Institute for Cognitive Systems
Technische Universität München
Prof. Gordon Cheng

Humanoid Sensors and Actuators - Tutorial 3

Microcontroller & Power Circuits (130 points)

In this tutorial we will learn:

- How to use the AVR timers and Interrupts
- How to generate Pulse Width Modulation (PWM) signals
- How to use PWM to control a DC motor
- How to use PWM to control servo actuators
- How to build a software Digital to Analogue Converter (DAC)

1 AVR Timers/Counters (20 points)

A timer is a special register whose value is incremented or decremented automatically[1] by the microcontroller, at a user-defined multiple of the clock frequency. A 8-bit timer for example, has values ranging from 0 to $(2^8 - 1) = 255$ and will therefore reach its maximum value in $K \times 255$ clock ticks, where K is the so called timer *clock prescaler*. When a timer reaches its maximum value, it "*overflows*". This means that its register value is rolled back to 0 and that a new count cycle is automatically initiated. When a timer overflows, an interrupt can be generated under certain conditions in order to inform the CPU. The AVR has 3 different timers: one of 16 bits and two of 8 bits.

a) Setup

- Download the documentation of the micro-controller:
  `http://ics.ei.tum.de/~flo/hsa-lecture/Tutorials/T3/Material/atmega32.pdf`
- Download the template project for micro-controller C programs:
  `http://ics.ei.tum.de/~flo/hsa-lecture/Tutorials/T3/Material/Atmega32Template.tar.gz`
- Include the AVR interrupt library within your code template: `#include <avr/interrupt.h>`

[1] For this reason, timers are also called counters.

b) Experimental protocol (10 points)

In this first exercise, we want to use the overflow interrupt of TIMER2 (8-bits) in order to blink a LED connected to the pin PC0 of the microcontroller, at exactly 0.5Hz:

T.1.1 (5 points) Within your "`main()`" function:

- Start by disabling the interrupts [2]. This is in general a good practice, which prevents the micro-controller from being interrupted while performing important operations.
- In the `TIMER2 Control Register` (TCCR2), initialize the timer clock prescaler to 64 [3].
- Set TIMER2 to issue an interrupt when an overflow event is detected. To do so, set the "`Overflow Interrupt Enable`" in the `Timer Interrupt Mask Register` (TIMSK)[4].
- Initialize the `TIMER2 Counter Register` (TCNT2) to 0. The value of this register will be automatically incremented by one unit at each tic of the prescaled clock, and will roll back to zero at overflow.
- Set C0 as the desired output pin.
- You can now re-enable the interrupts by setting the global interrupt enable.
- Define a dummy infinite loop in order to maintain your microcontroller active.

T.1.2 (5 points) Outside the "`main()`" function:

- Write an *interrupt service routine (ISR)* catching the overflow interrupt vector of TIMER2:

```
ISR(TIMER2_OVF_vect)
{
  ...
}
```

- Use a *global variable* into the ISR, so that the LED state changes after a defined number of counter overflow. Give this number into your report and justify.

c) Report (10 points)

R.1.1 (5 points) Explain with your own words what is an interrupt, an interrupt service routine and a busy wait state. Why are interrupts so interesting compared to busy wait states ?

R.1.2 (5 points) Knowing the clock frequency of the microcontroller and the size of the timer register, propose a new timer clock prescaler value, allowing to get as close as possible to the desired blinking frequency with the previously implemented method. Justify.

[2] ***Hint:*** take a look in `avr/interrupt.h`
[3] ***Hint:*** take a look p. 125-127 of the AVR manual.
[4] ***Hint:*** take a look p. 130 of the AVR manual.

2 Pulse Width Modulation (45 points)

a) Experimental protocol (30 points)

a).1 Generating a PWM signal (10 points)

Digital devices can only generate two voltage levels: HIGH=5V and LOW=0V. Pulse Width Modulation (PWM) is a widely used modulation technique which consists in varying the portion of the time a periodic digital signal spends in state "HIGH" versus the time that this signal spends in state "LOW". In this context, we define the so called *Duty cycle* of such as signal as:

$$D = \frac{T_{HIGH}}{T_{HIGH} + T_{LOW}} \times 100\% \tag{1}$$

PWM with different frequencies and duty cycles can be easily generated by exploiting some specific features of timers, namely the prescaler and the *Output Compare Register (OCR)*. When a timer is properly configured (see AVR datasheet), its value is incremented until it matches the OCR (defined by the user). At that point, the corresponding OCx pin is pulled LOW for the rest of the timer period before being set back to HIGH at overflow (see figure 1):

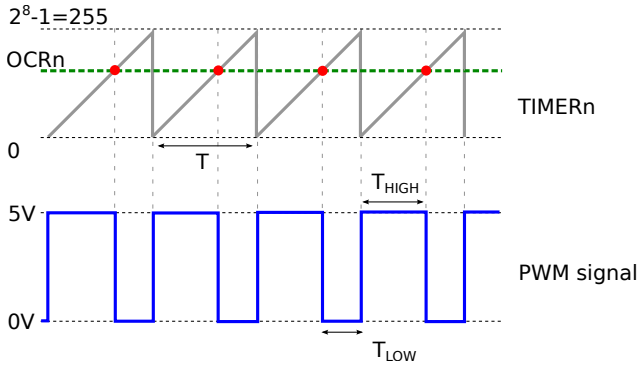

Figure 1: PWM generation principle on a microcontroller: the PWM signal (in blue) is here obtained with a 8-bits timer (`TIMERn` in gray) and output compare register `OCRn` (in dashed green). With appropriate filtering, one can get the PWM mean value (in orange).

T.2.1 (2 points) Initialize TIMER 0 with the following settings:

- Fast PWM in non-inverting mode
- Select a clock prescaler so that the PWM has a frequency close to 500Hz
- Set OC0 on compare match mode
- Set OC0 PIN as PWM output.

T.2.2 (3 points) Initialize the ADC MCU Block with the following settings:

- Use the AVCC with a capacitor on pin AREF (in our case the capacitor is not strictly needed)
- Use an ADC prescaler of 2 (the clock frequency of the ADC is half of the CPU frequency)

- Use the free running mode
- Use the ADC in the 8 bit mode
- Connect a potentiometer to the ADC0 pin, such that you can use the potentiometer to set voltages between 0V and 5V

T.2.3 (2 points) Use the value read by the ADC as duty cycle for your PWM signal.

T.2.4 (3 points) Light a LED with the obtained signal. What do you observe when you change the duty cycle ?

a).2 Simple DC motor controller (10 points)

PWM signals are widely used in power switching application. We here consider the control of a small DC motor. Neither the microcontroller not its USB power supply can directly drive such a motor. It is therefore necessary to use an external MOSFET transistor with an additional power supply, capable to withstand the motor current. A capacitor can also be added to the system for filtering purpose:

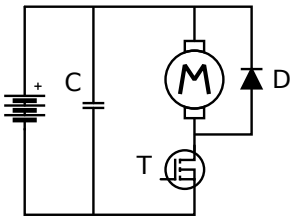

Figure 2: DC motor control circuit

- Download the motor and transistor datasheets: `http://ics.ei.tum.de/~flo/hsa-lecture/Tutorials/T3/Material`
- Ask a supervisor to check your circuit. Then switch on the 12V external power supply.

T.2.5 (10 points) Drive the N-MOS transistor T (cf: figure 2) using the PWM signal you previously generated in order to control the motor speed. Use the potentiometer and the ADC in order to change the motor speed.

a).3 Servomotor control (10 points)

Servo actuators are widely used in robotics. Most servos have se same standard three-wire connection: two wires for a DC power supply and one for control, carrying the PWM signal. The frequency of the control signal is 60Hz and the width of positive pulse controls the rotation angle. It is important to note that the huge majority of servos have the same neutral position at 1.5ms pulse width. The working principle of servos is illustrated in figure 3.
The mechanical limits of a servo must be taken into account prior to the PWM waveform generation in order to avoid damaging the gears or overheating the motor.

- Connect a 5V external power supply to the circuit. This should be the same power supply you used previously for the DC motor. Do not forget to set the output voltage to 5V ! Connecting the servo on a 12V power supply will destroy it !

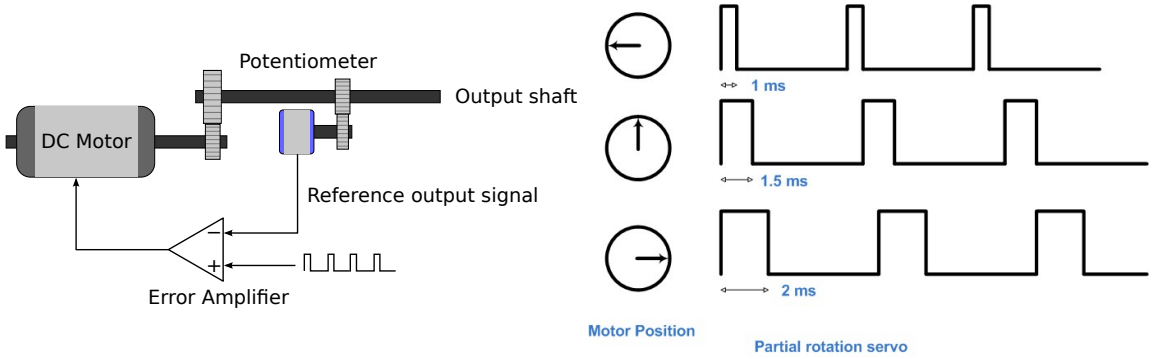

Figure 3: Working principle of servo actuators.

- Do not forget to connect the servo and the microcontroller GND pins together in order to avoid floating voltage levels on the PWM input.
- Ask a supervisor to check your circuit. Then switch on the 5V external power supply.

T.2.6 **(5 points)** Generate a 60Hz PWM signal, and control the pulse width using the ADC. Generating PWM at the wrong frequency will destroy the servo ! Therefore, be aware of any excessive noise or heat from the unit.

T.2.7 **(5 points)** Correctly set the upper and lower pulse width limits according to figure 3. You are encouraged to use safety error margins.

b) Report (15 points)

R.2.1 **(2 points)** Which AVR timers can be used for PWM generation ? Why ? (justify by refering to the relevant datasheet page numbers)

R.2.2 **(2 points)** Is a filter necessary for the LED in **T.2.4** ? Why ?

R.2.3 **(2 points)** What is the difference between the "`Fast_PWM`" and "`Phase_Correct_PWM`" proposed in the timer configuration registers ?

R.2.4 **(2 points)** What is the role of the diode D in the circuit 2 ?

R.2.5 **(2 points)** What may happen if this diode is disconnected while the motor is running ? Justify your answer (and of course do not try this on your circuit).

R.2.6 **(2 points)** Is it possible to reverse the rotation of the motor using the proposed circuit ? Why ?

R.2.7 **(3 points)** Which of the AVR timers is/are best suited for servo-motor control ? Why ?

3 Digital to Analog Converter (65 points)

As explained earlier, digital devices can only generate two voltage levels: HIGH=5V and LOW=0V. In many cases however, it is interesting to generate intermediate or even alternative voltage levels, in order – for example – to drive a loudspeaker. This is the role of a Digital to Analog Converter (DAC). The principle of a DAC is to generate a high-frequency – filtered – PWM signal, whose duty-cycle is modulated by the wave form of the desired signal. High quality hardware DACs are nowadays implemented in most microcontroller... but not in the AVR ! The goal of this part will therefore be to implement your own software-based DAC using PWM signals and a suitable filter in order to generate a sine wave. A continuous sine wave can be stored in a discrete form within a microcontroller, as a simple lookup table. Using a proper timer and interrupts, it is then possible to generate a PWM signal and to modulate its duty cycle according to the lookup table entries. The frequency of the sine wave will simply be the frequency at which the table is being browsed.

a) Experimental protocol (40 points)

- Download the `Atmega32TemplateDAC` template: `http://ics.ei.tum.de/~flo/hsa-lecture/Tutorials/T3/Material/Atmega32TemplateDAC.tar.gz`

T.3.1 Start by answering the questions **R.3.1** to **R.3.3**.

T.3.2 **(20 points)** Generate a 10 Hz sine wave by modulating the duty cycle of a PWM according to the provided lookup table entries[5]. Measure the filtered voltages with one of the microcontroller ADC and use the UART to send the measured voltages in binary format to the PC. Visualize the generated wave forms on Matlab. Include the graphs within your report.

T.3.3 **(20 points)** Implement the method proposed in **R.3.3** to generate a 100 Hz sine wave. Measure the filtered voltages with one of the microcontroller ADC and use the UART to send the measured voltages in binary format to the PC. Visualize the generated wave forms on Matlab. Include the graphs within your report.

b) Report (25 points)

R.3.1 **(5 points)** What is the maximum PWM frequency you can generate on the AVR with a 1MHz clock signal?

R.3.2 **(5 points)** Considering that the PWM duty cycle can only be changed once per timer overflow, deduce what is the maximum frequency of the sine wave obtained by going through the 256 elements of the table.

R.3.3 **(10 points)** Propose a method[6] allowing to generate signals with a higher frequency. **Justify your answer.**

R.3.4 **(5 points)** Take a look at the generated waveforms in the matlab plots:

- quantify the frequency error
- propose an explanation

[5] ***Hint:*** You may consider using two different timers, namely one in classic PWM mode for signal generation and one in normal mode with the output compare interrupt enabled to browse the table at a desired rate.

[6] ***Hint:*** Reducing the precision of the obtained signal is not problematic. Remember that you must anyway let the PWM counter the time to overflow and that it does not make sense to change the duty-cycle values several times before each overflow.