NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTK8 - Construction of Embedded Systems

Software for DC Motor Speed Controller Card for Differential Drive Robot

Author:
Thomas Andersen

Autumn 2021

Table of Contents

List of Figures

List of Tables

1 Summary and conclusion

Embedded software for a DC motor speed controller card for a differential drive robot based on a nRF52840 was implemented and tested. Bear-metal programming of the ATmega328p was the main objective of this project, this entails writing software drivers for motor actuation and reading encoder measurements, as well as a simple speed estimator and PID controller. As the hardware for the motor speed controller card was developed in parallel with the software, the Arduino Uno (based on the ATmega328p) was used for prototyping. Due to limited time, the three students working on the project were not able to assemble their contributions to the overall development of the motor speed controller card.

Using a separate microcontroller for speed control of the robot is beneficial for further development on the robot as it would decrease the software complexity of the nRF52840 based robot, enabling the robot to do more advanced computations (e.g. SLAM). This comes at the cost of increased overall hardware complexity.

2 Background information

Since 2004 students at the Department of Engineering Cybernetics have been assigned to work on the *robot-project* (also known as the *SLAM-project*) in order to do research and development for their specialization and master theses. As a supplement for student's specialization project, they are enrolled in TTK8 where they are to create something related to embedded systems relevant for their specialization project.

The goal of the robot-project is to be able to map an *a priori* unknown environment using multiple exploring autonomous robots connected to a central server. The robots are to send information to the server about their local surroundings, and the server should be able to stitch together the relevant information from the robots to make a global map.

Since the robots necessarily have to navigate autonomously through the environment there is an apparent need for a control system for the motion of the robots. The current robot is built upon a Nordic Semiconductor nRF52840 Developement Kit(DK). The robot moves using two wheels, each attached to a DC motor, which are controlled by a L298N motor driver using Pulse-Width-Modulated (PWM) signals from the nRF52840 DK for setting shaft speed and control signals for setting direction. The DC motors have built-in rotary encoders which can be used to measure the rotation speed and direction of the motor shaft. One of the main tasks the of the nRF52840 DK is motor speed control. The most recent software running on the device has a dedicated thread of execution running a PID controller for the rotational velocity of the wheels.

3 Problem statement

It is thought that creating an external motor card for controlling the speed of the wheels of the robot would be beneficial for future development of the robot. The envisioned system should:

1. Have a clear interface to the nRF52840 DK. The nRF52840 DK should be able to set speed references for the external motor card. The nRF52 DK should also be able to read the speed of the robot's wheels directly from the external motor card.
2. Interface the quadrature encoders of the DC motors to be used to estimate the robot's speed which is to be used as feedback in a control loop.
3. Control the motor speed to a given set point given by the nRF52840 DK.
4. Be encapsulated on a compact PCB, easy to mount on the current robot setup

Implementation of software for an external DC motor speed controller card is the main goal of my project for TTK8, this includes point (2) and (3) above. The main accomplishment by outsourcing the software implementation of the control algorithm currently running on the nRF52 DK to an external motor controller card is decreased complexity of the software running on the nRF52840 DK, which in turn should ease future software developement.

Figure 1 and 2 illustrate a high-level overview of the current system and the envisioned system, respectively.

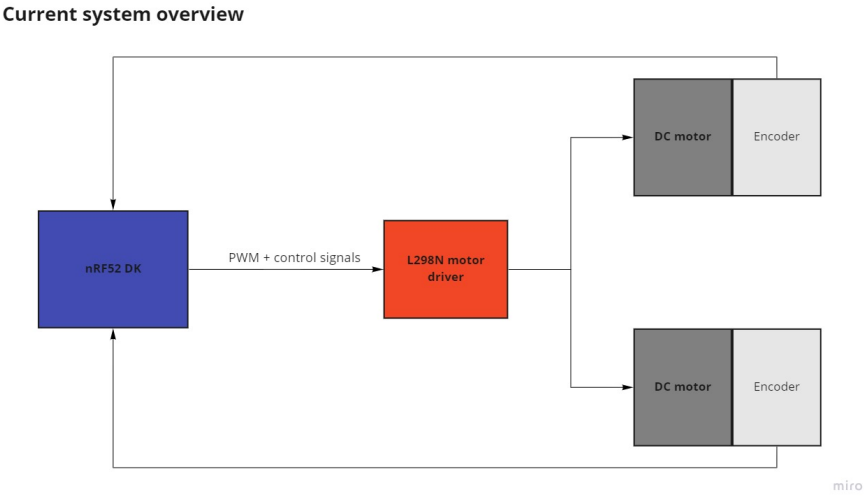


Figure 1: High-level overview of the current system

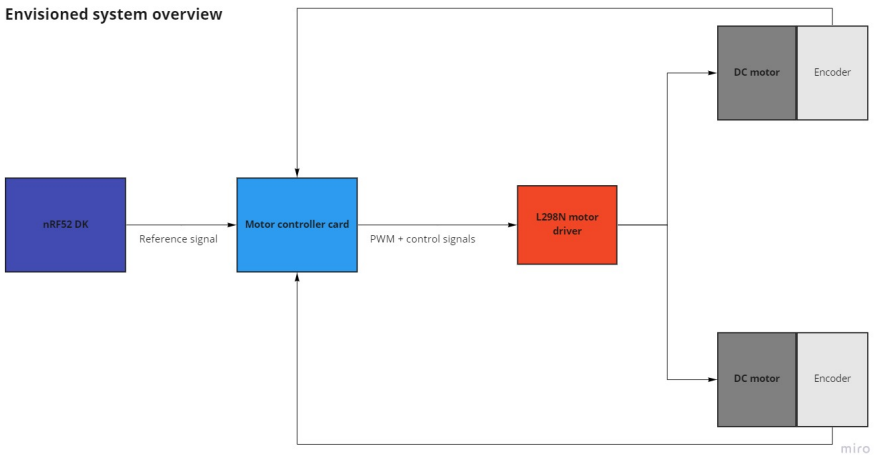


Figure 2: High-level overview of the envisioned system with an external motor card

4 Hardware

This section gives a brief description of the hardware components used in this project.

4.1 Arduino Uno

The Arduino Uno is an open-sourced development board based around the ATmega328P microcontroller. In development of the DC motor control card for this project, the Arduino Uno was used as a prototyping board for embedded software development. Using the Arduino Uno enabled fast development of software as no initial hardware setup and wiring was needed to start programming the atmega328p, which was to be used in the final PCB.

The Arduino Uno has 14 digital I/O-pins and 6 analog input-pins easily available from the on-board female headers. Moreover, the Arduino Uno board is integrated with a 16MHz ceramic oscillator (used as system clock), USB-connector for supplying the board with 5V and serial communication, an ICSP-header for flashing programs and debugging using the Atmel ICE debugger, several built-in LEDs, and a reset-button. [1]

4.2 ATmega328P

The ATmega328p is a 8-bit microcontroller based on the AVR enhanced RISC architecture produced by Microchip. The microcontroller has a rich set of features, including 32kB programmable flash memory, 1kB EEPROM, 2kB SRAM, 32 general purpose registers, two 8-bit Timer/Counters, one 16-bit Timer/Counter, internal and external interrupts, 6-channel 10-bit ADC, USART-, I2C-, and SPI-interfaces. [6]

4.3 L298N motor driver

The L298N motor driver is the hardware interface to the DC motors driving the two wheels of the robot. The L298N motor driver is based on the L298 integrated circuit, and is also equipped with a 78M05 5V regulator, LED indicators, protection diodes, and a heat sink. The L298N provides a simple interface to control speed and direction of two DC motors by exposing four control inputs for direction control for each wheel, and two PWM input for speed control for each wheel. [7] The input pin layout for the L298N is given in table 1. The truth table for the direction control signals is given in table 2.

L298N input pin	**Info**
ENA	PWM signal for motor A
ENB	PWM signal for motor B
IN1	Direction control pin for motor A.
IN2	Direction control pin for motor A.
IN3	Direction control pin for motor B
IN4	Direction control pin for motor B

Table 1: Input pin layout to L298N motor driver

IN1/IN3	**IN2/IN4**	**Motor A / Motor B direction**
0V	0V	off
0	1	forward
1	1	backward

Table 2: Truth table for direction control

4.4 Machifit 25GA370 DC motors

Each of the robot's wheels are powered by a separate 12V DC motor. The DC motors are stated to be able to run at 110 rpm. [3]

The motors have built-in quadrature encoders used to measure shaft displacement. Each encoder outputs a series of pulses on two channels, A and B, when the motor shaft is rotating, as shown in figure 3. The pulses on channel A and B are slightly phase-shifted enabling detection of the motor's rotational direction. Depending on the set speed of the motor, the encoder will output pulses at a frequency between (100 Hz and 15 kHz). The resolution of the encoders are $\sim$ 855 pulses per revolution, or $\sim$ 4 pulses per mm (given that the circumference of the wheels are 210 mm). Be aware that the number of pulses per revolution may vary from encoders of the same model. Therefore, one should always calibrate the encoders before usage.

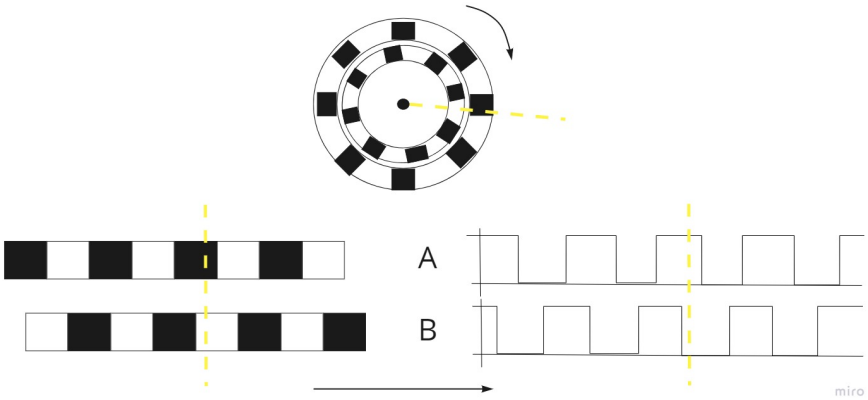


Figure 3: Working principal of a quadrature encoder.

5 System overview

The DC motor controller card is connected to four peripheral hardware modules: The nRF52840, an encoder for odometry measurements for each wheel, and the L298N motor driver. This is illustrated in the context diagram given in figure 4. The dotted lines in the context diagrams are there to illustrate that they are not a part of the scope of my work on this project, however, they are provided to show how the end product is designed.

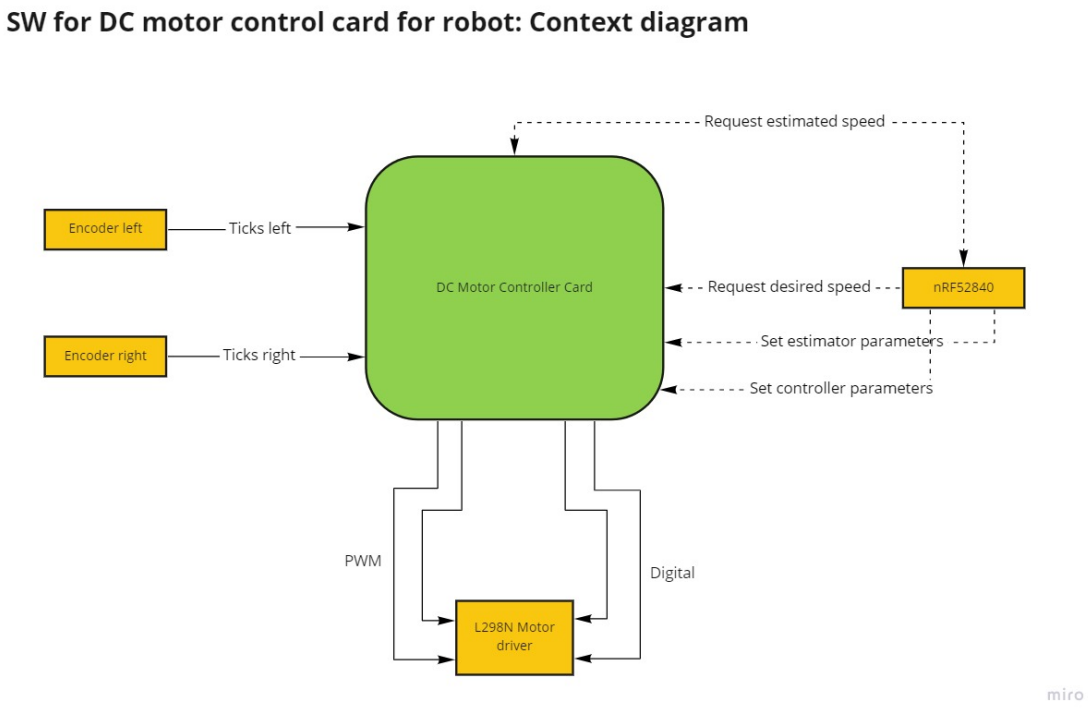


Figure 4: Context diagram

The software for the DC motor control card consists five main modules; an encoder driver, a speed estimator, a motor driver, and a timer. The encoder driver reads encoder pulses/ticks and feeds these measurements into the speed estimator where wheel speed is calculated. The speed controller, uses the wheel speed from each wheel to calculate a control action/output to the motor driver. The timer module is used to set a flag periodically to initialize computation of a control action. The motor driver outputs PWM and direction control signals to the L298N board. The interaction between the software modules can be seen in figure 5. For completeness, an inner analysis of the motor driver module is also given in figure 6.

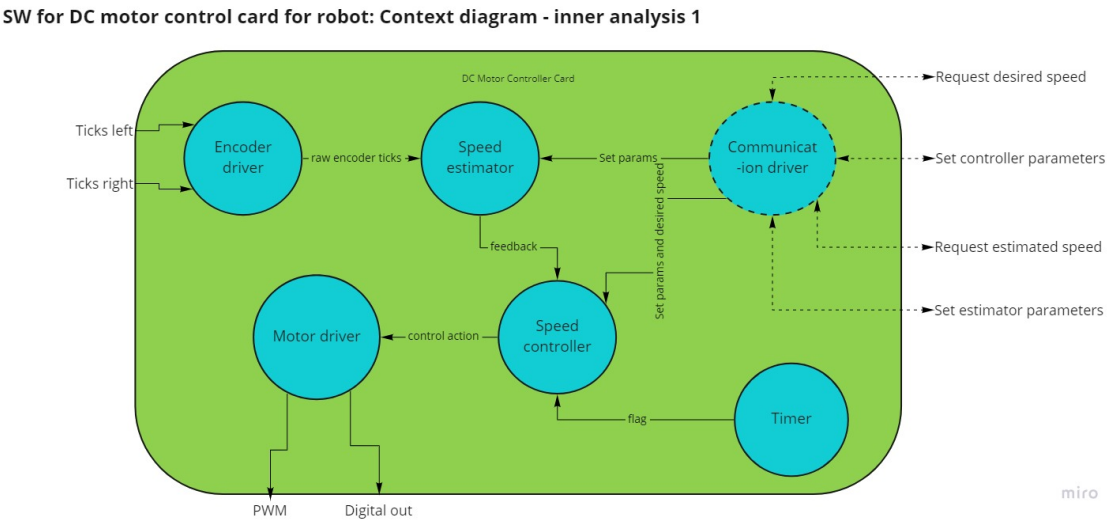


Figure 5: Context diagram: Inner analysis - DC motor controller card

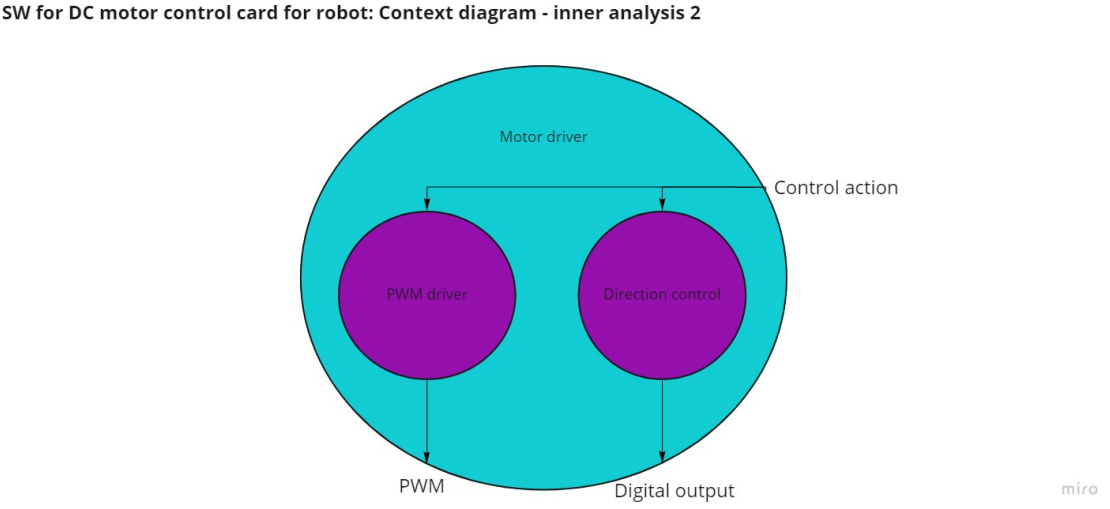


Figure 6: Context diagram: Inner analysis - Motor driver

6 Implementation details

This section goes into detail about the software modules introduced in section 5. For code documentation refer to Appendix A.

6.1 USART driver

A USART driver to enable serial communication was implemented to ease debugging, and enables plotting of measurements. As the Arduino Uno board is equipped with a pre-programmed ATmega16U2 microcontroller which serves as a USB interface to the ATmega328P, one can use USART for serial communication over USB to a COM port. This enables communication with a computer at the workstation by overriding the standard AVR *printf()* functionality. The baud rate of the serial communication is set to 76800 bits/s.

6.2 Encoder driver

The encoder driver's purpose is to measure the revolutions of the motor shaft in order to keep track of how far the robot has travelled. The accumulated number of encoder pulses will be used to estimate the speed of the robot.

External interrupts 0 and 1 on the atmega328p are used for rising edge detection of logical high on channel A for the left and right wheel, respectively. I.e. the *interrupt service routine* (ISR) for external interrupt 0 is triggered every time a logic high level is detected on channel A for the encoder attached to the left wheel. The ISR will read the logic value of channel B. If channel B is high, this indicates that the motor shaft has rotated in the clockwise direction. Thus, a variable representing left wheel encoder ticks is incremented by one. If channel B is low, the motor shaft has rotated in the counter clockwise direction, followingly the left wheel encoder ticks variable is decremented by one. The same logic follows for the right wheel implemented in the ISR for external interrupt 1. Figure 7 illustrates the implemented logic of measuring odometry from encoder values.

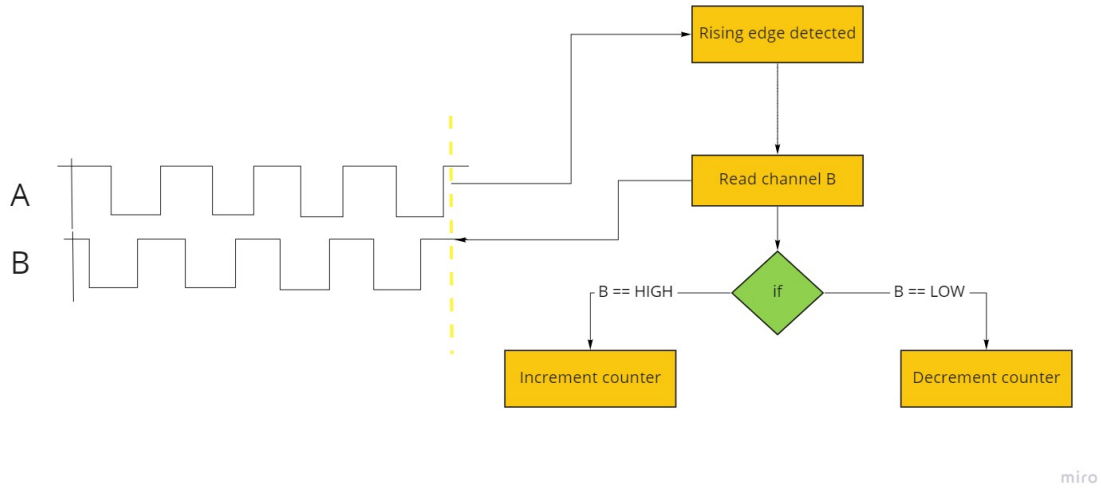


Figure 7: Using an ISR to measure accumulated encoder ticks.

6.3 PWM driver

The PWM driver is used to output PWM-signals to the L298N motor driver to control the speed of the motors.

The PWM-signals are generated using the 8-bit Timer/Counter 0 on the ATmega328P. Timer/-Counter 0 is initialized to run in the phase-correct PWM mode at a frequency of 31 kHz. Since the L298N motor driver is used to control the speed of the motor accurately by low-pass filtering the high-frequency PWM-signals, it was important to set the PWM frequency high enough in order to ensure that only the DC component of the PWM signal (i.e. the average) is observed by the L298N motor driver. Too low PWM frequency would result in jittery movement of the motor, however, too high frequency would result in the L298N motor driver to burn off unnecessary amounts of power. Another factor effecting the choice of PWM frequency is that frequencies < 20 kHz would produce unwanted audible noise.

6.4 Motor driver

The motor driver is the software interface to the L298N motor driver. It outputs 4 digital signals to the L298N board to control the rotational direction of both the motors. Moreover, the motor driver uses the PWM driver to control the speed of both the motors. This software module exposes easy to use functions for setting speed and direction at the same time for each wheel (i.e. the PWM driver is not intended to be used directly by the main application).

6.5 Timer

The timer software module is used to keep track of time since program start-up (for plotting), as well as to ensure that the control loop runs at a set frequency.

The timer module uses the 8-bit Timer/Counter 2 on the ATmega328P. It is implemented by incrementing a counting variable representing time in milliseconds since program startup. The counting variable is updated using an ISR triggered every millisecond. This is accomplished by initializing Timer/Counter 2 to clear on compare match with a value of 249. With 64 as prescaler and using the internal clock of the MCU (16 MHz), an ISR can be triggered every millisecond. Every time the ISR is triggered a timeout check will be conducted; if the elapsed time is equal to a set period, a flag will be set. This feature enables the speed controller to run at a given frequency.

6.6 Speed estimator

The speed estimator software module is used to compute the speed of each of the wheels of the robot using the encoder driver. The speed estimator samples encoder values at 50Hz (half of the lowest frequency of pulses from the encoders). By keeping track of the accumulated encoder pulses at a steady rate, wheel speed can be estimated accurately.

By initializing the 16-bit Timer/Counter 1 on the Atmega328P to clear on compare match with a value of 624, prescaler set to 256, and using the internal clock (16MHz) to drive Timer/Counter 1, an ISR can be triggered every 0.01 seconds. The ISR will store the previously sampled accumulated encoder pulses in a separate variable, before sampling the accumulated encoder pulses. Thus, one is able to compute the difference between the newest and previous sample of encoder pulses at a fixed rate.

Occasionally when the motor is rotating at a constant speed the ISR will not sample the same amount of encoder pulses between each sampling period due to the resolution of the encoder. This would result in periodic spikes in the speed estimates although the motors were rotating at a constant speed, especially for low speeds. One can minimize this effect by lowering the sampling frequency of the speed estimator, however, this would degrade the accuracy of the measurements at high speeds. A digital low-pass filter was implemented to smooth out the fully encoder-based speed estimates.

6.7 Speed controller

A PID controller was implemented for speed control of each wheel. The PID routine is widely used in industrial control systems due to it being easy to both implement and tune[5]. Tuned appropriately the PID controller is considered robust, has good disturbance rejection, and is able to track setpoints with fast response. (1) denotes the error between a given desired output and a measurement. The continous ideal PID controller in the time-domain is given in (2).

$$e(t) = r(t) - y(t) \tag{1}$$

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{2}$$

Discrete PID controllers come in a variety of forms mostly depending on the types of approximations used for integration and differention of the error $e(t)$. Using the notation for discrete outputs $u[kT] = u[k]$, where T is the period of the PID control loop and k is a discrete step, we can obtain the discrete PID controller in (5). (5) is derived using the approximations in (3) and (4).

$$\int_0^t e(t) \approx \sum_{j=0}^{k} e[k] \tag{3}$$

$$\frac{de(t)}{dt} \approx \frac{e[k] - e[k-1]}{T} \tag{4}$$

$$u[k] = K_p e[k] + K_i \sum_{j=0}^{k} e[k] + K_d \frac{e[k] - e[k-1]}{T} \tag{5}$$

The discrete PID controller given in (5) is prone to *integral-windup*, a condition which will occur when there is a large change in the input making the integral term accumulate. This is undesirable when the output becomes saturated as the integral term will take some time to unwind, thus, producing excessive overshoot. The robot will mostly be operating at top-speed until it suddenly has reached its target location, thus, if disregarding integral-windup, the robot would in most circumstances overshoot its target speed since the integral term's contribution to the controller output would have to unwind. Therefore, an anti-windup scheme was implemented. There are various ways one could implement such a scheme. The chosen implementation is given in [4, p.115-116], which will subtract from the integral exactly the amount needed to keep the output at the saturation bound. Pseudocode for the anti-windup scheme is given in Algorithm 1.

The implemented control loop frequency, by using the timer module, is set to 50Hz. It is important that the control loop frequency is not any higher than the frequency of speed measurements in order to not let clock cycles go to waist. As the nRF52840 uses successive loop-closure to implement pose control with the speed controller in the inner loop, it is important that the pose control loop frequency is at least lower than 50Hz.

Using the timer module to measure the speed of a control loop, i.e. measuring encoder ticks, computing speed, computing a control action and actuating the motors, it was found that the loop takes less than 1ms (given system clock speed at 16MHz). Thus, an alternative solution to keeping the control loop to run at a steady rate would be to place the control loop in an ISR triggered by a Timer/Counter, such as in [5]. However, the ATmega328p only has three Timer/Counters and they are all in use with the current software.

Algorithm 1 Anti-windup scheme

Ensure: $K_i ! = 0$
  $u = PID\_output(r, y)$
  $uLimited = u$
  if $|u| >$ saturationBound **then**
    $uLimited = sgn(u) \times saturationBound$
    $integralError = integralError - \frac{T}{K_i}(u - uLimited)$

7 Testing and verification

Testing of the encoder driver, motor driver, speed estimator, and speed controller was conducted. The schematics of the test setup along with a picture from the workstation can be seen in figure 9 and figure 8. (Dotted lines in the schematic show the pins which could be used for either SPI or I2C communication to the nRF52840).

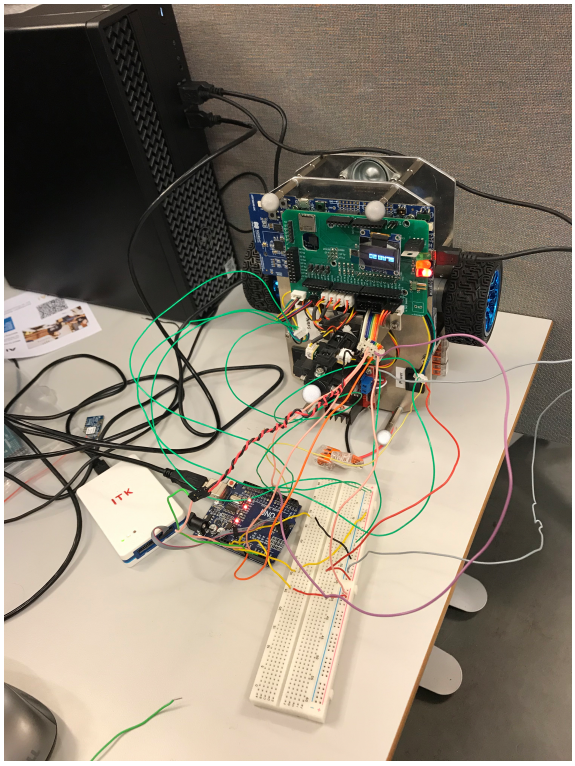


Figure 8: Test setup on workstation

7.1 Encoder and motor driver testing

In order to verify that the encoder and motor drivers work as expected, the following test was devised:

1. Use the motor driver to rotate the wheels backwards and forwards at different speeds.
2. Sample the accumulated encoder pulses whilst the wheels are rotating.
3. Send the accumulated encoder pulses over USART in order to plot the results.

As can be seen in 10, each wheel was rotated forwards and backwards at increasing speeds. Under ideal circumstances one would expect the accumulated encoder ticks at the last sample (time = 30000) to be exactly zero. This is (almost) the case for the right wheel, however, the left wheel encoder has drifted ~ 400 ticks, which is equivalent to approximately half of a rotation. The drift in encoder measurements is most likely a result of hardware inaccuracies of the motors, making the motors run at different speeds although the same PWM signal is applied.

7.2 Speed estimator testing

Verifying that the speed estimator was working as intended was done by:

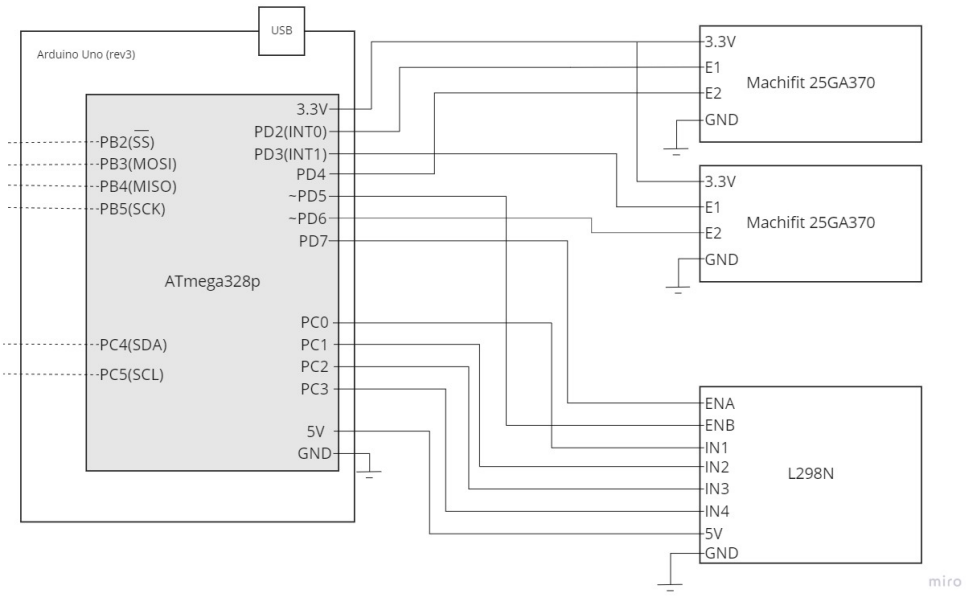


Figure 9: Schematic for test setup

1. Using the motor driver to rotate a wheel at different speeds.
2. Sample estimated rotational velocity.
3. Send the results over USART for plotting.

Whilst sampling the estimated rotational velocity of the wheels, the actual rotational velocity of the wheels were captured by recording the wheels using an Iphone 7. Using a program called *Tracker*, one can analyze the footage frame-by-frame to compute a *ground truth* for the rotational velocity of the wheel. For more details refer to Appendix E.

In figure 11 the rotational velocity estimates of the left wheel where sampled whilst using the motor driver function `motor_left(float d)` for `d=100`, `d=50` and `d=10`. The variable `d` is a variable for controlling the duty cycle of the PWM signal to the motor as well as direction control, i.e., `d=100` represents full throttle in the forwards direction, and `d=-100` is full throttle backwards.

In figure 12 the tracked rotational velocities of the left wheel computed by Tracker can be seen. The average rotational velocities of the wheel (disregarding acceleration from $0°/s$) are $\sim 561°/s$, $\sim 350°/s$, and $\sim 102°/s$ for figure 12a, 12b, 12c, respectively. Be aware that the results seen in figure 11 are given in rad/s whilst, in figure 12 they are given in $°/s$.

Comparing figure 11 and 12, the estimates of the rotational velocity are smoother at higher rotational velocities, however, do not appear to be more accurate such as expected following the explanation in section 6.7. At the higher velocities, the estimates deviate from the tracked velocities by $\sim 20°/s$. At low speeds the rotational velocity deviates less than $10°/s$ from the tracked velocity. The discrepancy in the accuracy of the estimates between high and low speeds can be due to inaccuracies in tracking the wheel caused by motion blur.

7.3 Speed controller testing

For verifying that the implemented speed controller works as intended the robot's wheels were actuated with a series of step responses. Estimated wheel speed, and control action were sent over USART for plotting. The resulting plots can be seen in figure 13.

The response obtained in figure 13 was obtained with the controller parameters, $K_p = 10$, $K_i = 50$ and $K_d = 0$. As can be seen from the figure the controller is able to track a setpoint with reasonable

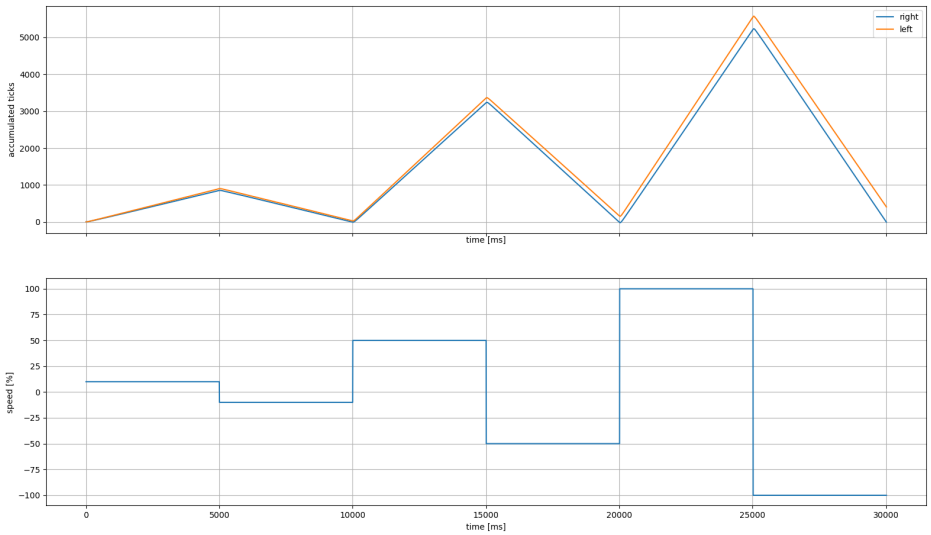
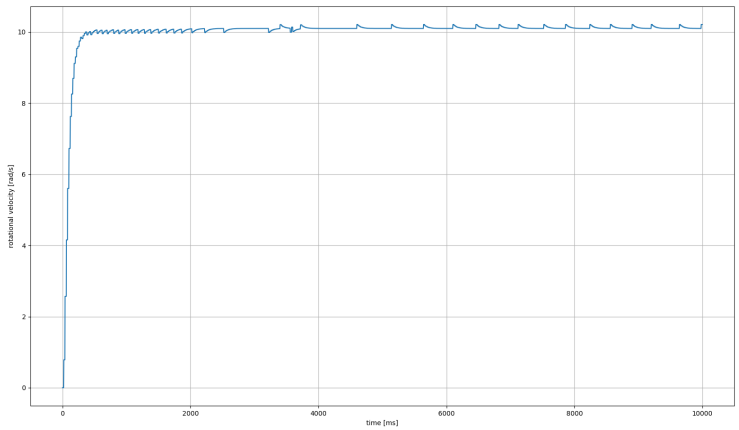


Figure 10: Encoder and motor driver test

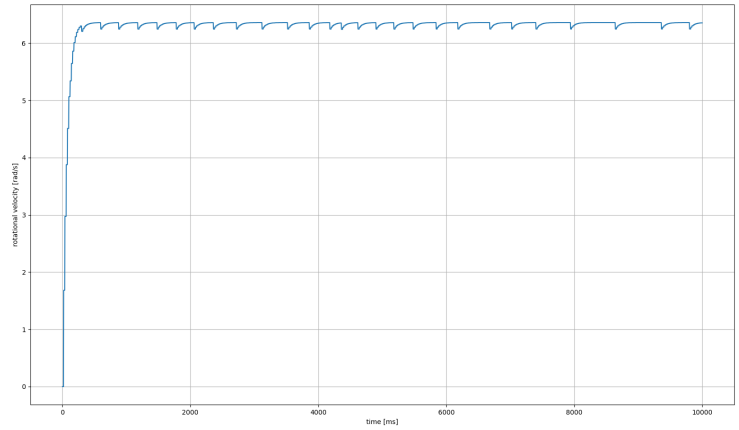
accuracy and speed. Tuning the controller was done experimentally by initially following the prosedure of Ziegler Nichols method by setting $K_i = K_d = 0$, and increasing K_p until the wheels would oscillate [2, p. 348]. K_p was then lowered slightly, and K_i was increased until a satisfactory response was obtained.

In order to verify the necessity of the anti-windup scheme a large reference was initially set in order to saturate the motors, making the integral error accumuluate. Then a new reference was set below the saturation limit of the motors. Estimated wheel speed, and control action were again sent over USART for plotting.

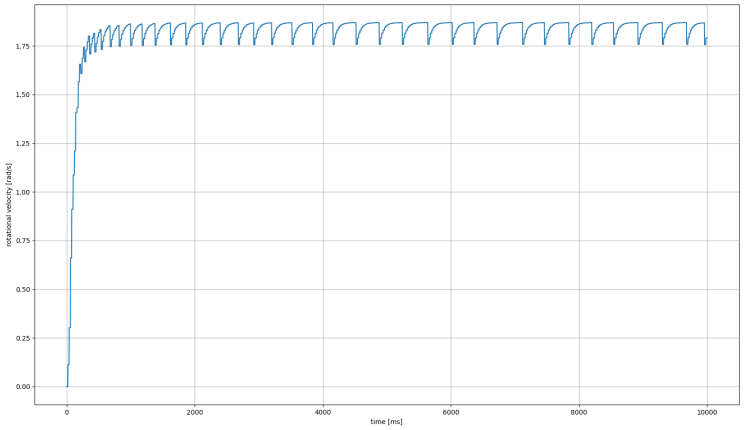
In figure 14 the anti-windup scheme is enabled, whilst in figure 15 it is disabled. Without the anti-windup scheme the rotational velocity will overshoot the new reference given at $t = 5s$ due to the large integral error having to unwind.



(a) Rotational velocity estimate - `motor_left(100)`

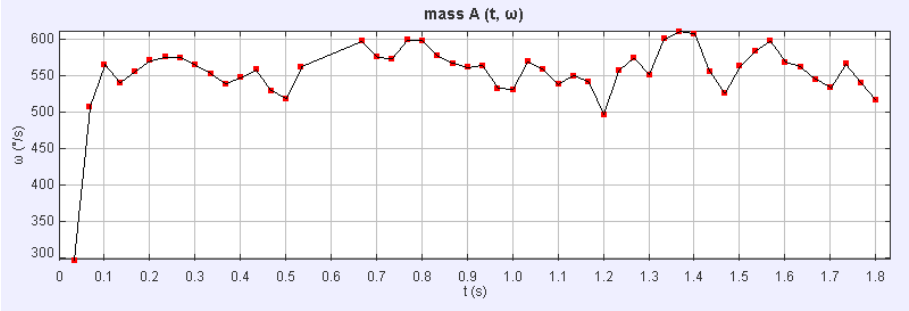


(b) Rotational velocity estimate - `motor_left(50)`

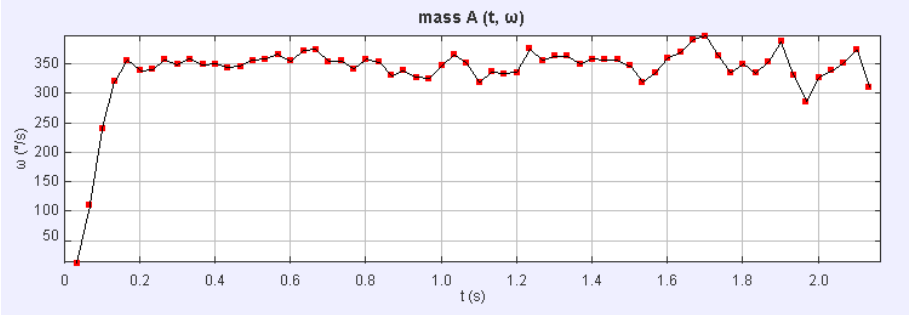


(c) Rotational velocity estimate - `motor_left(10)`

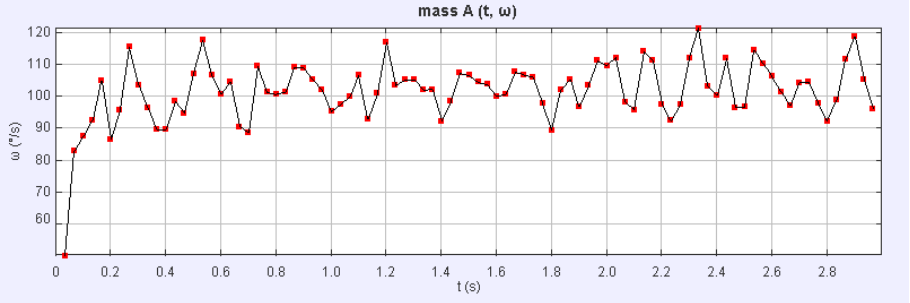
Figure 11: Rotational velocity estimates for different motor speeds



(a) Tracked rotational velocity - `motor_left(100)`



(b) Tracked rotational velocity - `motor_left(50)`



(c) Tracked rotational velocity - `motor_left(10)`

Figure 12: Tracked rotational velocities for different motor speeds

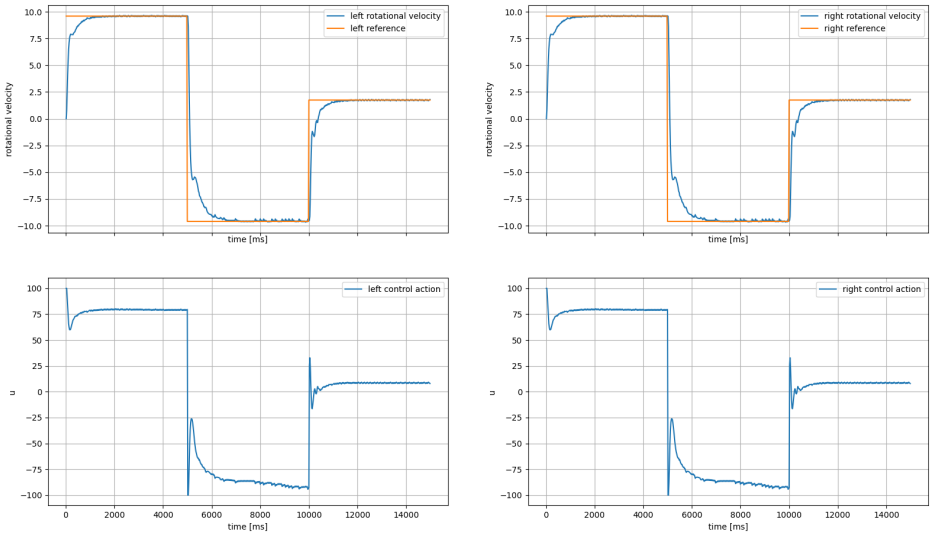


Figure 13: Step response

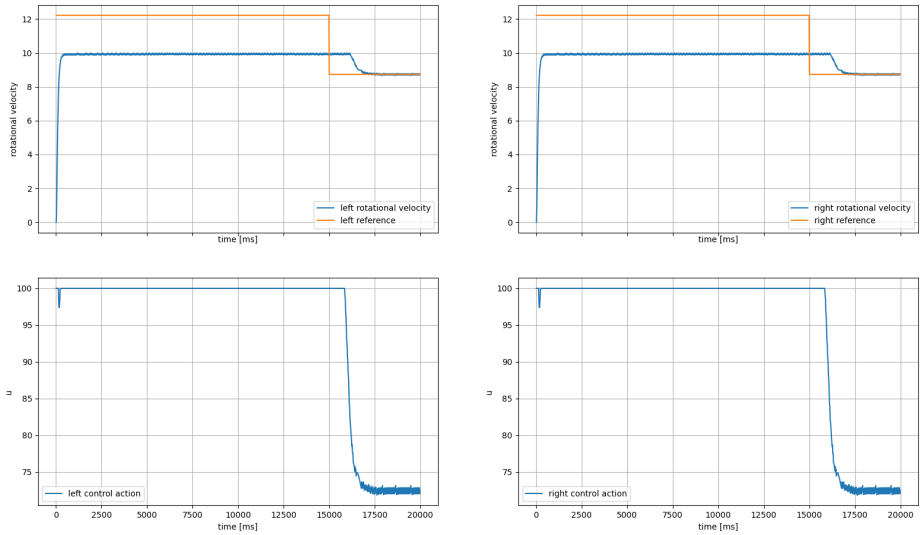


Figure 14: Anti-windup enabled

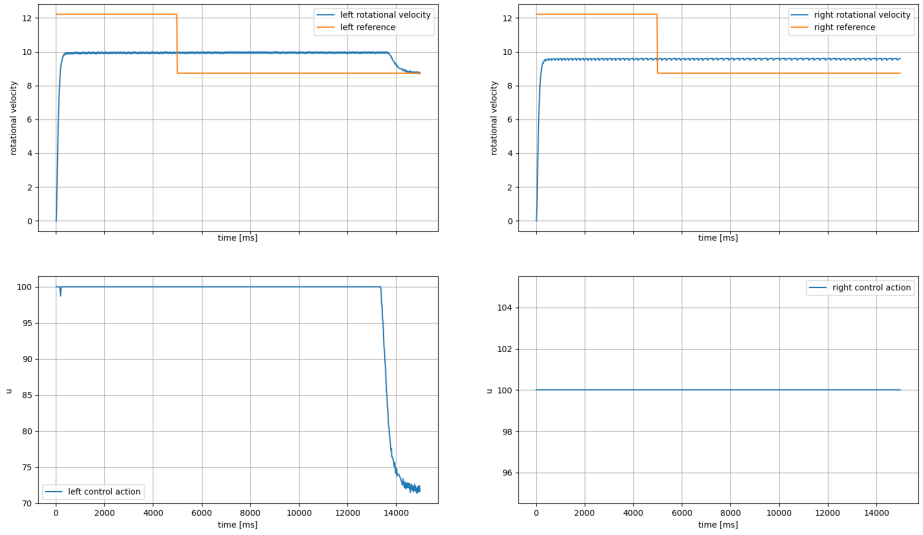


Figure 15: Anti-windup disabled

8 Further work

Further work on the project of creating the speed controller card should have its main focus on assembling the work done in parallel with this project. Specifically, design and assembly of hardware by Skinstad, and communication with nRF52840 by Buø.

Bibliography

[1] Arduino. *Arduino Uno Rev3*. URL: https://store.arduino.cc/products/arduino-uno-rev3 (visited on 10th Oct. 2021).

[2] Jens G. Balchen, Trond Andresen and Bjarne A. Foss. *Reguleringsteknikk, utgave 6*. Institutt for teknisk kybernetikk, 2016.

[3] Banggood. *Machifit 25GA370 DC 12V Micro Gear Reduction Encoder Motor with Mounting Bracket and Wheel - 110RPM*. URL: https://www.banggood.com/Machifit-25GA370-DC-12V-Micro-Gear-Reduction-Encoder-Motor-with-Mounting-Bracket-and-Wheel-p-1532242.html?utm_source=googleshopping&utm_medium=cpc_organic&gmcCountry=NO&utm_content=minha&utm_campaign=minha-no-en-pc&currency=NOK&cur_warehouse=CN&createTmp=1&ID=6157423&utm_source=googleshopping&utm_medium=cpc_union&utm_content=sandra&utm_campaign=sandra-ssc-no-all-0302&ad_id=337428064977&gclid=Cj0KCQjwnoqLBhD4ARIsAL5JedJP5LpKAdGG2ce0sGZKJwxHG02JDMGT-2iXCdv6jYSNI-es1Uh9RYQaAkKnEALw_wcB (visited on 10th Oct. 2021).

[4] Randal W. Beard and Timothy W. McLain. *Small Unmanned Aircraft: Theory and practice*. Princeton University Press, 2012.

[5] Microchip. *Servo control of a DC-brush motor*. URL: https://ww1.microchip.com/downloads/en/AppNotes/00532c.pdf (visited on 16th Nov. 2021).

[6] Microhip. *ATmega328P datasheet*. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (visited on 10th Oct. 2021).

[7] Handson Technology. *L298N Dual H-Bridge Motor Driver*. URL: http://www.handsontec.com/dataspecs/L298N%20Motor%20Driver.pdf (visited on 8th Nov. 2021).

Appendix

A Code documentation

A Encoder driver

Initialize the encoder driver.

```
void encoder_init(void);
```

Read the value of an input pin. `encoder_pin` is the desired pin to read. This function is used internally in `encoder_get_accumulated_ticks_left()` and `encoder_get_accumulated_ticks_right()`. Returns 1 (HIGH), or 0 (LOW).

```
int encoder_read_tick(int encoder_pin);
```

Returns the accumulated number of ticks from the left wheel encoder. Forward rotation of the wheel will increment the accumulated number of ticks. Backward rotation of the wheel will decrement the accumulated number of ticks.

```
long encoder_get_accumulated_ticks_left(void);
```

Returns the accumulated number of ticks from the right wheel encoder. Forward rotation of the wheel will increment the accumulated number of ticks. Backward rotation of the wheel will decrement the accumulated number of ticks.

```
long encoder_get_accumulated_ticks_right(void);
```

B PWM driver

The functions defined for the PWM driver are only used internally in the motor driver module.

Initialize PWM driver.

```
void pwm_init(void);
```

Set a PWM signal to drive the left wheel. `duty` is a float representing the duty cycle of the PWM signal (0-100).

```
void pwm_set_duty_cycle_left(float duty);
```

Set a PWM signal to drive the right wheel. `duty` is a float representing the duty cycle of the PWM signal (0-100).

```
void pwm_set_duty_cycle_right(float duty);
```

C Motor driver

Initialize motor driver.

```
void motor_init(void);
```

Set speed and direction of left motor. `duty` is a float between -100.0 and 100.0.

```
void motor_left(float duty);
```

Set speed and direction of right motor. `duty` is a float between -100.0 and 100.0.

```
void motor_right(float duty);
```

Stop left motor.

```
void motor_left_stop(void);
```

Stop right motor.

```
void motor_right_stop(void);
```

D Timer driver

Initialize general purpose timer. `timeout_ms` is the number of milliseconds before the timer times out.

```
void timer_init(unsigned int timeout_ms);
```

Check if the timer has timed out. Returns 1(true) if the timer has timed out, else 0(false).

```
unsigned int timer_timeout(void);
```

Reset after the timer timed out.

```
void timer_reset(void);
```

Returns time in milliseconds since `timer_init` was called. Resetting after the timer timed out will not reset the value returned from this function.

```
unsigned long timer_get_elapsed_ms(void);
```

E Speed estimator

Initialize speed estimator. `ticks_per_rot_left` and `ticks_per_rot_right` should be the number of ticks measured by manually turning the appropriate wheel a full rotation.

```
void speed_estimator_init(long ticks_per_rot_left, long ticks_per_rot_right);
```

Returns estimated rotational velocity of left wheel.

```
float speed_estimator_left_rad_per_s();
```

Returns estimated rotational velocity of right wheel.

```
float speed_estimator_right_rad_per_s();
```

F PID controller

Structure for holding PID controller parameters, and internal variables.

```
typedef struct PID_controller;
```

Set controller parameters. `*pid` is a pointer to a `PID_controller`, `P` is proportional gain, `I` is integral gain, `D` is derivative gain, `loop_period` is the period of the control loop.

```
void PID_controller_set_parameters(PID_controller *pid,
float P, float I, float D, float loop_period);
```

Returns a control action to be fed as input to the motor driver.

```
float PID_controller_get_control_action(PID_controller *pid, float error);
```

G Example application usage

```
#import <avr/io.h>
#import <stdio.h>
#include "drivers/usart/usart.h"
#include "drivers/timer/timer.h"
#include "drivers/motor/motor.h"
#include "speed_estimator/speed_estimator.h"
#include "PID_controller/PID_controller.h"

int main(void) {

    usart_init(UBRR);
    speed_estimator_init(850, 850);
    motor_init();

    float T = 0.02 // Control loop period [s]

    PID_controller left_motor;
    PID_controller right_motor;
    PID_controller_init(&left_motor);
    PID_controller_init(&right_motor);
    PID_controller_set_parameters(&left_motor, 10.0, 50.0, 0.0, T);
        PID_controller_set_parameters(&right_motor, 10.0, 50.0, 0.0, T);

        float left_error_rps; // left wheel speed error [rad/s]
        float left_speed_rps; // left wheel speed [rad/s]
        float left_speed_ref_rps = DEG2RAD*500; // left wheel reference speed [rad/s]
        float left_u; // left wheel control action

        float right_error_rps; // right wheel error [rad/s]
        float right_speed_rps; // right wheel speed [rad/s]
        float right_speed_ref_rps = DEG2RAD*500; // right wheel reference speed [rad/
        float right_u; // right wheel control action

    unsigned long time = 0; // keeps track of time

    timer_init(T*1000);

    int send_flag = 0; // Flag used to schedule sending of measurements over USART

    while(1) {
        // wheel speed control loop
                if (timer_timeout()) {
                        timer_reset();

                        left_speed_rps = speed_estimator_left_rad_per_s();
                        left_error_rps = left_speed_ref_rps - left_speed_rps;
                        left_u = PID_controller_get_control_action(&left_motor, left_e
                        motor_left(left_u);


                        right_speed_rps = speed_estimator_right_rad_per_s();
                        right_error_rps = right_speed_ref_rps - right_speed_rps;
                        right_u = PID_controller_get_control_action(&right_motor, rig
                        motor_right(right_u);

                        time = timer_get_elapsed_time();
```

```
                send_flag = 1;
            }
            if (send_flag) {
                send_flag = 0;
                // Send measurements over USART for plotting
                printf("%.2f", left_speed_rps);
            }

            // Here we can communicate with the nRF52840.
            // Send speed and odometry measurements
            // Receive new reference speeds, and parameters.
    }

    return 0;
}
```

B Getting started with further development

The source code for the project is available on GitHub from: https://github.com/andersenthomas98/TTK8.

A Microchip Studio

1. Download Microchip Studio from: https://www.microchip.com/en-us/development-tools-tools-and-software/microchip-studio-for-avr-and-sam-devices. During development of this project I have been using version 7.0.2542 with the installed package *Atmel Kits* (7.0.132).
2. In the Microchip Studio start-up view, click *Open Project...*
3. Open the *application.atsln* file from the source code downloaded from the GitHub repository above.

B Flashing source code to the Atmega328p

1. Plug the Atmel ICE and Arduino Uno into your computer with a USB cable.
2. Connect the ISP header from the port marked with AVR on the Atmel ICE to the Arduino Uno as seen in figure 16.
3. In the application view in Microchip Studio choose the configuration as seen in figure 17.
4. Flash the source code by pressing the green arrow in Microchip Studio (or by using the keyboard shortcut Ctrl+Alt+F5).

Figure 16: Atmel ICE ISP connection to Arduino Uno

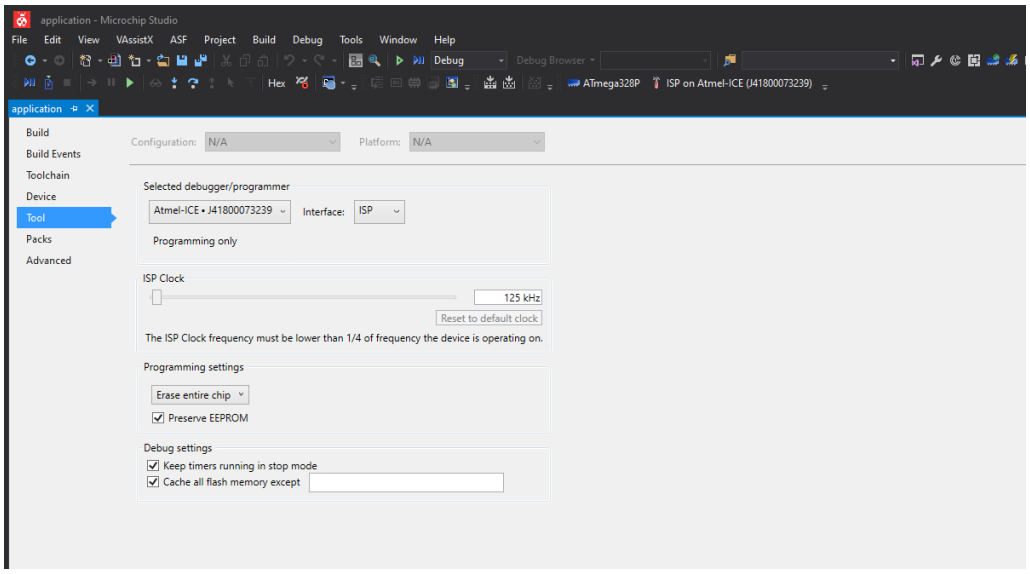


Figure 17: Application view configuration

C Pin layout

The current version of the software uses the pin layout given in table 3 for reading encoder pulses, writing PWM and control signals to the L298N motor driver.

D Encoder calibration

In order to calculate speeds accurately one has to know the number of pulses per wheel revolution. A program which will read out the accumulated encoder ticks was therefore made in order to calibrate the encoders. Before running the main application, run the calibrate-encoder program located in the source file directory, and open a terminal on your computer for serial communication (e.g one could use PuTTY). The program will write the accumulated encoder ticks over UART, the user must then manually rotate each wheel one revolution. After writing down the number of encoder pulses per wheel rotation, update the variables `TICKS_PER_ROT_LEFT` and `TICKS_PER_ROT_RIGHT`

	ATmega328P pin	Arduino Uno pin	Info
Left encoder channel A	PD2(INT0)	2	
Left encoder channel B	PD4	4	
Right encoder channel A	PD3(INT1)	3	
Right encoder channel B	PD7	7	
Left PWM-signal	PD6	6	
Right PWM-signal	PD5	5	
Left direction control 0	PC0	A0	
Left direction control 1	PC1	A1	
Right direction control 0	PC2	A2	
Right direction control 1	PC3	A3	

Table 3: Pin layout

corresponding to the left and right wheel respectively, in the defines.h file.

E Tracker

In order to find a ground truth to compare rotational velocity with the estimates from the speed estimator a program called Tracker was used. A screenshot of the Tracker program can be seen in figure 18.

1. Download Tracker from here: https://physlets.org/tracker/
2. Upload a video to Tracker by pressing Video → Import and browse for the video file.
3. To calibrate the software, create a calibration stick and drag it to a known distance in the camera view and specify the real distance.
4. Press Track → New → Point Mass, and hold shift while clicking the point you want to track.
5. Once you have clicked, the software will present you the next frame, and you can follow the same procedure as the point above.

The Tracker projects used in this project can be found in the folder called Tracker.

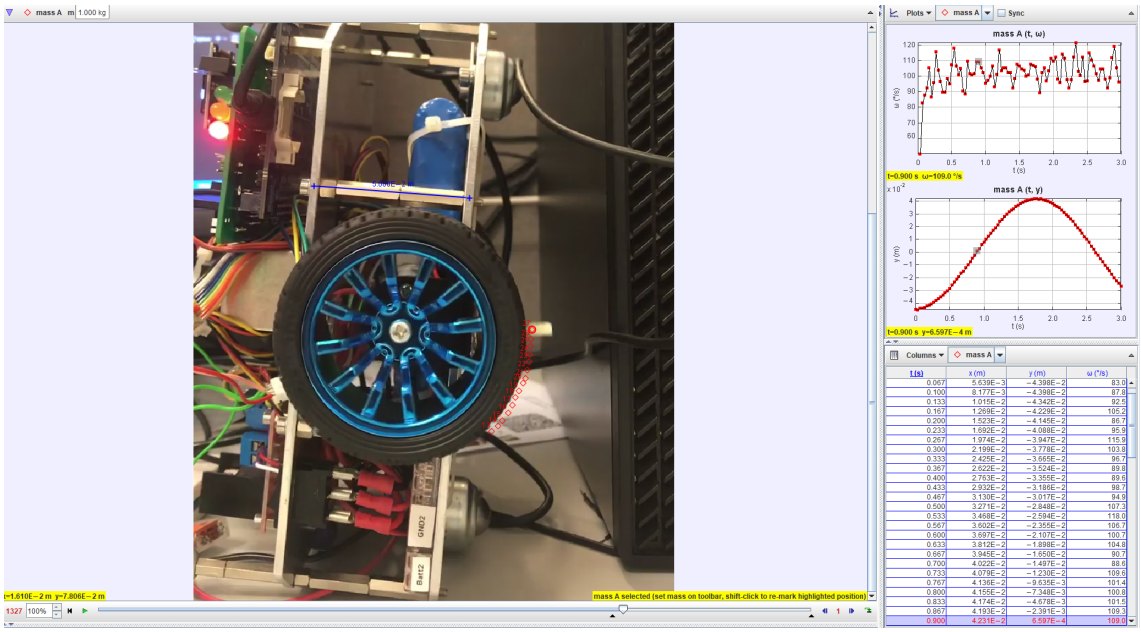


Figure 18: Compute rotational velocity of wheel using Tracker