1 Antiblockiersystem

1.1 Antiblockiersystem: Michael Wörner (30137), Christian Silfang (30147)

1.2 Aufgabenstellung und -ziel

Die Aufgabenstellung im Versuch *Antiblockiersystem* sieht es vor, eine automatische Bremsung eines Fahrradreifens zu implementieren, wobei das Rad möglichst nicht blockieren soll. Dabei muss der Antrieb des Rads vor dem Bremsvorgang ebenfalls selbst erzeugt werden.

1.3 Versuchsaufbau

Der Versuch besteht aus einem Fahrradreifen (1) welcher mit einer Vordergabel (2) auf einem Steg besfestigt ist. Um ein Aufschwingen des Rades zu verhindern wurde zusätzlich ein Gewicht (3) angebracht. Um das Rad zu Beschleunigen wird eine Antriebswelle (4) verwendet welche durch einen Motor (Brushless-Outrunner) (5) mittels Treibriemen angetrieben wird. Die Ansteuerung wird mit Hilfe eines ChipKit Uno 32 (6) realisiert.

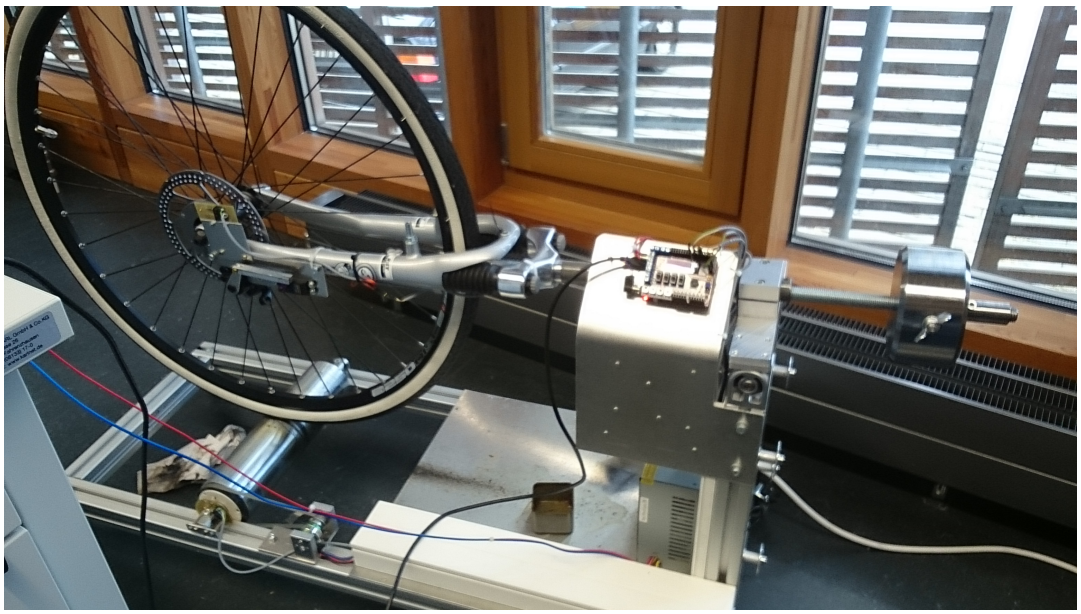

Abbildung 1.1: „Beweisfoto“ des Versuchs Antiblockiersystem

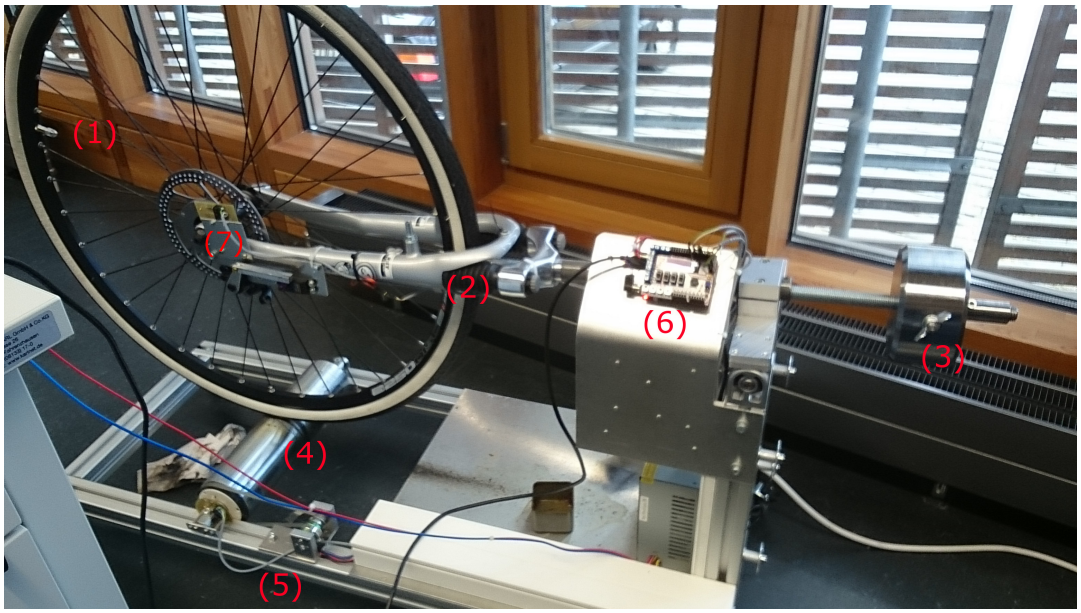

Abbildung 1.2: Aufbau des Antiblockiersystems mit einem Fahrradreifen

Die Bremse wird mittels eines Servos (7) angezogen.

Der Versuchsaufbau ist in Abb. 1.2 mit Markierungen dargestellt.

1.4 Modellierung

Dieser Abschnitt beschreibt die Ideen und theoretische Modellierungsschritte, die zur Implementierung des Versuchs nötig waren.

Die Grundidee basiert auf dem Prinzip, dass ab einer Zeit kontinuierlich gebremst wird. Die Bremse soll dann jedoch gelöst werden, wenn während dem Bremsen das Rad blockiert. Im Anschluss soll sofort wieder gebremst werden.

1.4.1 Beschleunigung der Antriebswelle

Das Anfahren des Rades wird über eine Puls-Breiten-Modulation erreicht. Dabei wird schrittweise die Pulsbreite erhöht. Listing 1.1 zeigt die exakte Implementierung der Motor-Beschleunigung.

1.4.2 Detektion eines blockierenden Rades

Im Versuchsaufbau haben Fahrradreifen und Antriebswelle ein Größenverhältnis von $\frac{1}{8}$. Das zieht den Schluss nach sich, dass das Rad blockiert, wenn gilt:

$$\text{WELLEN\_INTERRUPT} \quad > \quad 8 \tag{1.1}$$

Wobei WELLEN_INTERRUPT in der Implementierung (siehe Listing 1.2) eine Zählervariable für die Anzahl der Umdrehungen der Antriebswelle ist, die in einer zweiten Funktion bei jeder Umdrehung der Welle inkrementiert wird. Dadurch wird ein Referenzwert zum Vergleich generiert.

1.4.3 Regelung der Bremse

Das Regeln der Bremse wird über ein einfaches Toggling gesteuert. Um dies zu realisieren ist die vorherige Detektion eines blockierenden Rades unabdingbar. Über den aktuellen Zustand in welchem sich das Rad momentan befindet kann entschieden werden ob die Bremse geöffnet bzw. geschlossen werden muss. Das Listing 1.3 zeigt die Implementierung der Regelung der Bremse.

1.5 Implementierung

Dieser Abschnitt beschreibt die Implementierung der einzelnen Teile des gesamten Codes anhand der Vorüberlegungen, die bereits in 1.4 näher beschrieben wurden.

1.5.1 Setup des Boards

Pinbelegung:

Bremse PWM	4
Motor PWM	5

Tabelle 1.1: Pinbelegung des ChipKit Uno32

Interrupts:

Welle INTERRUPT PIN	7
Rad INTERRUPT PIN	8
Welle INTERRUPT NUMMER	2
Rad INTERRUPT NUMMER	3

Tabelle 1.2: Interrupts des ChipKit Uno32

1.5.2 Beschleunigung der Antriebswelle

Die Beschleunigung der Antriebswelle ist mittels einer `for`-Schleife implementiert. Dabei wird der Funktion `MotorBeschleunigung()` (Listing 1.1) als Parameter ein ganzzahliger Wert übergeben. Dieser Wert entspricht dem Wert der maximalen Beschleunigung die erreicht werden soll und ist die obere Grenze der `for`-Schleife.

Die Werte für den Antrieb selbst werden durch ein `analogWrite()` auf dem definierten PIN gesetzt.

```
1 void MotorBeschleunigung( uint8_t VMAX ){
2   for(int i = VMIN; i <= VMAX; i++){
3     analogWrite( MOTOR_PIN_PWM, i );
4     delay(100);
5   }
6 }
```

Listing 1.1: Beschleunigung der Antriebswelle

1.5.3 Detektion eines blockierenden Rades

Um das Blockieren eines Rades erkennen zu können wurden im Grunde zwei Funktionen benötigt (Listing 1.2).

Über die Funktion `WelleInterruptZaehler()` wird die Anzahl der Wellen-Umdrehungen gezählt. Dadurch werden lediglich die Interrupts einer Umdrehung gezählt und bei Eintreten dieser Interrupts die Variable `WELLE_INTTERUPT_ZAEHLER` erhöht.

Die zweite Funktion `RadInterruptHandler()` vergleicht letztlich die Anzahl der Umdrehungen der mitdrehenden Welle und die Umdrehungen des Rades. Dabei wird das Rad-Wellen-Verhältnis als Vergleichsreferenz verwendet um zu prüfen ob es mehr Umdrehungen der Welle gibt, wie der Werte des Rad-Wellen-Verhältnis groß ist.

Ist dies der Fall, wird der aktuelle Wert zur Bremsung so gesetzt, dass die Bremse geöffnet wird, das ein Blockieren stattfinden muss.

Wird das Rad-Wellen-Verhältnis jedoch nicht überschritten wird der aktuelle Bremswert so gesetzt, dass eine Bremsung stattfindet, da das Rad momentan nicht blockiert.

```
1  void WelleInterruptHandler(){
2    WELLE_INTERRUPT_ZAEHLER++;
3  }
4
5  void RadInterruptHandler(){
6    if( WELLE_INTERRUPT_ZAEHLER > RAD_WELLEN_VERHAELTNIS )
7      AKTUELLER_ABS_WERT = BREMSE_AUF;
8    else
9      AKTUELLER_ABS_WERT = BREMSE_ZU;
10   WELLE_INTERRUPT_ZAEHLER = 0;
11 }
```

Listing 1.2: Detektion eines blockierenden Rades

1.5.4 Regelung der Bremse

Die Regelung des Bremsvorgangs ist mittels der Funktion `bremseServoRegelung()` (Listing 1.3) realisiert. Dabei wird über das Toggling einer boolschen Variable abwechselnd ein Bremsen ausgelöst bzw. gestoppt. Die Zeit, in welcher diese Aktionen ausgeführt werden müssen, wird dabei mit der Hilfe der aktuellen Systemzeit + `CORE_TICK_RATE` gesetzt. Da für das Auslösen der Bremse eine bestimmte Pulsbreite benötigt wird, muss der jeweilige Wert berücksichtigt werden.

Der maximale Wert der Pulsbreite kann dabei 2ms betragen, minimal jedoch 1ms. Das Bremsen wird bei 1.5ms ausgelöst. Bei 1.75ms wird die Bremse nicht betätigt, bzw. falls diese verschlossen wäre, geöffnet.

```
1  uint32_t bremseServoRegelung( uint32_t time ){
2    static bool b = false;
3    b = !b;
4    if( b ){
5      digitalWrite( BREMSE_PIN_PWM, HIGH );
6      return time + CORE_TICK_RATE / 1 * AKTUELLER_ABS_WERT;
7    }
8    else{
9      digitalWrite( BREMSE_PIN_PWM, LOW );
10     return time + CORE_TICK_RATE / 1 * ( MAX_ABS -
         AKTUELLER_ABS_WERT );
11   }
12 }
```

Listing 1.3: Regelung der Bremse mit ABS

1.6 Quellcode

```
/*
**     Programmierung eingebetteter Systeme
**     ABS-Steuerung
**     M.Woerner (30137), C.Silfang (30147)
*/
const   float MAX_ABS = 2.0; // 2ms
const   float BREMSE_AUF = 1.75; // 1,75ms
const   float BREMSE_ZU = 1.5; // 1,5ms
  float AKTUELLER_ABS_WERT = 1.75; // Bremse am Anfang auf

// Pins
const int BREMSE_PIN_PWM = 4;
const int MOTOR_PIN_PWM = 5;

const int WELLE_INTERRUPT_PIN = 7;
const int RAD_INTERRUPT_PIN =8;

const int WELLE_INTERRUPT_NUMMER = 2;
const int RAD_INTERRUPT_NUMMER = 3;

// Geschwindigkeitsvorgaben
const int VMIN = 100; // Mindestgeschwindigkeit
const int VMAX = 200; // Hoechstgeschwindigkeit

// Zaehler
static int WELLE_INTERRUPT_ZAEHLER = 0;

// Rad-/Wellenverhaeltnis 1:8
// > Welle dreht sich 8 mal schneller als Rad
const int RAD_WELLEN_VERHAELTNIS = 8;
```

```
31
32 // Setup
33 void setup() {
34
35   // Pins setzen
36   pinMode( WELLE_INTERRUPT_PIN, INPUT );
37   pinMode( RAD_INTERRUPT_PIN, INPUT );
38   pinMode( BREMSE_PIN_PWM, OUTPUT );
39
40   Serial.begin(9600);
41
42   /*
43   ** Motor Beschleunigen: Start: VMIN - VMAX
44   ** anschliesend Beschleunigen beenden
45   */
46   MotorBeschleunigung( VMAX );
47   delay( 5000 );
48   analogWrite( MOTOR_PIN_PWM, 0 );
49   delay( 100 );
50
51   // Interrupt Handling
52   attachInterrupt( WELLE_INTERRUPT_NUMMER, WelleInterruptHandler,
         RISING );// Wellenumdrehungen zaehlen
53   attachInterrupt( RAD_INTERRUPT_NUMMER, RadInterruptHandler,
        RISING );// Check nach Radumdrehung
54
55   // Brems-Servo Regelung
56   attachCoreTimerService( bremseServoRegelung );
57 }
58
59 // LOOP
```

```
60 void loop() {
61
62 }
63
64 /*
65 ** Funktion um Motor zu Beschleunigen.
66 ** Die Beschleunigung beginnt bei VMIN
67 ** und wird bei einer maximalen Geschwindigkeit *
68 ** von VMAX beendet.
69 */
70 void MotorBeschleunigung( uint8_t VMAX ){
71   for(int i = VMIN; i <= VMAX; i++){
72     analogWrite( MOTOR_PIN_PWM, i );
73     delay(100);
74   }
75 }
76
77 /*
78 ** Funktion um auf Wellenimpuls zu reagieren.
79 ** Dazu wird die Zaehlervariable inkrementiert
80 ** und somit die Umdrehung der Welle gezaehlt.
81 */
82 void WelleInterruptHandler(){
83   WELLE_INTERRUPT_ZAEHLER++;
84 }
85
86 /*
87 ** Funktion um auf Radimpuls zu reagieren.
88 ** Dazu wird auf Rad-Blockierungen reagiert:
89 **   - 1:8 Bremse zu, da Rad blockieren muss da das Verhaeltnis
    unterschritten wird
```

```
90 **  - 9 > Bremse auf, da Umdrehungen groesser wie das Verhaeltnis
91 */
92 void RadInterruptHandler(){
93   if( WELLE_INTERRUPT_ZAEHLER > RAD_WELLEN_VERHAELTNIS )
94     AKTUELLER_ABS_WERT = BREMSE_AUF; // Rad blockiert nicht
95   else
96     AKTUELLER_ABS_WERT = BREMSE_ZU; // Rad blockiert
97   WELLE_INTERRUPT_ZAEHLER = 0;
98 }
99
100 /*
101 ** Funktion um den Servo fuer die Bremsung zu steuern.
102 ** Steuerung wird ueber die Zeit mittels der CORE_TICK_RATE
103 ** realisiert.
104 ** Die Steuerung der HIGH-LOW-Phasen geschieht ueber toggling.
105 */
106 uint32_t bremseServoRegelung( uint32_t time ){
107   static bool b = false;
108   b = !b; // toogling
109   if( b ){ // HIGH -> Bremse zu
110     digitalWrite( BREMSE_PIN_PWM, HIGH );
111     return time + CORE_TICK_RATE / 1 * AKTUELLER_ABS_WERT;
112   }
113   else{ // LOW -> Bremse auf
114     digitalWrite( BREMSE_PIN_PWM, LOW );
115     return time + CORE_TICK_RATE / 1 * ( MAX_ABS -
          AKTUELLER_ABS_WERT );
116   }
117 }
```