## Scope 1:

- **Schrittmotoransteuerung mit kontinuierlicher Frequenz in eine Richtung**

## Scope 2:

- **Aufzeichnen eines Bewegungsablaufs**
- **Mittels zwei Tastern wird eine Drehrichtung vorgegeben**
- **Mit einem Taster kann die Bewegung gestoppt werden**
- **Mit einem Taster wird die Sequenz abgespielt**

## Scope 3:

- **lineare Veränderung der Schrittfrequenz**
- **Mit zwei Tastern wird die Frequenz erhöht bzw. verringert**
- **0% - 100% und umgekehrt in ca. 2 Sekunden**
- **Wird 0% erreicht ändert sich die Drehrichtung und die Frequenz erhöht sich**
- **Position und Frequenz wird aufs Display ausgegeben**


# Schrittmotoransteuerung mit dem TMC2209-Treiber

**Dokumentation zum HWE-Softwareprojekt „Schrittmotoransteuerung“.**

**Verwendete Hardware:**

**Megacard**
**+**
**TMC2209-Stepper-Driver**
**+**
**17HS19-2004S1 2-Phase-Stepper**
**+**
**2-Line-LCD mit HD44780-Controller & PCF8574 – I2C I/O-Expander**

# Inhalt

# Abbildungsverzeichnis

# Hardware:

## TMC2209:

Der TMC2209 ist ein Treiber-Board für 2-Phasen-Schrittmotoren, basierend auf dem TMC2209 IC.

Features und Leistungen:

- Die Steuerung kann über eine UART-Schnittstelle oder über den Richtungs- und Schritt-Pin erfolgen.
- Bei jeder steigenden Flanke am Schritt-Pin wird ein Schritt durchgeführt (diese Methode wird verwendet).
- Die Drehrichtung des Motors kann über den Direktions-Pin bestimmt werden.
- Das Board benötigt zwei Spannungsversorgungen:
  - 5 Volt für den Chip
  - 4,78 bis 28 Volt für den Motor

Leistung:

- Kontinuierliche I-Phase = 1,4 ARMS möglich
- I-Phase bis zu 2,5 A Spitzenleistung für kurze Zeit möglich

## Ansteuerung:

### *Normalmodus:*

In dieser Anwendung wird der Normalmodus verwendet. Dabei wird der Motor über den Step-Pin betrieben, wobei jede steigende Flanke am Step-Pin einen Mikroschritt des Motors bedeutet. Die Größe dieser Mikroschritte kann mithilfe der MS-Pins wie folgt definiert werden:

| CFG2/MS2 | CFG1/MS1 | Steps | Interpolation |
|---|---|---|---|
| GND | GND | 1/8 | 1/256 |
| GND | VIO | 1/32 | 1/256 |
| VIO | GND | 1/64 | 1/256 |
| VIO | VIO | d1/16 | 1/256 |

*Abbildung 1: TMC2209 Mikroschritt-Einstellungen*

### *UART:*

Für die UART-Ansteuerung wird ein Ansteuerungsprogramm, wie zum Beispiel „ScriptCommunicators“, benötigt. Wird in einem solchen Programm der TMC2209 ausgewählt, können diverse Einstellungen, wie Frequenzanpassung oder Stromlimitierung, vorgenommen werden.

Für die unidirektionale UART-Kommunikation muss der Treiber wie folgt angeschlossen werden:

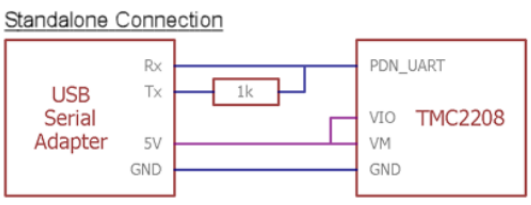


*Abbildung 3: TMC2209 Standalone UART-Connection*

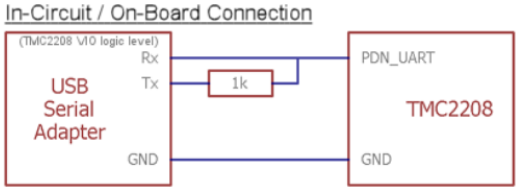


*Abbildung 2: TMC2209 In-Circuit UART-Connection*

## Pin-Out:

| Left | Signal | Right | Signal |
|---|---|---|---|
| 1 | GND | 9 | Dir |
| 2 | VIO | 10 | Step |
| 3 | M1B (Motor Phase B) | 11 | PDN |
| 4 | M1A (Motor Phase A) | 12 | UART |
| 5 | M2A (Motor Phase A) | 13 | SPRD |
| 6 | M2B (Motor Phase B) | 14 | MS2 |
| 7 | GND | 15 | MS1 |
| 8 | VM | 16 | EN |
| | | | |
| 17 | INDEX | 18 | DIAG |

*Abbildung 5: TMC2209 Pin-Out Tabelle*

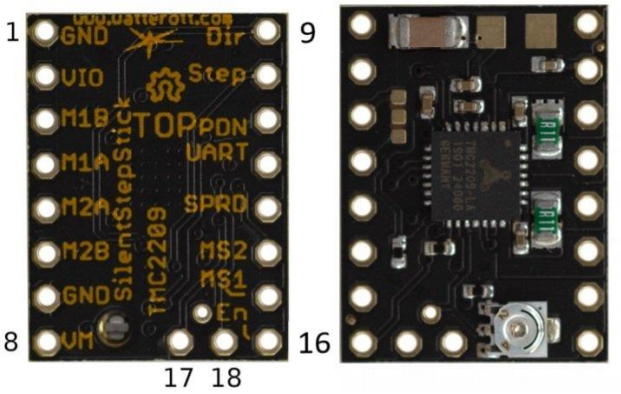


*Abbildung 4: TMC2209 Pin-Out Print*

## Schrittmotor:

Für die folgenden Aufgaben wird der 17HS19-2004S1 Schrittmotor verwendet. Es folgt eine kleine Übersicht wichtiger Eckdaten des Motors:

| SPECIFICATION / CONNECTION | BIPOLAR |
|---|---|
| AMPS/PHASE | 2.00 |
| RESISTANCE/PHASE(Ohms)@25°C | 1.40±10% |
| INDUCTANCE/PHASE(mH)@1KHz | 3.00±20% |
| HOLDING TORQUE(Nm)[lb-In] | 0.59[5.22] |
| STEP ANGLE(°) | 1.80 |
| STEP ACCURACY(NON-ACCUM) | ±5.00% |
| ROTOR INERTIA(g-cm²) | 82.00 |

*Abbildung 6: Stepper Spezifikationen*

## Anschlussdiagram:

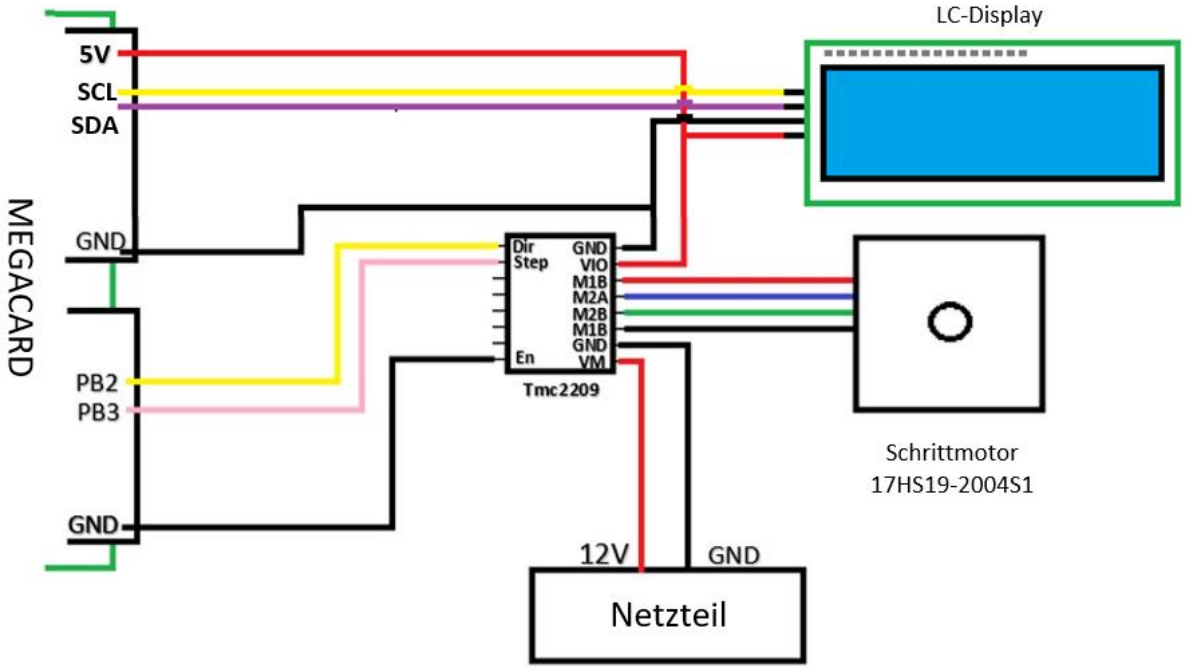


*Abbildung 7: Anschlussdiagramm*

# Scope 1

## Aufgabenstellung:

Im ersten Scope ist das zu erreichende Ziel ein sich mit konstanter Frequenz (1kHz) in eine Richtung drehender Schrittmotor.

## Theorie:

Um eine kontinuierliche Frequenz zu erreichen, wird der Timer0 des ATmega16 im Clear-Timer-on-Compare Modus verwendet. Durch die richtige Einstellung von Hardware- und Softwareteiler, sowie Compare-Wert im OCR0-Register wird probiert, so genau wie möglich die gewünschte Frequenz zu erreichen. Da eine Frequenz von 1000Hz nicht ohne Softwarevorteiler erreichbar ist, kann der PWM-Modus nicht verwendet werden und die Invertierung des Takt-Pins muss manuell in der ISR erfolgen. Aufgrund der manuellen Invertierung des Takt-Pins muss aber die ISR in der doppelten gewünschten Frequenz aufgerufen werden, um damit eine ganze Taktperiode mit steigender und fallender Flanke darzustellen.

Wird vom Hardwarevorteiler 1 ausgegangen und der OCR0-Wert auf 59 gesetzt, wird eine Interrupt-Frequenz von 200kHz erreicht, was ein Vielfaches der gewünschten Frequenz von 2kHz ist. Wird diese Frequenz durch 100 geteilt erreicht man die gewünschten 2kHz.

## Rechnung:

$$f_{ISR} = \frac{f_{sys}}{SW_{teiler} * HW_{teiler} * (OCR0 + 1)} = \frac{12MHz}{100 * 1 * (59 + 1)} = \frac{12000\cancel{000}}{6\cancel{000}} = \underline{2kHz}$$

*Abbildung 8: Scope 1 - Berechnung*

## Software:

### Timer-Initialisierung:

Die Konfigurierung des Timers geschieht hauptsätzlich durch das Setzen der richtigen Bits im TCCR0-Register und dem Festlegen des OCR0-Wertes. Anschließend muss nur noch das Interrupt, durch das Setzen des OCIE0-Bits im TIMSK-Register und dem Ausführen der sei() Methode, spezifisch und global freigegeben werden.

*TCCR0 (Timer/Counter Control Register):*

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | FOC0 | WGM00 | COM01 | COM00 | WGM01 | CS02 | CS01 | CS00 | TCCR0 |
| Read/Write | W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

*Abbildung 9: TCCR0-Register*

FOC0: 0 → Erzwungener Compare-Match abwählen

WGM00: 0
WGM01: 1 → CTC-WGM-Mode auswählen

COM01: 0
COM00: 1 → Normale Port Operation (Pin OC0 wird nicht beeinflusst)

CS02: 0
CS01: 0
CS00: 1 → Hardwarevorteiler auf 1 setzen

*TIMSK (Timer/Counter Interrupt Mask Register):*

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | OCIE2 | TOIE2 | TICIE1 | OCIE1A | OCIE1B | TOIE1 | OCIE0 | TOIE0 | TIMSK |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

*Abbildung 10: TIMSK-Register*

OCIE0: 1 → Timer Compare Interrupt Mask setzen (Interrupt spezifisch freigeben)

## Code:

Die essenziellen Code-Abschnitte dieser Realisierung beschränken sich auf die Initialisierungs- und Interrupt-Service-Routine des Timers. Die Gesamtimplementierung in einem funktionsfähigen C-Programm und dem dazugehörigen HEX-File ist unter Referenzen verlinkt.

### Timer-Initialisierung - Routine:

In der Timer-Initialisierung wird zusätzlich, neben den in Timer-Initialisierung erläuterten Definitionen, nur noch die Datenrichtung des Clock-Pins im Datenrichtungsregister als Ausgang definiert.

```
/*****************************************************************************/
/* Initialisiert Timer0 zur Clockausgabe an Pin PB3 (f=1kHz)                 */
void clockInit(){                           // 200kHz Interrupt Frequenz
        DDRB |= (1<<PB3);                   // Clock-Pin-Datenrichtung auf Ausgang
        OCR0 = 59;                          // CTC-Mode OCR0=59 mit 1 Vorteiler --> 200kHz
                                                Interruptfrequenz
        TCCR0 = (1<<CS00) | (1<<WGM01);     // Mode: CTC, Vorteiler: 1
        TIMSK = (1<<OCIE0);                 // OCIE0-Match interrupt spezifisch freigeben
        sei();                              // Interrupts global freigeben
}
```

*Abbildung 11: Scope 1 - Implementierung - Timer-Initialisierung*

### Timer-ISR:

In der ISR selbst wird bei jedem hundertsten Aufruf der Clock-Pin invertiert und somit die Frequenz am Pin PB3 erzeugt.

```
static int count = 0; // statische Zählvariable zur Softwareteilung der Frequenz

/*****************************************************************************/
/* Interrupt-Service-Routine für 1kHz Signal an Pin PB3                      */
ISR (TIMER0_COMP_vect){             // Timer Compare Match Interrupt-Routine
        count++;                    // Zählvariable inkrementieren
        if (count >= 100){          // Wurde Zählvariable 100-mal inkrementiert?
                PORTB ^= (1<<PB3);  // Pin PB3 Toggeln (Clock-Pin)
                count = 0;          // Zählvariable zurücksetzen
        }
}
```

*Abbildung 12: Scope 1 - Implementierung - Timer-ISR*

## Nachweis:

Als Nachweis zur erfolgreichen Frequenzerzeugung wurde die Frequenz mit dem Multimeter gemessen:


*Abbildung 13: Scope 1 - Frequenznachweis*

# Scope 2

Das Hauptaugenmerk des zweiten Scopes liegt darin, eine mit den Tastern abgespielte Frequenz zu speichern und mit dem Drücken eines weiteren Tasters, nach Rückkehr zur Ursprungsposition, wieder abzuspielen.

**Vorgenommene Taster-Implementierung:**
Taster 1 → Stopp
Taster 2 → Vorwärts
Taster 3 → Rückwärts
Taster 4 → Sequenz abspielen

## Theorie:

Die Herausforderung dieser Aufgabe besteht darin, zum einen die Schritte, die ein Motor, während er sich dreht zu zählen und zum anderen den Motor für eine Bestimmte Anzahl von Schritten wieder drehen zu lassen.
Zum Erreichen einer stabileren Frequenz, durch eine geringere Interrupt-Frequenz, wird eine Frequenz gewählt, welche gut ohne Softwareteiler realisierbar ist und in der Nähe von einem kHz liegt. Dadurch werden laufende Programmabläufe nicht ständig von der ISR unterbrochen und somit nicht verzögert.
Die Implementierung dieser Stepper-Routinen und Funktionen erfolgte in einer Header-File, um im Hauptprogramm ein gewisses Maß von Abstraktion zu wahren.

Der Grundsatz zur Implementierung des Timers ist in Scope 1 zu sehen. Es ändert sich hier lediglich die Frequenz wie die nachfolgende Rechnung zeigt:

$$f_{ISR} = \frac{f_{sys}}{HW_{teiler} * (OCR0 + 1)} = \frac{12MHz}{64 * (99 + 1)} = \frac{12000\cancel{00}}{64\cancel{00}} = \underline{1875Hz}$$

$$f_{CLKPIN} = \frac{f_{ISR}}{2} = \underline{937.5Hz}$$

*Abbildung 14: Scope 2 - Berechnung*

## Code:

Aufgrund der Abstrahierung im Hauptprogramm wird sowohl auf die Stepper-Bibliothek, in welcher hauptsächlich die Steuerung des Schrittmotors stattfindet, als auch auf das Hauptprogramm, in welchem die Sequenzverarbeitung stattfindet, eingegangen.

### Stepper-Bibliothek:

In der Stepper-Bibliothek befinden sich drei verschiedene „Funktionstypen“: die Timer-Initialisierung mit ISR, permanente Bewegung mit Schrittzählung und Bewegung mit vorgeschriebener Schrittzahl. Es wird jeweils auf ein markantes Element eingegangen, wenn mehrere Ähnlichkeiten besitzen (z.B. vorwärts und rückwärts bewegen). Eine detaillierte Code-Dokumentation ist im Source-Code durch die Kommentare gegeben.

Hinweis:

Aus Gründen der Implementierung bedeutet nur jeder 2. Schritt einen eigentlichen Schritt da die Anzahl der ISR-Aufrufe gezählt wird.

### *Timer0-ISR:*

In der ISR kann grob zwischen zwei Operationen entschieden werden. Ist die globale Variable nsteps gesetzt, so wird die Bewegung für eine bestimmte Anzahl an Schritten ausgeführt und die ISR stoppt den Timer nach der erfolgreichen Ausführung selbst. Ist die Variable nsteps nicht gesetzt so findet eine kontinuierliche Bewegung statt und jeder Schritt, bzw. jeder halbe Schritt, wird gezählt. Für Rückwärtsbewegungen wird dekrementiert und für Vorwärtsbewegung inkrementiert, somit ist jede Bewegung rückwärts negativ und vorwärts positiv.

Hinweis: Die Implementierung eines Softwareteilers wurde hier nur aus Portabilitätsgründen zur Verfügung gestellt, allerdings in der Anwendung nicht verwendet.

```
/*****************************************************************************/
/* Interrupt-Service-Routine der Signalerzeugung und Schrittzählung          */
/* Falls nsteps nicht 0 --> automatisch Schrittmodus                         */
/*                                                                           */
/* swScaler - Variable kann zur Frequenzänderung verwendet werden (nicht linear)  */
ISR (TIMER0_COMP_vect){      // Timer Compare Match Interrupt-Routine
       if (counter == swScaler){                  // Softwareteiler erreicht?
              if(nsteps != 0 && nsteps <= steps){ // Falls Betrieb mit vorgeschriebener
                                                  Schrittzahl und Schrittzahl erreicht
                     nsteps = 0;                  // vorgeschriebene Schrittzahl
                                                  zurücksetzen (Warteschleife in NSteps
                                                  Routinen wird abgebrochen)
                     TCCR0 = 0x00;                // Timer0 stoppen
              }else{
                     OUTPORT ^= (1<<CLOCKPIN);    // Clock-Pin Toggeln zur
                                                  Frequenzerzeugung
                     counter = 0;                 // Softwareteiler-Zählvariable
                                                  zurücksetzen
                     steps += stepsInkrementor;   // Schrittzähler mit stepsInkrementor in-
                                                  bzw. dekrementieren
              }
       }else{counter++;}                          // Softwareteiler-Zählvariable
                                                  inkrementieren
}
```

*Abbildung 15: Scope 2 - Implementierung - Timer-ISR*

### *Kontinuierliche Bewegung - Routine:*

In den Funktionen der kontinuierlichen Bewegung wird als erstes die Anzahl der Schritte einer möglichen vorherigen Bewegung gespeichert. Anschließend wird der Dir-Pin und der Inkrementor auf die jeweilige Drehrichtung eingestellt, die Clock gestartet und die Schrittanzahl der letzten Bewegung zurückgegeben. Nach Aufrufen der moveStop()-Funktion wird die gezählte Schrittanzahl wieder zurückgesetzt.

Hinweis: Die Anzahl der gezählten Schritte ist auf den Umfang eines signed-int-16 Wertes begrenzt.

```
/*****************************************************************************/
/* Motorbewegung vorwärts bis zu neuer Anweisung                             */
/*                                                                           */
/* Rückgabe: Schritte der letzten Aktion bis zur Ausführung dieser Routine   */
/*           (Rückwärts < 0 < Vorwärts)                                      */
int16_t moveForward(){
       int16_t s = moveStop();      // Motor stoppen und Schrittzahl der letzten Bewegung
                                    zwischenspeichern
       OUTPORTDDR |= (1<<DIRPIN);   // Richtungs-Pin im Datenrichtungsregister als Ausgang
                                    definieren
       OUTPORT |= (1<<DIRPIN);      // Richtungs-Pin auf High setzen
       stepsInkrementor = 1;        // Schrittzähl-Inkrementor auf 1 setzen
       clockInit();                 // Timer0 initialisieren (--> Bewegung starten)
       return s;                    // Schrittzahl der letzten Bewegung zurückgeben
                                    (Rückwärts < 0 < Vorwärts)
}
```

*Abbildung 16: Scope 2 - Implementierung - kontinuierliche Bewegungen*

### *Bewegung mit vorgeschriebener Schrittanzahl - Routine:*

Bei der Bewegung mit vorgeschriebener Schrittanzahl erfolgt neben der Richtungskonfiguration am Dir-Pin, das Beschreiben der nsteps Variable mit der gewünschten Anzahl an Schritten (Achtung: Hier wieder eigentlich doppelte Anzahl zu verwenden). Da die Schrittzählung hier lediglich den Betrag der durchgeführten Schritte erfassen soll, wird die Variable stepsInkrementor in allen Routinen auf 1 gesetzt. Zur Ausführung der Schritte erfolgt nur noch die Initialisierung der Clock und das Warten, bis die Bewegung beendet und nsteps zurückgesetzt wurde.

```
/*****************************************************************************/
/* Motorbewegung vorwärts für n Schritte (bzw. Mikroschritte je nach Treiber config)  */
/*                                                                           */
/* Eingabeparameter: Auszuführende Schrittanzahl                             */
void moveNStepsForward(uint16_t n){
        moveStop();                         // Motor stoppen
        OUTPORTDDR |= (1<<DIRPIN);          // Richtungs-Pin im Datenrichtungsregister als
                                            Ausgang definieren
        OUTPORT |= (1<<DIRPIN);             // Richtungs-Pin auf High setzen
        stepsInkrementor = 1;               // Schrittzähl Inkrementor auf 1 setzen
        nsteps = n;                         // Soll-Schrittvariable auf übergebenen Wert
                                            setzen
        clockInit();                        // Timer0 initialiseren (--> Bewegung starten)
        while(nsteps != 0){_delay_us(1);}   // warten bis Schritte gefahren wurden
        moveStop();                         // Bewegung stoppen
}
```

*Abbildung 17: Scope 2 - Implementierung - Bewegung mit vorgeschriebener Schrittzahl*

## Hauptprogramm:

Aufgrund der umfangreichen Implementierung der Stepper-Library muss im Hauptprogram nur noch die jeweilige Taster-Abfrage mit der dazugehörigen Funktion ein Element in das Sequenz-Array zu schreiben oder das Array abzuspielen realisiert werden. Die Rückkehr zum Ursprung des Bewegungsablaufes gestaltet sich aufgrund der vorzeichenabhängigen Schritte sehr einfach. Die einzelnen Schritte können als Richtungsvektor betrachtet und somit addiert werden, um die aktuelle Position relativ zum Ursprung zu bestimmen. Ist das Ergebnis zum Beispiel positiv mit einem Betrag von 50 Schritten, so befindet sich der Motor 50 Schritte in die Vorwärts-Richtung entfernt vom Ursprung. Um zum Ursprung zurückzukehren muss also, um den Betrag der Schritte, in die entgegengesetzte Richtung gefahren werden. Die Implementierung ist unter Referenzen verlinkt.

## Nachweis:

Leider wurde die Hardware von uns zu früh abgegeben, ohne daran zu denken ein etwaiges Demonstrationsvideo aufzunehmen. Es wird als Nachweis daher auf die im Rahmen des HWE-Unterrichts durchgeführte Projektvorstellung verwiesen.

# Scope 3

Die Aufgabe der letzten Stufe besteht darin, die Frequenz, mit welcher der Schrittmotor betrieben wird, linear zu verändern. Die Änderung soll über einen Tastendruck erfolgen und bei durchgehendem Drücken ungefähr 2 Sekunden von 0% - 100% dauern. Die aktuelle Position und Frequenz sollen auf ein Display ausgegeben werden.

## Theorie:

Die Schwierigkeit dieser Stufe teilt sich auf zwei Gebiete auf. Das erste Gebiet ist die Schwierigkeit der linearen Frequenzänderung und das zweite das der Ansteuerung des LCDs, wozu eine eigene LCD-Library erstellt wurde. Lediglich die I²C-Senderoutine und die dazugehörigen Routinen wie die I²C-Init Routine wurden aus der Library vom Herrn Zudrell-Koch entnommen.

Lineare Frequenzänderung:

Für eine lineare Änderung der Frequenz genügt es nicht einen Softwareteiler einzusetzen, um von der maximalen Frequenz runter-zu-skalieren. Es wurde eine ISR-Frequenz gewählte, dessen Periodendauer sich gut als Teiler der Periodendauern der gewünschten Frequenzen darbietet. In diesem Fall wurde eine Frequenz 10kHz gewählt. Die nachfolgende Rechnung zeigt auf, wie sich die ISR-Frequenz auf die Genauigkeit der Frequenzerzeugung auswirkt.

$$f_{ISR} = \frac{f_{sys}}{HW_{teiler} * (OCR0 + 1)} = \frac{12MHz}{8 * (149 + 1)} = \frac{12000000}{1200} = \underline{10kHz}$$

$$\rightarrow T_{ISR} = \frac{1}{f_{ISR}} = 0.0001s$$

$$\rightarrow T_{100Hz} = T(2 * 100Hz) = 0.005s \rightarrow \quad SW_{counter} = \frac{T_{100Hz}}{T_{ISR}} = \underline{50}$$

$$\rightarrow T_{95Hz} = T(2 * 95Hz) = 0.005263s \rightarrow \quad SW_{counter} = \frac{T_{95Hz}}{T_{ISR}} = \underline{52{,}63}$$

$$\rightarrow T_{5Hz} = T(2 * 5Hz) = 0.1s \rightarrow \quad SW_{counter} = \frac{T_{5Hz}}{T_{ISR}} = \underline{1000}$$

*Abbildung 18: Scope 3 - Berechnung*

In den Berechnungen ist zu erkennen das sich sowohl die Soll-Frequenz als auch die ISR-Frequenz auf die Genauigkeit auswirkt. Lässt sich die Periodendauer der Soll-Frequenz ohne Rest durch die der ISR-Frequenz teilen (siehe 5Hz oder 100Hz), so entsteht keine Abweichung durch eine etwaige Rundung. Lässt sich aber wie bei 95Hz nicht ohne Rest teilen, entsteht durch die Rundung ein Fehler. Dieser Fehler könnte zum Beispiel durch die Wahl einer geringeren ISR-Aufruffrequenz kleiner gemacht werden. Aufgrund der langen I2C Übertragungsdauer für das Display ist das hier leider nicht möglich.

Display:

Die Ansteuerung des Displays erfolgt über einen I2C-Portexpander. Da allerdings das Hauptaugenmerk dieser Aufgabe auf der Schrittmotorsteuerung liegen soll, folgt anschließend nur eine kurze Erläuterung und der Source-Code Kommentierung des Codes wird in Maßen gehalten.

## Code:

Anschließend erfolgen kurze Erläuterungen zu Schlüsselstellen im Code. Der gesamte kommentierte Source-Code ist unter Referenzen verlinkt.

## Hauptprogramm:

### *Timer0-ISR:*

Die ISR erfüllt zwei wesentliche Funktionen. Die erste Funktion stellt die Frequenzerzeugung mittels Softwareteiler dar, die zweite das Zählen der Schritte und das Ausgeben der Schrittanzahl nach jedem hundertsten Schritt. Die reserved-Variable gibt an, ob das Display momentan durch eine Frequenzänderung bearbeitet wird und sperrt damit das Update der Schrittanzahl.

```
/*****************************************************************************/
/* Interrupt-Service-Routine der Signalerzeugung (f=10kHz -> T=0.1ms)         */
/*                                                                           */
/* period-Variable kann zur Frequenzänderung verwendet werden (!Achtung! zur */
/* Frequenzerzeugung muss die doppelte Frequenz gewählt werden)              */
ISR (TIMER0_COMP_vect){
	counter++;                  // Softwarecounter inkrementieren
	if (counter >= period){     // halbe Periodendauer der aktuellen Frequenz abwarten
		OUTPORT ^= (1<<CLOCKPIN);          // Clock-Pin toggeln
		counter = 0;                       // Softwarecounter zurücksetzen
		if (OUTPORT & (1<<CLOCKPIN)){      // Clock-Pin logisch 1? (jeder zweite ISR
		                                   Aufruf -> 1 Step)
			steps = steps + inkrementor; // Schrittzähl-Variable inkrementieren

			if (steps == 0){               // Schrittzähl-Variable 0?
				inkrementor = 1;       // Schrittzähl-Inkrementor auf 1 setzen
				                       (-> Zählrichtungsumkehr)
				steps = 1;             // Schrittzähl-Variable auf 1 setzen
				                       (Inkrement für diesen Zyklus manuell
				                       durchführen)
			}
			if (steps % 100 == 0 && reserved != 1){    // Schrittzahl gerade durch
			                                           100 teilbar und Display
			                                           Zugriff nicht reserviert?
				char str[14];                          // String-Buffer definieren
				sprintf(str, "Step: %d   ", steps); // Schrittzählvariable als
				                                       String auf den String
				                                       Buffer speichern
				lcd_setCursor(LCD_LINE1ADDR);      // LCD-Cursor auf
				                                       Anfangsadresse setzen
				lcd_printString(str);              // String-Buffer auf
				                                       Display ausgeben
			}
			steps = steps % 1600;              // Falls Schrittzählvariable 1600
			                                   -> zurücksetzen
		}
	}
}
```

*Abbildung 19: Scope 3 - Implementierung - ISR*

### *Increase/Decrease Frequency - Routine:*

Die Increase- und Decrease-Frequency Funktionen sind sich in ihrem Aufbau sehr ähnlich. Die Decrease-Funktion unterscheidet sich darin, dass diese ebenfalls für die Richtungsumkehr beim Erreichen der 0Hz zuständig ist. Neben der Ausgabe der neuen Soll-Frequenz auf dem Display erfolgt nur noch die Berechnung des neuen SW-Teiler-Werts. Erreicht die Soll-Frequenz 0Hz, wird der Timer gestoppt.

```
/*****************************************************************************/
/* Routine zur Verringerung der Frequenz (falls f < 0 --> Richtungsumkehr)    */
void decreaseFrequency(){
	char frequencyArray[12];	// String-Buffer definieren
	if (frequency > 5){		// Frequenz größer 5?
		frequency -= 5;		// Frequenz dekrementieren
		period = (uint16_t)1/(frequency*2)/0.0001;	// Neue Periodendauer aus
								Frequenz berechnen
		sprintf(frequencyArray, "Freq: %dHz    ", (int16_t)frequency);  // Frequenz
							als String in String-Buffer speichern
		lcd_setCursor(LCD_LINE2ADDR);	// LCD-Cursor auf Startadresse der 2.
						Zeile setzen
		lcd_printString(frequencyArray);	// String-Buffer auf Display ausgeben
	}else{					// Frequenz kleiner 5?
		inkrementor *= -1;	// Inkrementor invertieren (Steps Anzeige zählt nun bis
					zum Erreichen von 0 rückwärts)
		stop_Timer0();		// Motor stoppen
		frequency = 0;
		OUTPORT ^= (1<<DIRPIN);		// Richtungspin invertieren
		lcd_setCursor(LCD_LINE2ADDR);	// LCD-Cursor auf Startadresse der 2.
						Zeile setzen
		lcd_printString("Freq: 0Hz    ");	// Frequenz auf Display ausgeben
	}
}
```

*Abbildung 20: Scope 3 - Implementierung - Increase/Decrease Frequency*

# LCD-Ansteuerung:

## I²C-Übertragung & I/O-Expander:

Zur I²C-Übertragung wurde, die vom Herrn Zudrell vorgefertigte Bibliothek verwendet. Die Ansteuerung des I/O-Expanders (Datenblatt unter Referenzen) erfolgt damit sehr simpel. Durch die Auswahl der richtigen Adresse müssen die am I/O-Expander zur Ausgabe gewünschten Daten nur noch per I²C geschrieben werden.

## LCD-Controller

Für den LCD-Controller gibt es grundsätzlich zwei verschiedene Betriebsarten, 4-Bit und 8-Bit Betrieb sind möglich. Durch die Anwendung des I/O-Expanders und dessen Aufbau ist der 4-Bit Modus zwingend nötig.

### *Initialisierung:*

Um das LCD mit diesem Modus zu initialisieren, muss folgende Sequenz durchgeführt werden (Mehr Informationen im Datenblatt unter Referenzen):

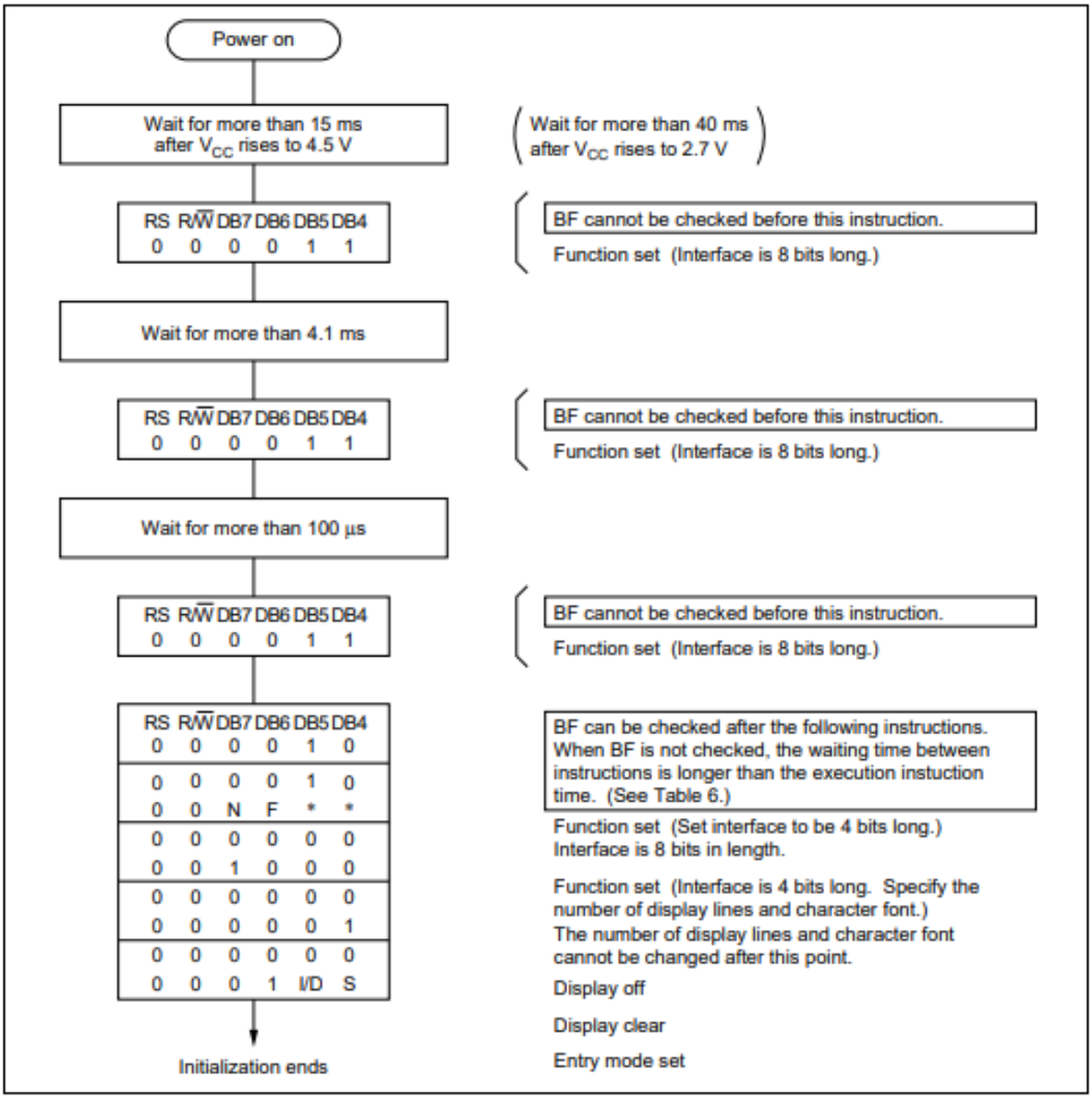


**Figure 24 4-Bit Interface**

*Abbildung 21: LCD-Initialisierung*

## Implementierung:

### LCD initialisieren – Routine:

```
// LCD-Initialisierungsroutine
void lcd_init(){
	// Initialisierungssequenz laut Datenblatt:
	iic_cmd4(3); _delay_ms(500);
	iic_cmd4(3); _delay_ms(20);
	iic_cmd4(3); _delay_ms(10);
	iic_cmd4(2); _delay_ms(10);

	iic_cmd8(2,8); // Function Set // 0100 1000
	iic_cmd8(0,8); // Display Off
	iic_cmd8(0,1); // Clear Display
	iic_cmd8(0,6); // Entry Mode Set
	iic_cmd8(0,14); // Display On
	iic_cmd8(0,2); // Return Home
}
```

*Abbildung 22: Scope 3 - Implementierung - LCD-Initialisierung*

### LCD 4-Bit Befehl – Routine:

```
// 4-Bit Befehls-Senderoutine
void iic_cmd4(uint8_t n){
	uint8_t data = (n<<4) | 8; //
	i2c_WriteNBytes(0x27, &data, 1);
	data = (n<<4) | 12;
	i2c_WriteNBytes(0x27, &data, 1);
	data = (n<<4) | 8;
	i2c_WriteNBytes(0x27, &data, 1);
}
```

*Abbildung 23: Scope 3 - Implementierung - LCD-4-Bit-Befehl*

### *4-Bit Steuerung:*

Die weitere Steuerung ergibt sich durch folgende im Datenblatt ersichtliche Sequenz (Mehr Informationen im Datenblatt):

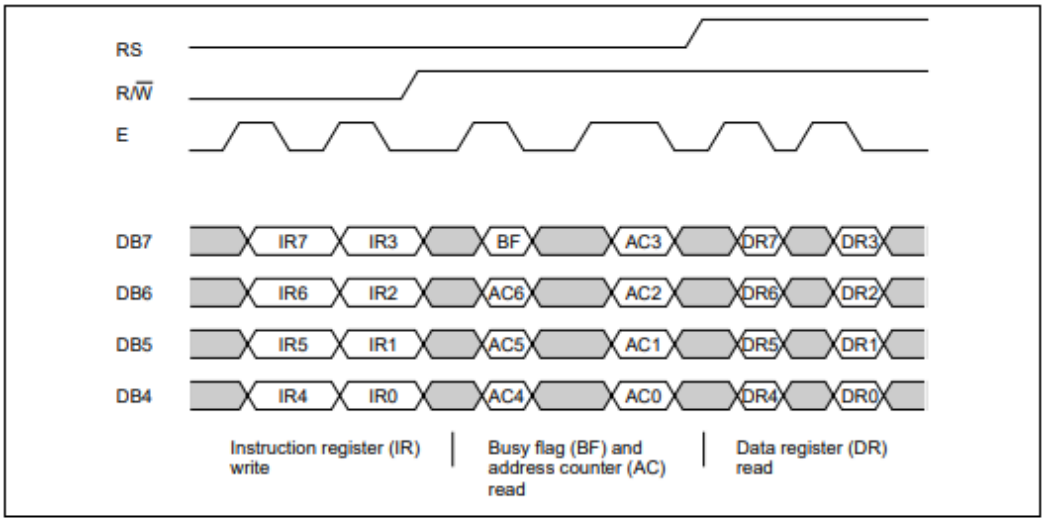


**Figure 9 4-Bit Transfer Example**

*Abbildung 24: Scope 3 - LCD 4-Bit-Steuerung*

## Implementierung:

### LCD 8-Bit Befehl - Routine:

```
// 8-Bit Befehls-Senderoutine für 4-Bit Modus mit seperierter Eingabe
void iic_cmd8(uint8_t hn, uint8_t ln){
	uint8_t data = (hn<<4) | 8;
	i2c_WriteNBytes(0x27, &data, 1);
	data = (hn<<4) | 12;
	i2c_WriteNBytes(0x27, &data, 1);
	data = (hn<<4) | 8;
	i2c_WriteNBytes(0x27, &data, 1);

	data = (ln<<4) | 8;
	i2c_WriteNBytes(0x27, &data, 1);
	data = (ln<<4) | 12;
	i2c_WriteNBytes(0x27, &data, 1);
	data = (ln<<4) | 8;
	i2c_WriteNBytes(0x27, &data, 1);

	_delay_ms(1);
}
```

*Abbildung 25: Scope 3 - Implementierung - LCD-8-Bit-Befehl*

### LCD 8-Bit-Hex Befehl – Routine:

```
// 8-Bit Befehls-Senderoutine für 4-Bit Modus
void iic_cmd8Hex(uint8_t n){
	uint8_t hn = (n>>4) & 0x0F;
	uint8_t ln = n & 0x0F;
	iic_cmd8(hn, ln);
}
```

### *Übertragen von Daten (Text):*

Die Sequenz für das Übertragen von Daten ist im Datenblatt wie folgt beschrieben (Mehr Informationen im Datenblatt):

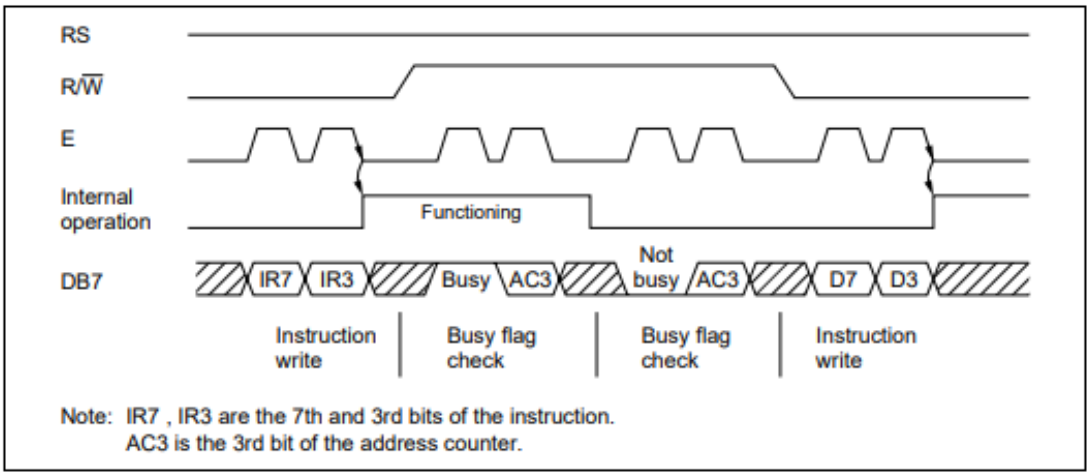


**Figure 17 Example of 4-Bit Data Transfer Timing Sequence**

*Abbildung 26: Scope 3 - LCD 4-Bit-Daten*

### LCD - Daten senden – Routine:

```
// Daten-Senderoutine
void iic_data(uint8_t hn, uint8_t ln){
	uint8_t data = hn | 9;
	i2c_WriteNBytes(0x27, &data, 1);
	data = hn | 13;
	i2c_WriteNBytes(0x27, &data, 1);
	data = hn | 9;
	i2c_WriteNBytes(0x27, &data, 1);

	data = ln | 9;
	i2c_WriteNBytes(0x27, &data, 1);
	data = ln | 13;
	i2c_WriteNBytes(0x27, &data, 1);
	data = ln | 9;
	i2c_WriteNBytes(0x27, &data, 1);
}
```

*Abbildung 27: Scope 3 - Implementierung - LCD - Daten senden*

### LCD - Buchstabe senden – Routine:

```
// Char Senderoutine
void iic_sendLetter(char letter){
	int asciiLetter = (int)letter;
	iic_data((asciiLetter & 0xF0), (asciiLetter & 0x0F)<<4); _delay_ms(1);
}
```

*Abbildung 28: Scope 3 - Implementierung - LCD Buchstabe senden*

### LCD - Wort senden – Routine:

```
// String Senderoutine
void iic_sendString(char* str) {
	int i;
	for (i = 0; i < strlen(str); i++) {
		if(i == 20){
			iic_cmd8Hex(LCD_LINE2ADDR | LCD_SETDDRAMADDR);
		}
		if(i == 0x54){
			iic_cmd8Hex(LCD_LINE3ADDR | LCD_SETDDRAMADDR);
		}
		if (i == 0x28){
			iic_cmd8Hex(LCD_LINE4ADDR | LCD_SETDDRAMADDR);
		}
		iic_sendLetter(str[i]);
	}
}
```

*Abbildung 29: Scope 3 - Implementierung - LCD Wort senden*

## Nachweis:

Die Frequenzwerte wurden gemessen und in nachfolgende Tabelle eingetragen. Ebenfalls wurde die theoretische errechnete Abweichung eingetragen um die erwarteten Abweichungen zu überprüfen. Die theoretische Berechnung ergibt sich aus dem gerundeten (ATmega rundet durch Type-Cast ab) Softwarevorteiler aus der Berechnung in Theorie auf die Frequenz umgeformt.

| FREQUENZ-SOLLWERT (IN HZ) | FREQUENZ-ISTWERT (IN HZ) | FREQUENZ-THEORETI-SCHER ISTWERT (IN HZ) |
|---|---|---|
| **0** | 0 | 0 |
| **5** | 5,00 | 5,00 |
| **10** | 10,00 | 10,00 |
| **15** | 15,01 | 15,02 |
| **20** | 20,00 | 20,00 |
| **25** | 25,00 | 25,00 |
| **30** | 30,12 | 30,12 |
| **35** | 35,21 | 35,21 |
| **40** | 39,99 | 40,00 |
| **45** | 45,04 | 45,05 |
| **50** | 49,99 | 50,00 |
| **55** | 55,55 | 55,55 |
| **60** | 60,23 | 60,24 |
| **65** | 65,78 | 65,79 |
| **70** | 70,47 | 70,42 |
| **75** | 75,75 | 75,76 |
| **80** | 80,63 | 80,65 |
| **85** | 86,19 | 86,20 |
| **90** | 90,9 | 90,90 |
| **95** | 96,14 | 96,15 |
| **100** | 99,99 | 100,00 |

*Abbildung 30: Scope 3 - Messtabelle*

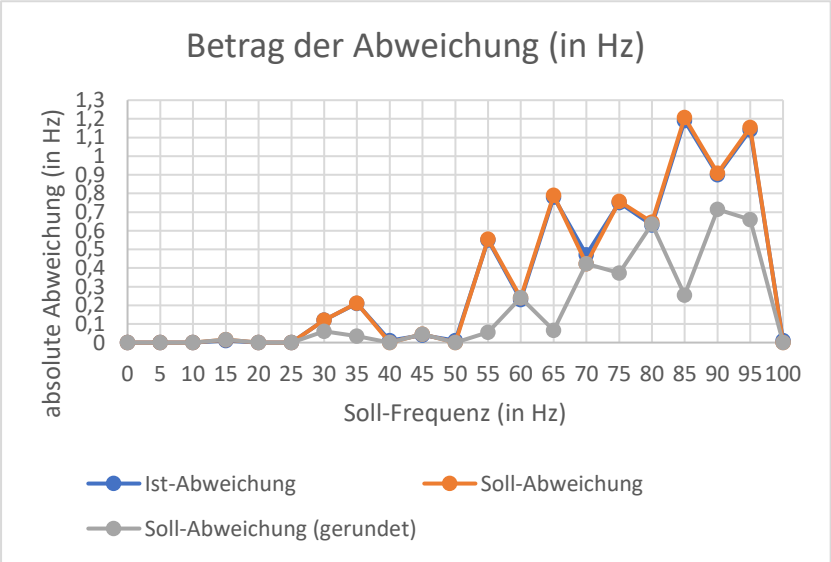


*Abbildung 31: Scope 3 - Abweichungsdiagramm*

Die berechnete Abweichung weicht von der tatsächlichen nur minimal ab, was ein sehr gutes Ergebnis ist. Durch die Analyse dieser Abweichungen wurde aber auch der erste Schritt zur Verbesserung der Genauigkeit erkannt. Wird die Funktion math.round() aus der Math-Bibliothek verwendet, werden Werte echt gerundet und die Genauigkeit erfährt dadurch einen enormen Vorteil. (graue Kurve)

Wie in Scope zwei wird zum Nachweis der Display-Funktion auf die Vorstellung verwiesen.

# Referenzen

- GitHub-Repository (https://github.com/Skh4rf/HWE-Schrittmotoransteuerung)

- Scope 1: Source-Code (https://github.com/Skh4rf/HWE-Schrittmotoransteuerung/tree/main/src/Metzler_J_4chel_HWE_Stepper_Scope1)

- Scope 2: Source-Code (https://github.com/Skh4rf/HWE-Schrittmotoransteuerung/tree/main/src/Metzler_J_4chel_HWE_Stepper_Scope2)

- Scope 3: Source-Code (https://github.com/Skh4rf/HWE-Schrittmotoransteuerung/tree/main/src/Metzler_J_4chel_HWE_Stepper_Scope3)

- TMC2209 Stepper-Driver (https://github.com/Skh4rf/HWE-Schrittmotoransteuerung/blob/main/doc/ref/Datasheet_TMC2209_Stepperdriver.pdf)

- PCF8574 Port-Expander (https://github.com/Skh4rf/HWE-Schrittmotoransteuerung/blob/main/doc/ref/Datasheet_PCF8574_I2C-Portexpander.pdf)

- HD44780 LCD-Controller (https://github.com/Skh4rf/HWE-Schrittmotoransteuerung/blob/main/doc/ref/Datasheet_HD44780_LCD-Controller.pdf)