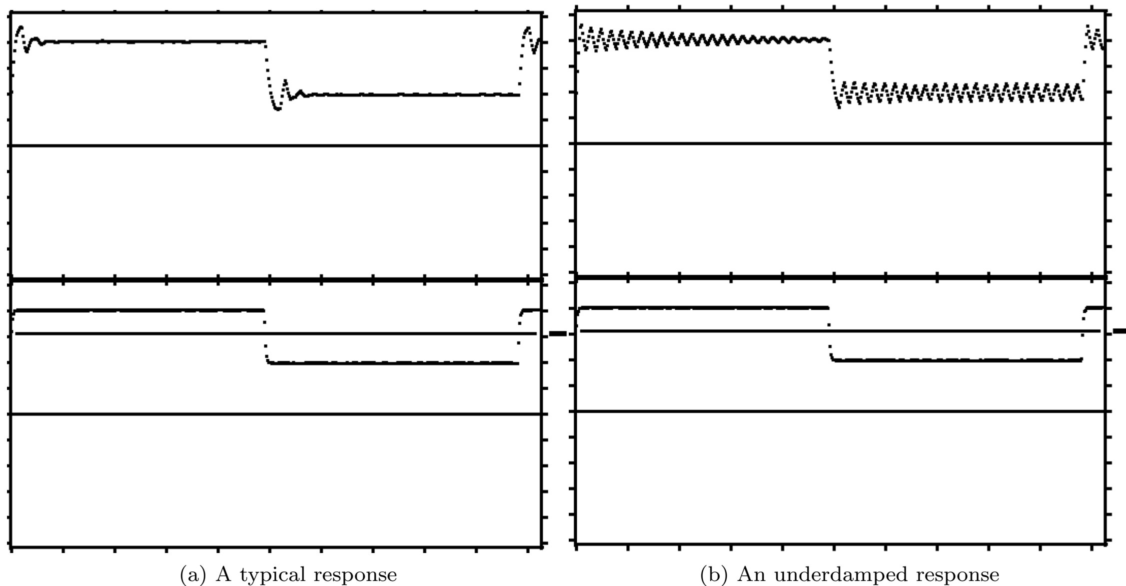
Assignment 5: Simple real-time control with the PIC32

Electronic submission due **before** 11:00 a.m. on February 21st

1 Introduction

In this assignment, you will be writing a real-time controller for an electronic system. Your code will involve multiple time-based interrupt service routines (ISRs), analog-to-digital conversion (ADC), pulse-width modulation (PWM) using output compare, and a library for interfacing with an LCD. The task is to build a closed-loop controller to set the voltage at the emitter of a phototransistor to a prescribed value regardless of external disturbances, such as varying ambient light conditions. A reference voltage signal will be generated that represents the desired voltage at the emitter using PWM and a low-pass filter (LPF). The phototransistor will then be controlled by modulating a PWM signal that is powering an LED. The ADC module will be used to sample both the reference signal and the actual signal. Using these samples, you will implement a proportional-integral (PI) controller that will automatically adjust what the PIC is doing to account for changes in the environment. So if, for example, you were to place a piece of tissue paper between the LED and the phototransistor, the PIC would automatically increase the power it is sending to the LED to adjust for this unexpected change. A range of useful sample code can be found at http://hades.mech.northwestern.edu/index.php/ME333_Sample_Code.

Any statements marked in bold and blue should be answered and turned in!



(a) A typical response

(b) An underdamped response

Figure 1: Above are two example plots showing the output signal on the top channel and the reference signal on the bottom channel. The plot in (b) shows an example of a poorly tuned controller.

2 Questions

For each question in this section, turn in a typed response.

1. You are setting up port B to receive analog input and digital input, and to write digital output. Here is how you would like to configure the port. (Pin x corresponds to RBx.)

   - Pin 0 is an analog input.
   - Pin 1 is a "typical" buffered digital output.
   - Pin 2 is an open-drain digital output.
   - Pin 3 is a "typical" digital input.
   - Pin 4 is a digital input with an internal pull-up resistor.
   - Pins 5-15 are analog inputs.
   - Pin 3 is monitored for change notification, and the change notification interrupt is enabled.

   Questions:

   (a) Which digital pin is most likely to have an external pull-up resistor? What would be a reasonable resistance to use?

   (b) To achieve the configuration described above, give the eight-digit hex values you should write to AD1PCFG, TRISB, ODCB, CNPUE, CNCON, and CNEN. (Some of these SFRs have unimplemented bits 16-31; you can just write 0 for those bits.)

2. Our PBCLK is running at 80 MHz. Give the four-digit hex values for T3CON and PR3 so that Timer3 is enabled, accepts PBCLK as input, has a 1:64 prescaler, and rolls over (generates an interrupt) every 16 ms. (Keep in mind that if you put x in the period register, the cycle duration is actually x+1, since counting starts at 0.)

3. Using a 32-bit timer (Timer23 or Timer45), what is the longest duration you can time, in seconds, before the timer rolls over? (Use the prescaler that maximizes this time.)

4. You will use Timer2 and OC1 to generate a PWM signal at approximately 20 kHz. You use a 1:8 prescaler on the PBCLK as input to Timer2. What value should you place in PR2 (in base 10)? If you want to create an approximately 25% duty cycle signal, what value should you place in OC1RS (in base 10)?

3 The Program

3.1 Before you begin

Before you attempt the programming assignment, you should create two MPLAB X projects. The first project will run the ME 333 sample code and the other project will have the code you end up submitting. We will refer to these projects as SampleCode and MyProject, respectively. Download

- ADC_Read2.c,
- ADC_Read2_LCD.c,
- TMR_16bit.c,
- LCD.c, LCD.h, LCDtest.c,
- and OC_square_wave.c

from the *ME333 Sample Code* wiki page. Place them in the SampleCode project folder. Instead of creating multiple projects, you will add and remove these files from the SampleCode project as instructed in the homework.

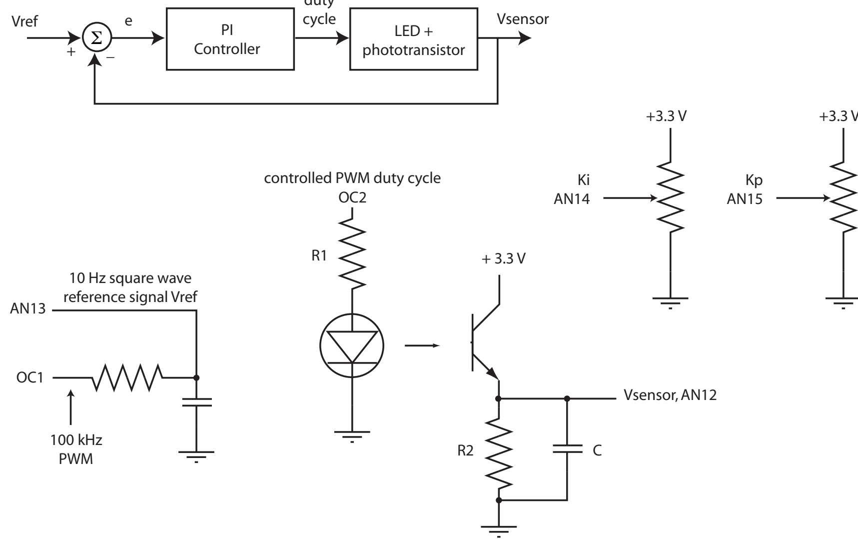


Figure 2: The circuit and control loop you will implement.

3.2 Analog Input

In this section, you will wire up two potentiometers that we will eventually use as inputs to the PIC32 to modify the controller gain parameters, K_p and K_i. You do not have to turn anything in for this section.

1. Wire terminals 1 and 3 of a potentiometer to 3.3 V and ground. Connect terminal 2, the output from the wiper, to pin AN15 on the PIC32. This will serve as our eventual input for K_p (see Figure 2).

2. Add ADC_Read2.c to the SampleCode project and run it on your PIC. Make sure that only ADC_Read2.c and no other sample file in Section 3.1 appears in the SampleCode project under the MPLAB X *Projects* window.[1] Verify that the code is working and outputting values over serial with the NU32_utility or a terminal program.

3. While the ADC code is a useful reference, it does seem like a waste to duplicate the same lines of code just to change which pin is connected to CH0SA. In this step we are going to structure MyProject in such a way that we increase code modularity and the possibility of code reuse in future assignments. In MyProject, add a new C source file, called MyControllerMain.c. This C file will contain your main program and your interrupt service routines (ISRs). Create two more files, one C source file called MyLibrary.c and a header file called MyLibrary.h. Add both of these to MyProject.

4. For now, there will be three functions in the project: a main function, an ADC initialization function (`ADCInitManual`), and an ADC read function (`ADCManualRead`). The main function will be in MyControllerMain.c, and the other two will go in MyLibrary.c. Take the ADC initialization code from ADC_Read2.c, three lines in this case, and place it in the function `void ADCInitManual(void)` in MyLibrary.c. Now write the function `int ADCManualRead(int pin)`, which takes in a desired analog pin number between 0 and 15 and returns the ADC value on that pin, which will be between 0 and 1023. You should define the function prototypes for the two ADC functions in MyLibrary.h, and include MyLibrary.h in MyLibrary.c and MyControllerMain.c.

[1] NU32.c and NU32.h should always be included in the SampleCode project, and procdefs.ld should be in both the SampleCode and the MyProject directory.

When creating header files, it is common practice to write an "inclusion guard".[2] To help you with the syntax, the following is a template for creating MyLibrary.h:

```
#ifndef __MYLIBRARY_H
#define __MYLIBRARY_H

void ADCInitManual(void);
#endif // __MYLIBRARY_H
```

Don't forget to add the function prototype for `ADCManualRead`.

5. You can test your functions by calling `ADCManualRead(15)` in your main function and sending the ADC value on AN15 to your PC so you can view the value in a terminal. You should be able to get voltages from 0 to 3.3 volts.

6. Wire a second potentiometer and connect its wiper to AN14. Send the value of both pots to your PC over serial. For now, just make sure you are reading both pots correctly by sending the readings to your computer over UART1.

3.3 The LCD

Now that you have the potentiometers wired up and you are successfully reading the analog voltages, we are going to get the LCD, included in your kit, set up to display the current values of the potentiometers. You do not have to turn anything in for this section.

1. The first step is to wire up the LCD and test that you have it functioning correctly. Follow the wiring directions at `http://hades.mech.northwestern.edu/index.php/NU32:_16x2_LCD` to get the LCD wired up.

2. Did you wire everything correctly? In your SampleCode project, remove ADC_Read2.c and add LCD.c, LCD.h, and LCDtest.c to the project. LCDtest.c contains the `main` function, and LCD.c and LCD.h contain useful utility functions for interfacing with the LCD. Build this project and load the code onto your PIC32. If you wired everything successfully you should see output similar to that in NU32: 16x2 LCD page on the wiki.

3. In MyProject, print the values of the ADC to the LCD instead of sending the values over serial to your PC. Be careful to send no more than 16 characters per line! Don't forget to copy LCD.h and LCD.c into the MyProject folder, and add them to MyProject.

3.4 Timer ISRs

You will now write two timer interrupt service routines in MyControllerMain.c. You should name the first interrupt routine `ControlLoopISR`, which will handle your 1 kHz control loop logic. The second ISR should be named `OutputISR`; it will be responsible for printing to the LCD. The `ControlLoopISR` should have higher priority than the `OutputISR`.

1. TMR_16bit.c is a useful piece of code to understand for this section. In order to run the code, remove the LCD files from the SampleCode project and replace them with TMR_16bit.c. Compile the program and run it on the PIC. Notice how the ISR blinks an LED to communicate to the world that it's running. Toggling a pin or blinking an LED is a useful way to know that an ISR is being called.

[2]The role of the inclusion guard is to prevent functions and variables from getting defined more than once. For example, say you have three files, s1.c, h1.h and h2.h. If s1.c includes h1.h and h2.h, and h1.h includes h2.h, then in effect, h2.h has been included in s1.c twice. The inclusion guard prevents this from causing compilation errors.

2. In MyLibrary.c, write a `TMRInitFixedFreq()` function that initializes TMR3 to run at 1 kHz and TMR4 to run at 20 Hz. Don't forget to add a prototype to MyLibrary.h. **What value should PR3 be if you know that `T3CON<TCKPS>` = 1 and that TMR3 is suppose to run at 1 kHz? Similarly, what should PR4 be if you know that `T4CON<TCKPS>` = 7 and that it runs at 20 Hz?** Don't forget that we want to write timer ISRs, so remember to configure both timers to interrupt when a rollover event is detected.

3. Now write your two ISRs, `ControlLoopISR` and `OutputISR`, in MyControllerMain.c. Test that they work by toggling NU32LED1 in `OutputISR` and NU32LED2 in `ControlLoopISR`. Verify that you set the frequency of each ISR correctly by connecting your NUScope to pins A4 and A5, which correspond to the LEDs. If your ISR is running correctly the square wave generated by the pins toggling should be half the frequency of your ISR.

4. Define K_p and K_i as two global `int`s in MyControllerMain.c. In `ControlLoopISR` read AN15 using `ADCManualRead` and assign its value to K_p. Then read in AN14 and assign its value to K_i. In `OutputISR`, print the values of K_p and K_i to the LCD. From now on, you should only print K_p and K_i to the LCD in the `OutputISR` function. Remove any previous instances that print the values of K_p and K_i to the LCD or serial UART in MyMainController.c. Test your ISRs again by verifying that you can print the values of K_p and K_i to the LCD in ADC counts.

5. Now declare two local `int` variables inside of `ControlLoopISR`, call them `actual_signal` and `ref_signal`. Read in AN12 and AN13 using `ADCManualRead` and assign their values to `actual_signal` and `ref_signal`, respectively. You might have realized that there is nothing connected to AN12 and AN13. For now, you can leave AN12 and AN13 floating. We'll give them real inputs soon.

3.5 Generating a Reference Signal with a Low-pass Filter

In this section, we will write the code that generates the reference signal, V_{ref}, using PWM. Specifically, we will generate a 100 kHz PWM signal out of OC1 and change its duty cycle every 50 ms. We'll then build a low-pass filter on the OC1 pin to generate V_{ref} at 10 Hz between 1 V and 2 V. Our strategy will be to modify OC_square_wave.c, so that it meets the specs and then copy the changes into MyProject.

1. Remove TMR_16bit.c from the SampleProject and add OC_square_wave.c. This program fills an array with duty cycles that produce a 1 Hz square wave with voltages between 1 and 2 V. It uses the array to "playback" the square wave. You should be able to see that this code is general enough to modulate the PWM signal to create arbitrary signals. Construct a low-pass filter on the OC1 PWM output pin D0. We found a resistance value of 10 kΩ and $f_{\text{cutoff}} \approx 723$ Hz satisfactory. **Using this information, what value was our capacitor?** You should recall from class that $f_{\text{cutoff}} = \frac{1}{2\pi RC}$. If you don't have these R and C values in your kit, then choose a resistor that will safely draw less than 18 mA, which is the maximum amount of current an individual pin on the PIC can supply. After choosing a resistor, use a capacitor that will produce a cutoff frequency of around 1 kHz.

2. Set `WAVE_PERIOD` to 0.1 and `WAVE_SAMPS` to 2. This will increase the frequency of V_{ref} to 10 Hz. Run the program with these modifications and verify with your NUScope that V_{ref} is now a 10 Hz square wave.

3. We want to eventually copy pieces of OC_square_wave.c into MyLibrary.c, but instead of copying sections of OC_square_wave.c into our more complicated project, MyProject, let's keep making the modifications in OC_square_wave.c. If something isn't working, we have less code to find mistakes in. As is, the function `startPWM()` relies on two macros to set the PWM frequency of OC1. This doesn't make `startPWM()` as modular as it can be; if we copied it into our library, we would have to also copy the definitions of `TMR2_PS_VAL` and `TMR2_PR2`. Instead, modify `startPWM` so that the prescaler and period match values are inputs to the function. This involves changing the function header so it's `void startPWM(int prescaler, int period)`. In OC_square_wave.c, OC1RS and OC1R are set to

the first value in the global `Signal` array. In our modified function, set them both to zero. This means the square reference wave won't begin until our `OutputISR` begins modulating the output duty cycle, but this causes no problems.

Test the program, making any necessary changes in the file to get it to compile, and make sure that you can still generate the 10 Hz signal. You can still use the macros when calling the function. For example, in main, you could replace the old call to `startPWM` with `startPWM(TMR2_PS_VAL, TMR2_PR2);`.

4. The array `Signal` stores the signal we want to playback as a series of counts corresponding to the PWM duty cycle. Our `buildSignal` function doesn't need to know how voltages map to a particular duty cycle, just the final value in counts. Modify `buildSignal` so that its function header is `void buildSignal(unsigned *signal, unsigned samples, int duty1, int duty2)`. While `buildSignal` does not have to worry about how voltages are converted to counts, you do, so make sure you call `buildSignal` with the correct duty cycles that will create the desired voltage levels. After making your changes, verify that you can still generate the same 10 Hz signal.

5. Copy your modified versions of `buildSignal` and `startPWM` from OC_square_wave.c into MyLibrary.c. In order to generate the PWM signal from within MyProject, modify MyControllerMain.c so that your main function calls `buildSignal` and `startPWM` and your `OutputISR` function correctly steps through the signal. Use OC_square_wave.c as a guide to help you figure out what variables and macros you should declare to get PWM working from within MyControllerMain.c. Don't forget to add prototypes for `buildSignal` and `startPWM` to MyLibrary.h. When you have everything working, connect V_{ref} to AN13.

3.6 The Phototransistor Circuit

In this section, you are going to build and test the circuit for the LED and phototransistor pair. The LED is going to point at the base of the phototransistor, and the brightness of the LED will modulate how much current flows through the transistor. The LED that we are using is the large clear one that shines red light, the red "super bright" LED. The phototransistor that we are using is the SFH310 phototransistor. You can find pictures of these parts and their datasheets on the *What is in the NU32 Kit* wiki page. In your circuit, use R1 = 100 Ω, C = 0.1 μF, R2 = 10 kΩ (see Figure 2). There is nothing to hand in for this section.

1. Wire the circuit shown in Fig. 2 for the LED and phototransistor. For the LED, the *cathode is the shorter lead*[3]. For the phototransistor, the *shorter lead is the collector*. From a practical wiring perspective (Figure 3), the LED and the phototransistor should be placed about 1.5 to 2 inches apart (we counted 16 holes separating the two in our breadboard). They should both be bent to point "towards" each other.

2. In MyLibrary.c, expand `startPWM` so that it also initializes PWM on OC2. Initialize OC2 so that it starts with a duty cycle of 50%. Because OC1 and OC2 have their own separate registers for changing the duty cycle, they can both use TMR2 to generate the base frequency without interfering with each other. You can use initialization code for OC1 as a guide for writing the initialization code for OC2.

3. Wire the output of the phototransistor circuit (V_{sensor} in Figure 2) to AN12.

4. If everything is wired correctly, you should see the LED glowing red. Plug your NUScope into V_{sensor}. If you are able to see the voltage level toggle by blocking the light (with your finger) and allowing it to pass, then your circuit is working.

[3]Remember that current flows from anode to cathode in an LED and from collector to emitter in an NPN transistor.

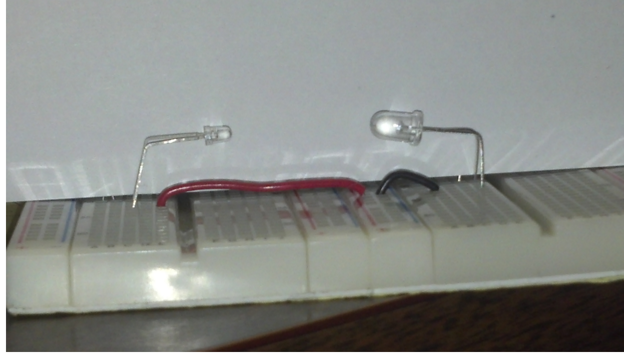


Figure 3: A picture of the phototransistor (left) and LED (right). A sheet of paper was added to make the pair more visible in the picture.

3.7 A brief recap

Let's recap what we have working so that we can describe what we still need to do. We are reading in four analog inputs. Two of these inputs are from our pots, which we will use as knobs to dynamically change the values of our control gains. The other two analog inputs read in our desired signal, V_{ref}, and the actual signal, V_{sensor}, from our LED/phototransistor pair. For outputs, we are using two 100 kHz PWM signals to generate our desired and actual signals. Because our desired signal is much lower frequency than the 100 kHz PWM, we change the duty cycle every $\frac{1}{20}$ of a second to get a filtered 10 Hz signal. Because we needed to perform this periodic task, we set up a 20 Hz ISR (`OutputISR`). This rate is appropriate for the LCD as well because we don't need a high refresh rate, so we update the LCD inside the same ISR. All we have left to implement is the control, which we do at a speed of 1 kHz. Earlier we set up a 1 kHz timer ISR (`ControlISR`), so all we have left to implement is the logic inside the routine for doing PI control.

3.8 The controls

In this section, we add the final piece to our program. Before we get into the coding aspect, we introduce a few issues with implementing a control law and how to deal with them in practice.

1. In class, we showed you how to implement a PI controller in C. Ideally, you would only need to write

   ```
   e = ref_signal - actual_signal;  // calculate the amount of error.
   eint += e; // calculate the accumulated error.
   u = Kp * e + Ki * eint; // calculate the control law.
   ```

   Of course, it's a different story in practice. There are two major issues associated with saturation that we have to address. First, our output, the PWM duty cycle, can only take on values between 0 and 799. Assume that `Kp` = 1 and `Ki` = 0, what are realistic signal values that will

   (a) cause `u` to be negative?

   (b) cause `u` to exceed 799?

   Remember that the signals are read from the 10-bit ADC, so they have a fixed range of values too. A simple way to deal with this problem is to cap the minimum and maximum values that `u` can take on.

   The second problem is known as integrator windup and is unique to the integral term, `eint`. The concept is simple enough, you can't integrate forever, so cap `eint` at a minimum and maximum value.

2. In control systems, we typically want a wide range of gain values to tune our control with. For tracking V_{ref}, you should be able to find a pair of gains for K_p and K_i between 0.1 and 100. Naturally we'd choose a `float` or a `double` data type to represent our gains. However, we know that multiplying and dividing these data types takes over 50 cycles per operation, which potentially makes them too slow in a high-speed control loop. Instead, we will perform all of our mathematical operations using integer data types. One issue is how to perform integer calculations with precision that is less than one. Consider the following two methods for calculating the control law u with only proportional gain

Method 1: `u = (Kp / KDIV) * e;` (1)

Method 2: `u = (Kp * e) / KDIV;`. (2)

Mathematically they are identical, but because of the subtleties of integer math we will see that performing the division as the final operation (Method 2) has higher precision.

   (a) Assume that `e` is fixed at 67 and `KDIV` is 10, fill in the missing entries in the table below for Method 1 and Method 2 based on the integer math that your C program would have performed. Compare these two methods with the first row, where the mathematically equivalent operations were computed by "hand." How accurate is each method relative to the numbers in the first row, for example, are the results within ± 5 of the first row?

K_p pot reading	10	11	12	13	14	15
`u` rounded to nearest tenth	67	73.7	80.4	87.1	93.8	100.5
Method 1: `u = (Kp / KDIV) * e;`	67					
Method 2: `u = (Kp * e) / KDIV;`	67					

Table 1: Results of integer math when dividing too early (Method 1) and at the end (Method 2).

   (b) We know that `Kp` can take on values between 0 and 1023 and that dividing by `KDIV` at the end of our calculations preserves some precision in our result. For nonzero values of `Kp` (i.e., $1 \leq$ `Kp` ≤ 1023), what is the effective range of gains that we can represent with the second method, i.e., $K_{\text{eff}} =$ `Kp / KDIV`? Compute K_{eff} by "hand", that is $\frac{1023}{10} = 102.3$.

   (c) Overflow can become an issue if the numbers we are multiplying are too large. What is the maximum positive value that the product `Kp * e` can produce? Remember that `Kp` is in ADC counts and `e` is the difference of two ADC counts. Can this value safely be stored in a signed 32-bit integer? In practice, you should also consider the maximum negative value, but for us the result doesn't change.

3. Below is a template to help guide you with the control loop ISR, you should fill in the question marks with actual code. The template gives good starting values for `WINDUP` and `KDIV`, but you may need to tweak these values depending on your circuit and code.

```
#define WINDUP (800)
#define UMIN (???)
#define UMAX (???)
#define KDIV (10)

void __ISR(_TIMER_3_VECTOR, ???) ControlISR(void) {
  static eint = 0;
  int e;
  // define any additional variables you need here
```

```
  ???
  // read all four analog signals using ADCManualRead(...)
  ???
  // calculate the error terms
  ???

  // avoid wind-up issues
  if (eint > WINDUP) {
    eint = WINDUP;
  } else if (eint <-WINDUP) {
    eint = -WINDUP;
  }

  // calculate the control law
  u = ((Kp * e) + (Ki * eint)) / KDIV;

  // avoid saturation issues
  if (u < UMIN) {
    u = UMIN;
  } else if (u > UMAX) {
    u = UMAX;
  }

  // update the actual signal's duty cycle
  // don't forget to cast to an unsigned data type!
  ???
  NU32LED2 = !NU32LED2;
  IFS0bits.T3IF = 0; // clear interrupt flag
}
```

4. Connect V_{ref} to channel A of the NUScope and V_{sensor} to channel B. Make sure both gains are initially set to zero. In order to get good tracking behavior, you should start by increasing `Kp` first. Once making `Kp` higher doesn't minimize the error, start to increase `Ki` until you are satisfied with the results. If the actual signal appears to always be at the rails[4] for all values of K_p or K_i, then you might have a sign error. Try flipping the sign for K_p and/or K_i and try tuning your circuit again. If increasing the integral term causes immediate saturation, consider decreasing the `WINDUP` constant.

5. **When you are satisfied with your gains, take a screenshot showing the V_{ref} and V_{sensor} signals on your NUScope.**

4 What to turn in

For the questions, you must submit typed responses. Place your typed responses, the required code (MyControllerMain.c, MyLibrary.h, and MyLibrary.c), and the NUScope screenshot in a **zip** file and submit the zip file through Blackboard before class on the date the assignment is due. The name of the zip file you submit will be of the form lastname_firstname_a5.zip.

In class you will connect your output signals (V_{ref} and V_{sensor}) to the NUScope and demo your circuit to the TAs.

[4] Stuck either high or low