Exercise 5: Motor Speed Control

Overview

In the previous sessions, we have seen how to generate PWM signals and how to use the input capture functionality of a microcontroller. In this exercise, we will combine these features in a real-world example.

Imagine you have a model car that is driven by a battery-powered DC motor. We want the car to drive at a constant speed, even if there is load on the motor because we are currently driving uphill or the battery power is low. Of course we can not always guarantee that the motor will reach the designated speed, but in this exercise we will at least try to do so.

Since this exercise is only about the controller of the motor speed, we do not have a real model car here, but we have the essential components. For completing the tasks, you receive an extra printed circuit board with a DC motor, a light barrier and respective driver circuits. We will develop a control application that measures the current motor speed using the light barrier and adapts the control signal until the speed reaches the designated value.

Driving a DC Motor

In the previous exercise, you have developed a PWM signal generator. We will now use a PWM signal to control the speed of the DC motor. Note that you should never directly attach a component with a high power consumption (like a motor) to a microcontroller, because the high current that occurs while running it would damage the controller. To circumvent this issue, we use special electronic components on the motor board that allow us to directly attach it to a PWM signal generated by the microcontroller.

Driving a DC Motor with ATmega8515

Exercise 5.1

a) Attach the DC motor board to STK500 so that you can feed a PWM signal to the motor. Do not forget to also connect the power supply pins on the motor board (`+5V`, `GND`), which is needed for the electronic components on the board to work properly. Which pin connections do you need?

b) Use the program from exercise 4.5 **d)** to test the motor at different speeds.

- Find out the approximate minimum duty cycle at which the motor runs and the maximum duty cycle from where increasing the value does not have any visible effect (listen to the sound of the motor). When finding the minumum duty cycle, is there a difference between the motor already turning at a higher speed or turning it on from stand?
- What happens if you hold the motor so that the axis is arranged vertically?

Determining the Speed of a Motor

There are multiple methods that can be used to measure the speed of a motor. One method is to attach a magnet to the motor axis and use a sensor that detects the magnetic field to measure the number of rounds per minute. The approach we will use here today is an optical way of measuring the motor speed. There is a small light barrier mounted on the motor board. In addition to that, a disk is mounted on the motor axis. One half of the disk is opaque and the

other one is transparent. When the opaque area of the disk is in between the light barrier, the light is blocked. Otherwise, the light can be detected by the receiving part of the light barrier. This optical information is transformed to a voltage by the electrical circuit on the motor board, which can be finally evaluated by the microcontroller. Note that you can not see the light that the light barrier produces, as it is in the infrared spectrum.

Determining the Motor Speed with ATmega8515

Exercise 5.2

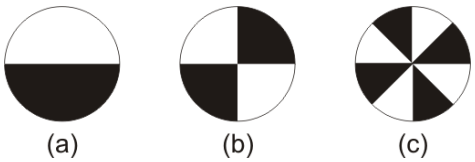
a) Attach the signal called `Schmitt-Trigger OUT` to the input capture pin of the microcontroller. Write a program that will measure the time between two successive input capture events (it is up to you whether you want to recognize the falling and/or rising edge(s)). Do all the calculations in the ISR and store the most recent result in a global variable. In the main program, periodically dump the current value in microseconds to the debug console.

b) Which resolution (in degrees) can we achieve when...

i) ...only considering the rising or falling edge...

ii) ...considering the rising and falling edge...

...of the signal from the light barrier when using the following encoder disks:



Hints

- To use PWM signal to drive the DC motor, use the rather small period value (e.g. 100 μs), otherwise the light barrier signal may become noisy and distort your input capture events. You might need to use small prescaler as well to achieve better resolution.
- We had some bad experience with floating point computations using AVR GCC compiler, especially for small numbers. So when calculating the motor speed, try to avoid using floating point numbers. If the number is smaller than 1, you may need to scale it up using a proper scaling factor and then use fix point (integers) operations. For example, let's assume the time stamps of two successive input capture events are $x_1 = 100$, $x_2 = 200$ and we wanted to compute the time interval in μs between the two events. The timer overflow value (maximum value of timer counter) in the example is $T = 1000$ and the corresponding real time (timer period) is $P = 500\,\mu s$. The code to Compute the interval using floating point may look like:

```
/* x1,x2: time stamp of input capture events */
float f1=x1,f2=x2;
...
/* Compute the fraction in one timer period
 * Note that the variable interval must be float, otherwise the
 * result is always 0, since (x2-x1)/T is smaller than 1 */
float interval = (x2-x1)/T;
/* Compute the real time in us */
interval = interval*P;
```

To avoid using floating point numbers, we can use fixed-point arithmentic:

```
/* x1,x2: time stamp of input capture events
 * scale up: note that 1000 is just for demonstration, you should pick up
 * a value that is large enough */
uint32_t s1=x1*1000;
uint32_t s2=x2*1000;
...
/* Compute the fraction, now the absolute value of the result can be safely
 * stored in an integer since it is much large than one. */
uint32_t interval = (x2-x1)/T;
/* Compute the real time in us */
interval = interval*P;
/* Scale down */
interval = interval/1000;
```

Control Theory

One of the requirements for our (so far virtual) model car was that the speed of the motor should be constant. Until now, we have seen that the motor speed may change even if we do not change the duty cycle of the PWM signal. Hence, we need to dynamically adapt the PWM frequency given the current speed of the motor.

This is a typical problem from the control systems area. We will now implement a simple yet effective controller to achieve the desired task. One of the essential features of a controller is a feedback loop, which means that the current state of the system is fed back to controller after the controller has set a new control value.

The controller we will use is a PI controller. PI means proportional plus integral controller. It basically works as follows: The controller has thee input values. The first one is the so called system deviation d, that is the difference of the measured value (so-called *actual value*) v_m to the desired value (so-called *setpoint value*) v_s. The second and third inputs are the proportial coefficient c_p and the integral coefficient c_i. The output s is the new setpoint for the system. A pseudo-code algorithm for a PI controller is shown in listing 1.

Motor Control with ATmega8515

Exercise 5.3

a) Extend your previous program(s) so that it regulates the speed of the DC motor using a PI controller. Output the current rounds per minute to the debug cosole (about every second). It should still be possible to input the new desired value (in rounds per minute) over the serial interface. Start with control parameters $c_p = 0.1$ and $c_i = 0$. Describe the behavior of the motor.

b) Now set $c_p = c_i = 0.1$. What happens?

c) Optimize the control parameters so that the motor reacts as fast as possible when you change the desired value.

d) What happens now if you hold the motor so that the axis is arranged vertically?

Listing 1: PI Controller Pseudocode

```
input(c_p); input(c_i)        // Retrieve control parameters
t_old ← time()
v ← 0

forever {
  input(v_m); input(v_s)      // Retrieve actual value and setpoint value

  t_cur ← time()              // Calculate how much time has passed
  Δt ← t_cur − t_old
  t_old ← t_cur

  d ← v_m − v_s               // PI controller
  v ← v + c_i · d
  s ← c_p · d + v · Δt

  output(s)                   // Output control value
}
```

Notes

- Try to write your program to save as much program memory as possible (but still document it appropriatly). Otherwise, you might hit the flash memory size limitation of 8K.