

Übungs-Nr.:	10	Übungsbezeichnung:
Übungsdatum:	01. Dezember 2014	
Lehrer:	Prof. Dipl.-Ing. Helge Frank	Mikrocontroller
Gruppe:	3	Mitglieder:
Klasse:	4 AHET	Labenbacher Michael, Kieninger Dominik, Ibrahim Ibragimov

Protokollführer	Unterschriften	Note
Labenbacher		

Inhaltsverzeichnis

1 Einleitung	4
2 Verwendete Geräte & Betriebsmittel	5
3 Mikrocontroller	6
3.1 Aufgabenstellungen	6
3.2 Sägezahn-Signal mit arithmetischem Mittelwertbildner	7
3.2.1 Schaltungsentwicklung	7
3.2.2 Programmentwicklung	13
3.2.3 Schaltungsaufbau, Messvorgang & Auswertung	14
3.2.4 Messbericht	14
3.3 Dreieck-Signal mit aktivem Tiefpass 2.Ordnung	15
3.3.1 Schaltungsentwicklung	15
3.3.2 Programmentwicklung	22
3.3.3 Schaltungsaufbau, Messvorgang & Auswertung	24
3.3.4 Messbericht	24
3.4 Sinus-Signal mit R2R-Netzwerk	25
3.4.1 Schaltungsentwicklung	25
3.4.2 Programmentwicklung	27
3.4.3 Schaltungsaufbau, Messvorgang & Auswertung	29
3.4.4 Messbericht	31
3.5 3 ~ Sinus-Signal mit aktivem Tiefpass 2.Ordnung	32
3.5.1 Schaltungsentwicklung	32
3.5.2 Programmentwicklung	36
3.5.3 Schaltungsaufbau, Messvorgang & Auswertung	39
4 Resümee	40
5 Literatur- und Quellenverzeichnis	41

Tabellenverzeichnis

1	Geräte & Betriebsmittel	5
---	-----------------------------------	---

Abbildungsverzeichnis

1	Blockschaltbild zur Erzeugung des Sägezahn-Signales	7
2	Schaltung des OPV-Tiefpasses 1.Ordnung	9
3	Schaltung des invertierenden Verstärkers	11
4	Schaltung zur Erzeugung des Sägezahn-Signales	12
5	Oszilloskopeaufnahme des Sägezahn-Signals bei Potentiometerstellung=45%	14
6	Proteussimulation des Sägezahn-Signals bei Potentiometerstellung=45%	14
7	Blockschaltbild zur Erzeugung des Dreieck-Signales	15
8	Schaltung des OPV-Tiefpasses 2.Ordnung	17
9	Schaltung des Subtrahierers	19
10	Schaltung des invertierenden Verstärkers*	20
11	Schaltung zur Erzeugung des Dreieck-Signales	21
12	Schaltung zur Frequenzverstellung (+Referenzspannung)	21
13	Oszilloskopeaufnahme des Dreieck-Signals bei Potentiometerstellung=20%	24
14	Proteussimulation des Dreieck-Signals bei Potentiometerstellung=20%	24
15	Blockschaltbild zur Erzeugung des Sinus-Signales mit R2R-Netzwerk	25
16	Schaltung zur Erzeugung des Sinus-Signales	26
17	Schaltung zur Frequenz- & Amplitudenverstellung (+Referenzspannung)	26
18	Oszilloskopeaufnahme des Sinus-Signals 10Hz Potentiometerstellung=100%	29
19	Proteussimulation des Sinus-Signals 10Hz Potentiometerstellung=100%	29
20	Oszilloskopeaufnahme des Sinus-Signals 1,33Hz Potentiometerstellung=20%	30
21	Proteussimulation des Sinus-Signals 1,33Hz Potentiometerstellung=20%	30
22	Aufbau des R2R-Netzwerkes	31
23	Blockschaltbild zur Erzeugung des 3~Sinus-Signales	32
24	Schaltung zur Frequenzverstellung (+Referenzspannung)	34
25	Schaltung zur Erzeugung des 3~Sinus-Signales	35
26	Oszilloskopeaufnahme "3~Sinus-Signals" 20Hz Potentiometerstellung=20%	39
27	Proteussimulation 3~Sinus-Signals 20Hz Potentiometerstellung=20%	39

1 Einleitung

Dieses Projekt soll im Wesentlichen dazu dienen, die Kenntnisse vom Theorieunterricht (in Fachspezifische Informatik FI, Automatisierungstechnik AUT, Informationselektronik IE,...) praktisch überprüfen und besser verstehen zu lernen.

Bei allen Übungen handelt es sich darum, einen digitalen Wert, in eine analoge Spannung verschiedener Art umzuwandeln. Die einzelnen Projekte sind chronologisch nach dem Schwierigkeitsgrad geordnet. Verwendung finden in diesen Übungen unter anderem μ C, OPVs,....

Des Weiteren soll eine mögliche Dimensionierung von Bauteilen aufgezeigt werden und auch das Erstellen eines Programmes für die Umwandlung der Signale soll erläutert werden.

2 Verwendete Geräte & Betriebsmittel

Bezeichnung/Nr.	Gerät/Betriebsmittel	Beschreibung/Typ	Geräte-Nr.
O1	Oszilloskope	Tektronix TDS2004C	AA-4/2
N1	Doppelnetzgerät	Doppelnetzgerät DF1731 5B3A	IA-4/1
N2	Doppelnetzgerät	Doppelnetzgerät DF1731 5B3A	R2-5/3
yC1	Arduino Uno R3	ATmega328P ¹	—
OPV1	Operationsverstärker	LM741CN	AS-2/7
OPV2	Operationsverstärker	LM741CN	AS-2/1
OPV3	Operationsverstärker	LM741CN	R3-1/4
OPV4	Operationsverstärker	LM741CN	A5-2/6
OPV5	Operationsverstärker	LM741CN	A5-2/7
OPV6	Operationsverstärker	LM741CN	A5-2/8
OPV7	Operationsverstärker	LM741CN	A5-2/1
OPV8	Operationsverstärker	LM741CN	A5-2/2
OPV9	Operationsverstärker	LM741CN	A5-2/3

Tabelle 1: Geräte & Betriebsmittel

¹Weitere Informationen über den μ C, z.B. dessen Pinnbelegung,... entnehmen Sie dem (auch online verfügbarem) Datenblatt

3 Mikrocontroller

3.1 Aufgabenstellungen

Als kleiner Einstieg in das Thema Mikrocontroller soll ein einfaches Sägezahn-Signal, mittels Pulsweitenmodulation, von 0 bis $\pm U^2$ und einer konstanten Frequenz von 1Hz erzeugt werden. Dabei ist der μ C, Atmega328P, sprich der Arduino Uno R3, zu verwenden.

Als zweiten Schritt, ist ein Dreiecksignal mittels PWM, mit verstellbarer Amplitude von -U bis +U und einer variablen Frequenz von 0 bis 10Hz , mittels dem Arduino Uno R3 zu erzeugen.

Des Weiteren soll ein Sinus-Signal erzeugt werden, dessen Amplitude von 0 bis 5V und dessen Frequenz im Bereich von 0 bis 10Hz variabel eingestellt werden kann. Hier ist jedoch, im Gegensatz zu den beiden vorherigen Übungen, keine Pulsweitenmodulation anzuwenden, sondern ein R2R-Netzwerk zu berechnen und aufzubauen.

Als letztes Projekt ist ein $3 \sim$ Sinus durch PWM auszugeben. Die Frequenz dieses Signales soll bis zu 20Hz einstellbar und die Amplitude im Bereich von -U bis +U veränderbar sein.

² $\pm U = U_B \dots$ ist jeweils die Betriebsspannung der OPVs

3.2 Sägezahn-Signal mit arithmetischem Mittelwertbildner

3.2.1 Schaltungsentwicklung

Allgemeine Vorüberlegungen:

Für dieses Projekt werden im Wesentlichen folgende Komponenten benötigt:

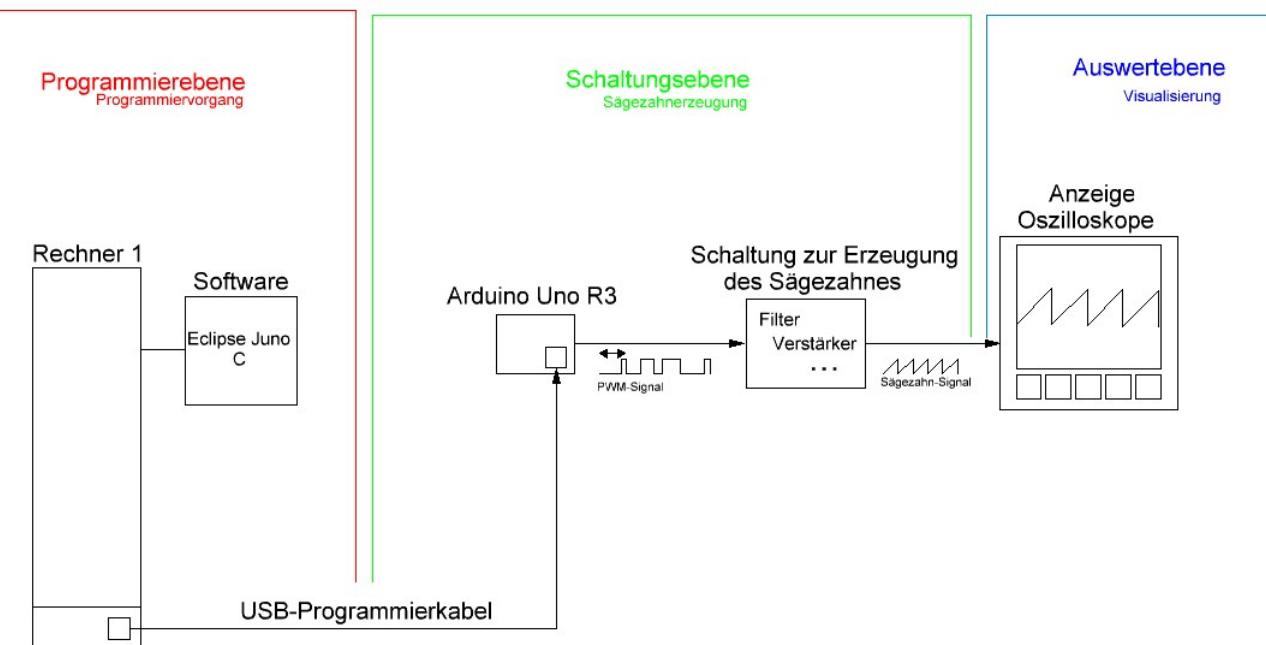


Abbildung 1: Blockschaltbild zur Erzeugung des Sägezahn-Signales

Am Beginn muss das Programm für den μ C entwickelt werden und im Anschluss darauf ist eine Schaltung zur Umsetzung des Vorhabens zu entwerfen. Danach kann mit der Überprüfung mittels Oszilloskope begonnen und kontrolliert werden. Des Weiteren folgt dann die Auswertung der gemessenen Signale.

Frequenzen berechnen:

Mittels dem nachfolgenden Programm im Abschnitt 3.2.2 wird an einem Port (PD6), des Arduino Uno R3's ein pulsweitenmoduliertes Signal ausgegeben und dieses muss nun in eine analoge Spannung, der Form eines Sägezahns umgewandelt werden.

Anders gesagt, muss ein Filter bestimmt werden, welches das PWM-Signal sozusagen glättet, also die hochfrequenten Anteile rausfiltert. → Tiefpass

Das entworfene Programm im Abschnitt 3.2.2 liefert ein PWM-Signal mit einer Frequenz, welcher nach folgender Formel berechnet werden kann:

$$f_{OC0} = \frac{f_{CPU}}{\text{Prescaler}0 \cdot (\text{TOP}_{FASTPWM0} + 1)} \quad (1)$$

- f_{OC0} Taktfrequenz am Ausgang des Timer0
- f_{CPU} CPU – Frequenz
- $\text{Prescaler}0$ Vorteiler des Timer 0
- $\text{TOP}_{FASTPWM0}$ Maximalwert des Timer 0 bei Fast PWM → 255

Das ergibt bei unserem Projekt, bei einer CPU-Frequenz von 16Mhz und einem gewählten Prescaler von 8 und bei 8Bit Fast PWM eine Taktfrequenz von:

$$f_{OC0} = \frac{16000000}{8 \cdot (255 + 1)} = 7812,5 \text{ Hz}$$

Das OCR0A-Register des Timer 0 soll bei einem Sägezahn-Signal von 1Hz, sprich innerhalb 1s, von 0 bis 255 hochzählen. Mit der nachfolgenden Formel lässt sich die Änderungsfrequenz, ermitteln.

$$f_{Änderung0} = \frac{f_{OC0}}{z_0} \quad (2)$$

- $f_{Änderung0}$ Frequenz mit der sich der Wert im OCR0A – Register erhöht
- f_{OC0} Taktfrequenz am Ausgang des Timer 0
- z_0 Gewählte Änderung des PWM – Signales (von 0 – 255) → 256

Dies ergibt explicit in unserem Fall eine Änderungsfrequenz von:

$$f_{Änderung0} = \frac{7812,5}{256} = 30,5 \text{ Hz}$$

Filterberechnung:

Es wird hier ein normaler OPV-Tiefpass 1. Ordnung verwendet, da dieser hier sicherlich ausreichend ist. Die allgemeine Schaltung sieht folgendermaßen aus:

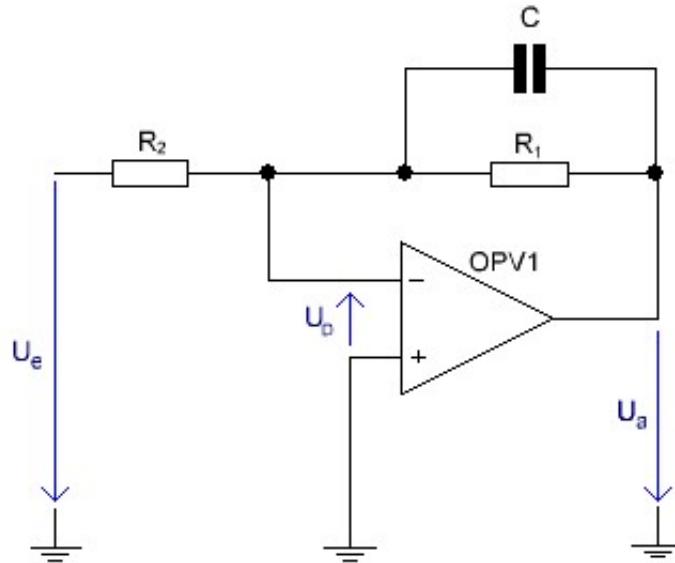


Abbildung 2: Schaltung des OPV-Tiefpasses 1. Ordnung

Nun müssen die Widerstände und Kapazitäten ermittelt werden. Die Formeln zur Berechnung dieser Schaltung³ lauten wie folgt:

$$G_1(s) = -\frac{R_1}{R_2} \cdot \frac{1}{1+sT} \quad \text{mit } T = R_1 C \quad (3)$$

$$K_1 = \frac{R_1}{R_2} \cdot \frac{1}{\sqrt{1 + (\omega T)^2}} \quad (4)$$

$$\omega_K = \frac{1}{T} \quad (5)$$

$$f_K = \frac{\omega_K}{2\pi} \quad (6)$$

$G_1(s)$	<i>Übertragungsfunktion</i>
K_1	<i>Verstärkung, also der Betrag von $G_1(s)$</i>
ω_K	<i>Knickkreisfrequenz</i>
f_K	<i>Kreisfrequenz</i>
R_1, R_2	<i>Widerstände</i>
C	<i>Kondensator</i>
T	<i>Zeitkonstante</i>

³In diesem Projekt wurde auf die Herleitung der Formeln verzichtet, da dies schon im Theorieunterricht erfolgte.

Berechnung:

Nun wird einfach als Kreisfrequenz die vorher bestimmte Änderungsfrequenz und ein beliebiger Kondensator gewählt. Es wird deswegen ein Kondensator C gewählt und nicht der Widerstand R_1 , da man das Ergebnis für R_1 leichter realisieren kann. Der Kondensator C sollte sich rasch auf- und entladen können und somit relativ klein sein. zB.: 100nF.

Damit kann der Widerstand R_1 wie folgt mit den Formeln 5 & 6 berechnet werden:

$$R_1 = \frac{1}{\omega_K C} = \frac{1}{2 \cdot \pi \cdot 30,5 \text{Hz} \cdot 100 \text{nF}} = 52,15 \text{k}\Omega$$

Gewählt wurde somit ein einziger Widerstand von $47 \text{k}\Omega$, da so die Schaltung übersichtlich ist und dieser noch im 10%igen Toleranzbereich liegt.

Der Widerstand R_2 wurde gleich groß gewählt, damit eine Verstärkung von ungefähr 1, bei niedrigen Frequenzen vorliegt.

Verstärker berechnen:

Da dieser Tiefpass jedoch das Ausgangssignal invertiert und nicht verstärkt muss noch ein invertierender Verstärker, mit verstellbarer Verstärkung nachgeschalten werden, welcher, wie folgt, realisiert wurde:

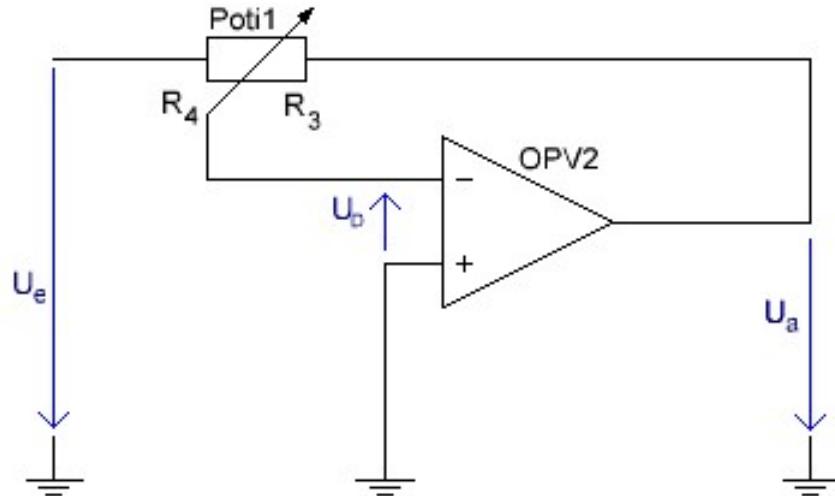


Abbildung 3: Schaltung des invertierenden Verstärkers

Berechnungsformeln:

$$G_2(s) = -\frac{R_3}{R_4} \quad (7)$$

$$K_2 = \frac{R_3}{R_4} \quad (8)$$

$G_2(s)$ Übertragungsfunktion des Invertierers

K_2 Verstärkung des Invertierers

R_3, R_4 Widerstände (eigentlich Poti 1, diese Darstellung dient nur der Einfachheitshalber)

Es wurde hier einfach ein Potentiometer Poti 1 mit $100k\Omega$ gewählt, und man kann somit den Sägezahn-Maximalwert im Prinzip beliebig von 0 bis $+U_B$ verstetigen.⁴

⁴Hier ist schon anzumerken, dass die Betriebsspannungen aller Operationsverstärker im gesamten Projekt $\pm 15V$ beträgt (=ideal, real $\approx 12V$) und aus übersichtlichen Gründen nie eingezeichnet wurde, aber immer mittels dem Netzgerät N2 zur Verfügung gestellt wurde.

Die fertige Gesamtschaltung sieht nun folgendermaßen aus:

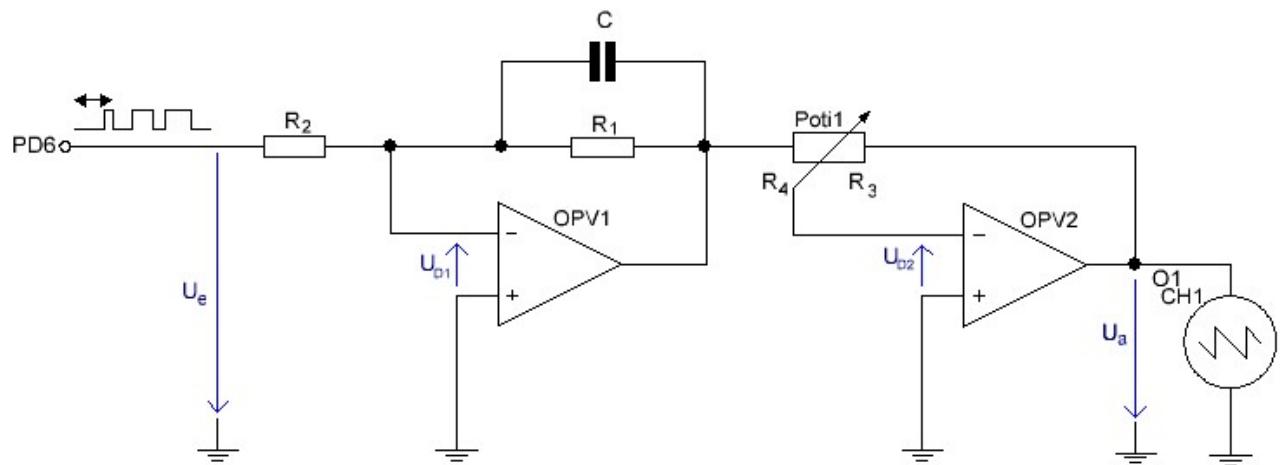


Abbildung 4: Schaltung zur Erzeugung des Sägezahn-Signales

Berechnungsformeln:

$$G(s) = \frac{R_3}{R_4} \cdot \frac{R_1}{R_2} \cdot \frac{1}{1+sT} \quad \text{mit } T = R_1 C \quad (9)$$

$$K = \frac{R_3}{R_4} \cdot \frac{R_1}{R_2} \cdot \frac{1}{\sqrt{1 + (\omega T)^2}} \quad (10)$$

$G(s)$ Gesamtübertragungsfunktion

K Gesamtverstärkung

R_1, R_2, R_3, R_4 ... Widerstände

Diese Schaltung konnte nun mit den zuvor bestimmten Kapazitäts- und Widerstandswerten aufgebaut und in Betrieb genommen werden.

3.2.2 Programmentwicklung

Mittels dem Programm Eclipse Juno wurde mit der Hochsprache C folgendes, dokumentiertes Programm, zur Erzeugung eines Sägezahn-PWM-Signals, erstellt:

```

1  /* SägezahnPWM.c
2   * Created on: 01.12.2014
3   * Author: Michael Labenbacher */
4 //1. "Packages" laden
5 //Bibliothek für den Prozessor laden
6 #include<avr/io.h>
7 //Includedatein für die Funktionen zur Interrupt–Verarbeitung einbinden
8 #include<avr/interrupt.h>
9 //2. Interruptroutinen & (Unterprogramme) & (Höherwertige Variablen definieren)
10 //Höherwertige Variablen
11 volatile int icnt; //Overflow–Zähler
12 ISR(TIMER0_OVF_vect){ //Overflow–Interrupt–Routine
13 icnt++; //Die Overflows werden damit gezählt
14 }
15 //3. Hauptprogramm
16 int main(void){
17 //3.1 Initialisierung
18     int a=0; //Hilfsvariable zur genaueren Realisierung der Frequenz (optional)
19     //Port PD6 (OCR0A–Ausgang) als Ausgang definieren
20     PORTD=(1<<PD6); //Register zum Ansteuern des Ausganges (zB. am Beginn auf High)
21     DDRD=(1<<PD6); //Datenrichtung (Ausgang) festlegen
22     /*Timer 0 Einstellungen*/
23     /* Mittels den WGMxx kann die Betriebsart gewählt werden , hier Fast PWM
24      * mit Top 0xFF PWM–Signal an OCR0A non–inverted */
25     TCCR0A=(1<<COM0A1)|(1<<WGM01)|(1<<WGM00);
26     /*Des weiteren wird mittels den CSxx kann der Prescaler eingestellt werden ,
27      * hier clk/8 dies ergibt bei 16MHz → 16MHz/8=2MHz */
28     TCCR0B=(1<<CS01);
29     /* Nun muss nur noch beim Überlauf des Timers ein Overflow Interrupt
30      * ausgelöst werden , kurz: Timer Overflow Interrupt Enable 0 aktivieren */
31     TIMSK0=(1<<TOIE0);
32     sei(); //Globale Interruptfreigabe
33 //3.2 Endlosschleife
34     while(1){
35         /* 2MHz/256 (von0–255→256 Schritte) Interrupts sind notwendig , dass eine
36          * Sekunde vergeht . OCR0A zählt von 0–255, dies soll eine Sekunde dauern
37          * → 1MHz/256 /256 */
38         if((icnt>=30 && a==0)|| (icnt>=31 && a==1)){
39             icnt=0; //Nun muss der Overflow–Zähler wieder zurückgesetzt werden.
40             OCR0A++; //Wert im OCR0A–Register jede rund 1/30,5 Sekunde um 1 erhöhen
41             /* Der nachfolgende Programmabschnitt ist nicht erforderlich , dient aber
42              * dazu , dass eine Frequenz von 1Hz exakter realisiert wird , da die
43              * berechnete Frequenz im Mittel 30,5 beträgt */
44             if(a==0){a=1;}
45             else{a=0;}
46         }
47     }
48 }
```

3.2.3 Schaltungsaufbau, Messvorgang & Auswertung

Nach dem Aufbau der Schaltung Abb.4 wurde das fertige Programm in den μ C1 geladen und mittels dem Oszilloskop O1 konnte das Sägezahn-Signal erfasst werden.

Das ganze kann auch mittels der Software Proteus simuliert werden und es wurde das Potentiometer auf 45% eingestellt und ergab folgende Auswertungen:⁵

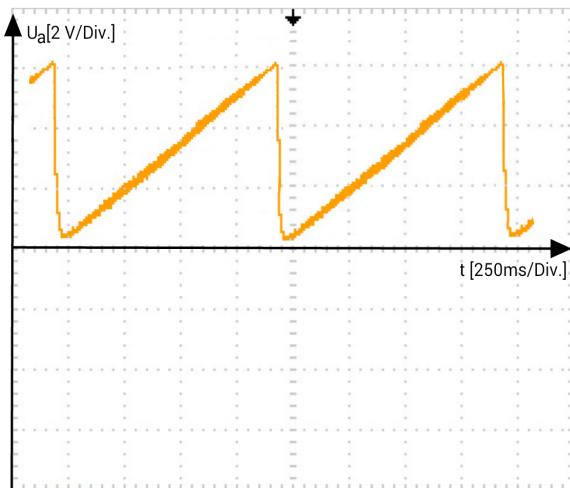


Abbildung 5: Oszilloskopeaufnahme
des Sägezahn-Signals bei
Potentiometerstellung=45%

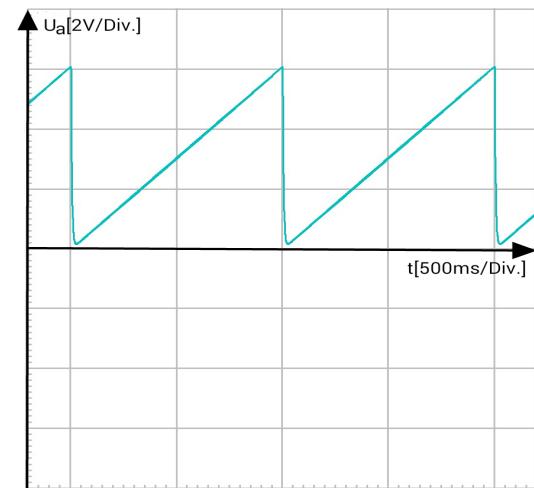


Abbildung 6: Proteussimulation
des Sägezahn-Signals bei
Potentiometerstellung=45%

Es lässt sich an der Oszilloskopeaufnahme erkennen, dass mittels dem Programm und der zugehörigen Schaltung, wie gewünscht, ein Sägezahn-Signal mit 1Hz im Prinzip korrekt ausgegeben wird. Jedoch ist am ansteigenden Teil des Sägezahns das Laden & Entladen des Kondensators zu erkennen, was darauf schließen lässt, dass ein bisschen ein kleinerer Kondensator zu wählen wäre.

3.2.4 Messbericht

Zusammenfassend, bezogen auf den 1.Abschnitt des Gesamtprojektes, lässt sich sagen, dass ein solches Sägezahn-Signal nun für weitere Anwendungen zur Verfügung steht. Genauso kann man ein solches vorhandenes Sägezahn-Signal mittels einem Sinussignal und einem OPV wieder in ein pulsweitenmoduliertes Signal rückwandeln.

⁵Alle an den Abbildungen angegebenen Potentiometerstellungen im gesamten Protokoll beziehen sich auf die Amplitudenverstellung!

3.3 Dreieck-Signal mit aktivem Tiefpass 2. Ordnung

3.3.1 Schaltungsentwicklung

Wiederum starteten wir mit einigen allgemeinen Überlegungen, wie dieses Projekt realisiert werden kann. Dabei ergab sich schließlich folgende Form:

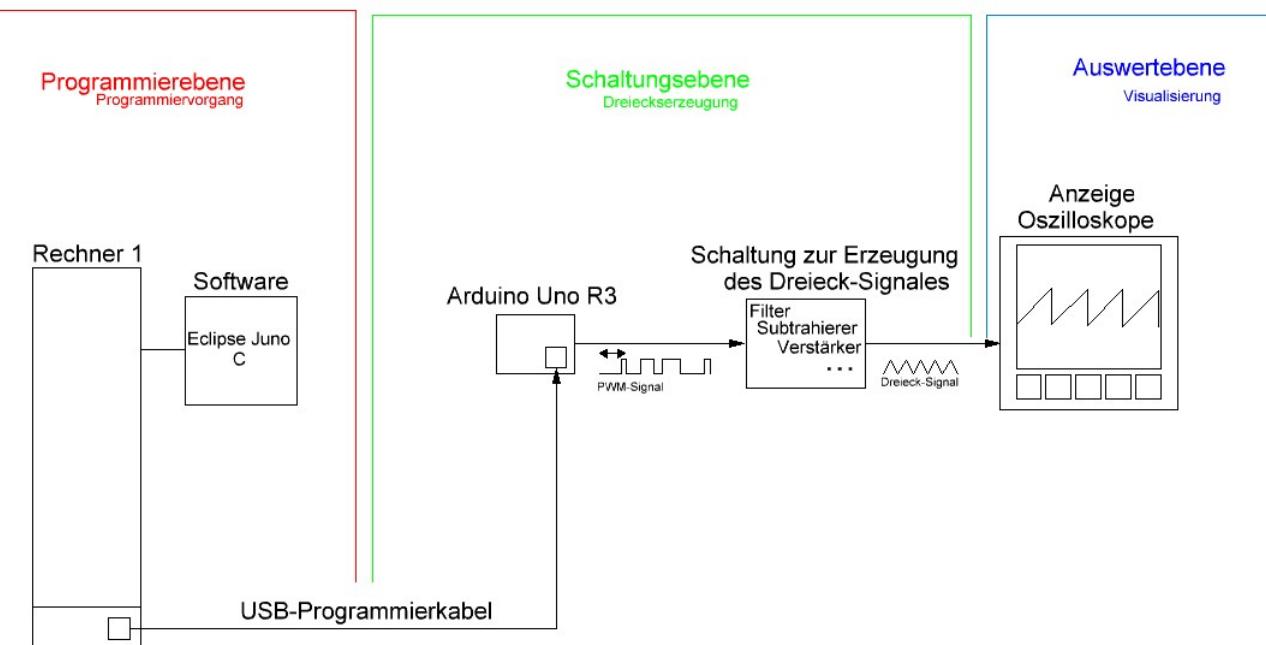


Abbildung 7: Blockschaltbild zur Erzeugung des Dreieck-Signales

Zuerst wird das Programm, für die Erzeugung eines Dreiecksignals für den Mikrocontroller entwickelt und danach kann eine Schaltung für dessen Umsetzung entwickelt werden. Im Anschluss ist dies mittels dem Oszilloskope wieder zu überprüfen und mit der Simulation in Proteus auszuwerten.

Frequenz berechnen:

Erneut wird mit dem Programm, im Abschnitt 3.3.2, ein PWM-Signal an Port D6 ausgegeben, da dies ein Vergleichsausgang des Timer 0 ist. Diese ist in eine analoge, dreiecksförmliche Spannung umzuwandeln.

Da, dieses mal, Frequenzen bis zu 10Hz auftreten sollen, genügt ein Tiefpass 1.Ordnung mit nur -20dB/Deckade nicht mehr und somit muss einer zB. mit -40dB/Deckade herangezogen werden.

→ Tiefpass 2.Ordnung

Es wurde eine CPU-Frequenz von erneut 16Mhz und ein Prescaler von 8 gewählt, bei 8Bit Fast PWM. Dies ergibt mit der Formel 1:

$$f_{OC0} = \frac{16000000}{8 \cdot (255 + 1)} = 7812,5 \text{ Hz}$$

Das OCR0A-Register des Timer 0 soll bei einem Dreieck-Signal von max. 10Hz, von 0 bis 255 hoch und wieder bis 1 runterzählen. Dabei wurde im Programm nach jeweils 15 Interrupts, also nach $\frac{15}{7812,5}$ Sekunden der Wert im Register erhöht, bei 1Hz um eins und bei max. 10Hz um 10. Daraus folgt nun eine gewählte Änderung von max. $15 \cdot 10 = 150$. Die (maximale) Änderungsfrequenz lässt sich mit der Formel 2 bestimmen:

$$f_{Änderung0} = \frac{7812,5}{150} = 52,08 \text{ Hz}$$

Aus diesen Erkenntnissen kann nun mit der Filterberechnung begonnen werden.

Filterberechnung

Es wird hier ein OPV-Tiefpass 2. Ordnung verwendet, welcher durch folgende Schaltung realisiert wurde:

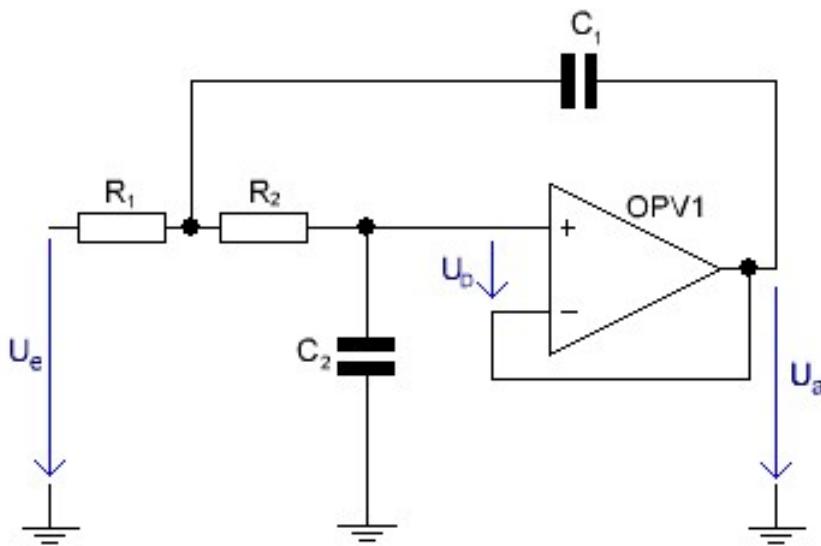


Abbildung 8: Schaltung des OPV-Tiefpasses 2. Ordnung

Die dazu verwendeten Formeln:

$$G_1(s) = \frac{1}{1 + sC_2(R_1 + R_2) + s^2C_2C_1R_2R_1} \quad (11)$$

$$K_1 = \frac{1}{\sqrt{(1 - \omega_C C_1 R_2 R_1)^2 + (\omega_C C_1 R_2 R_1)^2}} \quad (12)$$

$$\omega_K = \frac{1}{\sqrt{C_2 C_1 R_2 R_1}} \quad (13)$$

$$f_K = \frac{\omega_K}{2\pi} \quad (14)$$

$G_1(s)$	Übertragungsfunktion des Tiefpasses 2. Ordnung
K_1	Verstärkung, also der Betrag von $G_1(s)$ des Tiefpasses 2. Ordnung
ω_K	Knickkreisfrequenz
f_K	Kreisfrequenz
R_1, R_2	Widerstände
C_1, C_2	Kondensatoren

Berechnung:

Nun wird wieder die Änderungsfrequenz der Kreisfrequenz gleich gesetzt und beliebige Kondensatoren gewählt, welche im Bereich von 10-1000nF liegen sollten, wobei kleinere zu bevorzugen sind, wie z.B. $C_2 = 100nF$ und $C_1 = 220nF$. Daraus ergibt sich nun das Produkt von den Widerständen durch die Formeln 13 und 14:

$$R_1 R_2 = \frac{1}{(2 \cdot \pi \cdot 52,08Hz)^2 \cdot 100nF \cdot 220nF} = 424,44M\Omega^2$$

Da sich der Wert im OCR0A-Register um 10 jeweils maximal erhöht muss davon ein zentel, also 42,444MΩ gewählt werden. Somit kann zB. R_2 mit 10kΩ und R_1 mit 4,7kΩ genommen werden, da dessen Produkt $10k\Omega \cdot 4,7k\Omega = 47M\Omega^2$ ist und somit leicht im Toleranzbereich, durch die verwendeten Widerstände, liegt.

Subtrahierer berechnen

Die Dreiecksspannung liegt nach dem Tiefpass zwischen 0V bis 5V vor und dieser Gleichanteil von 2,5V muss nun entfernt werden, mit zB. folgendem Subtrahierer:

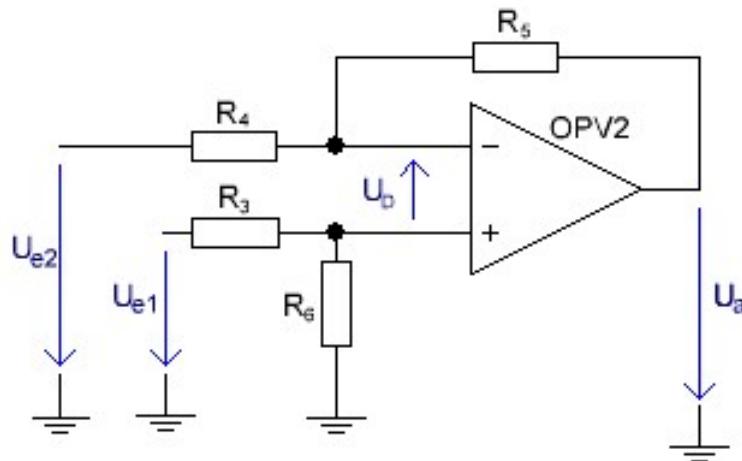


Abbildung 9: Schaltung des Subtrahierers

Berechnungsformeln:

$$U_a = U_{e1} - U_{e2} \quad (15)$$

$R_3, R_4, R_5, R_6 \dots$ Widerstände

Die Spannung kann bei einer einfachen Versuchsschaltung durch ein Netzgerät zur Verfügung gestellt werden, wie bei uns durch N1. Für diverse Anwendungen dann, sollte natürlich entweder ein Spannungsteiler, Z-Diode oder weiteres verwendet werden, um diese Spannung stabil zu halten.

Für die Dimensionierung der Widerstände ist nicht viel zu sagen außer, dass gleich große verwendet werden müssen und im Bereich von $1k\Omega$ und $1M\Omega$ liegen sollten, also zB.

$R_3 = R_4 = R_5 = R_6 = 10k\Omega$.

Falls es noch nicht offensichtlich war: Im späteren Verlauf ist für die Spannung U_{e1} die Ausgangsspannung des Tiefpasses und für U_{e2} die Spannung von 2,5V zu verwenden.

Verstärker berechnen:

Dieser Verstärker ist analog zu Punkt 3.2.1, dabei wurde die Invertierung beim vorherigem Projekt benötigt, hier hat sie prinzipiell keine wichtige Wirkung, sondern nur die Verstärkung ist für uns hier wichtig.

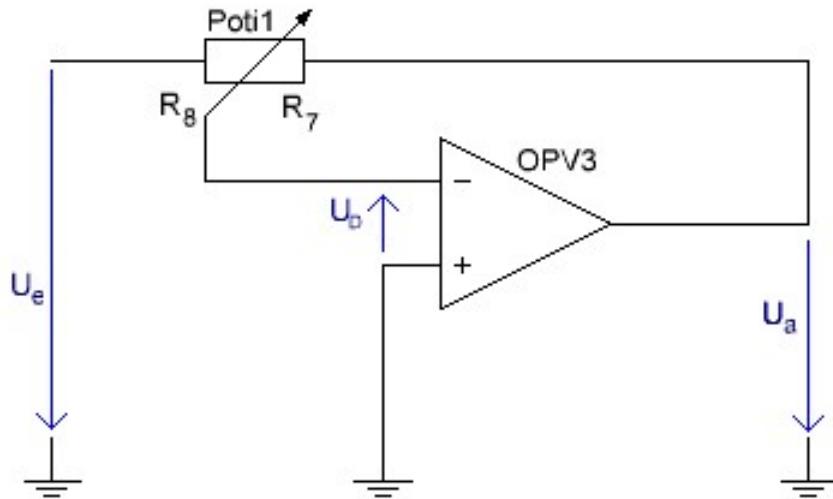


Abbildung 10: Schaltung des invertierenden Verstärkers*

Berechnungsformeln:

$$G_2(s) = -\frac{R_7}{R_8} \quad (16)$$

$$K_2 = \frac{R_7}{R_8} \quad (17)$$

$G_2(s)$ Übertragungsfunktion des Invertierers

K_2 Verstärkung des Invertierers

R_7, R_8 Widerstände(Poti1)

Es wurde hier einfach wieder das Potentiometer Poti 1 mit $100k\Omega$ gewählt, und man kann somit den Sägezahn-Maximalwert im Prinzip beliebig von $-U_B$ bis $+U_B$ verstetzen.

Die fertige Gesamtschaltung hat nun folgenden Aufbau:

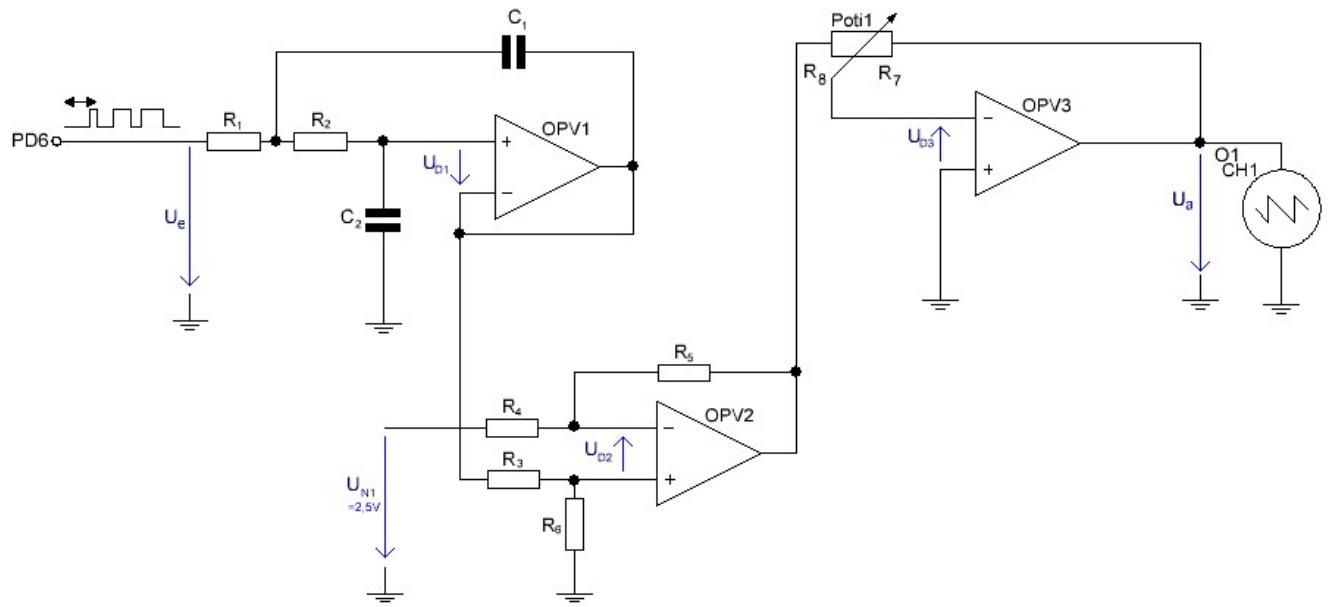


Abbildung 11: Schaltung zur Erzeugung des Dreieck-Signales

Diese Schaltung konnte nun mit den zuvor bestimmten Kapazitäts- und Widerstandswerten aufgebaut und in Betrieb genommen werden.

Natürlich darf nicht auf die Referenzspannung und dem Poti 2, welches zur Verstellung der Frequenz dient, vergessen werden:

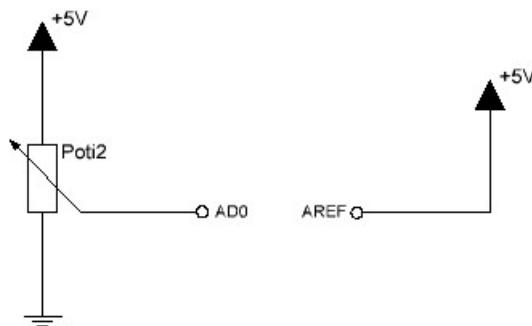


Abbildung 12: Schaltung zur Frequenzverstellung (+Referenzspannung)

Dabei sollte die Größe des Potentiometers im Bereich von $1k\Omega - 1M\Omega$ liegen, wie zB.: $10k\Omega$.

3.3.2 Programmentwicklung

Das dazugehörige Programm, dient zur Erzeugung eines Dreieck-Signales mittels einem Atmega 328P:

```

1  /*
2   * SägezahnPWM.c
3   * Created on: 01.12.2014
4   * Author: Michael Labenbacher
5   */
6 //1. "Packages" laden
7 //Bibliothek für den Prozessor laden
8 #include<avr/io.h>
9 //Includedatein für die Funktionen zur Interrupt-Verarbeitung einbinden
10 #include<avr/interrupt.h>
11 //2. Interruptroutinen & (Unterprogramme) & (Höherwertige Variablen definieren)
12 //Höherwertige Variablen
13 volatile int merker=1; //Merker-Variablen
14 volatile int icnt; //Overflow-Zähler
15 //Interruptroutinen
16 ISR(TIMER0_OVF_vect){ //Overflow-Interrupt-Routine
17     //Jeder Overflow Interrupt soll gezählt werden (2MHz / 256 Schritte = icnt+1)
18     icnt++;
19 }
20 //3. Hauptprogramm
21 int main(void){
22 //3.1 Initialisierung
23     //Allgemeine Variablen
24     //Der Rohwert von der ADC-Messung wird hier zwischengespeichert
25     //zur Weiterverarbeitung
26     double Rohwert;
27     //Dies ist direkt Proportional der Frequenz,
28     //sprich 1Hz->frequenz=1 / 10Hz->frequenz=10 / ...
29     int frequenz=1;
30     //Port PD6 (OCR0A-Ausgang) als Ausgang definieren
31     //Register zum Ansteuern des Ausganges (zB. am Beginn auf High)
32     PORTD=(1<<PD6);
33     //Datenrichtung (Ausgang) festlegen
34     DDRD=(1<<PD6);
35     //ADC-Port AD0 als Eingang zur Messung am Kanal 0 festlegen (Datenrichtung)
36     DDRC&=~(1<<PC0);
37     /* Timer 0 Einstellungen*/
38     /* Mittels den WGMxx kann die Betriebsart gewählt werden, hier
39      * Fast PWM mit Top 0xFF PWM-Signal an OCR0A non-inverted */
40     TCCR0A=(1<<COM0A1)|(1<<WGM01)|(1<<WGM00);
41     /* Des weiteren wird mittels den CSxx kann der Prescaler eingestellt werden,
42      * hier clk/8 dies ergibt bei 16MHz --> 16MHz/8=2MHz */
43     TCCR0B=(1<<CS01);
44     /* Nun muss nur noch beim Überlauf des Timers ein Overflow Interrupt
45      * ausgelöst werden, kurz: Timer Overflow Interrupt Enable 0 aktivieren */
46     TIMSK0=(1<<TOIE0);
47     //Globale Interruptfreigabe
48     sei();
49     /* Nun muss noch der ADC eingeschalten werden mittels ADEN=Enable und
50      * mit der tiefsten Frequenz einfach mal anfangen */
51     ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);

```

```

1 //3.2 Endlosschleife
2 while(1){
3     //ADC-Messung am Kanal 0 (AD0)
4     ADCSRA|=(1<<ADSC); //Start der Messung
5     while((ADCSRA & (1<<ADSC))!=0){ //Messung läuft (warten bis diese fertig ist)
6     }
7     //Der gemessene Wert steht im Register ADCW (oder in —> ADCL & ADCH)
8     Rohwert=ADCW;
9     //—> Frequenz berechnet werden, welche im Bereich von 0–10Hz liegen soll
10    frequenz=Rohwert/1023*10;
11    //—> keine Frequenz existieren welche Null ist,
12    //da immer ein Dreieck entstehen soll
13    if(frequenz<=0){frequenz=1;}
14    /* 2MHz/256 (von 0–255—>256 Schritte des Timer 0) Interrupts sind notwendig ,
15     * dass eine Sekunde vergeht. OCR0A zählt von 0–255–1, dies soll
16     * eine Sekunde dauern —> 2MHz/256 /511 */
17    if(icnt>=15){
18        //Die Overflow-Zählvariable natürlich wieder rücksetzen
19        icnt=0;
20        //Einmal raufzählen mit der Frequenz
21        if(merker==1){OCR0A+=frequenz;}
22        //Einmal runterzählen mit der Frequenz
23        else{OCR0A-=frequenz;}
24        //Falls das Maximum erreicht wird, muss man runterzählen
25        if(OCR0A>=255-frequenz){
26            merker=0;
27        }
28        //Falls das Minimum erreicht wird, muss man raufzählen
29        if(OCR0A<=1+frequenz){
30            merker=1;
31        }
32    }
33 }
34 }
```

3.3.3 Schaltungsaufbau, Messvorgang & Auswertung

Nach dem Aufbau der Schaltung Abb. 11 wurde das fertiggestellte yC-Programm in den Arduino mittels dem Programmierkabel hineingeladen und es konnte mittels dem Oszilloskop der Signalverlauf erfasst werden und ergab bei einer Potentiometerstellung von 20% folgende Aufnahme: (die Potentiometerstellung bezieht sich auf das Poti 1)

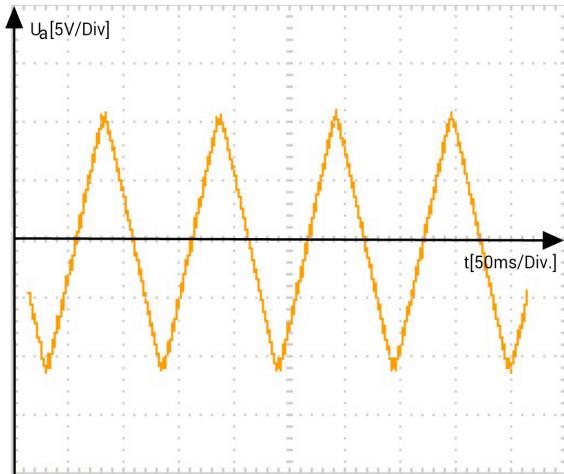


Abbildung 13: Oszilloskopeaufnahme
des Dreieck-Signals bei
Potentiometerstellung=20%

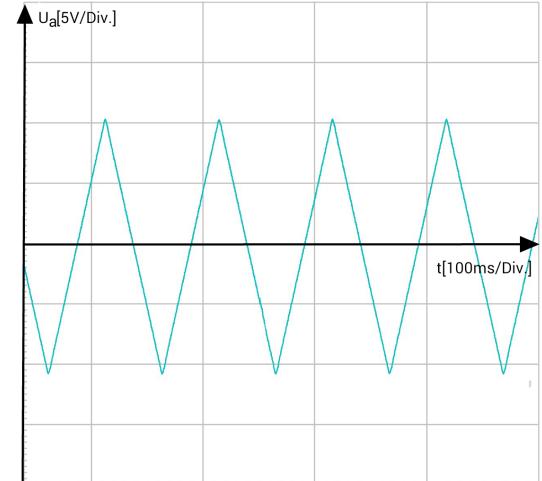


Abbildung 14: Proteussimulation
des Dreieck-Signals bei
Potentiometerstellung=20%

Der Vergleich der beiden Abbildungen 13 & 14 zeigt eine eindeutige Übereinstimmung, was darauf schließen lässt, dass ein Dreieckssignal relativ einfach und genau bis zu 10Hz realisiert werden kann.

3.3.4 Messbericht

Die Messungen zeigen, dass das Dreiecks-Signal Ein solches dreieckförmige Signal kann für viele weiter Anwendungen nun zur Verfüzung gestellt werden, wie zum Beispiel: zur Erzeugung von PWM-Signalen wiederum,...

3.4 Sinus-Signal mit R2R-Netzwerk

3.4.1 Schaltungsentwicklung

Das nächste Projekt lässt sich mit folgender Vorüberlegung realisieren:

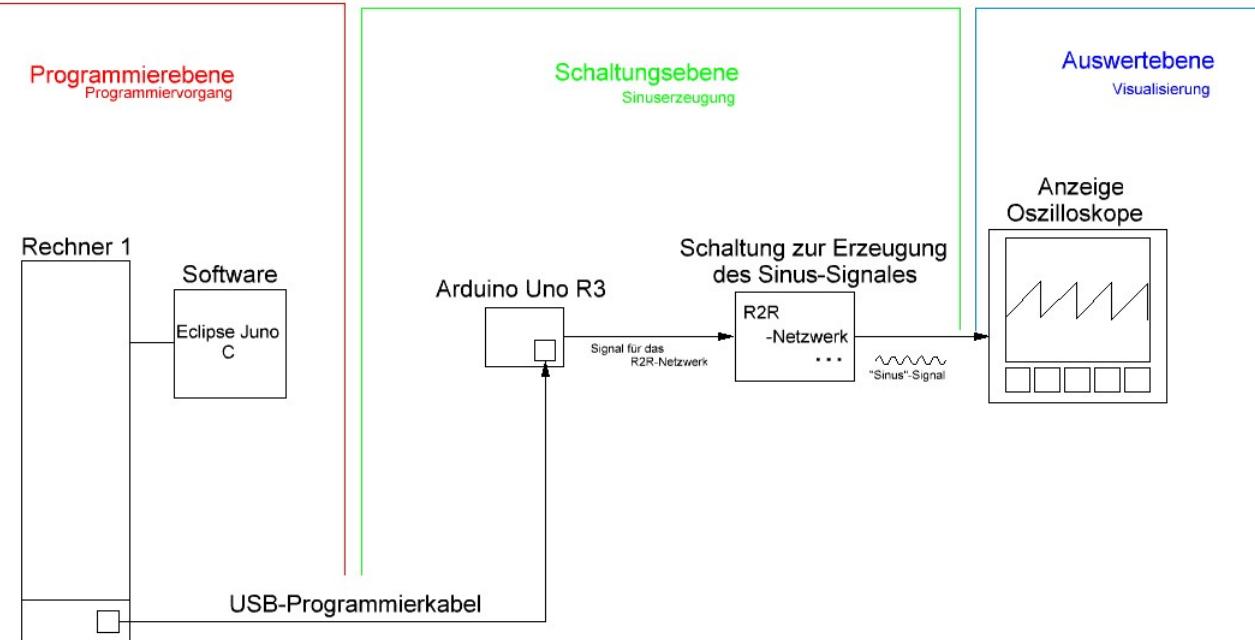


Abbildung 15: Blockschaltbild zur Erzeugung des Sinus-Signales mit R2R-Netzwerk

Nun kann mit der Entwicklung des Programmes und der zugehörigen Schaltung problemlos begonnen werden.

Ein R2R-Netzwerk besteht aus relativ vielen Widerständen, welche möglichst geringe Toleranzen aufweisen sollten, da diese so gleich wie möglich sein sollen, vor allem die für die höherwertigen Bits. Eine solche Digital-Analog-Umsetzung braucht natürlich eine Referenzspannung, welche auf keinen Fall vergessen werden darf!

Die entwickelte Schaltung hat nun folgende Form:

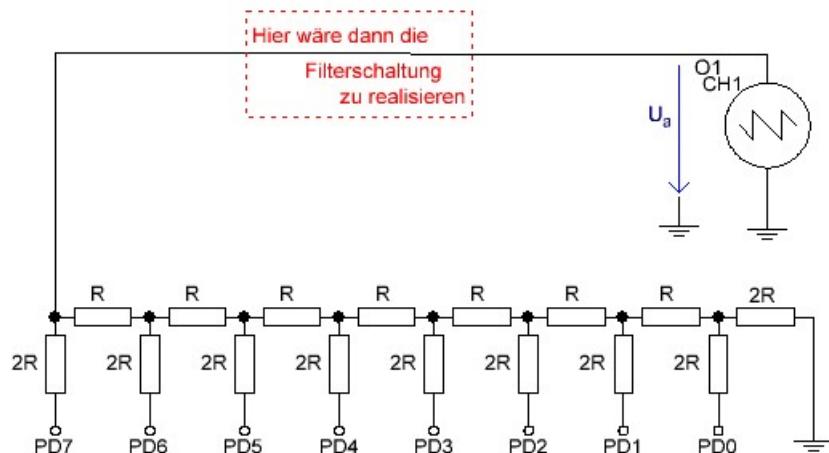


Abbildung 16: Schaltung zur Erzeugung des Sinus-Signales

Die Wahl der Widerstände sollte im Bereich von $1k\Omega < R < 1M\Omega$ liegen, wir entschieden uns für $R = 10k\Omega \rightarrow 2R = 20k\Omega = 2 \cdot 10k\Omega$.

Es darf natürlich wieder nicht auf die Referenzspannung, Frequenz- und der Amplitudenverstellungsschaltung vergessen werden:

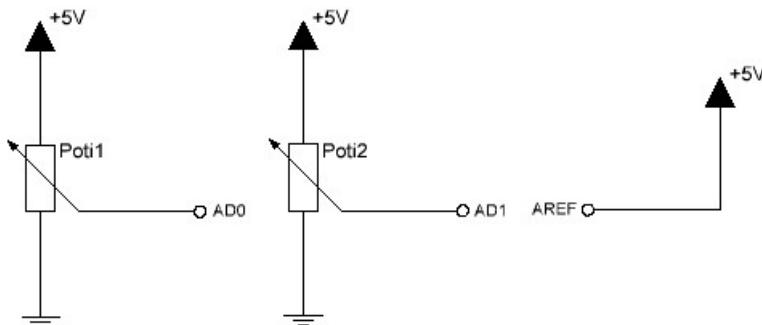


Abbildung 17: Schaltung zur Frequenz- & Amplitudenverstellung (+Referenzspannung)

Nach dieser Entwicklungsphase konnte das dazugehörige Programm mittels C geschrieben werden.

3.4.2 Programmentwicklung

```

1  /* SinusR2R.c
2   * Created on: 01.12.2014
3   * Author: Labenbacher Michael */
4 //1. "Packages" laden
5 //Bibliothek für den Prozessor laden
6 #include<avr/io.h>
7 //Includedatein für die Funktionen zur Interrupt–Verarbeitung einbinden
8 #include<avr/interrupt.h>
9 //Mathebibliothek laden
10 #include<math.h>
11 //Die Zahl PI ungefähr (= mind. 5Kommastellen) definieren
12 #define PI 3.141592654f
13 //2. Interruptroutinen & (Unterprogramme) & (Höherwertige Variablen definieren)
14 //Höherwertige Veriablen
15 volatile int icnt=0;
16 //Timer Overflow Interrupt , wird alle 1ms ausgelöst
17 //Zähler TCNT0 zählt von 6 bis 255, also 250 Schritte ,
18 //dann kommt das Timer Overflow Interrupt
19 ISR(TIMER0_OVF_vect){
20     TCNT0=6;//Startwert wieder auf 6
21     icnt++; } //bei jedem Überlauf um 1 erhöht , d.h. er zählt die ms
22 //3. Hauptprogramm
23 int main(void){
24 //3.1 Initialisierung
25     //Allgemeine Variablen
26     double Winkel=0; //Winkel vom Sinus
27     double Amplitude=1; //Amplitude vom Sinus
28     double frequenz=1; //Frequenz für den Sinus
29     double Rohwert; //ADC–Messrohwert
30     double Sinus; //Sinus (laut Einheitskreis)
31     //Die Ports für das R2R–Netzwerk festlegen als Ausgänge
32     //Alle Ausgänge am Beginn auf 0 setzen
33     PORTD=0x00;
34     //Datenrichtung (Ausgang) festlegen
35     DDRD=0xFF;
36     //ADC–Port AD0 als Eingang zur Messung am Kanal 0 festlegen (Datenrichtung)
37     DDRC&=~(1<<PC0);
38     /* Timer 0 Einstellungen*/
39     /* Normaler Betriebsmodus mit Top 0xFF */
40     TCCR0A=0;
41     /* Des weiteren wird mittels den CSxx kann der Prescaler eingestellt
42     * werden , hier clk/64 dies ergibt bei 16MHz —> 16MHz/64=250kHz */
43     TCCR0B=(1<<CS01)|(1<<CS00);
44     /* Nun muss nur noch beim Überlauf des Timers ein Overflow Interrupt
45     * ausgelöst werden , kurz: Timer Overflow Interrupt Enable 0 aktivieren */
46     TIMSK0=(1<<TOIE0);
47     //Startwert des Timers auf 6 setzen (prinzipiell optional)
48     TCNT0=6;
49     //Globale Interruptfreigabe
50     sei();
51     /* Nun muss noch der ADC eingeschalten werden mittels ADEN=Enable und
52     * mit der tiefsten Frequenz einfach mal anfangen */
53     ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);

```

```

1 //3.2 Endlosschleife
2 while(1){
3     //ADC-Messung am Kanal 0 (AD0) ((Frequenz einlesen))
4     ADMUX&=~(1<<MUX0);
5     ADCSRA|=(1<<ADSC); //Start der Messung
6     while((ADCSRA & (1<<ADSC))!=0){ //Messung läuft (warten bis diese fertig ist)
7 }
8     //Der gemessene Wert steht im Register ADCW (oder --> ADCL & ADCH)
9     Rohwert=ADCW;
10    //Daraus kann die Frequenz berechnet werden,
11    //welche im Bereich von 0–10Hz liegen soll
12    frequenz=Rohwert/1023*10;
13    //Hier soll keine Frequenz existieren welche Null ist,
14    //da immer ein Sinus entstehen soll
15    if(frequenz<=0){frequenz=1;}
16    //ADC-Messung am Kanal 1 (AD0) ((Amplitude einlesen))
17    ADMUX|=(1<<MUX0);
18    ADCSRA|=(1<<ADSC); //Start der Messung
19    while((ADCSRA & (1<<ADSC))!=0){ //Messung läuft (warten bis diese fertig ist)
20 }
21    //Der gemessene Wert steht im Register ADCW (oder --> ADCL & ADCH)
22    Rohwert=ADCW;
23    //Daraus kann die Amplitude gesetzt werden
24    Amplitude=Rohwert;
25    //Hier soll keine Amplitude existieren welche Null ist,
26    //da immer ein Sinus entstehen soll
27    if(Amplitude<=0){Amplitude=1;}
28    //Berechnungstakt
29    if(icnt>=10){ //nach 10ms -->
30        icnt=0; //Overflow-Zähler
31        //sinus berechnen (eingelesener Amplitudenwert wieder
32        //durch 1023 und mal sin(Winkel)
33        Sinus=Amplitude*sin(Winkel)/1023;
34        //Winkel berechnen (100 mal --> 1s, darum durch 100 dividieren)
35        Winkel+=2*frequenz*PI/100;
36        //Beim Erreichen von 360 Grad soll rückgesetzt werden, zur Sicherheit
37        if(Winkel==2*PI){Winkel=0;}
38    }
39    //Ausgabe auf R2R-Netzwerk
40    PORTD=Sinus*127+128;
41 }
42 }
```

3.4.3 Schaltungsaufbau, Messvorgang & Auswertung

Nach der Entwicklung folgt nun der Aufbau der Schaltung Abb. 16. Somit kann dann mit dem Oszilloskope das durch das R2R-Netzwerk gebildete Sinus-Signal mittels dem Oszilloskope erfasst und mit der Simulation verglichen werden. (Die Potentiometerstellung bezieht sich auf das Poti 1 und die Hertz wurden mittels dem Poti 2 eingestellt)

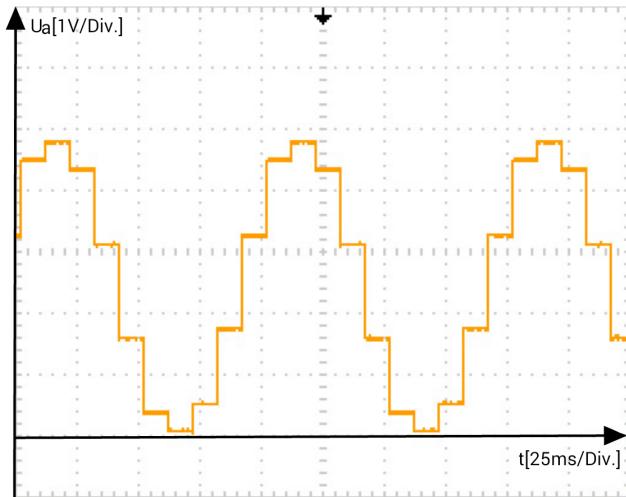


Abbildung 18: Oszilloskopeaufnahme
des Sinus-Signals 10Hz
Potentiometerstellung=100%

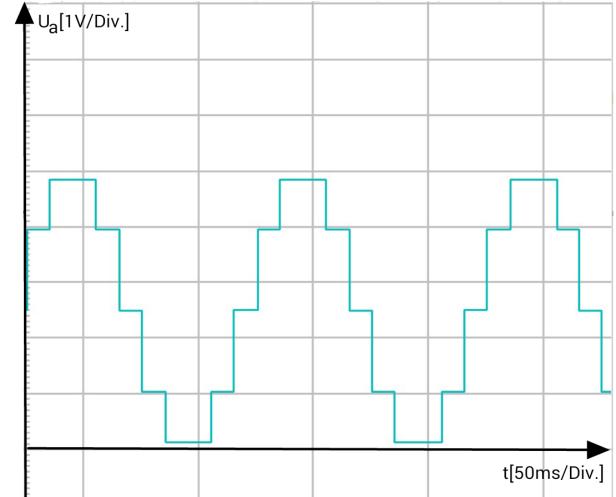


Abbildung 19: Proteussimulation
des Sinus-Signals 10Hz
Potentiometerstellung=100%

Man erkennt den sinusförmigen Verlauf des Signales, jedoch führt das Programm nur 10 Berechnungen pro Periode bei 10Hz durch. Diese sind in beiden Bildern 19 & 18 deutlich ersichtlich.

Des Weiteren wurde analog zur 1. Messung, das selbe mit verringter Frequenz und Amplitude als Kontrolle der Funktionalität durchgeführt.

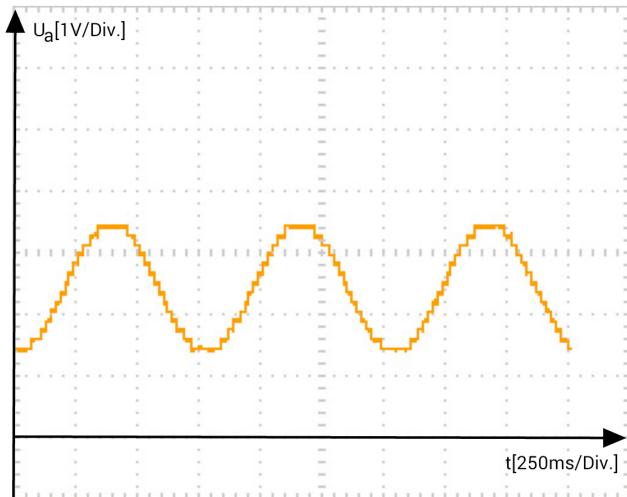


Abbildung 20: Oszilloskopeaufnahme
des Sinus-Signals 1,33Hz
Potentiometerstellung=20%

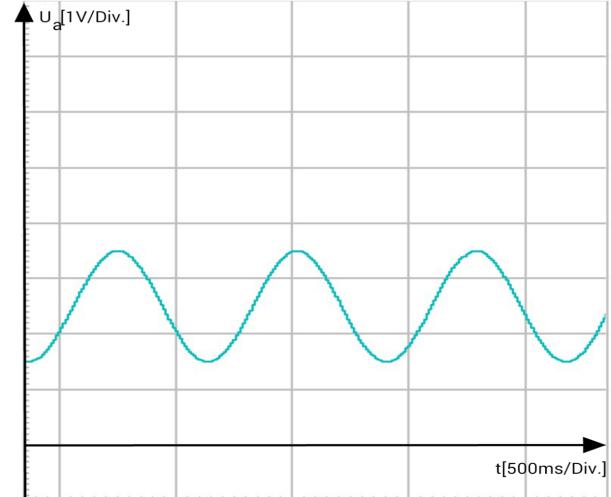


Abbildung 21: Proteussimulation
des Sinus-Signals 1,33Hz
Potentiometerstellung=20%

Dieses Mal lässt sich das Sinus-Signal schon genauer erkennen, da die Frequenz verringert wurde und somit mehr Berechnungsschritte pro Periode durchgeführt werden, was wiederum in den Abbildungen 21 & 20 erkennbar ist.

3.4.4 Messbericht

Hier einmal ein kleines Foto über den Aufbau auf den Steckbrettern:

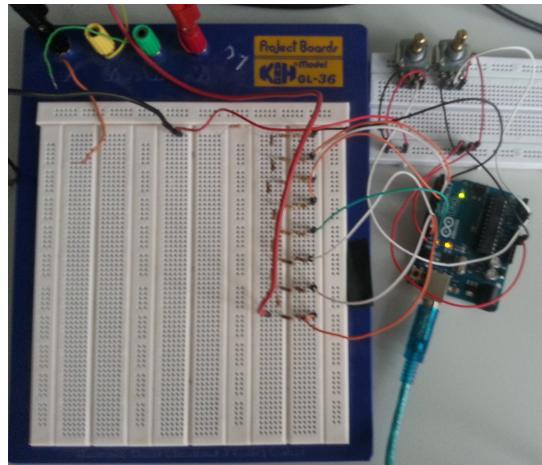


Abbildung 22: Aufbau des R2R-Netzwerkes

Um auch bei größeren Frequenzen einen “schönen” Sinus zu erzeugen müssen mehr Berechnungsschritte, was mittels dem Programm leicht zu realisieren ist, durchgeführt werden. Um das Sinus-Signal noch besser zu erzeugen, müsste ein Filter, wie in Abb. 16, eingebaut werden.

Der große Vorteil dieser Realisierung, gegenüber von zB. Filterschaltungen, ist die hohe Geschwindigkeit welche erzielt werden kann, der Nachteil jedoch ist die hohe Anzahl an möglichst genauen Bauteilen welche benötigt werden.

3.5 3 ~ Sinus-Signal mit aktivem Tiefpass 2. Ordnung

3.5.1 Schaltungsentwicklung

Nach den bisherigen Übungen, folgt nun die letzte, bei der ein 3~Sinus-Signal mit bis zu 20Hz zu erzeugen ist, dabei wurde folgendermaßen vorgegangen:

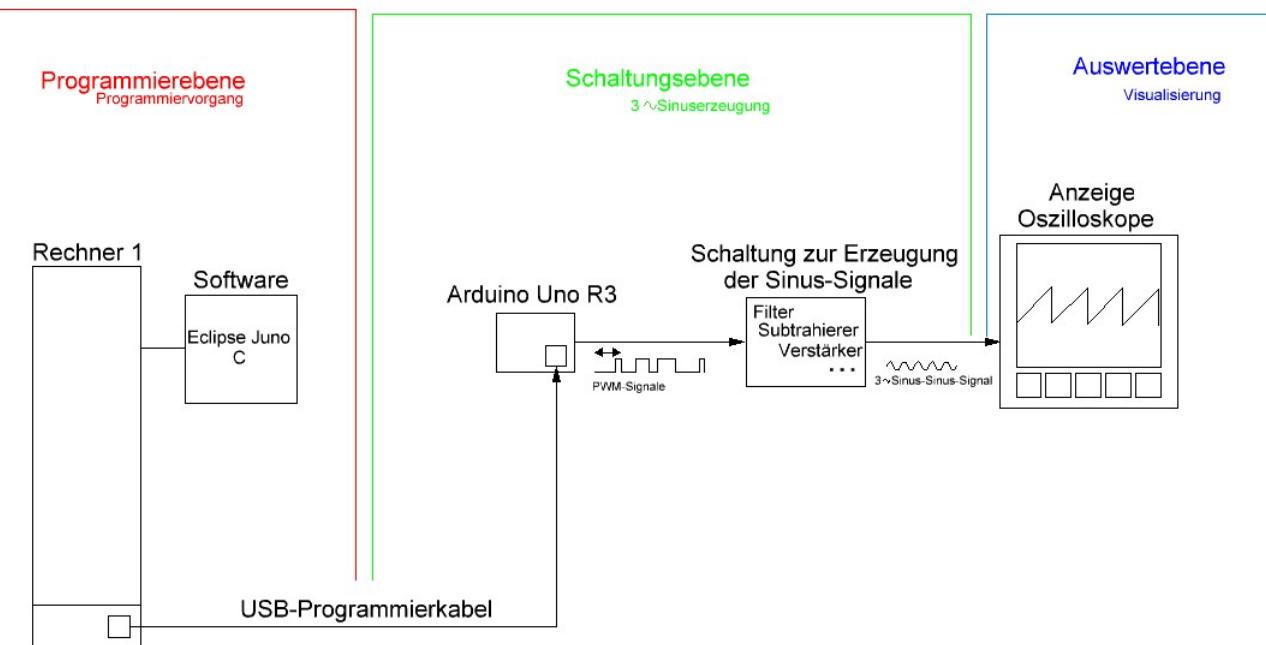


Abbildung 23: Blockschaltbild zur Erzeugung des 3~Sinus-Signales

Für die Erzeugung einer analogen Spannung aus einem PWM-Signal wurde im Prinzip gleich wie im Kapitel 3.3 vorgegangen.

Frequenz berechnen

Das nachfolgende Programm im Abschnitt 3.5.2 erzeugt an den Ports PD6, PD5 und PB1 jeweils ein pulsweitenmoduliertes Signal, welche in eine analoge Spannung umgewandelt werden soll.

Das entworfene Programm weißt folgende Taktfrequenz an den Ausgängen der Timer 0/1, bei 16MHz CPU-Frequenz und einem Prescaler von 8 bei 8Bit Fast PWM, auf:

$$f_{OC0} = f_{OC1} = \frac{16000000}{8 \cdot (255 + 1)} = 7812,5 \text{ Hz}$$

f_{OC0} *Taktfrequenz am Ausgang des Timer0*
 f_{OC1} *Taktfrequenz am Ausgang des Timer1*

Diesesmal kurz gesagt: der Wert im OCR0A-Register erhöht sich laut Programm bei jeweils 21 Interrupts um maximal $128 \cdot \sin(2 \cdot \frac{380}{372}) = 14,..$, somit folgt:

$$f_{\ddot{A}\ddot{n}d\ddot{e}rung0} = f_{\ddot{A}\ddot{n}d\ddot{e}rung1} = \frac{7812,5}{21 \cdot 14} = 26,57 \text{ Hz}$$

$f_{\ddot{A}\ddot{n}d\ddot{e}rung0}$ *Frequenz mit der sich der Wert im OCR0A/OCR0B – Register erhöht*
 $f_{\ddot{A}\ddot{n}d\ddot{e}rung1}$ *Frequenz mit der sich der Wert im OCR1A – Register erhöht*

Mit diesen Erkenntnissen kann der Filter wieder berechnet werden.

Filterberechnung

Hierbei wird jeweils (also 3 mal) die Schaltung Abb. 8 verwendet. Die Änderungsfrequenz wird nun der Kreisfrequenz gleich gesetzt und erneut beliebige Kondensatoren im Bereich von 10-1000nF gewählt. zB: $C_2 = 100\text{nF}$ und $C_1 = 220\text{nF}$. Daraus ergibt sich das Produkt der Widerstände mittels den Formel 14 und 13:

$$R_1 R_2 = \frac{1}{(2 \cdot \pi \cdot 26,57\text{Hz})^2 \cdot 100\text{nF} \cdot 220\text{nF}} = 1630\Omega^2$$

Da sich der Wert im OCR0A-Register um hier max. 14 erhöht, muss davon $\frac{1}{14}$ stel, also $116,3\text{M}\Omega$ verwendet werden. Somit kann zB. $R_2 = R_1 = 10\text{k}\Omega$ genommen werden, da dessen Produkt $100\text{M}\Omega$ ergibt und im Toleranzbereich der verwendeten Bauteile liegt.

Subtrahierer berechnen

Hier gibt es nicht viel zu sagen, da einfach der Subtrahierer Abb. 9 im Kapitel 3.3.1 verwendet wurde mit den gleichen Bauteilen.

→ $R_3 = R_4 = R_5 = R_6 = 10\text{k}\Omega$.

Verstärker berechnen

Auch dafür wurde der gleiche Verstärker Abb. 3 zur Verstellung der Amplitude hergenommen wurde. Also das Poti 1 jeweils wurde erneut mit $100\text{k}\Omega$ gewählt.

Die Striche über den Widerständen im nachfolgendem Gesamtbild bedeuten nur, dass diese jeweils gleich groß dimensioniert wurden, sprich also $R_1 = R'_1 = R''_1, R_2 = R'_2 = R''_2, \dots$

Des Weiteren muss auch diese Schaltung wiederrum existieren:

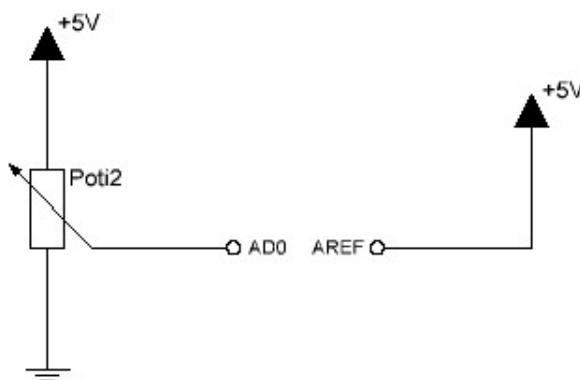


Abbildung 24: Schaltung zur Frequenzverstellung (+Referenzspannung)

Das Poti 2 wurde genau so wie im Abschnitt 3.2.1 dimensioniert, also mit $10\text{k}\Omega$.

Die gesamte Schaltung hat nun folgende Form:

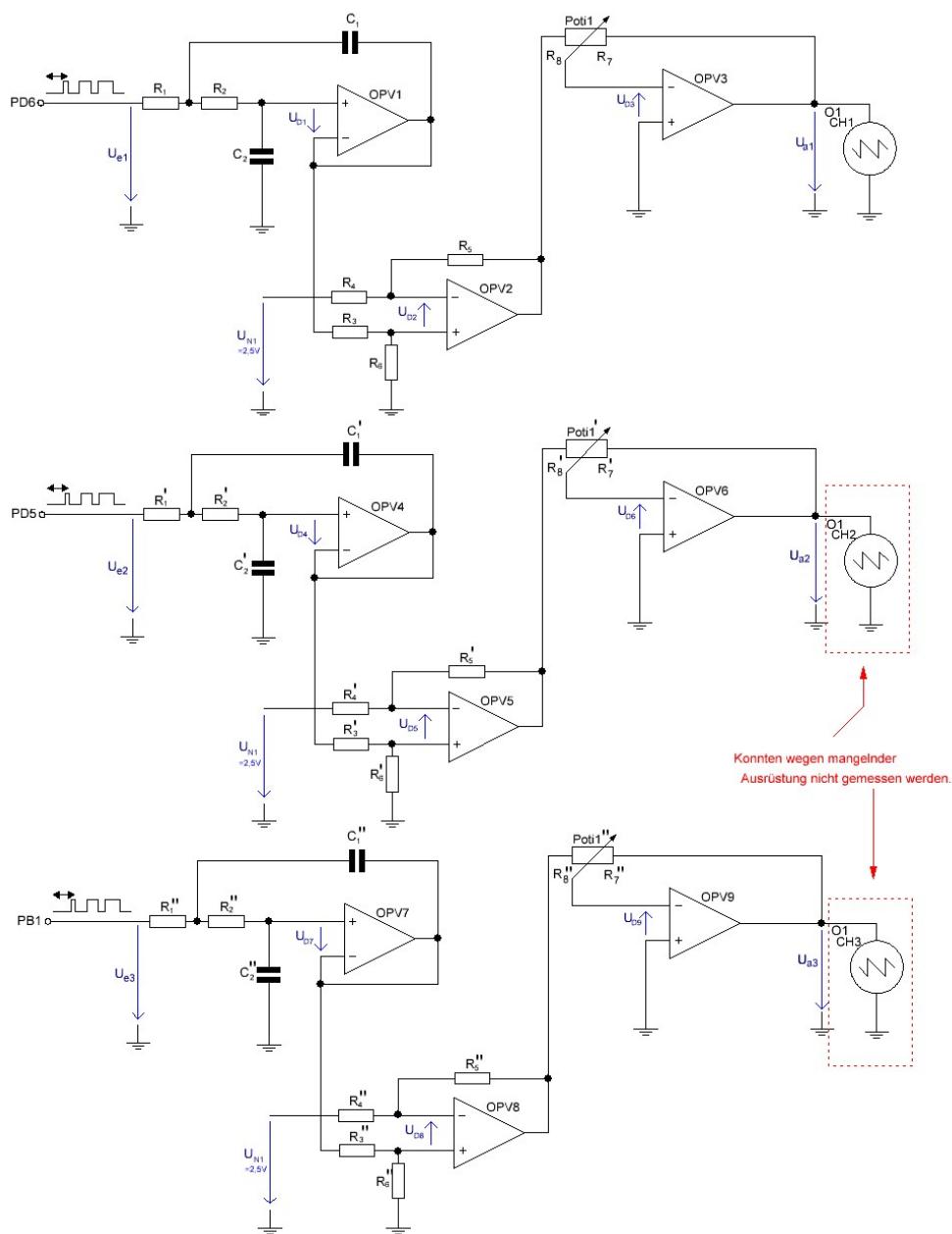


Abbildung 25: Schaltung zur Erzeugung des 3~Sinus-Signales

3.5.2 Programmierung

Folgendes Programm wurde zur Realisierung des 3~Sinus Signales entworfen:

```

1  /* DreiphasenSinus.c
2   * Created on: 01.12.2014
3   * Author: Labenbacher Michael */
4 // 1. "Packages" laden
5 // Bibliothek für den Prozessor laden
6 #include<avr/io.h>
7 // Includedatein für die Funktionen zur Interrupt-Verarbeitung einbinden
8 #include<avr/interrupt.h>
9 // Mathebibliothek laden
10 #include<math.h>
11 // Die Zahl PI ungefähr (= mind. 5Kommastellen) definieren
12 #define PI 3.141592654 f
13 // 2. Interruptroutinen&(Unterprogramme)&(Höherwertige Variablen definieren)
14 // Höherwertige Variablen
15 volatile int icnt=0;//Overflow-Zähler
16 //Overflow-Interrupts des Timer 0
17 ISR(TIMER0_OVF_vect){
18     icnt++;}// Jeder Overflow Interrupt soll gezählt werden
19 //Overflow-Interrupt des Timer 1 (wird eigentlich nicht benötigt
20 ISR(TIMER1_OVF_vect){}

```

```

1 //3. Hauptprogramm
2 int main(void){
3     //3.1 Initialisierung
4     double frequenz=50;      //Frequenz
5     double Rohwert;          //Rohwert der ADC-Messung
6     double WinkelOC0A=0;     //Winkel für den 1. Sinus
7     double sinusOC0A=0;      //Sinuswert des 1. Sinus
8     double WinkelOC0B=0;     //Winkel für den 2. Sinus
9     double sinusOC0B=0;      //Sinuswert des 2. Sinus
10    double WinkelOC1A=0;     //Winkel für den 3. Sinus
11    double sinusOC1A=0;      //Sinuswert des 3. Sinus
12    int startOC0B=0;         //Hilfsvariable für den Startwert des 2. Sinus
13    int startOC1A=0;         //Hilfsvariable für den Startwert des 3. Sinus
14    PORTD=(1<<PD6)|(1<<PD5); //Ausgang am Beginn auf High
15    DDRD= (1<<PD6)|(1<<PD5); //Datenrichtungen als Ausgang von OC0A/B festlegen
16    PORTB=(1<<PB1);        //Ausgang auf Beginn auf High
17    DDRB= (1<<PB1);        //Datenrichtung als Ausgang von OC1A festlegen
18    DDRC=0; //Datenrichtung für den Eingang PC0=ADC0 festlegen (gleich für alle)
19    /* Timer 0 Einstellungen*/
20    /* Mittels den WGMxx kann die Betriebsart gewählt werden,
21     * hier Fast PWM mit Top 0xFF PWM-Signal an OCR0A non-inverted */
22    TCCR0A=(1<<COM0A1)|(1<<COM0B1)|(1<<WGM01)|(1<<WGM00);
23    /* Des weiteren wird mittels den CSxx kann der Prescaler eingestellt
24     * werden, hier clk/8 dies ergibt bei 16MHz --> 16MHz/8=2M */
25    TCCR0B=(1<<CS01);
26    /* Nun muss nur noch beim Überlauf des Timers ein Overflow Interrupt
27     * ausgelöst werden, kurz: Timer Overflow Interrupt Enable 0 aktivieren */
28    TIMSK0=(1<<TOIE0);
29    /* Timer 1 Einstellungen*/
30    /* Mittels den WGMxx kann die Betriebsart gewählt werden, hier
31     * Fast PWM mit Top 0xFF PWM-Signal an OCR0A non-inverted */
32    TCCR1A=(1<<COM1A1)|(1<<WGM10);
33    /* Des weiteren wird mittels den CSxx kann der Prescaler eingestellt
34     * werden, hier clk/8 dies ergibt bei 16MHz --> 16MHz/8=2M */
35    TCCR1B=(1<<WGM12)|(1<<CS11);
36    /* Nun muss nur noch beim Überlauf des Timers ein Overflow Interrupt
37     * ausgelöst werden, kurz: Timer Overflow Interrupt Enable 0 aktivieren */
38    TIMSK1=(1<<TOIE1);
39    //Globale Interruptfreigabe
40    sei();
41    /* Nun muss noch der ADC eingeschalten werden mittels ADEN=Enable und
42     * mit der tiefsten Frequenz einfach mal anfangen */
43    ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);

```

```

1 //3.2 Endlosschleife
2 while(1){ //es müssen 2MHz/256 /(zB.:)21=372 Interrupts vergehen
3     // (=fast gerade Zahl), bis der nächste Wert gesetzt wird
4     if(icnt >=21){
5         icnt=0;//Interrupt-Zähler wieder rücksetzen
6         double PLUS=2*PI*(frequenz/372); //Hilfsvariable zum Erhöhen
7         /*Sinus 1 auf OC0A "1. Halbwelle"*/
8         if(sinusOC0A>0){OCR0A=sinusOC0A*128+127;}
9         /*Sinus 1 auf OC0A "2. Halbwelle"*/
10        else{OCR0A=sinusOC0A*128-127;}
11        /*Der dazugehörige Winkel wird jedesmal erhöht*/
12        WinkelOC0A+=PLUS;
13        /*Der Sinus-Wert berechnet sich aus dem Winkel*/
14        sinusOC0A=sin(WinkelOC0A);
15        /*Falls der Winkel eine Kreisumdrehung überschreitet
16         * wird er rückgesetzt (optional)*/
17        if(WinkelOC0A>=2*PI){WinkelOC0A=0;}
18        /*Sinus 2 auf OC0B soll 120 Grad später starten*/
19        if((WinkelOC0A>=(2*PI/3)) || (startOC0B>0)) {
20            startOC0B=1; //Selbsthaltung
21            /*Sinus 2 auf OC0B "1. Halbwelle"*/
22            if(sinusOC0B>0){OCR0B=sinusOC0B*128+127;}
23            /*Sinus 2 auf OC0B "2. Halbwelle"*/
24            else{OCR0B=sinusOC0B*128-127;}
25            /*Der dazugehörige Winkel wird jedesmal erhöht*/
26            WinkelOC0B+=PLUS;
27            /*Der Sinus-Wert berechnet sich aus dem Winkel*/
28            sinusOC0B=sin(WinkelOC0B);
29            /*Falls der Winkel eine Kreisumdrehung überschreitet
30             * wird er rückgesetzt (optional)*/
31            if(WinkelOC0B>=2*PI){WinkelOC0B=0;}
32        }
33        /*Sinus 3 auf OC1A soll 240 Grad später starten*/
34        if((WinkelOC0A>=(4*PI/3))||(startOC1A>0)){
35            startOC1A=1;//Selbsthaltung
36            /*Sinus 3 auf OC1A "1. Halbwelle"*/
37            if(sinusOC1A>0){OCR1AL=sinusOC1A*128+127;}
38            /*Sinus 3 auf OC1A "2. Halbwelle"*/
39            else{OCR1AL=sinusOC1A*128-127;}
40            /*Der dazugehörige Winkel wird jedesmal erhöht*/
41            WinkelOC1A+=PLUS;
42            /*Der Sinus-Wert berechnet sich aus dem Winkel*/
43            sinusOC1A=sin(WinkelOC1A);
44            /*Falls der Winkel eine Kreisumdrehung überschreitet
45             * wird er rückgesetzt (optional)*/
46            if(WinkelOC1A>=2*PI){WinkelOC1A=0;}
47        }
48    } //Messung für den Kanal 0 ADC-Wandlung starten (Bit ADSC auf 1 setzen)
49    ADCSRA |= (1<<ADSC);
50    // warten bis ADC-Wandlung fertig (Bit ADSC ist wieder 0)
51    while (ADCSRA & (1<<ADSC)) {}
52    Rohwert=ADCW; //Rohwert der Messung
53    //Maximal 20Hz —> Messung mal 20 rechnen und durch "10Bit" dividieren
54    frequenz=Rohwert/1023*20;
55}
}

```

3.5.3 Schaltungsaufbau, Messvorgang & Auswertung

Nachdem diese Schaltung Abb. 25 vollständig aufgebaut wurde, und das Programm in den Arduino Uno R3 hineingeladen wurde, konnte mit dem Messen begonnen werden. (Die angegebenen Potentiometerstellungen beziehen sich hier auf die Amplitude, also dem Poti 1) Leider stand unserer Laborgruppe nur ein Oszilloskopkabel zur Verfügung und somit konnten wir nur jeweils eine Phase, der drei einwandfrei funktionierenden, zur Anzeige bringen ☺:

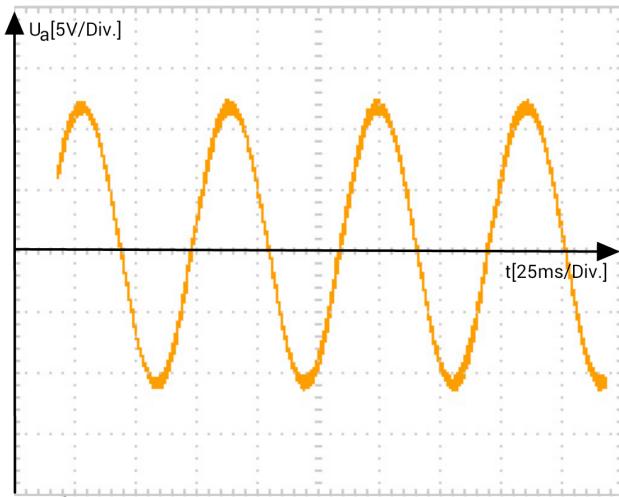


Abbildung 26: Oszilloskopeaufnahme
"3~Sinus-Signals" 20Hz
Potentiometerstellung=20%

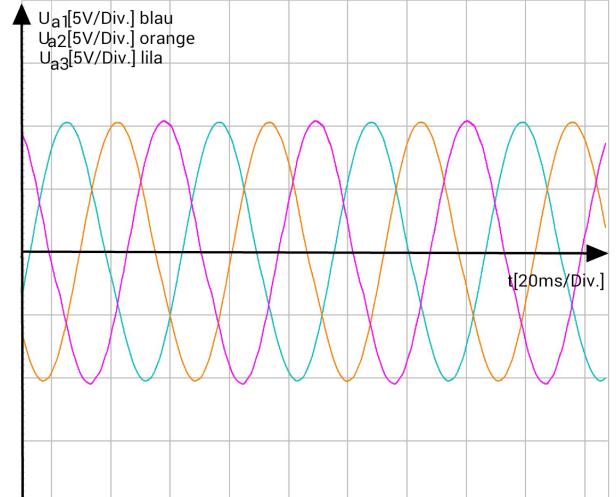


Abbildung 27: Proteussimulation
3~Sinus-Signals 20Hz
Potentiometerstellung=20%

Man erkennt, prinzipiell der sinusförmige Verlauf einwandfrei funktioniert, jedoch ob die Phasenverschiebung in der Realität auch genauso wie in der Simulation vorliegt, konnte leider nicht überprüft werden.

4 Resümee

Im Allgemeinen kann man sagen, dass alle Schaltungen & Programme funktionierten, vor allem beim R2R-Netzwerk ließ sich erkennen, dass das Programm für noch viel größere Frequenzen erweitert werden kann.

Des Weiteren fanden wir heraus, dass die einzelnen Filterschaltungen mit steigender Frequenz immer schwieriger zu realisieren sind, und oft ein einfacher RC-Filter nicht mehr ausreichend ist.

Zusammenfassend lässt sich sagen, dass mit kleinen Programmänderungen große Auswirkungen auf das Verhalten der Signale erzielt werden können.

5 Literatur- und Quellenverzeichnis

[1] <http://de.academic.ru/>

[2] <http://www.krucker.ch/>