ME 333: Introduction to Mechatronics

Assignment 5: Simple real-time control with the PIC32

Electronic submission due **before** 12:30 p.m. on February 15th

1 The PIC32 ADC Peripheral

Conceptually, the ADC peripheral is very simple: sample an analog pin for a specified amount of time and then hold the voltage level for a number of cycles so the circuitry can convert the analog signal into a 10-bit number. However, with its plethora of options, the ADC peripheral is the most complicated peripheral we will work with in ME 333. Ironically, the functions provided by the peripheral library can make it harder to program the ADC for general purpose usage. Instead, we will directly program a few of the SFRs corresponding to the ADC peripheral, so that we can manually sample an analog pin. This isn't as hard as it sounds. Many of the options that we want are enabled by default whenever the PIC is reset, like the manual sampling mode. If you are interested in what the default values are for the ADC refer to Section 17. 10-Bit A/D Converter of the Reference Manual, keep in mind that after a PIC reset all ADC registers are zero by default. The options that we need to explicitly enable are:

- Setting the output buffer, `ADC1BUF0`, to return the sample in a 32-bit number that ranges from 0–1023. The Reference Manual refers to the name of this option as `FORM`, which is located in `AD1CON1`,
- Connecting an analog pin, other than the default, `AN0`, to MUXA for sampling (`CH0SA` in `AD1CHS`), and
- Turning the ADC peripheral on (`ON` in `AD1CON1`).

Questions

(a) For this question refer to Algorithm 1. In line 9 of the algorithm, assume SAMPLE_TIME is set to 8. How long are we theoretically sampling `AN15` for? You should find that it exceeds the minimum sampling time of 132ns mentioned on the NU32v2: Analog Input wiki page. This code also shows two examples of polling, specifically lines 14 and 16, where the CPU continuously checks a register until an event happens. For the case of the timer in line 14, we are waiting for a rollover event and for line 16 we are waiting for the conversion from analog to digital to finish.

(b) Calibrating your sensor is an important step in control applications. For this assignment we are interested in the relationship between the PWM duty cycle at 20kHz and the brightness of the LED as measured by the phototransistor. Wire up the LED and phototransistor as in Figure 1 with R1 = 100Ω and C = 10nF. You will treat R2 as a design parameter in tuning the circuit. Ideally, your choice of R2 would allow for the use of the full PWM duty cycle range without saturating the transistor under normal and dark lighting conditions. Instead,

Algorithm 1 ADC Manual Conversion of Samples with SFRs. Important Microchip-defined ADC constants are in color

```
1: int sample;
2:
3: // set up ADC to read from AN15 and store the sample, so that we can read it using an int
4: AD1CON1bits.FORM = 4; // set output format to range from 0–1023
5: AD1CHSbits.CH0SA = 15; // Connect pin AN15 to MUXA for sampling
6: AD1CON1bits.ADON = 1; // Turn on A/D converter
7:
8: // use TMR3 to ensure that we sample the input for at least 132ns
9: OpenTimer3(T3_ON | T3_PS_1_2, SAMPLE_TIME);
10:
11: while(1) {
12:   AD1CON1bits.SAMP = 1; // start sampling
13:   WriteTimer3(1); // we set TMR3 = 1 (instead of 0), so delay loop works properly
14:   while(ReadTimer3()) {}; // sample for >= 132ns (this is a timed delay loop)
15:   AD1CON1bits.SAMP = 0; // stop sampling and start conversion to a 10-bit number
16:   while(!AD1CON1bits.DONE) {}; // wait for the conversion process to finish
17:   sample = ADC1BUF0; // read the buffer location where the result is stored
18:
19:   // send result over serial
20:   sprintf(RS232_Out_Buffer, "%f (%u)\r\n", (VOLTS_PER_SAMPLE*sample), sample);
21:   WriteString(UART3, RS232_Out_Buffer);
22: }
```

under normal lighting conditions, find a value for R2 in the range of 100kΩ–500kΩ that allows you to use at least 80% of your duty cycle before saturating the transistor. What value of R2 did you use? With this value of R2, turn in a representative plot of phototransistor output voltage vs. PWM duty cycle (you can either use NU32v2_plot or print a small set of representative points to a terminal on your PC and plot those values). Clearly show at what voltage level, if any, you are saturating at. This voltage level also represents the max voltage that you can expect to track as a reference signal. You should also test that your value for R2 works in a darkly lit room (to simulate this situation, just put your hand over the LED).

2 Real-time control with (of course) blinking lights

2.1 Overview

For this programming assignment, you will set the brightness of an LED based on a square wave reference signal. Instead of reading a reference signal from an external source, you will create an array that stores the voltage levels of the reference signal at equally spaced intervals in time. As you try to track the reference signal with your actual signal, the output from the phototransistor, there will be some error between the two readings. In order to drive the error to zero you will write a control loop that will implement a Proportional-Integral controller, more commonly known as a PI controller. A discretized version of a PI control law can be expressed as:

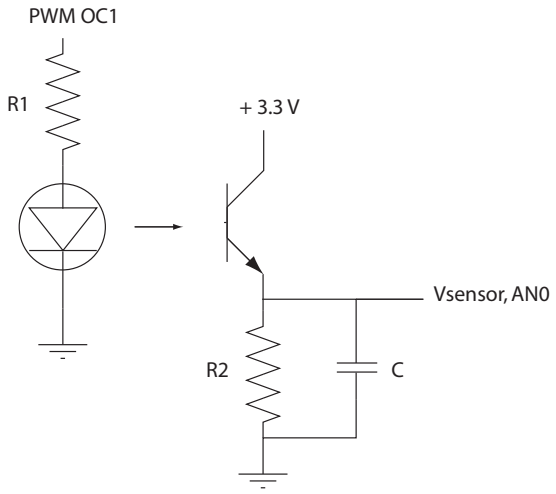

Figure 1: Circuit diagram for LED/Phototransistor, R1 = 100Ohms, C = 10nF, see Question 1.1 (a) for R2

$$u[n] = K_p \times error[n] + K_i \times \sum_{i=0}^{n} error[i]$$

where K_p and K_i are your gain values (these are parameters that you get to tune), $error[n]$ is the difference between the reference and actual signal at time step n, and $u[n]$ is your control law at time step n. As you can see the PI controller is aptly named. It has a term that is proportional to the error and a second term that is proportional to the integral of the error. Refer to Algorithm 2 for one way to implement a PI controller in code. Note that the algorithm is in pseudocode (a mixture of real code and English). It is up to you to fill in the details.

2.2 Programming assignment

For this assignment you can assume that you will always receive well-formatted inputs (i.e., the user is perfect and will not give your program invalid inputs). You should use `AN0` to read the output signal of the phototransistor and `OC1` to generate a PWM signal. If you haven't already, wire up your components as in Figure 1. In the circuit use R1 = 100Ω, C = 10nF, and use your value from Question 1(b) for R2. Here are the basics of the program:

- Create and initialize an array that stores the reference signal values. You should be able to fill the array with a square wave profile that runs at `REF_FREQ`, has voltage levels `V_LO` or `V_HI`, and a duty cycle specified by `REF_DUTY`.
- Write a control loop that executes at 1kHz, reads in a sample from `AN0`, computes the new duty cycle of `OC1` based on the PI control law, and stores the sample in an array for later display (see Algorithm 2).
- Approximately every second send the samples of the actual signal generated, reference voltage, and their index over serial, so that NU32v_plot can display the results.

There is also a template provided with this assignment in the Assignment 5 zip folder. You are free to modify the entire file. Finally, for the demo on Tuesday, you will show us how well you can track a 2Hz square wave, between 1V and 2V, and with a 50% duty cycle.

Questions

(a) With a reference signal at 2Hz, alternating between 1V and 2V, at 50% duty cycle, what gain values K_p and K_i did you use for your control law? Include a snapshot of how well your control law performed at tracking the reference signal.

Algorithm 2 Implementing a PI controller inside an interrupt routine

```
... code that reads the analog signal on AN0 ...
... store the current value, ADCVal, into an array ...
error = refVal - ADCVal;
dutyCycleProp = Kp * error; // this is the proportional controller
errorInt = errorInt + error;
... we can't integrate forever, so cap errorInt at reasonable values to prevent overflow issues ...

dutyCycleInt = Ki * errorint; // this is the integral controller
dutyCycleOutput = dutyCycleProp + dutyCycleInt;
... update OC1 duty cycle ...
```

2.3 Extra Credit

In order to receive extra credit, the original assignment must still meet the stated specifications under Section 2.2 Programming Assignment.

- Generate the reference signal using OC3 and read it in using ANx (where x $\neq$ 0). You will need to refer to the Reference Manual (section 17) in order to connect ANx to MUXB and to enable alternating between MUXA and MUXB.
- Display the current error on the Nokia LCD. Animations or text that enhance the information content of what is going on inside the program will generate more points.

3 What to turn in

For the questions in Section 1, you must submit typed responses. Place your responses and your .c file from the programming assignment 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 lastname_a5.zip.

In addition to submitting your assignment via Blackboard, you will also demonstrate your program to the teaching staff at the beginning of class on the date the assignment is due. For this assignment you will demo your ability to track a 2Hz square wave, alternating between 1V and 2V, at 50% duty cycle. You will display your results using NU32v2_plot.

For full credit, you must follow these instructions.