CHAPTER 24

Feedback Control of LED Brightness

In this chapter you will use feedback control to control the brightness of an LED. This project uses counter/timer, output compare, and analog input peripherals, as well as the parallel master port for the LCD screen.

Figure 24.1 shows an example result of this project. The LED's brightness is expressed in terms of the analog voltage of the brightness sensor, measured in ADC counts. The desired brightness alternates between 800 ADC counts (bright) and 200 ADC counts (dim) every half second, shown as a square wave reference in Figure 24.1. A successful feedback controller results in an actual brightness that closely follows the reference brightness.

Figure 24.2 shows the LED and sensor circuits and their connection to the OC1 output and the AN0 analog input. A PWM waveform from OC1 turns the LED on and off at 20 kHz, too fast for the eye to see, yielding an apparent averaged brightness between off and full on. The phototransistor is activated by the LED's light, creating an emitter current proportional to the

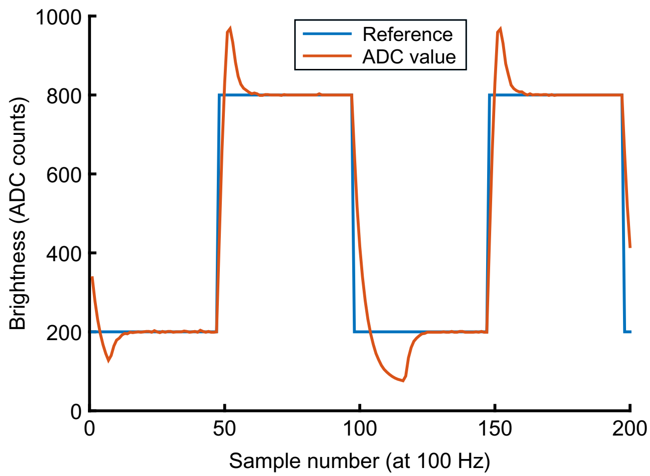

Figure 24.1
A demonstration of feedback control of LED brightness. The square wave is the desired LED brightness, in sensor ADC counts, and the other curve is the actual brightness as measured by the sensor. Samples are taken at 100 Hz, and the duration of the plotted data is 2 s.

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00024-X

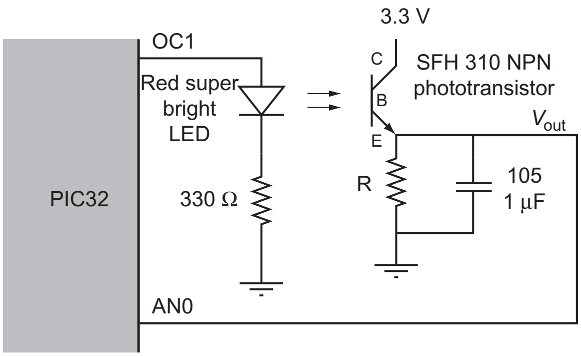

Figure 24.2
The LED control circuit. The long leg of the LED (anode) is connected to OC1 and the short leg (cathode) is connected to the 330 Ω resistor. The short leg of the phototransistor (collector) is attached to 3.3 V and the long leg (emitter) is attached to AN0, the resistor *R*, and the 1 µF capacitor.

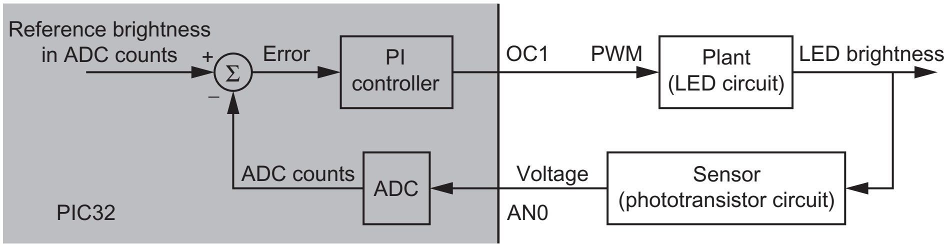

Figure 24.3
A block diagram of the LED brightness control system.

incident light. The resistor R turns this current into a sensed voltage. The 1 µF capacitor in parallel with R creates a low-pass filter with a time constant $\tau = RC$, removing high-frequency components due to the rapidly switching PWM signal and instead giving a time-averaged voltage. This filtering is similar to the low-pass filtering of your visual perception, which does not allow you to see the LED turning on and off rapidly.

A block diagram of the control system is shown in Figure 24.3. The PIC32 reads the analog voltage from the phototransistor circuit, calculates the error as the desired brightness (in ADC counts) minus the measured voltage in ADC counts, and uses a proportional-integral (PI) controller to generate a new PWM duty cycle on OC1. This control signal, in turn, changes the average brightness of the LED, which is sensed by the phototransistor circuit.

Your PIC32 program will generate a reference waveform, the desired light brightness measured in ADC counts, as a function of time. Then a 1 kHz control loop will read the sensor voltage (in ADC counts) and update the duty cycle of the 20 kHz OC1 PWM signal, attempting to make the measured ADC counts track the reference waveform.

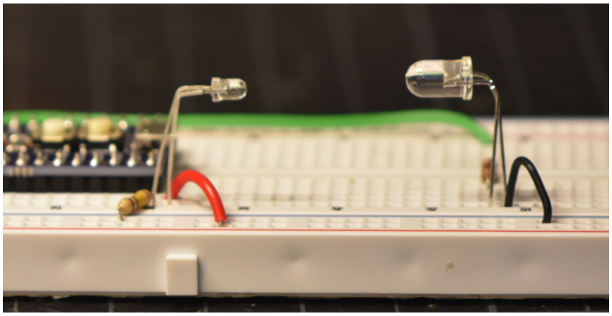

Figure 24.4
The phototransistor (left) and LED (right) pointed toward each other.

This project requires the coordination of many peripherals to work properly. Therefore, it is useful to divide it into smaller pieces and verify that each piece works, rather than attempting the whole project all at once.

24.1 Wiring and Testing the Circuit

1. **LED diode drop.** Connect the LED anode to 3.3 V, the cathode to a 330 Ω resistor, and the other end of the resistor to ground. This is the LED at its maximum brightness. Use your multimeter to record the forward bias voltage drop across the LED. Calculate or measure the current through the LED. Is this current safe for the PIC32 to provide?
2. **Choose** R. Wire the circuit as shown in Figure 24.2, except for the connection from the LED to OC1. The LED and phototransistor should be pointing toward each other, with approximately one inch separation, as shown in Figure 24.4. Now choose R to be as small as possible while ensuring that the voltage V_{out} at the phototransistor emitter is close to 3 V when the LED anode is connected to 3.3 V (maximum LED brightness) and close to 0 V when the LED anode is disconnected (the LED is off). (Something in the 10 kΩ range may work, but use a smaller resistance if you can still get the same voltage swing.) Record your value of R. Now connect the anode of the LED to OC1 for the rest of the project.

24.2 Powering the LED with OC1

1. **PWM calculation.** You will use Timer3 as the timer base for OC1. You want a 20 kHz PWM on OC1. Timer3 takes the PBCLK as input and uses a prescaler of 1. What should PR3 be?
2. **PWM program.** Write a program that uses your previous result to create a 20 kHz PWM output on OC1 (with no fault pin) using Timer3. Set the duty cycle to 75%. Get the following screenshots from your oscilloscope:

a. The OC1 waveform. Verify that this waveform matches your expectations.
b. The sensor voltage V_{out}.
c. Now remove the 1 μF capacitor and get another screenshot of V_{out}. Explain the difference from the previous waveform.

Insert the 1 μF capacitor back into the circuit for the rest of the project.

24.3 Playing an Open-Loop PWM Waveform

Now you will modify your program to generate a waveform stored in an `int` array. This array will eventually be the reference brightness waveform for your feedback controller (the square wave in Figure 24.1), but not yet; here this array will represent a PWM duty cycle as a function of time. Modify your program to define a constant `NUMSAMPS` and the global `volatile int` array `Waveform` by putting the following code near the top of the C file (outside of `main`):

```
#define NUMSAMPS  1000                      // number of points in waveform
static volatile int Waveform[NUMSAMPS];  // waveform
```

Then create a function `makeWaveform()` to store a square wave in `Waveform[]` and call it near the beginning of `main`. The square wave has amplitude `A` centered about the value `center`. Initialize `center` as (PR3+1)/2 and `A` as for the PR3 you calculated in the previous section.

```
void makeWaveform() {
  int i = 0, center = ???, A = ???; // square wave, fill in center value and amplitude
  for (i = 0; i < NUMSAMPS; ++i) {
    if ( i < NUMSAMPS/2) {
      Waveform[i] = center + A;
    } else {
      Waveform[i] = center - A;
    }
  }
}
```

Now configure Timer2 to call an ISR at a frequency of 1 kHz. This ISR will eventually implement the controller that reads the ADC and calculates the new duty cycle of the PWM. For now we will use it to modify the duty cycle according to the waveform in `Waveform[]`. Call the ISR `Controller` and make it interrupt priority level 5. It will use a `static` local `int` that counts the number of ISR entries and resets after 1000 entries.[1] In other words, the ISR should be of the form

[1] Recall that a `static` local variable is only initialized once, not upon every function call, and the value of the variable is retained between function calls. For global variables, the `static` qualifier means that the variable cannot be used in other modules (i.e., other `.c` files).

```
void __ISR(_TIMER_2_VECTOR, IPL5SOFT) Controller(void) { // _TIMER_2_VECTOR = 8
  static int counter = 0;         // initialize counter once

  // insert line(s) to set OC1RS

  counter++;                      // add one to counter every time ISR is entered
  if (counter == NUMSAMPS) {
    counter = 0;                  // roll the counter over when needed
  }
  // insert line to clear interrupt flag
}
```

In addition to clearing the interrupt flag (which we did not show in our example), your `Controller` ISR should set OC1RS to be equal to `Waveform[counter]`. Since your ISR is called every 1 ms, and the period of the square wave in `Waveform[]` is 1000 cycles, your PWM duty cycle will undergo one square wave period every 1 s. You should see your LED become bright and dim once per second.

1. Get a screenshot of your oscilloscope trace of V_{out} showing 2-4 periods of what should be an approximately square-wave sensor reading.
2. Turn in your code.

24.4 Establishing Communication with MATLAB

By establishing communication between your PIC32 and MATLAB, the PIC32 gains access to MATLAB's extensive scientific computing and graphics capabilities, and MATLAB can use the PIC32 as a data acquisition and control device. Refer to Section 11.3.5 for details about how to open a serial port in MATLAB and use it to communicate with `talkingPIC.c`, the basic communication program from Chapter 1.

1. Make sure you can communicate between `talkingPIC` on the PIC32 and `talkingPIC.m` in MATLAB. Do not proceed further until you have verified correct communication.

24.5 Plotting Data in MATLAB

Now that you have MATLAB communication working, you will build on your code from Section 24.3 by sending your controller's reference and sensed ADC data to MATLAB for plotting. This information will help you see how well your controller is working, allowing you to tune the PI gains empirically.

First, add some constants and global variables at the top of your program. The PIC32 program will send to MATLAB `PLOTPTS` data points upon request from MATLAB, where the constant `PLOTPTS` is set to 200. It is unnecessary to record data from every control iteration, so the program will record the data once every DECIMATION times, where DECIMATION is 10. Since the control loop is running at 1000 Hz, data is collected at 1000 Hz/DECIMATION = 100 Hz.

We also define the global int arrays ADCarray and REFarray to hold the values of the sensor signal and the reference signal. The int StoringData is a flag that indicates whether data is currently being collected. When it transitions from TRUE (1) to FALSE (0), it indicates that PLOTPTS data points have been collected and it is time to send ADCarray and REFarray to MATLAB. Finally, Kp and Ki are global floats with the PI gains. All of the variables have the specifier volatile because they are shared between the ISR and main and static because they are not needed in other .c files (good practice, even though this project only uses one .c file).

So you should have the following constants and variables near the beginning of your program:

```
#define NUMSAMPS  1000      // number of points in waveform
#define PLOTPTS   200       // number of data points to plot
#define DECIMATION  10      // plot every 10th point

static volatile int Waveform[NUMSAMPS];   // waveform
static volatile int ADCarray[PLOTPTS];    // measured values to plot
static volatile int REFarray[PLOTPTS];    // reference values to plot
static volatile int StoringData = 0;      // if this flag = 1, currently storing
                                          // plot data
static volatile float Kp = 0, Ki = 0;     // control gains
```

You should also modify your main function to define these local variables near the beginning:

```
char message[100];              // message to and from MATLAB
float kptemp = 0, kitemp = 0;   // temporary local gains
int i = 0;                      // plot data loop counter
```

These local variables are used in the infinite loop in main, below. This loop is interrupted by the ISR at 1 kHz. The loop waits for a message from MATLAB, which contains the new PI gains requested by the user. When a message is received, the gains from MATLAB are stored into the local variables kptemp and kitemp. Then interrupts are disabled, these local values are copied into the global gains Kp and Ki, and interrupts are re-enabled. Interrupts are disabled while Kp and Ki are assigned to ensure that the ISR does not interrupt in the middle of these assignments, causing it to use the new value of Kp but the old value of Ki. In addition, since sscanf takes longer to execute than simple variable assignments, it is called outside of the period when interrupts are disabled. We want to keep the time that interrupts are disabled as brief as possible, to avoid interfering with the timing of the 1 kHz control loop.

Next, the flag StoringData is set to TRUE (1), to tell the ISR to begin storing data. The ISR sets StoringData to FALSE (0) when PLOTPTS data points have been collected, indicating that it is time to send the stored data to MATLAB for plotting. Your infinite loop in main should be the following:

```
while (1) {
  NU32_ReadUART3(message, sizeof(message));       // wait for a message from MATLAB
  sscanf(message, "%f %f" , &kptemp,  &kitemp);
  __builtin_disable_interrupts();  // keep ISR disabled as briefly as possible
  Kp = kptemp;                     // copy local variables to globals used by ISR
  Ki = kitemp;
  __builtin_enable_interrupts();   // only 2 simple C commands while ISRs disabled
  StoringData = 1;                 // message to ISR to start storing data
  while (StoringData) {            // wait until ISR says data storing is done
    ; // do nothing
  }
  for (i=0; i<PLOTPTS; i++) {      // send plot data to MATLAB
                             // when first number sent = 1, MATLAB knows we're done
    sprintf(message, "%d %d %d\r\n", PLOTPTS-i, ADCarray[i], REFarray[i]);
    NU32_WriteUART3(message);
  }
}
```

Finally, you will need to write code in your ISR to record data when `StoringData` is TRUE. This code will use the new local `static int` variables `plotind`, `decctr`, and `adcval`. `plotind` is the index, 0 to `PLOTPTS-1`, of the next set of data to collect. `decctr` counts from 1 up to `DECIMATION` to implement the once-every-`DECIMATION` data storing. `adcval` is set to zero for now, until you start reading the ADC.

Your code should look like the following. You only need to insert lines to set OC1RS and to clear the interrupt flag.

```
void __ISR(_TIMER_2_VECTOR, IPL5SOFT) Controller(void) {
  static int counter = 0;          // initialize counter once
  static int plotind = 0;          // index for data arrays; counts up to PLOTPTS
  static int decctr = 0;           // counts to store data one every DECIMATION
  static int adcval = 0;           //

 // insert line(s) to set OC1RS

  if (StoringData) {
    decctr++;
    if (decctr == DECIMATION) {    // after DECIMATION control loops,
      decctr = 0;                  // reset decimation counter
      ADCarray[plotind] = adcval;  // store data in global arrays
      REFarray[plotind] = Waveform[counter];
      plotind++;                   // increment plot data index
    }
    if (plotind == PLOTPTS) {      // if max number of plot points plot is reached,
      plotind = 0;                 // reset the plot index
      StoringData = 0;             // tell main data is ready to be sent to MATLAB
    }
  }
  counter++;                       // add one to counter every time ISR is entered
  if (counter == NUMSAMPS) {
    counter = 0;  // rollover counter over when end of waveform reached
  }

  // insert line to clear interrupt flag
}
```

The MATLAB code to communicate with the PIC32 is below. Load your new PIC32 code and, in MATLAB, use a command like

```
data = pid_plot('COM3', 2.0, 1.0)
```

where you should replace `'COM3'` with the appropriate `COM` port name from your `Makefile`. The `2.0` is your K_p and the `1.0` is your K_i. Since your program does not do anything with the gains yet, it does not matter what gains you type. If all is working properly, MATLAB should plot two cycles of your square wave duty cycle waveform and zero for your measured ADC value (which you have not implemented yet).

Code Sample 24.1 `pid_plot.m`**. MATLAB Code to Plot Data from Your PIC32 LED Control Program.**

```
function data = pid_plot(port,Kp,Ki)
%   pid_plot plot the data from the pwm controller to the current figure
%
%   data = pid_plot(port,Kp,Ki)
%
%   Input Arguments:
%       port - the name of the com port.  This should be the same as what
%               you use in screen or putty in quotes ' '
%       Kp - proportional gain for controller
%       Ki - integral gain for controller
%   Output Arguments:
%       data - The collected data.  Each column is a time slice
%
%   Example:
%       data = pid_plot('/dev/ttyUSB0',1.0,1.0) (Linux)
%       data = pid_plot('/dev/tty.usbserial-00001014A',1.0,1.0) (Mac)
%       data = pid_plot('COM3',1.0,1.0) (PC)
%

%% Opening COM connection
if ~isempty(instrfind)
    fclose(instrfind);
    delete(instrfind);
end
fprintf('Opening port %s....\n',port);
mySerial = serial(port, 'BaudRate', 230400, 'FlowControl', 'hardware');
fopen(mySerial); % opens serial connection
clean = onCleanup(@()fclose(mySerial)); % closes serial port when function exits

%% Sending Data
% Printing to matlab Command window
fprintf('Setting Kp = %f, Ki = %f\n', Kp, Ki);

% Writing to serial port
fprintf(mySerial,'%f %f\n',[Kp,Ki]);

%% Reading data
fprintf('Waiting for samples ...\n');

sampnum = 1; % index for number of samples read
read_samples = 10; % When this value from PIC32 equals 1, it is done sending data
```

```
while read_samples > 1
    data_read = fscanf(mySerial,'%d %d %d'); % reading data from serial port

    % Extracting variables from data_read
    read_samples=data_read(1);
    ADCval(sampnum)=data_read(2);
    ref(sampnum)=data_read(3);

    sampnum=sampnum+1; % incrementing loop number
end
data = [ref;ADCval]; % setting data variable

%% Plotting data
clf;
hold on;
t = 1:sampnum-1;
plot(t,ref);
plot(t,ADCval);
legend('Reference', 'ADC Value')
title(['Kp: ',num2str(Kp),'  Ki: ',num2str(Ki)]);
ylabel('Brightness (ADC counts)');
xlabel('Sample Number (at 100 Hz)');
hold off;
end
```

1. Turn in a MATLAB plot showing `pid_plot.m` is communicating with your PIC32 code.

24.6 Writing to the LCD Screen

1. Write the function `printGainsToLCD()`, and its function prototype `void printGainsToLCD(void);`. This function writes the gains `Kp` and `Ki` on your LCD screen, one per row, like

   ```
   Kp: 12.30
   Ki:  1.00
   ```

   This function should be called by `main` just after `StoringData` is set to 1. Verify that it works before continuing to the next section.

24.7 Reading the ADC

1. Read the ADC value in your ISR, just before the `if (StoringData)` line of code. The value should be called `adcval`, so it will be stored in `ADCarray`. Turn in a MATLAB plot showing the measured `ADCarray` and the `REFArray`. You may wish to use manual sampling and automatic conversion to read the ADC.

24.8 PI Control

Now you will implement the PI controller. Change `makeWaveform` so that `center` is 500 and the amplitude `A` is 300, making a square wave swinging between `200` and `800`. This waveform is now the desired brightness of the LED, in ADC counts. Use the `adcval` read from the ADC and the reference waveform as inputs to the PI controller. Call `u` the output of the PI controller.

The output `u` may be positive or negative, but the PWM duty cycle can only be between 0 and PR3. If we treat the value `u` as a percentage, we can make it centered at 50% by adding 50 to the value, then saturate it at 0% and 100%, by the following code:

```
unew = u + 50.0;
if (unew > 100.0) {
  unew = 100.0;
} else if (unew < 0.0) {
  unew = 0.0;
}
```

Finally we must convert the control effort `unew` into a value in the range 0 to PR3 so that it can be stored in OC1RS:

```
OC1RS = (unsigned int) ((unew/100.0) * PR3);
```

We recommend that you define the integral of the control error, `Eint`, as a global `static volatile int`. Then reset `Eint` to zero in `main` every time a new `Kp` and `Ki` are received from MATLAB. This ensures that this new controller starts fresh, without a potentially large error integral from the previous controller.

1. Using your MATLAB interface, tune your gains K_p and K_i until you get good tracking of the square wave reference. Turn in a plot of the performance.

24.9 Additional Features

Some other features you can add:

1. In addition to plotting the reference waveform and the actual measured signal, plot the OC1RS value, so you can see the control effort.
2. Create a new reference waveform shape. For example, make the LED brightness follow a sinusoidal waveform. You can calculate this reference waveform on the PIC32. You should be able to choose which waveform to use by an input argument in your MATLAB interface. Perhaps even allow the user to specify parameters of the waveform (like `center` and `A`).
3. Change the PIC32 and MATLAB code so that MATLAB sends over the 1000 samples of an arbitrary reference trajectory. Then you can use MATLAB code to flexibly create a wide variety of reference trajectories.

24.10 Chapter Summary

- Control of the brightness of an LED can be achieved by a PWM signal at a frequency beyond that perceptible by the eye. The brightness can be sensed by a phototransistor and resistor circuit. A capacitor in parallel with the resistor low-pass filters the sensor signal with a cutoff frequency $f_c = 1/(2\pi RC)$, rejecting the high-frequency components due to the PWM frequency and its harmonics while keeping the low-frequency components (those perceptible by the eye).

- A reference brightness, as a function of time, can be stored in an array with N samples. By cyclically indexing through this array in an ISR invoked at a fixed frequency of f_{ISR}, the reference brightness waveform is periodic with frequency f_{ISR}/N.
- Since LED brightness control by a PWM signal is a zeroth-order system (the PWM voltage directly changes the LED current and therefore brightness, without any integrations), a good choice for a feedback controller is a PI controller.
- When accepting new gains `Kp` and `Ki` from the user, interrupts should be disabled to ensure that the ISR is not called in the middle of updating `Kp` and `Ki`. Interrupts should be disabled as briefly as possible, however, to avoid interfering with expected ISR timing. This can be achieved by keeping the relatively slow `sscanf` outside the period that interrupts are disabled. Interrupts are only disabled during the short period that the values read by `sscanf` are copied to `Kp` and `Ki`.

24.11 Exercises

Complete the LED brightness control project as outlined in the chapter.