Technische Universität Berlin
Institute for Process an Chemical Engineering
Chair of Control Engineering

IESS – Laboratory 5

Motor control with the Arduino platform

The last building stone before implementing your controllers is to go through the actuators of the system. In this mini-segway, there are two DC-motors and in the first part of this lab you will have a look on how DC-motors can be controlled.

Pulse Width Modulation

In order to control the speed of a DC motor, a technique called Pulse Width Modulation (PWM) can be used. PWM is a method of controlling the amount of power that is supplied to a device by turning the power on and off very rapidly.

To control the speed of a DC motor using PWM, the DC voltage that is supplied to the motor is turned on and off rapidly by a switching device, such as a H-bridge. The on/off cycle is controlled by varying the duty cycle of the PWM signal. The duty cycle is the ratio of the time that the voltage is on to the total time of one cycle. For example, a duty cycle of 50% means that the voltage is on for half the time and off for the other half.

This in turns means that the averaged supplied voltage on the motor is 50% to that of the voltage from the voltage supply. By switching the The advantages of PWM-controlled DC motors include their low cost, simplicity, and high efficiency. They are commonly used in a wide range of applications, such as robotics, industrial automation, and automotive systems.

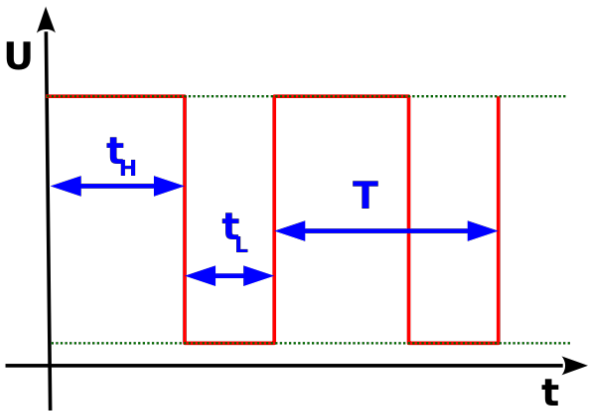

Figure 1: PWM-signal with duty-cycle $(T - t_L)/T$ %.

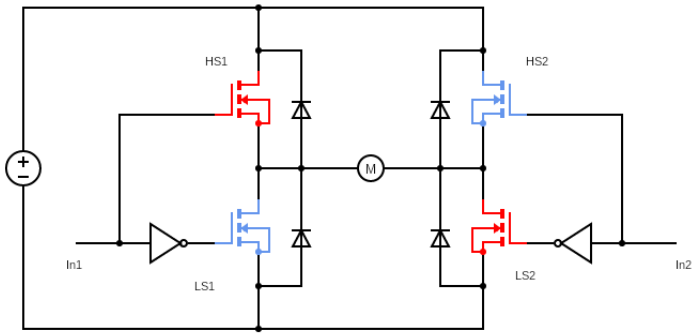

Figure 2: Schematic of a H-bridge, when HS1 and LS2 are both on the current flows from left to right through the motor. The diodes are called flyback-diodes and protects the transistors against back-EMF generated in the inductive motor by switching the voltage.

H-Bridge

A H-bridge is an electronic circuit that is commonly used to control the direction and speed of a DC motor. It consists of four switching elements that are arranged in a specific configuration to allow the current to flow in either direction through the motor.

The four transistors in the H-bridge are arranged in two pairs, with each pair consisting of a high-side switch and a low-side switch. The high-side switch is connected between the positive supply voltage and the load (in this case, the DC motor), while the low-side switch is connected between the load and ground.

To control the direction of the motor, the control switches the transistors on and off in a specific sequence. When the transistors in one pair are switched on and the other pair is switched off, the current flows in one direction through the motor. When the other pair of transistors is switched on and the first pair is switched off, the current flows in the opposite direction through the motor. In addition, by switching the transistors with a PWM signal also the speed may be controlled, as the average voltage over the motor can be varied.

Task 1 – PWM signals

In this first task you will familiarize yourself with the PWM-signals generated from the Arduino. The first step is to measure the PWM-signal using an oscilloscope. If the software Digilent WaveForms is not installed on your computer, please install it now.

1. Detach the MinSeg Shield and attach an oscilloscope-probe to a PWM-enabled output pin and generate a PWM-signal using 'analogWrite()'.

2. Setup the oscilloscope with correct input (AC or DC) and make a measurement. Measure the PWM-signal's voltage, frequency, duty cycle and resolution. Verify

this with the documentation.
Note: you may have to make multiple measurements with varying duty cycles to find everything.

3. Go to `https://drive.google.com/file/d/1cj9GJP6VNnoNMeMLXaRS_VNAamLaPOVb/view` and take note of which pins drive the motors. The MinSeg uses a DRV8833 motor driver, find the datasheet to this driver and have a look at Table 1-2 to learn about the operations of the motor driver.

4. Attach the shield again and write an Arduino script to verify which direction corresponds to pin M1A, M1B, M2A and M2B being set to high or low (or some other value using analogWrite()).

Counters & Timers

The MinSeg uses an ATmega 2560 as microcontroller, the ATmega2560 is a microcontroller from the AVR family, a common family of microcontrollers. For the purpose of implementing a sampling frequency it features several 8-bit and 16-bit counters known as Timer/Counters, which can be used for various timing and counting applications, including PWM-generators.

The ATmega2560 has six Timer/Counters, numbered from 0 to 5. Each Timer/-Counter can operate in different modes, such as Normal mode, Clear Timer on Compare Match (CTC) mode, and Pulse Width Modulation (PWM) mode.

In Listing 1 you can see code to configure Timer 5 to cause an interrupt with a predefined interval. The timer works by incrementing the number of counts once every clock cycle and comparing this to a stored value. As soon as the number of counts is greater than the compared value the timer fires an interrupt. In order to calculate the value to use in the counter the following formula is used

$$\#\text{Counts} = \frac{F_{\text{CPU}}}{\text{PRESCALER} \cdot F_S} - 1 \tag{1}$$

Listing 1: Arduino code for configuring a timer to throw an interrupt on a fixed interval.

```
#undef F_CPU
#define F_CPU 16000000UL // 16MHz clock frequency
#define PRESCALER 1024UL
#define F_S 1UL  // 1Hz sampling frequency
volatile uint8_t countFlag = 0;
int counts = F_CPU/(PRESCALER*F_S) - 1; //
volatile bool Mode = 0;
unsigned int i;

void setup() {
  //Used for verification/debugging
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
```

```
14
15
16    /*
17       Timers and interrupts
18    */
19    // Set a 1Hz Timer for clocking controller
20    TCCR5A = 0;
21    TCCR5B = (1 << WGM52) | (1 << CS52) | (0 << CS51) | (1 << CS50);
22    OCR5A = F_CPU / (PRESCALER * F_S) - 1; //Set the number of counts
          to compare the counter with
23    TIMSK5 |= (1 << OCIE5A);
24    TCNT5 = 0;
25  }
26
27  void loop() {
28
29    if (countFlag) {
30      countFlag = 0;
31      i = TCNT5;
32
33      // Prints out counts and number of counts since last interrupt
           for debugging purposes
34      Serial.print(counts);
35      Serial.print("␣");
36      Serial.println(i);
37      digitalWrite(LED_BUILTIN, Mode);
38    }
39  }
40
41  /* Interrupt Service Routine called by the counter
42   * Be careful with what code you place in this routine! If this
         routine takes too long to execute it may cause some complicated
         errors.
43   */
44  ISR(TIMER5_COMPA_vect) {
45    countFlag = 1;
46    Mode = !Mode;
47  }
```

The code in Listing 1 provides a general overview of using a counter on the ATmega2560 microcontroller. However, the exact implementation details and register configurations can vary depending on the specific requirements of your application and the programming language or environment you are using (such as Arduino IDE or Atmel Studio).

When setting up a sampling frequency the following steps may help

- Decide on a timer. Look into the pinout for the MinSeg-shield, the Arduino Mega

and the datasheet of a AtMega 2560 to see which timers are already being used by the motor control.

- Decide on an appropriate timer mode and timer pre-scaling. Due to the pre-scaling it may be that you cannot set the sampling frequency exactly as desired.
- Implement an Interrupt Service Routine, ISR, for your interrupt. This routine should execute as fast as possible to avoid nested interrupts.

Task 2 – Setting a sampling frequency

1. Decide on a suitable sampling frequency and what number of counts this would correspond to for the various prescalers available. Verify that this number of counts can be stored in a 16-bit (unsigned) integer. *Note: the ATmega2560 provides a variety of prescaler options, such as dividing the clock by 1, 8, 64, 256, or 1024.*
2. Configure the timer mode in the code provided, you will need to change the number of counts in *OCR5A* and the prescaler. See `https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega64datasheet.pdf`, Table 17-6 on page 157, to find out how to set the prescaler.
3. You can read the current value of the counter at any time to determine the elapsed time or the number of counts. The number of counts is stored in *TCNT5*. Verify that your counter works for your desired sampling frequency.

Task 3 – Draw a block diagram of the closed-loop system

Up until this point you have implemented a Kalman filter, an estimation of the angular position and angular velocity of the wheels. You have configured a timer to set a desired sampling frequency. The last part before connecting all parts is to come up with a structure of the closed-loop system.

1. Draw a block diagram of the entire closed-loop system, including controller, observer, control allocation, actuators, and any other relevant component.
2. Add which signals are being passed between the blocks and their dimensions.
3. Find the dimensions of your model, observer, controller, and other relevant matrices.

Task 4 – Implementation of controller

With your implemented matrix operations and sampling time you should now be able to implement your controller from the block diagram. Exactly how to implement it is your choice. However, we suggest you add blocks and verify their functionality one-by-one.

Extension 1 – Mathematical operations on an 8-bit micro controller

Since the Arduino is an 8-bit controller it works poorly with mathematical operations on floating-point numbers (floats) and other larger data types. To speed up your controller, try to rework your calculations and avoid performing mathematical operations on floating-point numbers as far as possible. For instance, multiplications with constants such as π should be avoided or done in only one step.

You can also estimate what ranges of values you will receive and the resolution of these, this might help to select suitable data types for your variables. To further optimize the speed of your controller, try to reduce the accuracy of your calculations until you notice a significant loss in performance.

Example: the constant 'PI' is stored with double precision as 3.14159265358979323846264338327 95, this means that the multiplication $2 \cdot \text{PI}$ can take quite a significant number of operations, on the other hand the constant 'TWO_PI' is already defined. Further, since $2 \cdot \text{PI}$ and 'COUNTS_PER_REVOLUTION' are constants we could also perform the operations $2 \cdot \text{PI}/\text{COUNTS\_PER\_REVOLUTION}$ in advance.

Extension 2 – Implementation of matrix operations

Depending on how you implemented your controller or observer you may or may not have used matrices. If you have not, one way to clean up your code would be to implement functions for the matrix operations you use, i.e. addition, subtraction, multiplication and inverse. In this extension you can have a look at doing this.

1. Implement a function to calculate the inverse of a matrix.
   Tip: look up the dimensions of the matrix you need to take the inverse of.

2. Implement functions for matrix addition, subtraction and multiplication.
   Tip: look up the dimensions of the matrices you need to add, subtract and multiply and start by implementing functions for the operations on these matrices.

3. Verify that your implementations work by printing the results to the serial monitor and compare with the results from MATLAB.

Acknowledgments

This laboratory was developed by Magnus Axelson-Fisk, Roxanne R. Jackson, and Tom J. Jüstel. ChatGPT (AI-generated text) was used to assist in writing the content of this laboratory.