Exercise 5: PWM and Control Theory

Overview

In the previous sessions, we have seen how to use the input capture functionality of a microcontroller to capture external events. This functionality can also be used to measure the frequency of external signals (the measurement of the button press time was a very simple example for this scenario). In this exercise, we will learn how to generate control signals for driving external devices and how to combine both features.

Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is an efficient way to control analog circuits using digital signals. It is used in a wide variety of applications. As its name suggests, the control information is modulated in the width of a pulse. Figure 1 depicts three example PWM signals. Signal a) is a PWM signal with 25 % *duty cycle*, that is, the signal is *high* for one quarter of the time in one *period* and is *low* for the rest of the time. Signals b) and c) have a duty cycle of 50 % and 75 %, respectively.

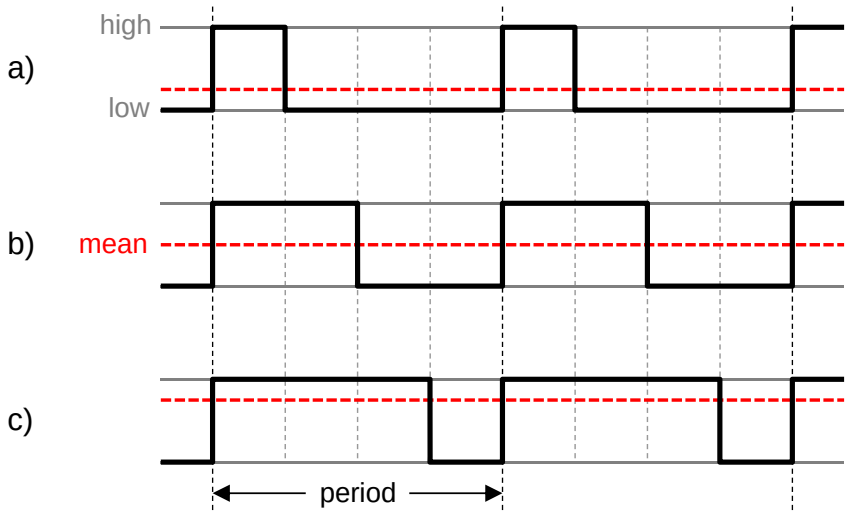

Figure 1: PWM Signals with 25 %, 50 % and 75 % Duty Cycle and respective Mean Values.

The mean value of a PWM signal (shown as dashed horizontal lines in figure 1) is an analog voltage value between 0 V and the high voltage of the digital signal, typically 5 V. This means that if the period of the PWM signal is reasonable short and the device attached to it behaves inertially, it will actually respond as if controlled by an analog signal with that analog voltage!

Exercise 5.1

We will now develop an application to control the brightness of LEDs. Atmega168 provides several mechanisms to generate PWM signals. For simplicity, we will stick to *Fast PWM Mode*.

a) What are the mean voltages corresponding to the PWM signals shown in figure 1, given a high voltage of 5 V?

b) Read the manual to understand how the Fast PWM Mode works. Find out the following information from the relevant sections in chapter 15 of the manual:

- Which pins can be used for PWM output when using Timer1? Which configuration do we need on those pins (e.g., do we need to configure them as output pins)?
- If we use the WGM 14, how does the PWM generator work for Compare Channel A? How can the period and the duty cycle be controlled?
- What is the difference between inverted and non-inverted PWM? How can we configure the microcontroller to generate such signals?

c) Develop a `pwm_init()` function that takes the period in milliseconds as input parameter and initializes the timer to generate a PWM signal with 50 % duty cycle on `OC1A` using WGM 14. It is acceptable if the range for the input parameter is between 0 ms and 1,000 ms. Use the LEDs to visualize the PWM output.

d) Develop a `pwm_set()` function that can be used to change the duty cycle of the PWM signal. It should receive the new duty cycle value (between 0 and 100) as input parameter. The PWM period should not be changed by the function.

e) As you have seen in the previous steps, the blinking of the LEDs is visible for large period values. Reduce the period of the PWM signal at a duty cycle of 50 % to find the value where the blinking is no longer visible. Which frequency does the period value correspond to?

f) Develop an application that receives commands on the serial port and adjusts the duty cycle of PWM signal accordingly. For example, the duty cycle of PWM signal is increased upon receiving a "`+`" character and decreased upon receiving a "`-`" character. Use a period of about 10 ms. Make sure that you can control the brightness of LEDs by configuring the duty cycle.

Driving a DC Motor

We will now combine the functionality of input capture and PWM signal generation in an illustrating 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 the car is 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.

We will use a PWM signal to drive the motor. The goal is to measure how fast the motor drives and compare this with the desired speed. If the motor is too slow, we will increase the PWM duty cycle, otherwise we will decrease it. This nebavior actually describes a simple controller that tries to minimize the difference ("error") between the desired speed and the actual speed.

Unfortunately this exercise does not feature a real model car, but you are of course free to apply your results to a setup of your own ;) For completing the tasks, you receive an extra printed circuit board with a DC motor and a light barrier. A disc is attached to the motor axis. Some parts of the disc are transparent, some are not. We will use the disc in combination with the light barrier to measure the current motor speed.

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 destroy the controller. To circumvent this issue, we use special electronic components on the printed circuit board that allow us to directly attach it to a PWM signal generated by the microcontroller.

Exercise 5.2

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`), they are needed for the electronic components on the board to work properly. Which pin connections do you need?

b) Use the program from exercise 5.1 **f)** 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 minimum duty cycle, is there a difference between the motor already turning at a higher speed or turning it on from stand?

c) 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 are using 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. Half of the disk is opaque and the other half 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. We will use input capture interrupts for this purpose. Note that you can not see the light that the light barrier produces, as it is in the infrared spectrum.

Exercise 5.3

a) Which resolution (in degrees) can we achieve from the light barrier when using the following encoder disks and if...

i) ...only the rising *or* falling edges are considered?

ii) ...rising *and* falling edges are considered?

b) Although not of interest in the current setup, can you imagine a way to measure the direction in which the motor is turning?

c) Attach the signal called `Schmitt-Trigger OUT` to the input capture pin of the microcontroller. Make sure that jumper `JP2` is set to the position `1-2` (*not* `2-3`). Extend your program from exercise **5.1 f)** so that measures 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)). From that on, calculate the time per round (depending on the number of edged per round) and the rounds per minute. Make sure that the rounds per minute is calculated correctly, you will need it for the subsequent exercises.

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 values to the debug console.

Note that you should disable interrupts in the main program before accessing the global variables to ensure that the ISR does not write to the values in parallel. Since we are using input capture, this does not reduce the precision as long as interrupts are only disabled for a short amount of time.

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.
- If you happen to encounter problems with floating point computations for small numbers, you might want to try using fixed-point numbers instead of floating-point numbers: for intermediate calculations, you can scale the values up by a proper scaling factor, use integer arithmetic and then scale them down again afterwards.

  Listing 1 shows how the *simplified* original code to compute the time interval between two capture events may look like using floating-point numbers, where t_1 and t_2 are the timestamps of two successive captured events, T is the MAX value of the timer and p is the associated real-time interval for T. To avoid using floating-point numbers, we could use fixed-point arithmetic as shown in listing 2.

Listing 1: Floating-point based calculation of motor speed

```
1  float calcRoundsPerMinute(float t1, t2, p)
2  {
3      /* Compute the fraction in one timer period. Note that the interval  */
4      /* variable must be of type float, otherwise the result is always 0! */
5      float interval = (t2-t1)/T;
6
7      /* Compute the real time in us */
8      interval *= p;
9
10     /* Return rounds per minute */
11     return interval/1000 * ...;
12 }
```

Listing 2: Fix-point based calculation of motor speed

```
1  float calcRoundsPerMinute(float t1, t2, p)
2  {
3      /* Scale numbers up. The value 1000 is just for demonstration, you   */
4      /* should pick up a value that is large enough for your application. */
5      uint32_t s1 = t1*1000;
6      uint32_t s2 = t2*1000;
7
8      /* Compute the fraction in one timer period. Now the absolute */
9      /* value of the result can be safely stored in an integer.    */
10     uint32_t interval = (s2-s1)/T;
11
12     /* Compute the real time in us */
13     interval = interval * p;
14
15     /* Scale down an return rounds per minute */
16     return interval/1000 * ...;
17 }
```

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 and 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 3.

Listing 3: PI controller pseudocode

```
1  input(c_p); input(c_i)          // Retrieve control parameters
2  t_old ← time()
3  v ← 0
4
5  forever {
6      input(v_m); input(v_s)      // Retrieve actual value and setpoint value
7
8      t_cur ← time()              // Calculate how much time has passed
9      Δt ← t_cur − t_old
10     t_old ← t_cur
11
12     d ← v_m − v_s               // PI controller
13     v ← v + c_i · d
14     s ← c_p · d + v · Δt
15
16     output(s)                   // Output control value
17 }
```

Motor Speed Control

Exercise 5.4

a) Extend your previous program(s) so that it controls 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) In contrast to your previous experience from exercise **5.2 c)**, what happens if you hold the motor so that the axis is arranged vertically?