JOHANNES KEPLER
UNIVERSITÄT LINZ

Protokoll

Praktikum Digitale Schaltungen
LVA 336.003/19/20, SS20

Vorname	Nachname	Matrikelnummer
Michael	Rynkiewicz	k11736476

Ich bestätige hiermit, dieses Protokoll eigenhändig angefertigt zu haben. Plagiierte Protokolle werden nicht bewertet und bewirken einen negativen Abschluss des Praktikums mit entsprechenden Konsequenzen.

2020-10-08 ,
Datum


Unterschrift

Bewertung:

	A.1	A.2	A.3	A.4	A.5	A.6	A.7	Form	$\sum$
Max.:	9	9	9	9	9	9	9	7	70
Pkt.:									

Inhaltsverzeichnis

1 Einleitung

Das Wissen um die Funktionsweise eines Mikrocontrollers ist wesentlicher Bestandteil eines Informatik-Studiums. Im Zuge Dieses werden in diesem Protokoll verschiedene Experimente beschrieben und evaluiert. Insgesamt werden sieben Experimente begutachtet und diskutiert, welche im Zeitraum vom 28.September 2020 und 2.Oktober 2020 durchgeführt wurden.

2 Verwendete Materialien

Jede Sektion gibt die in ihr verwendeten Materialien mitsamt ihrer Stückzahl an. In dieser Sektion werden alle Materialien im Allgemeinen gelistet und beschrieben.

2.1 LED

Eine Leuchtdiode (LED) ist ein Bauelement der Elektronik. Wird sie von elektrischen Strom durchflossen beginnt sie Licht auszustrahlen. Die Wellenlänge, d.h., die Farbe des Lichts sowie ob es für das menschliche Auge sichtbar ist oder nicht, hängt von den benutzten Materialien im inneren der LED ab. Die in der LED verwendeten Materialien sind für dieses Protokoll nicht weiter von Bedeutung. Für die beschriebenen Aufgaben wurden die folgenden LEDs verwendet.

Tabelle 1: LEDs - Farben, Flussspannung, Maximalstrom [5]

Farbe	Durchflussspannung	Maximalstrom
Rot	1,6V - 2,2V	20mA
Gelb	1,9V - 2,5V	20mA
Grün	1,9V - 2,5V	20mA

2.2 Widerstände

Widerstände werden verwendet, um einen Spannungsabfall in einem Stromkreis zu verursachen. Damit kann die Stromstärke in einem Stromkreis begrenzt beziehungsweise verringert werden. Oft wird bei der Verwendung von

Tabelle 2: Farbcodierung von Widerständen

Farbe	1.Ring (10er Stelle)	2.Ring (1er Stelle)	3.Ring (Multiplikator)	4.Ring (Toleranz)
Silber	-	-	-	±10%
Gold	-	-	0.1	±5%
Braun	1	0	10	±1%
Rot	2	2	100	±2%
Gelb	4	4	10 000	-
Grün	5	5	100 000	±0.5%
Blau	6	6	1 000 000	±0.25%
Violett	7	7	10 000 000	±0.1%
Grau	8	8	100 000 000	±0.05%
Weiß	9	9	1 000 000 000	-

LEDs ein sogenannter Vorwiederstand verwendet, um die Stromstärke soweit abzusenken, dass die LED nicht beschädigt wird.

In den Aufgaben wurden Widerstände der sogenannten E6-Reihe verwendet. E-Reihen sind normierte Widerstandsgrößen, wobei die Zahl die Stufen zwischen den Potenzen angibt. D.h., bei der E6-Reihe sind sechs verschiedene Widerstandsgrößen zwischen 10Ω und 100Ω, zwischen 100Ω und 1000Ω, u.s.w. bis zum oberen Limit von $10M\Omega$.

Die verwendeten Widerstände sind Farbcodiert, um sie voneinander unterscheiden zu können. Die Farbkodierung von Widerständen wird in den Aufgaben, in denen sie verwendet werden, angegeben. Die Bedeutung der Codierung der verwendeten Widerstände kann in Tab. 2 abgelesen werden.

2.3 Mikrocontroller [1]

Mikrocontroller werden verwendet, um komplexere Logik in eine elektronische Schaltung zu integrieren. Die in diesem Protokoll verwendete Mikrocontroller sind Mikrocontroller vom Typ Arduino Uno Rev 3. Diese Mikrocontroller besitzen 14 digitale Pins, an denen eine Spannung von entweder 0V oder 5V angelegt oder ausgegeben werden kann. Des Weiteren besitzt es sechs analoge Pins, welche auch auf Spannungsschwankungen reagieren können. Weitere wichtige Pins sind In-Pins, welche eine Spannung von 5V liefern können und Out-Pins, welche eine Verbindung zur Erdung herstellen.

2.4 Logikbausteine

Um verschiedene Logikgatter in Schaltungen verwenden zu können, ohne komplizierte Schaltkreise bauen zu müssen, werden Logikbausteine verwenden. Diese Bausteine implementieren verschiedene Gatter. D.h., bei zwei Eingängen a und b, wird durch das Anwenden der Funktion, die der Baustein implementiert, der Ausgang y erzeugt. Diese Funktionen bilden logische Operationen ab. Die Erklärung von logischen Funktionen würden über den Umfang dieses Protokolls hinausgehen und werden daher nicht weiter erklärt. In Tabelle 3 sind die verwendeten Bausteine und die Funktionen die sie implementieren gelistet.

Tabelle 3: Bausteine und deren logische Funktionen

Bausteinbezeichnung	Funktion
74HC00	NAND
74HC02	NOR
74HC08	AND
74HC32	OR
74HC86	XOR

2.5 Kondensatoren [3]

Kondensatoren nehmen elektrische Ladungen auf und können sie zu einem späteren Zeitpunkt wieder abgeben. Die Kapazität wird mit der Einheit Farad angegeben. Die verwendeten Kondensatoren sind gepolte Elektrolytkondensatoren. Durch die Polung können die Kondensatoren beschädigt werden, sollten sie falsch verbaut werden.

2.6 Motortreiber

Motortreiber werden verwendet, um Gleichstrommotoren zu betreiben. Der Treiber legt eine Versorgungsspannung an den Motor an, wenn ein Signal an einem seiner Eingangspins anliegt. Der hier verwendete Treiber besitzt zwei Eingänge pro Motor und kann zwei Motoren betreiben. Je nach verwendeten Eingang wird der Motor entweder in oder gegen den Uhrzeigersinn betrieben.

2.7 Ultraschallsensor

Ultraschallsensoren verwenden Ultraschall um die Entfernung zu Objekten zu messen. Es wird zuerst ein Ultraschallsignal ausgesendet und die Zeit gemessen, wie lange es dauert, bis ein Echo empfangen wird. Durch die bekannte Geschwindigkeit von Schall in Luft kann dann die Entfernung gemessen werden.

2.8 7-Segment Anzeige

Eine 7-Segment Anzeige verwendet sieben Segmente, mit LEDs, um Zahlen anzuzeigen. Die einzelnen Segmente können über Eingangspins angesprochen werden. Des Weiteren enthält die verwendete Anzeige einen Dezimalpunkt, um Nachkommastellen anzeigen zu können, sollten mehrere Anzeigen verwendet werden.

3 Begriffe

3.1 Pulswellenmodulation (PWM)

Bei der Pulswellenmodulation wird ein Signal mit hoher Frequenz ein- und ausgeschaltet. Dadurch kann die durchschnittliche Leistung reguliert werden. Dies ermöglicht z.B. das Dimmen einer LED oder das Steuern der Geschwindigkeit eines Motors.

3.2 PullUp

Ein PullUp-Input ist eine Schaltungstechnik, mit der ein Taster an einen Pin eines Mikrocontroller geschaltet werden kann. Durch das Verwenden eines Widerstands zwischen Taster und Versorgungsspannung, um den Mikrocontroller vor hohen Spannungen zu schützen.

3.3 Baudrate

Die Baudrate gibt an, wie viele Symbole pro Sekunde über eine Schnittstelle übertragen werden können. Im Fall des verwendeten Mikrocontrollers ist ein Symbol ein Bit. Die Baudrate ist hier also gleichzusetzen mit der Bitübertragungsrate.

3.4 Wahrheitstabelle

Eine Wahrheitstabelle wird verwendet, um logische Ausdrücke auszuwerten. Einige logische Operationen wie AND oder OR sind vordefiniert. Mittels Wahrheitstabellen können komplexere Verknüpfungen aufgelöst und ausgewertet werden.

4 Aufgabe 1 - Ampelsteuerung

Es soll eine Ampelsteuerung implementiert und getestet werden. Die Ampel wird mithilfe von drei LEDs, in den Farben Rot, Gelb und Grün, aufgebaut. Weiters soll die Steuerung folgende Funktionsweise implementieren:

Phase 1 soll die Ampel auf Rot setzen. D.h., die rote LED wird eingeschaltet. Dieser Zustand soll vier Sekunden lang gehalten werden.

Phase 2 soll zusätzlich zur roten LED die gelbe einschalten. Dieser Zustand soll eine Sekunde lang gehalten werden.

Phase 3 soll die rote sowie die gelbe LED ausschalten, während die Grüne eingeschaltet wird. Dieser Zustand soll vier Selunden lang gehalten werden.

Phase 4 soll die grüne LED ausschalten während die Gelbe eingeschaltet wird. Dieser Zustand soll eine Sekunde lang gehalten werden. Nach Ablauf der vier Sekunden soll die grüne LED erlöschen und der Ablauf bei Phase 1 neu gestartet werden.

4.1 Materialien

Tabelle 4: Aufgabe 1 - Verwendete Materialien

Bezeichnung	Eigenschaften	Menge
Widerstand	150Ω	3
	Braun - Grün - Braun - Gold	
LED	Rot	1
LED	Gelb	1
LED	Grün	1
Mikrocontroller	Arduino Uno R3	1

4.2 Vorbereitung

Für den Schaltkreis müssen die Vorwiederstände für die LEDs berechnet werden. Folgende Angaben sind bekannt.

- Ausgangsspannung der Pins des Mikrocontrollers: $V_{out} = 5V$

- Diodenspannung der LEDs: $U_D = 2V$
- Diodenstrom der LEDs: $I_D = 15mA$

Zur Berechnung wird folgender Stromkreis angenommen.

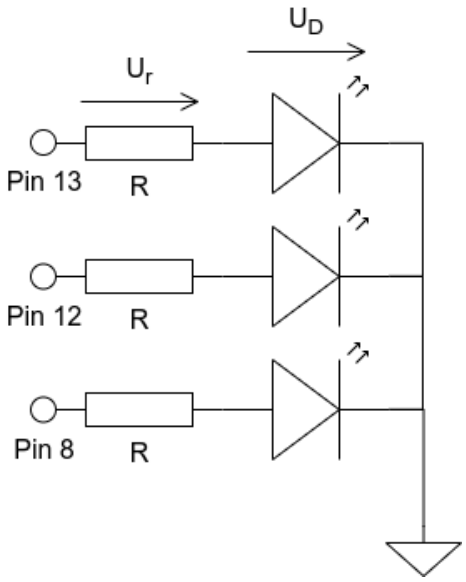

Abbildung 1: Stromkreis A1

Die Formel zur Berechnung des Stroms durch eine Diode ist bekannt.

$$I_D = \frac{1}{R_d} * (U - U_D) \tag{1}$$

Damit kann der benötigte Vorwiderstand berechnet werden.

$$\begin{aligned} I_D = \frac{1}{R_D} * (U - U_D) \Rightarrow \\ R_D = \frac{1}{I_D} * (U - U_D) \\ = \frac{1}{15mA} * (5V - 2V) \\ = 200\Omega \end{aligned} \tag{2}$$

Nachdem in der E6 Reihe keine 200Ω Widerstände vorhanden sind, wurden 150Ω gewählt. Diese Wahl erfolgt aus folgenden Überlegungen.

1. Laut angabe benötigt die LED $2V$ um zu schalten.
2. Es existieren die Widerstände 150Ω und 220Ω in der E6-Reihe
3. Bei einem Widerstand von 220Ω würden, laut Gleichung 1, $I_D = \frac{3V}{220\Omega} = 13,636mA$ durch die LED fließen.
4. Bei einem Widerstand von 150Ω würden, laut Gleichung 1, $I_D = \frac{3V}{150\Omega} = 20mA$ durch die LED fließen.

Da die Leuchtkraft einer LED von der Stromstärke abhängt und der Maximalstrom bei $20mA$ liegt, wurden 150Ω gewählt. Bei dieser Größe wird die LED bei theoretisch voller Leuchtkraft betrieben ohne die Lebensdauer markant zu verkürzen.

4.3 Praktikumsaufgabe

Der Stromkreis wurde laut Abbildung 1 implementiert. Die Implementierung wird in Abbildung 2 gezeigt.

Wie in Abbildung 2 gezeigt, wurden die Pins 13, 12 und 8 gewählt. Für die Wahl wurden die digitalen Pins herangezogen, da eine analoge Ausgabe nicht erforderlich war. Die Pins die mit einer Tilde (∼) markiert sind, sind in der Lage ein PWM-Signal zu liefern. Da ein solches Signal nicht benötigt wird, wurden auch diese ausgeschlossen. Von oben nach unten betrachte, wurden nun drei Pins ausgewählt, diese sind 13, 12, und 8. Die genannten Pins werden, im Code, als digitale Output-Pins konfiguriert.

Im nachfolgendem Text wird der Programmcode der Aufgabe erläutert. Einzelne Teile des Codes werden ausgewählt und beschrieben. Am Ende dieser Sektion befindet sich der vollständige Programmcode.

```
1 const int PIN_RED = 13;
2 const int PIN_YELLOW = 12;
3 const int PIN_GREEN = 8;
```

Listing 1: Konstanten für Aufgabe 1

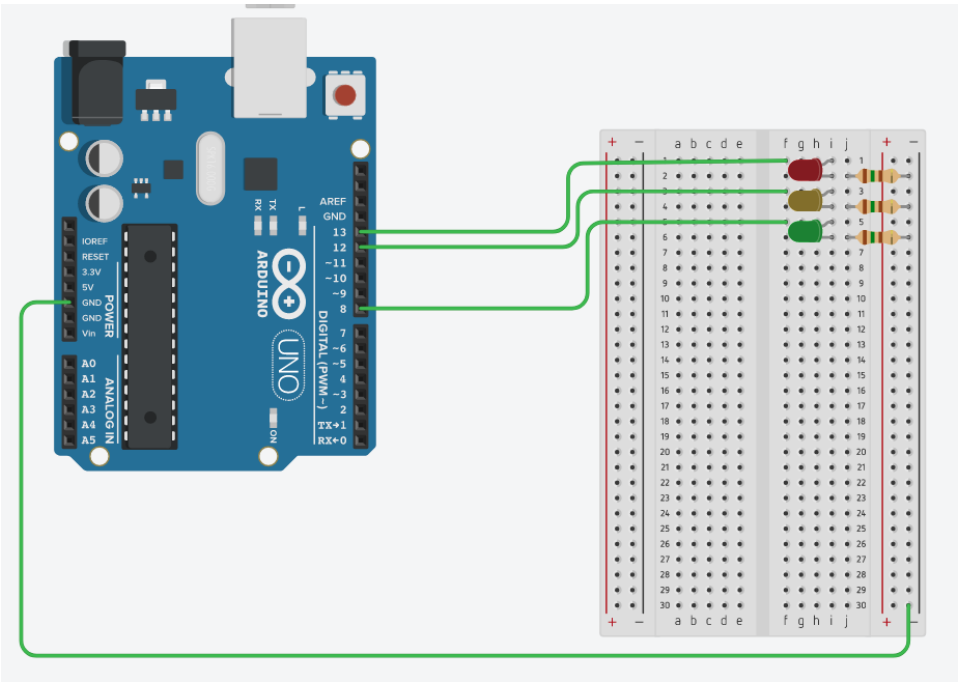

Abbildung 2: Implementiert Stromkreis von Aufgabe 1

In Listing 1 werden die Pins definiert. Durch das Anlegen von Konstanten für die Pins, kann der Code aussagekräftiger gestaltet werden. Des weiteren entsteht dadurch die Möglichkeit, andere Pins zu verwenden, ohne den gesamtem Code durchgehen zu müssen. Es reicht die Nummer an einer Stelle zu ändern.

```
void setup()
{
  pinMode(PIN_RED, OUTPUT);
  pinMode(PIN_YELLOW, OUTPUT);
  pinMode(PIN_GREEN, OUTPUT);
}
```

Listing 2: Setup für Aufgabe 1

In Listing 2 werden die Pins angelegt und konfiguriert. Wie oben beschrieben werden die Pins als Output-Pins, d.h., als Spannungsquelle, angelegt.

```
void off(int pin) {
  digitalWrite(pin, LOW);
}
```

```
4
5  void on(int pin) {
6    digitalWrite(pin, HIGH);
7  }
```

Listing 3: On- und Off Methoden für Aufgabe 1

In Listing 3 werden die Pins entweder ausgeschalten (LOW) oder eingeschalten (HIGH). Durch das Verwenden dieser Methoden ist es einfacher, die Phasen zu definieren bzw., im Code zu erkennen.

```
1  void loop()
2  {
3    off(PIN_YELLOW);
4    on(PIN_RED);
5    delay(4 * 1000);
6
7    on(PIN_YELLOW);
8    delay(1 * 1000);
9
10   off(PIN_RED);
11   off(PIN_YELLOW);
12   on(PIN_GREEN);
13   delay(4 * 1000);
14
15   off(PIN_GREEN);
16   on(PIN_YELLOW);
17   delay(1 * 1000);
18 }
```

Listing 4: Programmschleife für Aufgabe 1

In Listing 4 werden die einzelnen Phasen implementiert. Nach jeder Leerzeile, d.h., nach Zeile 6, 9, und 14 beginnt jeweils eine neue Phase.

Zu Beginn wird Phase 1 implementiert. Diese Schaltet die gelbe LED, Pin 12, aus, falls vorher Phase 4 aktiv war und schaltet die rote LED, Pin 13, an. Danach wird die $delay(x)$ Funktion aufgerufen, welche die Programmausführung für x Millisekunden unterbricht. Die Angabe von x als Berechnung aus $Sekunde * 1000$ wurde gewählt, um im Code besser erkenntlich zu machen, dass es sich um Millisekunden handelt. Der Programmfluss wird daher für vier Sekunden unterbrochen.

Dann folgt Phase 2, welche zusätzlich zur Roten auch die gelbe LED einschaltet. Der Programmfluss wird für eine Sekunde unterbrochen.

Es folgt Phase 3. Die Rote und die gelbe LED werden ausgeschaltet. Die

Grüne, Pin 8, wird eingeschaltet. Der Programmfluss wird für weitere vier Sekunden unterbrochen.

Schlussendlich folgt Phase 4. Es wird die grüne LED wieder ausgeschaltet, während die Gelbe aktiv wird. Es folgt wieder eine Unterbrechung des Programms für eine Sekunde. Am Ende der *loop*() Funktion wird sie wieder von Beginn an, d.h., von Phase 1 aus, ausgeführt.

```
const int PIN_RED = 13;
const int PIN_YELLOW = 12;
const int PIN_GREEN = 8;

void setup()
{
  pinMode(PIN_RED, OUTPUT);
  pinMode(PIN_YELLOW, OUTPUT);
  pinMode(PIN_GREEN, OUTPUT);
}

void loop()
{
  off(PIN_YELLOW);
  on(PIN_RED);
  delay(4 * 1000);

  on(PIN_YELLOW);
  delay(1 * 1000);

  off(PIN_RED);
  off(PIN_YELLOW);
  on(PIN_GREEN);
  delay(4 * 1000);

  off(PIN_GREEN);
  on(PIN_YELLOW);
  delay(1 * 1000);
}

void off(int pin) {
  digitalWrite(pin, LOW);
}

void on(int pin) {
  digitalWrite(pin, HIGH);
}
```

Listing 5: Vollständiger Programmcode für Aufgabe 1

4.4 Fehlerdiskussion

Es wurden während dieser Aufgabe keine Fehler gefunden.

4.5 Zusammenfassung

5 Aufgabe 2 - Reaktionsspiel

Es soll eine Schaltung implementiert werden, welche ein lichtbasiertes Reaktionsspiel darstellt. Die Schaltung soll aus fünf LEDs bestehen, welche der Reihe nach leuchten und erlöschen. D.h., LED 1 leuchtet, LED 1 erlischt und LED 2 leuchtet, LED 2 erlischt und LED 3 leuchtet, u.s.w.. Sobald die letzte LED erlischt soll die erste LED wieder leuchten und der Rythmus von neuen beginnen.

Die LEDs werden in der Reihenfolge Rot - Gelb - Grün - Gelb - Rot geschaltet und angeordnet. Des weiteren wird ein Taster eingebaut. Wird der Taster gedrückt, wenn die grüne LED leuchtet, halbiert sich die Zeit zwischen der die LEDs geschaltet werden. D.h., bleibt eine LED fünf Sekunden eingeschaltet, bevor die nächste LED geschaltet wird, so bleibt sie danach nur 2,5 Sekunden lang eingeschaltet. Wird der Taster erneut gedrückt, wenn die grüne LED leichtet, halbiert sich die Zeit erneut auf 1,25 Sekunden, u.s.w..

Wird der Taster gedrückt, wenn die grüne LED nicht leuchtet, wird die Zeit wieder auf den ursprünglichen Wert gesetzt. In diesem Beispiel also zurück auf fünf Sekunden. In diesem Fall sollen auch alle LEDs gleichzeitig drei mal blinken, bevor das Spiel schlussendlich von vorne beginnt.

5.1 Materialien

Tabelle 5: Aufgabe 2 - Verwendete Materialien

Bezeichnung	Eigenschaften	Menge
Widerstand	150Ω Braun - Grün - Braun - Gold	5
Widerstand	$10k\Omega$ Braun - Schwarz - Orange - Gold	1
LED	Rot	2
LED	Gelb	2
LED	Grün	1
Taster	4 Polig	1
Mikrocontroller	Arduino Uno R3	1

5.2 Vorbereitung

Für den ersten Teil muss wieder der Vorwiderstand der LEDs berechnet werden. Die Berechnung und des Widerstands und die reale Auswahl aus der E6-Reihe erfolgt wie in Sektion 4.2. D.h., es wurden 200Ω berechnet und der 150Ω Widerstand aus der E6-Reihe ausgewählt.

Der zweite Teil bestand aus zwei Fragestellungen. Erstens, wie kann ein Mikrocontroller-Programm unterbrochen werden, um ,z.B., einen Druck auf einen Taster zu erkennen. Dies kann mithilfe eines sogenannten *Interrupts* implementiert werden. Dafür wird ein Pin als Input-PullUp-Pin angelegt und für diesen Pin eine Funktion definiert, welche aufgerufen wird, sobald der Taster gedrückt wird.

Zweitens, wie kann ermittelt werden, ob der Taster zum richtigen oder falschen Zeitpunkt gedrückt wurde. Dafür kann eine Variable mit dem *volatile*-Schlüsselwort angelegt werden. Damit wird die Variable nicht in einem Zwischenspeicher behalten, sondern wird immer vom Hauptspeicher gelesen, wenn sie verwendet wird. Diese Variable kann die Werte "wahr", bzw. true", und falsch", bzw false", annehmen. Wird der Button gedrückt kann die Variable auf "wahr"gesetzt werden, ansonsten auf falsch". In der Programmlogik selbst kann dann überprüft werden, welchen Wert die Variable momentan besitzt. Damit kann ausgewertet werden, ob sie zum richtigenS-Zeitpunkt den Wert "wahr"hat, oder nicht.

5.3 Praktikumsaufgabe

In Abbildung 3 ist die implementierte Schaltung zu sehen. Zur Klärung sei gesagt, dass das blaue Kabel nur mit Spalte 15 verbunden ist und nicht mit einem der Widerstände, obwohl das Bild es vermuten lässt.

Weiters ist die Wahl der Pins zu erkennen. Die Wahl viel auf die Pins 9 bis 13 zur Steuerung der LEDs und auf Pin 2 zur Erkennung des Tastendrucks. Die Pins zur Ansteuerung der LEDs wurde willkürlich getroffen. Es handelt sich nur von oben nach unten um die ersten, zur Verfügung stehenden Pins. Pin 2 wurde gewählt, da nur Pin 2 und 3 in der Lage sind, an einen Interrupt gekoppelt zu werden.

Im Schaltkreis ist weiters der Taster zu sehen, der mit einem Pull-Up-Widerstand, wie in Sektion 3.2 zu sehen, versehen ist. Die Terminals an

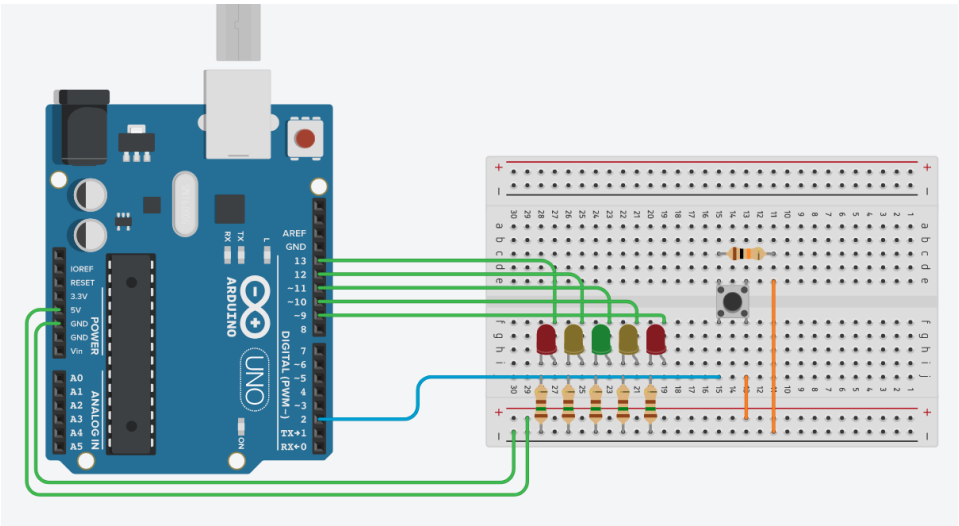

Abbildung 3: Implementierter Stromkreis von Aufgabe 2

der Spalte 15 auf beiden Seiten sind verbunden, die Terminal an den Spalten 15 und 13 werden bei einem Tastendruck verbunden. Damit ergibt sich das Teilschaltbild des Tasters, dass in Abbildung 4 zu sehen ist. Die alleinstehenden Zahlen geben die zugehörige Spalte des Steckbrettes an. Des weiteren sei gesagt, dass in der Abbildung zwei Schalter zu sehen sind, diese repräsentieren die zwei vorhanden Terminals des echten Schalters. Bei Tastendruck werden beide gleichzeitig geschlossen.

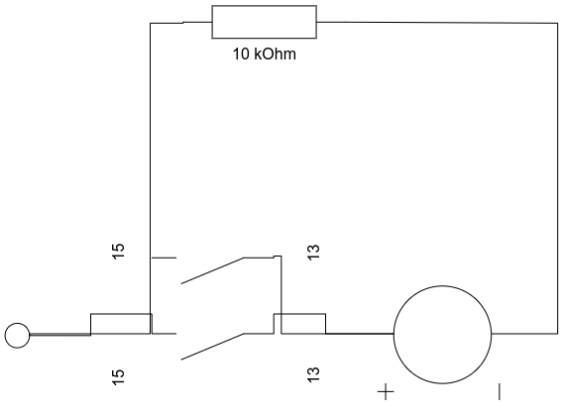

Abbildung 4: Schaltung des Tasters mit Pull-Up Widerstand

Im nachfolgendem Text wird der Programmcode der Aufgabe erläutert. Einzelne Teile des Codes werden ausgewählt und beschrieben. Am Ende dieser Sektion befindet sich der vollständige Programmcode.

```
onst int NUM_PINS = 5;
const int WINNING_PIN_INDEX = 2;
const int PINS[] = {
  13, 12, 11, 10, 9
};
const int PIN_BUTTON = 2;
```

Listing 6: Setzen der Pin-Konstanten

In Listing 6 werden die Nummern der Pins angelegt. Die Besonderheit gegenüber der vorherigen Sektion, Sektion 4, ist die Verwendung eines Arrays. In diesem werden die verwendeten Pinnummern zu Ansteuerung der LEDs angegeben. Mit NUM_PINS wird die Länge des Arrays angegeben. Solch eine Angabe ist nützlich, da die Länge eines Arrays in der gegebenen Programmiersprache, C, nicht trivial findbar ist.

Um in C die Länge eines Arrays zu bestimmen, müsste zuerst mittels $sizeof(PINS)$ die Menge an Speicher ermittelt werden, die das Arrays belegt. Dieser Wert müsste sodann durch das Ergebnis von $sizeof(int)$, die Menge an Speicher die ein Integer-Wert im Speicher belegt, dividiert werden. Das Ergebnis aus $sizeof(PINS)/sizeof(int)$ würde dann die Länge des Arrays ergeben. Einfacher ist es daher, die Größe als Konstante anzugeben, wenn diese, wie in diesem Fall, bekannt ist. Mit der Konstante PIN_BUTTON wird die Pinnummer des Pins angegeben, welcher später den Zustand des Tasters erkennt.

Die Konstante $WINNING\_PIN\_INDEX$ gibt an, welcher Index des Pin-Arrays die grüne LED darstellt, D.h., die der Taster gedrückt, wenn der Pin von $PINS[WINNING\_PIN\_INDEX]$ eingeschaltet ist, ist zum richtigen Zeitpunkt gedrückt worden.

```
const unsigned long INTERVAL_DEFAULT = 5000;
```

Listing 7: Standard Interval zum LED wechsel

Mit der Konstanten die in Listing 6 angegeben ist, wird das Standardinterval angegeben, mit dem bei den LEDs weitergesprungen wird. D.h., nach Ablauf der Zeit von $INTERVAL\_DEFAULT$, in Millisekunden, wird zur nächsten LED gesprungen, wenn das Spiel neu gestartet wurde.

```
unsigned long current_millis = 0;
```

```
unsigned long previous_millis = 0;
unsigned long interval = INTERVAL_DEFAULT;
```

Listing 8: Variablen zur Zeitmessung

Die Variablen in Listing 8 werden dazu verwendet, die vergangene Zeit im Programm mitzuverfolgen. In *current_millis* wird gespeichert, wie viel Zeit seit dem Programmstart vergangen ist. Der Wert in *previous_millis* gibt an, zu welchem Zeitpunkt die letzte Aktion durchgeführt wird. Die Bedeutung dieser Variable wird im Verlauf des Programmcodes klarer. In *interval* ist das derzeitige Interval zwischen Wechseln der LEDs gespeichert. Zu Beginn wird sie mit der Konstante aus Listing 7 initialisiert. Sie wird bei jedem Neustart des Spiels auf diesen Wert zurückgesetzt.

```
int pin_index = 0;

volatile bool button_pressed = false;
volatile bool game_over = false;
```

Listing 9: Variablen zur Zustandsbestimmung des Spiels

Durch die Variablen in Listing 9 wird der derzeitige Zustand des Spiels mitverfolgt. Sie haben zusätzlich zum Typ noch das Schlüsselwort *volatile*, welches im zweiten Teil der Sektion 5.2 beschrieben wurde. Diese Schlüsselwort benötigen sie, da die Variablen in einem Interrupt verändert werden und dadurch direkt vom Speicher ausgelesen werden müssen. Eine Annäherung des mit diesen Variablen bewirkten Zustandsverlauf kann in Abbildung 5 abgelesen werden.

Die Variable *pin_index* ist nicht mittels *volatile* markiert, da sie sich nur in der Hauptschleife des Programms ändert. Nach Ablauf von *interval* wird sie um 1 erhöht bis zu einem Maximum von $NUM\_PINS - 1$. Würde sie den Wert NUM_PINS annehmen, wird sie wieder auf 0 zurückgesetzt. Diese Variable gibt an, welcher Index, von 0 beginnend, aus $PINS$ verwendet wird, um eine LED anzusteuern. D.h., $pin\_index = 0 \Rightarrow Pin13 = HIGH$, $pin\_index = 1 \Rightarrow Pin12 = HIGH$, u.s.w..

```
void setup() {
  Serial.begin(9600);
```

Listing 10: Einstellen der seriellen Schnittstelle

Mit dem Kommando in Listing 10 wird eine serielle Verbindung zu einem verbundenem Computer hergestellt. Damit kann später im Programm Text

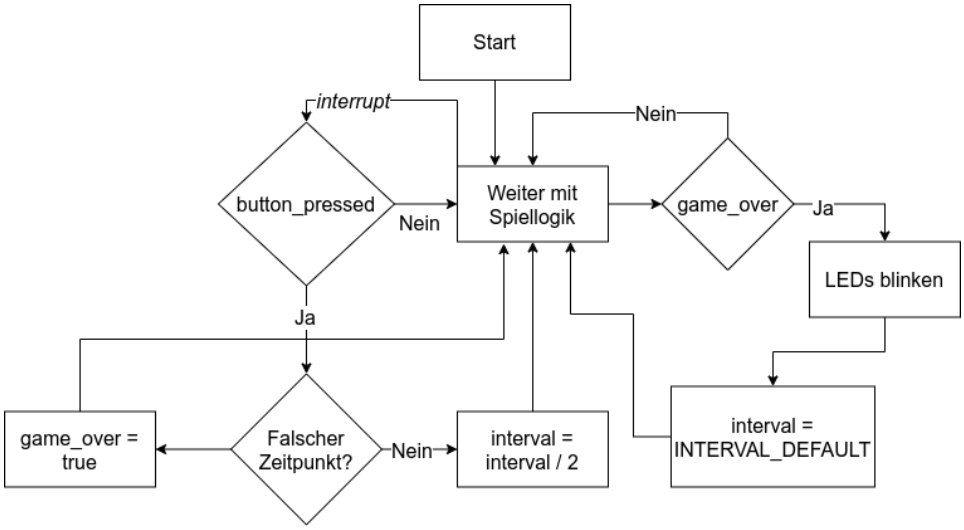

Abbildung 5: Annäherung des Spielverlaufs

an den Computer übertragen oder vom Computer auf den Mikrocontroller gesendet werden. Das Argument der Methode, 9600 gibt die Baudrate an.

```
int i;
for (i = 0; i < NUM_PINS; i++) {
    pinMode(PINS[i], OUTPUT);
}
```

Listing 11: Konfiguration der Pins

In Listing 11 werden die Pins durchgegangen. Hier wird das in Listing 6 angelegte Array zum ersten Mal verwendet. Es werden hier alle Pinnummern in dem Arrays durchgelaufen und als Output eingestellt. Dadurch muss nicht jede Pinnummer einzeln eingerichtet werden.

```
pinMode(PIN_BUTTON, INPUT_PULLUP);

attachInterrupt(
    digitalPinToInterrupt(PIN_BUTTON),
    on_button_change,
    CHANGE);
```

Listing 12: Einstellen des Buttons

Der Code in Listing 12 setzt nun den Pin, der an dem Taster angeschlossen ist, auf Input-PullUp und verknüpft ihn mit einem Interrupt. Die Funktion

on_button_change wird nun aufgerufen, jedes mal wenn der Taster manipuliert wird. Der Parameter dritte Parameter der *attachInterrupt*-Methode, in diesem Fall $CHANGE$, gibt an was passieren muss damit die Methode aufgerufen wird. $CHANGE$ beduetet, dass jedes mal wenn sich das Potenzial ändert, daher entweder von $5V$ auf $0V$ fällt oder von $0V$ auf $5V$ steigt, der Interrupt ausgeführt wird.

Des weiteren gibt es noch weitere Werte, die übergeben werden können, welche verschieden Zustände abbilden.

- LOW, der Interrupt wird ausgeführt, wenn am Pin $0V$ anliegen
- $CHANGE$, der Interrupt wird ausgeführt, wenn sich das Potenzial am Pin ändert
- $RISING$, der Interrupt wird ausgeführt, wenn die Spannung am Pin von $0V$ auf $5V$ steigt
- $FALLING$, der Interrupt wird ausgeführt, wenn die Spannung am Pin von $5V$ auf $0V$ fällt

```
1   init_game();
2 }
3
4 void init_game() {
5   game_over = false;
6   interval = INTERVAL_DEFAULT;
7   pin_index = 0;
8   digitalWrite(PINS[0], HIGH);
9 }
```

Listing 13: Setzen der Standardwerte

Listing 13 zeigt die Methode, die aufgerufen wird, wenn das Spiel neu gestartet wird. Der Aufruf von *init_game*() zu Begin des Listings ist noch in der *setup*()-Methode zu finden. Dadurch werden noch vor Spielbeginn die Werte richtig gesetzt.

In der *init_game*()-Methode wird des Zustand mittels *game_over* zurückgesetzt. Das Intervall zwischen den LED wechseln wird auf den Standardwert gesetzt. Die aktive LED wird mittels $pin\_index = 0$ auf die Erste gesetzt und mit dem Aufruf von $digitalWrite(\dots)$ eingeschaltet.

```
void on_button_change() {
  int button_state = digitalRead(PIN_BUTTON);
  if (button_state == HIGH) {
    on_button_rising();
  } else {
    on_button_falling();
  }
}

void on_button_rising() {
  button_pressed = true;

  if (pin_index == WINNING_PIN_INDEX &&
     !game_over) {
    interval = interval / 2;
  } else {
    game_over = true;
  }
}

void on_button_falling() {
  button_pressed = false;
}
```

Listing 14: Interrupt Methoden des Tasters

Obwohl die Methoden, welche die Logik des Interrupts darstellen, am Ende des Codes liegen, werden sie hier besprochen. Damit kann ein Kontext für andere Codeteile erzeugt werden.

In Listing 14 wird die in Listing 12 verwendete Methode gezeigt. Zu Beginn von *on_button_change*() wird der Zustand des Tasters ausgelesen. Liegen zum Zeitpunkt des Aufrufs $5V$ am Pin an, so wird *on_button_rising*() aufgerufen, ansonsten wird *on_button_falling*() verwendetet. Dieser Distinktion muss verwendet werden, da nur ein Interrupt pro Pin angelegt werden kann. Es können die entsprechenden Methoden nicht gleichzeitig direkt an den Pin angehängt werden. Daher muss die Methode *on_button_change*() diese Unterscheidung durchführen.

In *on_button_rising*() wird zuerst der Zustand von *button_pressed* auf *true* gesetzt. Damit kann in der Programmschleife erkannt werden, ob der Taster gedrückt wird oder nicht. Danach wird überprüft welche LED zurzeit leuchtet, indem der Wert in *pin_index* überprüft wird und ob das Spiel bereits vorbei ist. Die letztere Überprüfung wird gemacht, da der Taster auch dann getätigt werden kann, wenn, z.b., alle LEDs blinken. Wurde der Taster nicht

zum richtigen Zeitpunkt getätigt, so wird der Zustand des Spiels auf "Game Over"gesetzt, indem die enstprechende Variable auf *true* gesetzt wird.

```
void blink_all() {
  int j;
  for (j = 0; j < 3; j++) {
    int i;
    write_to_all(HIGH);
    delay(500);
    write_to_all(LOW);
    delay(500);
  }
}

void write_to_all(int state) {
  int i;
  for (i = 0; i < NUM_PINS; i++) {
    digitalWrite(PINS[i], state);
  }
}

void increase_pin_index() {
    pin_index++;
    if (pin_index >= NUM_PINS){
      pin_index = 0;
    }
}
```

Listing 15: Hilfsmethoden

In Listing 15 werden die verwendeten Hilfsmethoden gezeigt. Für *write_to_all* kann der gewünschte Zustand übergeben werden, $HIGH$ oder LOW, und dieser wird, ähnlich wie in *setup*(), mittels einer Schleife an alle Pins im $PINS$-Array übernommen. *blink_all* wird verwendet, um das Blinken der LEDs am Ende des Spiels zu implementieren. Es werden hier drei mal alle Pins im $PINS$-Array erst mittels $write\_to\_all(HIGH)$ eingeschaltet. Danach wird 500ms lang mit $delay(500)$ gewartet. Mit $write\_to\_all(LOW)$ und anschließenden $delay(500)$ werden die LEDs wieder ausgeschaltet und 500ms lang gewartet.

increase_pin_index() hat die Aufgabe, die Variable *pin_index* zu erhöhen. Würde die Variable das Limit von $NUM\_PINS - 1$ überschreiten, wird die Variable wieder auf 0 gesetzt.

```
void loop()
{
  current_millis = millis();
```

```
4
5   if (current_millis - previous_millis >= interval
6       && !button_pressed
7       && !game_over) {
8     digitalWrite(PINS[pin_index], LOW);
9     increase_pin_index();
10    digitalWrite(PINS[pin_index], HIGH);
11  previous_millis = current_millis;
12  } else if (game_over) {
13    blink_all();
14    init_game();
15    previous_millis = current_millis;
16  }
17 }
```

Listing 16: Programmschleife

In Listing 16 wird die Programmschleife beschrieben. Hier wird nun die Variable *current_millis* verwendet, um die vergangene Zeit zu speichern. Ist der Abstand zwischen *current_millis* und *previous_millis*, zu Beginn 0, größer oder gleich dem derzeitigen Intervall, so wird zuerst die momentan aktive LED ausgeschaltet. Danach wird die Methode *increase_pin_index* ausgeführt, um den verwendeten Index auf den nächsten Pin zu setzen. Die LED die als nächstes leuchten soll, wird sodann eingeschalten. Schlussendlich wird der Wert von *previous_millis* auf den Wert von *current_millis* gesetzt, um wieder den Zustand $current\_millis - previous\_millis < interval$ zu erreichen. Dieser Ablauf wird allerdings nur dann durchgeführt, wenn weder der Taster gedrückt, noch das Spiel vorbei ist.

Ist das Spiel vorbei so wird die vorher beschriebene Methode *blink_all*() aufgerufen, um alle LEDs blinken zu lassen. Mittels *init_game*() werden alle globalen Werte wieder auf ihren Standard zurückgesetzt. Schlussendlich wird der Wert von *previous_millis* auf den Wert von *current_millis* gesetzt, um das selbe Intervall wie bei einem Spielstart zu haben. Würde dies nicht gemacht, könnte das Spiel früher von der ersten LED auf die Zweite überspringen als gewollt.

```
1 const int NUM_PINS = 5;
2 const int WINNING_PIN_INDEX = 2;
3 const int PINS[] = {
4   13, 12, 11, 10, 9
5 };
6 const int PIN_BUTTON = 2;
7 const unsigned long INTERVAL_DEFAULT = 5000;
8
```

```
unsigned long current_millis = 0;
unsigned long previous_millis = 0;
unsigned long interval = INTERVAL_DEFAULT;

int pin_index = 0;

volatile bool button_pressed = false;
volatile bool game_over = false;

void setup()
{
  Serial.begin(9600);

  int i;
  for (i = 0; i < NUM_PINS; i++) {
    pinMode(PINS[i], OUTPUT);
  }

  pinMode(PIN_BUTTON, INPUT_PULLUP);

  attachInterrupt(
    digitalPinToInterrupt(PIN_BUTTON),
    on_button_change,
    CHANGE);

  init_game();
}

void init_game() {
  game_over = false;
  interval = INTERVAL_DEFAULT;
  pin_index = 0;
  digitalWrite(PINS[0], HIGH);
}

void loop()
{
  current_millis = millis();

  if (current_millis - previous_millis >= interval
       && !button_pressed
       && !game_over) {
    digitalWrite(PINS[pin_index], LOW);
    increase_pin_index();
    digitalWrite(PINS[pin_index], HIGH);
  previous_millis = current_millis;
  } else if (game_over) {
    blink_all();
    init_game();
```

```
58      previous_millis = current_millis;
59    }
60  }
61
62  void blink_all() {
63    int j;
64    for (j = 0; j < 3; j++) {
65      int i;
66      write_to_all(HIGH);
67      delay(500);
68      write_to_all(LOW);
69      delay(500);
70    }
71  }
72
73  void write_to_all(int state) {
74    int i;
75    for (i = 0; i < NUM_PINS; i++) {
76      digitalWrite(PINS[i], state);
77    }
78  }
79
80  void increase_pin_index() {
81      pin_index++;
82      if (pin_index >= NUM_PINS){
83        pin_index = 0;
84      }
85  }
86
87  void on_button_change() {
88    int button_state = digitalRead(PIN_BUTTON);
89    if (button_state == HIGH) {
90      on_button_rising();
91    } else {
92      on_button_falling();
93    }
94  }
95
96  void on_button_rising() {
97    button_pressed = true;
98
99    if (pin_index == WINNING_PIN_INDEX &&
100       !game_over) {
101     interval = interval / 2;
102   } else {
103     game_over = true;
104   }
105 }
106
```

```
107 void on_button_falling() {
108   button_pressed = false;
109 }
```

Listing 17: Vollständiger Programmcode für Aufgabe 2

5.4 Fehlerdiskussion

Es wurde der Fehler gemacht, dass in Listing 16 auch auf den Zustand des Tasters geprüft wird. D.h., der Taster darf nicht gedrückt sein, damit das Spiel weiter läuft. Dieser Fehler verursacht, dass das Lauflicht pausiert und nicht weiterläuft, wenn der Taster nicht mehr losgelassen wird.

Ein weiterer glücklicher Zufall konnte einen weiteren Fehler verhindern. Es wurde nämlich die Funktionsweise des Tasters falsch verstanden. Die Annahme war nicht, dass wie in Abbildung 4 dargestellt, die Spalte 15, bzw. die Spalte 13, verbunden ist, sondern dass Spalte 15 mit Spalte 13 verbunden ist. Würde dies zutreffen, so würde die Schaltung nicht funktionieren. Nachdem jedoch auch die gegebene Schaltung eines Pull-Down Widerstands falsch gelesen wurde, wurde sie unbeabsichtigt richtig implementiert.

5.5 Zusammenfassung

6 Aufgabe 3 - Realisierung eines Logic Analyzers

Es ist eine Schaltung zu entwerfen, welche Logikbausteine erkennt. Da bei zwei Signalen a und b als Eingänge je nach Baustein, bzw. Gatter, ein anderes Signal y am Ausgang anliegt, kann ein unbekanntes Gatter analysiert und benannt werden.

6.1 Materialien

Tabelle 6: Aufgabe 3 - Verwendete Materialien

Bezeichnung	Eigenschaften	Menge
Widerstand	150Ω	2
	Braun - Grün - Braun - Gold	
74HC00	NAND	1
74HC02	NOR	1
74HC08	AND	1
74HC32	OR	1
74HC86	XOR	1
Mikrocontroller	Arduino Uno R3	1

6.2 Vorbereitung

Zur Vorbereitung dieser Aufgabe, müssen die Wahrheitstabellen der zu untersuchenden Funktionen bekannt sein. Dafür werden die Tabellen für die verwendeten Funktionen aufgestellt. Es gilt die Annahme, dass die Funktionen bekannt sind und werden daher nicht näher erklärt. Eine Erklärung der logischen Funktionen würde über den Umfang dieses Protokolls hinaus gehen. Die Tabellen können in den Tabellen 7 angelesen werden.

6.3 Praktikumsaufgabe

Für die Aufgabe wurde nun die in Abbildung 6 gezeigte Schaltung implementiert. Die Pins 13 und 12 bilden hierfür jeweils die Eingänge für das Logikgatter. Der Pin 8 liest dann den Ausgang aus. Die Widerstände von 150Ω wurden verwendet, um die Logikbausteine vor zu hohen Stromflüssen

Tabelle 7: Wahrheitstabellen verwendeter Logikfunktionen

	AND	
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

	OR	
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

	NAND	
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

	NOR	
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

	XOR	
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

zu schützen. Mittels des angeschlossenen Multimeters kann der Ausgang manuell ausgelesen werden und das Ergebnis des Programmcodes überprüft werden.

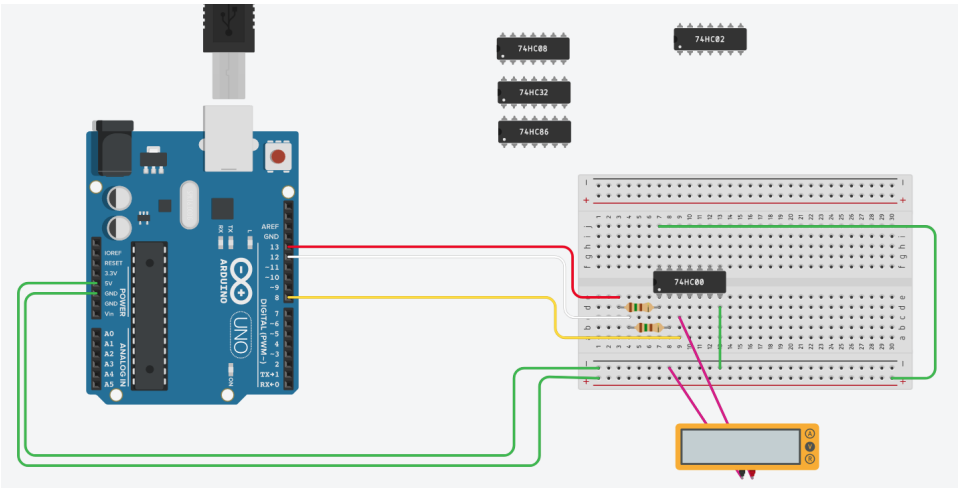

Abbildung 6: Implementierter Stromkreis von Aufgabe 3

Im nachfolgendem Text wird der Programmcode der Aufgabe erläutert. Einzelne Teile des Codes werden ausgewählt und beschrieben. Am Ende dieser Sektion befindet sich der vollständige Programmcode.

```
1 /*
```

```
(Input1, Input2) => [Index]
(0, 0) => [0]
(0, 1) => [1]
(1, 0) => [2]
(1, 1) => [3]
*/
bool TRUTH_TABLE[] = { false, false, false, false };
```

Listing 18: Variable für die Wahrheitstabelle

In Listing 18 wird ein Array angelegt, welches die Ergebnisse der Evaluierung speichert. Im Kommentar sind die Abbildungen der Eingänge auf den jeweiligen Index zu sehen. D.h., werden beide Eingänge auf 0, bzw. *LOW*, geschalten, so wird das Ergebnis an Index 0 gespeichert.

```
digitalWrite(PIN_OUTPUT_1, LOW);
digitalWrite(PIN_OUTPUT_2, LOW);

delay(2000);
TRUTH_TABLE[0] = digitalRead(PIN_INPUT) == HIGH;

digitalWrite(PIN_OUTPUT_1, LOW);
digitalWrite(PIN_OUTPUT_2, HIGH);

delay(2000);
TRUTH_TABLE[1] = digitalRead(PIN_INPUT) == HIGH;

digitalWrite(PIN_OUTPUT_1, HIGH);
digitalWrite(PIN_OUTPUT_2, LOW);

delay(2000);
TRUTH_TABLE[2] = digitalRead(PIN_INPUT) == HIGH;

digitalWrite(PIN_OUTPUT_1, HIGH);
digitalWrite(PIN_OUTPUT_2, HIGH);

delay(2000);
TRUTH_TABLE[3] = digitalRead(PIN_INPUT) == HIGH;
```

Listing 19: Ausführen der Logikfunktionen

Der Code in Listing 19 setzt nacheinander die Eingänge des Gatters auf die Werte der Wahrheitstabellen. D.h., es werden beide Eingänge auf 0 gesetzt und das Ergebnis im Array gespeichert. Danach wird ein Eingang auf 1 und der andere auf 0 gesetzt und das Ergebnis wieder im Array gespeichert, u.s.w..

Zwischen den einzelnen Schaltungen der Eingänge wird mittels Aufruf von

delay(2000) zwei Sekunden lang gewartet. Dieser Aufruf ist dazu da, um ein manuelles Ablesen des Multimeters zu ermöglichen und ist für die korrekte Ausführung des Programms nicht notwendig.

```
Serial.println("");
Serial.println("|---|---|---|");
Serial.println("| a | b | y |");
Serial.println("|---|---|---|");

Serial.print("| 0 | 0 | ");
Serial.print(TRUTH_TABLE[0]);
Serial.println(" |");

Serial.print("| 0 | 1 | ");
Serial.print(TRUTH_TABLE[1]);
Serial.println(" |");

Serial.print("| 1 | 0 | ");
Serial.print(TRUTH_TABLE[2]);
Serial.println(" |");

Serial.print("| 1 | 1 | ");
Serial.print(TRUTH_TABLE[3]);
Serial.println(" |");
Serial.println("|---|---|---|");
Serial.println("");
```

Listing 20: Ausgabe der Wahrheitstabelle

Die Ergebnisse die in der Variable aus Listing 18 gespeichert wurden, werden in Listing 20 ausgegeben. Diese Ausgabe wird gemacht, um eine manuelle Überprüfung des Ergebnisses machen zu können. D.h., sie wird gemacht um ein Debuggen des Codes zu ermöglichen, ist aber für eine korrekte Ausführung des Programms nicht notwendig. Das Programm gibt hier die resultierende Wahrheitstabelle, in der Form einer Tabelle wie in Tabellen 7 dargestellt, aus.

```
if (!TRUTH_TABLE[0]
&& !TRUTH_TABLE[1]
&& !TRUTH_TABLE[2]
&& TRUTH_TABLE[3]) {
    Serial.println("AND - Gate");
} else if (!TRUTH_TABLE[0]
&& TRUTH_TABLE[1]
&& TRUTH_TABLE[2]
&& TRUTH_TABLE[3]) {
    Serial.println("OR - Gate");
} else if (TRUTH_TABLE[0]
```

```
&& TRUTH_TABLE[1]
&& TRUTH_TABLE[2]
&& !TRUTH_TABLE[3]) {
    Serial.println("NAND - Gate");
} else if (TRUTH_TABLE[0]
&& !TRUTH_TABLE[1]
&& !TRUTH_TABLE[2]
&& !TRUTH_TABLE[3]) {
    Serial.println("NOR - Gate");
} else if (!TRUTH_TABLE[0]
&& TRUTH_TABLE[1]
&& TRUTH_TABLE[2]
&& !TRUTH_TABLE[3]) {
    Serial.println("XOR - Gate");
} else {
    Serial.println("Unrecognized Gate");
}
```

Listing 21: Evaluierung der Ergebnisse

Die Ergebnisse werden in Listing 21 evaluiert. Je nachdem welche Ausgaben der Logikbaustein getätigt hat, kann nun die entsprechende Funktion gefunden werden. Dadurch wird das Ergebnis mit den Ergebnissen aus den Tabellen 7 verglichen. Gleichen die Ergebnisse denen der Y-Spalte einer Tabelle, so wurde das jeweilige Gatter erkannt und der Name wird ausgegeben. Sind die Ergebnisse nicht zuzuordnen, so wird die Meldung Ünrecognized Gate", zu Deutsch Ünbekanntes Gatter", ausgegeben.

```
void loop()
{
}
```

Listing 22: Programmschleife

Die Programmschleife in Listing 22 ist leer, da der Code für jedes Logikgatter nur einmal ausgeführt werden muss. Da für jedes Logikgatter die Schaltung umgebaut werden muss, sprich, es muss das Gatter ausgetauscht werden, ist das Ausführen des Codes in einer Schleife nicht zielführend.

```
const int PIN_OUTPUT_1 = 13;
const int PIN_OUTPUT_2 = 12;
const int PIN_INPUT = 8;

/*
(Input1, Input2) => [Index]
(0, 0) => [0]
(0, 1) => [1]
```

```
(1, 0) => [2]
(1, 1) => [3]
*/
bool TRUTH_TABLE[] = { false, false, false, false };

void setup()
{
  Serial.begin(9600);

  pinMode(PIN_OUTPUT_1, OUTPUT);
  pinMode(PIN_OUTPUT_2, OUTPUT);
  pinMode(PIN_INPUT, INPUT);

  digitalWrite(PIN_OUTPUT_1, LOW);
  digitalWrite(PIN_OUTPUT_2, LOW);

  delay(2000);
  TRUTH_TABLE[0] = digitalRead(PIN_INPUT) == HIGH;

  digitalWrite(PIN_OUTPUT_1, LOW);
  digitalWrite(PIN_OUTPUT_2, HIGH);

  delay(2000);
  TRUTH_TABLE[1] = digitalRead(PIN_INPUT) == HIGH;

  digitalWrite(PIN_OUTPUT_1, HIGH);
  digitalWrite(PIN_OUTPUT_2, LOW);

  delay(2000);
  TRUTH_TABLE[2] = digitalRead(PIN_INPUT) == HIGH;

  digitalWrite(PIN_OUTPUT_1, HIGH);
  digitalWrite(PIN_OUTPUT_2, HIGH);

  delay(2000);
  TRUTH_TABLE[3] = digitalRead(PIN_INPUT) == HIGH;

  Serial.println("");
  Serial.println("|---|---|---|");
  Serial.println("| a | b | y |");
  Serial.println("|---|---|---|");

  Serial.print("| 0 | 0 | ");
  Serial.print(TRUTH_TABLE[0]);
  Serial.println(" |");

  Serial.print("| 0 | 1 | ");
  Serial.print(TRUTH_TABLE[1]);
  Serial.println(" |");
```

```
58
59    Serial.print("| 1 | 0 | ");
60    Serial.print(TRUTH_TABLE[2]);
61    Serial.println(" |");
62
63    Serial.print("| 1 | 1 | ");
64    Serial.print(TRUTH_TABLE[3]);
65    Serial.println(" |");
66    Serial.println("|---|---|---|");
67    Serial.println("");
68
69    Serial.println("Result: ");
70
71    if (!TRUTH_TABLE[0]
72          && !TRUTH_TABLE[1]
73          && !TRUTH_TABLE[2]
74          && TRUTH_TABLE[3]) {
75      Serial.println("AND - Gate");
76    } else if (!TRUTH_TABLE[0]
77          && TRUTH_TABLE[1]
78          && TRUTH_TABLE[2]
79          && TRUTH_TABLE[3]) {
80      Serial.println("OR - Gate");
81    } else if (TRUTH_TABLE[0]
82          && TRUTH_TABLE[1]
83          && TRUTH_TABLE[2]
84          && !TRUTH_TABLE[3]) {
85      Serial.println("NAND - Gate");
86    } else if (TRUTH_TABLE[0]
87          && !TRUTH_TABLE[1]
88          && !TRUTH_TABLE[2]
89          && !TRUTH_TABLE[3]) {
90      Serial.println("NOR - Gate");
91    } else if (!TRUTH_TABLE[0]
92          && TRUTH_TABLE[1]
93          && TRUTH_TABLE[2]
94          && !TRUTH_TABLE[3]) {
95      Serial.println("XOR - Gate");
96    } else {
97      Serial.println("Unrecognized Gate");
98    }
99  }
100
101 void loop()
102 {
103 }
```

Listing 23: Vollständiger Programmcode der Aufgabe 3

6.4 Fehlerdiskussion

Während des Testens des Programms konnten alle Gatter bis auf das NOR-Gatter nicht erkannt werden. Der Grund dafür war, dass bei gleicher ausrichtung des NOR-Gatters, die Ein- und Ausgänge vertauscht waren. D.h., die Reihenfolge der Pins der Bausteine war Ëingange 1 - Eingang 2 - Ausgang", außer am NOR-Gatter. Hier ist die Reihenfolge Äusgang - Eingang 1 - Eingang - 2". Um das NOR-Gatter richtig zu erkennen, muss daher die Schaltung entsprechend angepasst werden. Da die Schaltung theoretisch ein unbekanntes Gatter erkennen soll, müsste daher die Schaltung geändert werden, wenn das Gatter nicht erkannt wurde. Wird das Gatter danach immer noch nicht erkannt, handelt es sich tatsächlich um ein unbekanntes Gatter, ansonsten um ein NOR-Gatter.

6.5 Zusammenfassung

7 Aufgabe 4 - Wertbestimmung von Widerstand und Kondensator

In dieser Aufgabe geht es um Widerstände und Kondensatoren. Diese sind Grundbausteine der Elektrotechnik. Neben diversen Berechnungen sind folgende Fragen zu beantworten.

- Können Unterschiede zwischen Messungen am Mikrocontroller und manuellen Methoden gefunden werden?
- Wenn Ja, welche und warum?
- Mit welchen Prinzipien und Überlegungen wurden die Widerstände gemessen?
- Welche Abweichungen ergeben sich beim Messen bekannter Kondensatoren?

7.1 Materialien

Tabelle 8: Aufgabe 4 - Verwendete Materialien

Bezeichnung	Eigenschaften	Menge
Widerstand	$10k\Omega$	2
	Braun - Schwarz - Orange - Gold	
Widerstand	unbekannt	k.A.
Kondensator	unbekannt	k.A
Mikrocontroller	Arduino Uno R3	1

In der Tabelle 8 sind unbekannte Widerstände in nicht angegebener Menge gelistet. Damit sind Testwiderstände gemeint, welche durch das Testen an der Schaltung gemessen werden können. Beim Versuch an dieser Schaltung sind die Widerstandswerte normalerweise bekannt, um die Richtigkeit der Schaltung zu bestätigen. Die Werte und Anzahl der verwendeten Widerstände sind jedoch nicht relevant. Selbiges gilt für die Kondensatoren.

7.2 Vorbereitung

7.2.1 Aufgabe 1

In dieser Aufgabe sind Widerstände an Hand ihrer Farbcodes zu bestimmen. Drei Widerstände sind gegeben. Anhand der Tabelle in Sektion 2 kann der Wert bestimmt werden.

Tabelle 9: Widerstandswerte

Ring 1	Ring 2	Ring 3	Ring 4	Widerstandswert	Toleranz
Gelb	Violett	Rot	Gold	$4,7k\Omega$	$\pm 5\%$
Rot	Weiß	Grün	Gold	$3,9M\Omega$	$\pm 5\%$
Blau	Grau	Rot	Silber	$6,8k\Omega$	$\pm 10\%$

7.2.2 Aufgabe 2

In Aufgabe 2 ist die Schaltung aus Abbildung 7 gegeben.

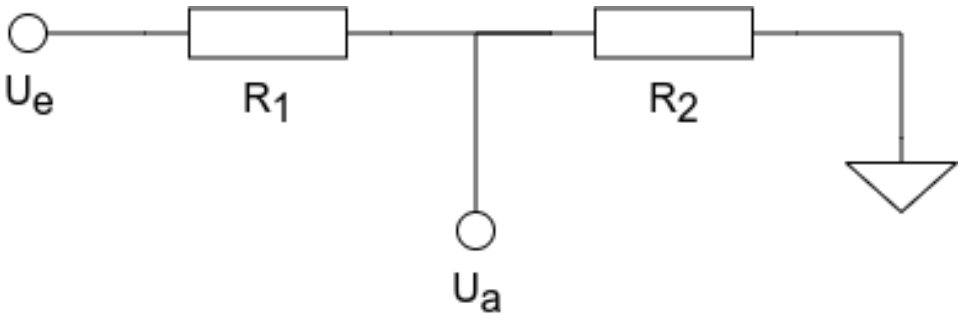

Abbildung 7: Schaltung aus der Angabe aus Aufgabe 4.2

Für den ersten Teil sind folgende Werte gegeben.

- $U_e = 5V$
- $U_a = 3V$

Nun soll ermittelt werden, welches Verhältnis zwischen den Widerständen R_1 und R_2 herrschen muss, damit der gegeben Zustand möglich ist. Um den

Spannungsabfall über einen Widerstand zu ermitteln, kann die Formel des Spannungsteilers verwendet werden.

$$U_R = \frac{U}{R_{ges}} * R \tag{3}$$

Unter berücksichtigung der Angaben können die Werte eingesetzt werden.

$$U_a = \frac{U_e}{R_1 + R_2} * R_2 \tag{4}$$

Nun kann umgeformt werden, um ein Verhältnis zwischen R_2 und R_{ges} zu erhalten.

$$U_a = \frac{U_e}{R_1 + R_2} * R_2 \Rightarrow \tag{5}$$

$$\frac{U_a}{U_e} = \frac{R_2}{R_{ges}} \Rightarrow \tag{6}$$

$$\frac{3}{5} = \frac{R_2}{R_{ges}} \tag{7}$$

Daher gilt:

$$\frac{3}{5} = \frac{R_2}{R_{ges}} \Rightarrow \frac{2}{5} = \frac{R_1}{R_{ges}} \Rightarrow \tag{8}$$

$$\frac{3}{5} * R_{ges} = R_2 \text{ und } \frac{2}{5} * R_{ges} = R_1 \Rightarrow \tag{9}$$

Das Verhältnis zwischen den Widerständen kann nun berechnet werden.

$$\frac{R_1}{R_2} = \tag{10}$$

$$\frac{\frac{2}{5} * \cancel{R_{ges}}}{\frac{3}{5} * \cancel{R_{ges}}} = \tag{11}$$

$$\frac{\frac{2}{\cancel{5}}}{\frac{3}{\cancel{5}}} = \tag{12}$$

$$\frac{2}{3} \tag{13}$$

Nun muss berechnet werden, wie hoch der Widerstand R_2 sein muss, wenn $R_1 = 10k\Omega$ gilt.

$$(10) \wedge (13) \Rightarrow \tag{14}$$

$$\frac{R_1}{R_2} = \frac{2}{3} \Rightarrow \tag{15}$$

$$\frac{3 * 10k\Omega}{2} = R_2 = 15k\Omega \tag{16}$$

Nun soll ein unbekannter Widerstand R_x berechnet werden, wenn $R_1 = 10k\Omega$ und $U_a = 1V$ gilt. Für den Spannungsteiler gilt nun folgendes.

$$U_a = \frac{U_e}{R_x + R_1} * R_x \Rightarrow \tag{17}$$

$$U_e = \frac{U_a * (R_x + R_1)}{R_x} \Rightarrow \tag{18}$$

$$\frac{U_e}{U_a} = \frac{R_x}{R_x} + \frac{R_1}{R_x} \Rightarrow \tag{19}$$

$$\frac{U_e}{U_a} - 1 = \frac{R_1}{R_x} \Rightarrow \tag{20}$$

$$R_x = \frac{R_1}{\frac{U_e}{U_a} - 1} \Rightarrow \tag{21}$$

$$R_x = \frac{R_1}{\frac{U_e - U_a}{U_a}} \Rightarrow \tag{22}$$

$$R_x = \frac{\frac{R_1 * U_a}{U_a}}{\frac{U_e - U_a}{U_a}} \Rightarrow \tag{23}$$

$$R_x = \frac{R_1 * U_a}{U_e - U_a} \tag{24}$$

Durch einsetzen der gegebenen Werte ergibt sich nun ein Wert für R_x.

$$(24) => \tag{25}$$

$$R_x = \frac{R_1 * U_a}{U_e - U_a} = \tag{26}$$

$$\frac{10k\Omega * 1V}{5V - 1V} = \tag{27}$$

$$\frac{10k\Omega V}{4V} \Rightarrow \tag{28}$$

$$R_x = 2,5k\Omega \tag{29}$$

7.2.3 Aufgabe 3

In dieser Aufgabe wird ein Kondensator über einen Wiederstand $R = 10k\Omega$ an einer Spannungsquelle U_q geladen. Die Spannung des Kondensators ist $U_c(0) = 0V$ und $U_c(0,1s) = 0,63 * U_q$. Es soll die Kapazität des Kondensators ermittelt werden.

Die Formel zur Berechnung der Spannung des Kondensators lautet folgendermaßen.

$$U_c(t) = \frac{1}{C} * I * t \tag{30}$$

Dies kann folgendermaßen umgeformt werden.

$$U_c(t) = \frac{1}{C} * I * t = \tag{31}$$

$$\frac{1}{C} * \frac{U_q}{R} * t \Rightarrow \tag{32}$$

$$C = \frac{U_c(t)}{\frac{U_q}{R} * t} \tag{33}$$

Nun können die Werte der Angabe eingesetzt werden.

$$(33) \Rightarrow \tag{34}$$

$$C = \frac{0,63 * U_q}{\frac{U_q}{10k\Omega} * 0,1s} = \tag{35}$$

$$\frac{0,63}{\frac{1}{10k\Omega} * 0,1s} = \tag{36}$$

$$15\mu F \tag{37}$$

Nun soll ermittelt werden, wie lange es dauert, denn Kondensator vollständig zu laden. Wir wissen:

$$\tau = RC \Rightarrow \tag{38}$$

$$\tau = 0,15s \tag{39}$$

Weiters ist bekannt

$$U_c \geq 0,99U_Q \Leftrightarrow \text{nach } 5\tau \tag{40}$$

daraus folgt

$$t_{99\%} = 5\tau = 0,75s \tag{41}$$

7.3 Praktikumsaufgabe

7.3.1 Aufgabe 1

Es wurde die in Abbildung 8 gezeigte Schaltung implementiert. Am Steckbrett ist eine ähnliche Schaltung wie in Abbildung 7 zu sehen. Der Pin A0 ist zwischen den bekannten $10k\Omega$ und den unbekannten Widerstand geschaltet. Nun sind folgende WGrößen bekannt.

- V_{out} des Mikrocontrollers $V_{out} = 5V$
- Der bekannte Widerstand zwischen V_{out} und A0, $R_1 = 10k\Omega$

Durch die Überlegung von (26) kann im Mirocontroller ein Programm implementiert werden, welches den unbekannten Widerstand errechnet.

Im nachfolgendem Text wird der Programmcode der Aufgabe erläutert. Einzelne Teile des Codes werden ausgewählt und beschrieben. Am Ende dieser Sektion befindet sich der vollständige Programmcode.

```
1 const int PIN_VOLTAGE = A0;
2 const float U_IN = 5;
3 const float RESOLUTION = U_IN / 1024;
4 const int RESISTOR = 10 * 1000;
```

Listing 24: Konstanten der Aufgabe 4.1

Durch den Code in Listing 24 werden die bekannten Größen angelegt. *U_IN* gibt die Spannung in Volt an, während *RESISTOR* den Widerstandswert abbildet. Die Konstante *RESOLUTION* bedarf genauerer Erklärung.

Die analogen Pins des Microkontroller können Spannungen zwischen 0V und 5V auslesen. Um diese als Zahl darzustellen, wird ein Wert zwischen 0 und 1024 zurückgegeben. Hierbei steht 0 für 0V und 1024 für 5V. Das Interval $(0V, 5V)$ wird daher auf das Interval $(0, 1024)$ abgebildet. Um die Spannung

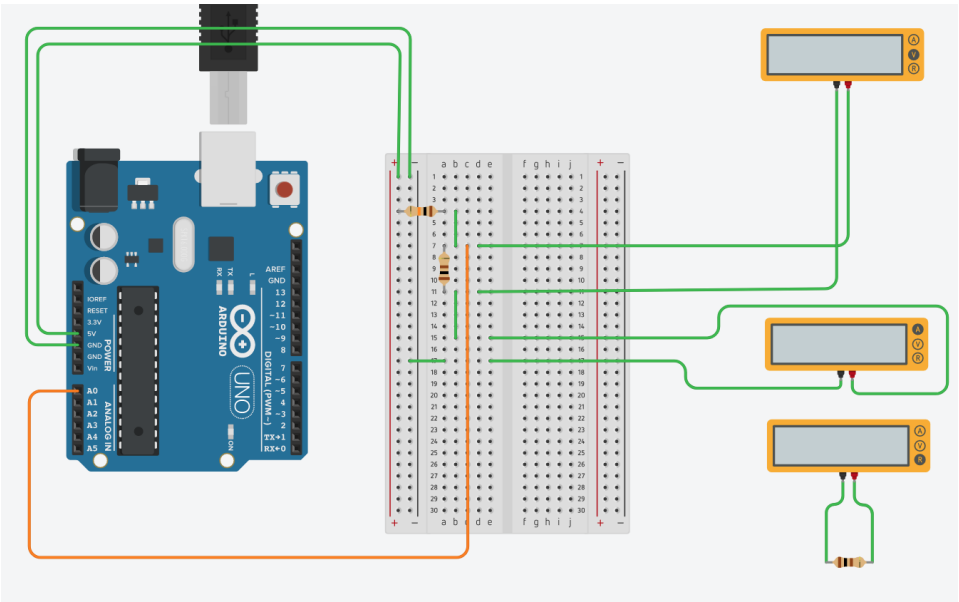

Abbildung 8: Implementierter Stromkreis von Aufgabe 4.1

nun in Volt zu erhalten, muss mit der Wert aus A0 mit $\frac{5V}{1024}$ multipliziert werden. Dieser Multiplikator wird in der Konstante *RESOLUTION* gespeichert.

Der verwendete Microkontroller konfiguriert analoge Pins standardmäßig als Input-Pins. Eine explizite Konfiguration als solche in der *setup*() Funktion ist also nicht notwendig.

```
1  void loop()
2  {
3      int value = analogRead(PIN_VOLTAGE);
4      float u_a = value * RESOLUTION;
5      float r = (u_a * RESISTOR) / (U_IN - u_a);
6
7      Serial.println("");
8      Serial.print(u_a);
9      Serial.println(" V");
10     Serial.print(r);
11     Serial.println(" Ohm");
12     delay(1000);
13 }
```

Listing 25: Programmschleife der Aufgabe 4.1

In Listing 25 wird zuerst der derzeitige Wert des Pins A0 ausgelesen und in der variable *value* abgelegt. Dieser Wert wird nun mit den in Listing 24 angelegten Multiplikator *RESOLUTION* multipliziert, um die Spannung U_a, vergleiche 1, zu erhalten. Durch die Überlegung aus (mal26) kann nun der Widerstand R_2, bzw. R_x, errechnet werden.

Danach erfolgt die Ausgabe über der errechneten Größen über die serielle Schnittstelle. Ein Pausieren des Programms wird durchgeführt, um ein angenehmeres Lesen der Ausgabe zu ermöglichen.

```
1  const int PIN_VOLTAGE = A0;
2  const float U_IN = 5;
3  const float RESOLUTION = U_IN / 1024;
4  const int RESISTOR = 10 * 1000;
5
6  void setup()
7  {
8      Serial.begin(9600);
9  }
10
11 void loop()
12 {
13     int value = analogRead(PIN_VOLTAGE);
14     float u_a = value * RESOLUTION;
15     float r = (u_a * RESISTOR) / (U_IN - u_a);
16
17     Serial.println("");
18     Serial.print(u_a);
19     Serial.println(" V");
20     Serial.print(r);
21     Serial.println(" Ohm");
22     delay(1000);
23 }
```

Listing 26: Vollständiger Programmcode der Aufgabe 4.1

7.3.2 Aufgabe 2

In dieser Aufgabe, soll die Kapazität eines Kondensators ermittelt werden. Es wurde die Schaltung aus Abbildung 9 implementiert. Im Unterschied zur Schaltung aus Abbildung 8, wurde der unbekannte Widerstand durch einen Kondensator ersetzt, welcher eine unbekannte Kapazität besitzt.

Im nachfolgendem Text wird der Programmcode der Aufgabe erläutert. Einzelne Teile des Codes werden ausgewählt und beschrieben. Am Ende dieser

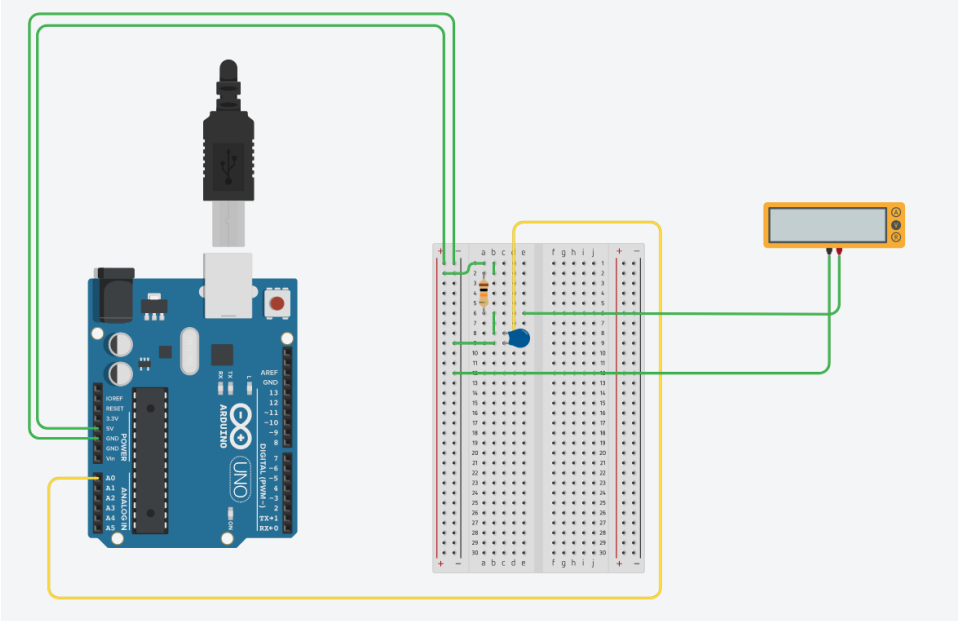

Abbildung 9: Implementierter Stromkreis von Aufgabe 4.2

Sektion befindet sich der vollständige Programmcode.

```
#include <math.h>
```

Listing 27: Preprozessoranweisung

In Listing 27 wurde eine Preprozessoranweisung verwendet. Diese fügt eine C-Bibliothek hinzu, die verschiedene mathematische Funktionen zur Verfügung stellt. Die Funktion die in dieser Aufgabe verwendet wurde, ist die *log*() Funktion. Es werden außerdem die selben Konstanten wie in Listing 24 verwendet.

```
double c = 0.0;

void loop()
{
  int value = analogRead(PIN_VOLTAGE);
  double u_a = value * RESOLUTION;

  if (u_a < 3.6) {
    double t = (double)millis() / 1000.0;
    if (u_a > 0) {
      c = -t / (log(1 - (u_a / U_IN)) * RESISTOR);
```

```
12          Serial.print(c, 6);
13          Serial.println(" F");
14      }
15    }
16 }
```

Listing 28: Programmschleife

Im Listing 28 wird nun die Kapazität berechnet. Dafür wird zusätzlich zum Wert des Pins A0 die vergangene Zeit miteinbezogen. Die Berechnung der Kapazität ergibt sich aus folgender Überlegung.

Die Formel der Spannung eines Kondensators ist gegeben durch:

$$U_c(t) = U * (1 - e^{\frac{-t}{RC}}) \tag{42}$$

Durch Umformen kann auf die Kapazität C umgeformt werden.

$$U_c(t) = U * (1 - e^{\frac{-t}{RC}}) \Rightarrow \tag{43}$$

$$\frac{U_c(t)}{U} - 1 = -e^{\frac{-t}{RC}} \Rightarrow \tag{44}$$

$$ln(1 - \frac{U_c(t)}{U}) = \frac{-t}{RC} \Rightarrow \tag{45}$$

$$\frac{log_{10}(1 - \frac{U_c(t)}{U})R}{-t} = \frac{1}{C} \Rightarrow \tag{46}$$

$$C = \frac{-t}{log_{10}(1 - \frac{U_c(t)}{U})R} \tag{47}$$

Um ein akkurates Ergebnis zu erhalten, wird die Formel under den folgenden Bedingungen verwendet. Die Spannung am Kondensator u_a muss unter $3,6V$ betragen, aber höher als $0V$ sein. Letztere Bedingung wurde gewählt, da ansonsten eine Division durch 0 möglich wäre. Erstere, da beim Testen des Programms festgestellt wurde, dass die errechnete Kapazität mit fortschreitender Zeit ungenauer, bzw. falsch wurde. Der in Listing 28 verwendete Wert von $3,6V$ wurde verwendet, da dieser in der Ausgabe am Monitor den genauesten Wert zu liefern schien.

```
1 #include <math.h>
```

```
2
3  const int PIN_VOLTAGE = A0;
4
5  const double U_IN = 5;
6  const double RESOLUTION = U_IN / 1024;
7  const double RESISTOR = 10 * 1000;
8
9  double c = 0.0;
10
11 void setup()
12 {
13   Serial.begin(9600);
14 }
15
16 void loop()
17 {
18   int value = analogRead(PIN_VOLTAGE);
19   double u_a = value * RESOLUTION;
20
21   if (u_a < 3.6) {
22     double t = (double)millis() / 1000.0;
23     if (u_a > 0) {
24       c = -t / (log(1 - (u_a / U_IN)) * RESISTOR);
25         Serial.print(c, 6);
26         Serial.println(" F");
27     }
28   }
29 }
```

Listing 29: Vollständiger Programmcode der Aufgabe 4.2

7.4 Fehlerdiskussion

Für die Berechnung der Kapazität wurde zuerst die Überlegung aus (33) verwendet. Die Verwendung dieser Formel führte jedoch zu unbefriedigenden Messungen und wurde mit der Überlegung aus (47) ersetzt. Diese führte ebenso zu unbefriedigenden, jedoch genaueren Werten als (33). Erst als das Limit von $u\_a < 3,6$ eingeführt wurde, wurden befriedigende Messergebnisse erzielt. Womöglich könnte mit dieser Einschränkung auch mit (33) ein gleich gutes Ergebnis erreicht werden.

Des weiteren wurde der Wert von $3,6V$ für die Obergrenze von u_a durch experimentieren, d.h., willkürlich, festgelegt. Nachdem der Wert von $3,6$ nahe an $0,63U = 0,63 * 5V = 3,15V$ liegt, ist es naheliegend das ein weniger willkürlicher Wert $3,15V$ sein könnte. Dieser Wert ergibt sich aus der

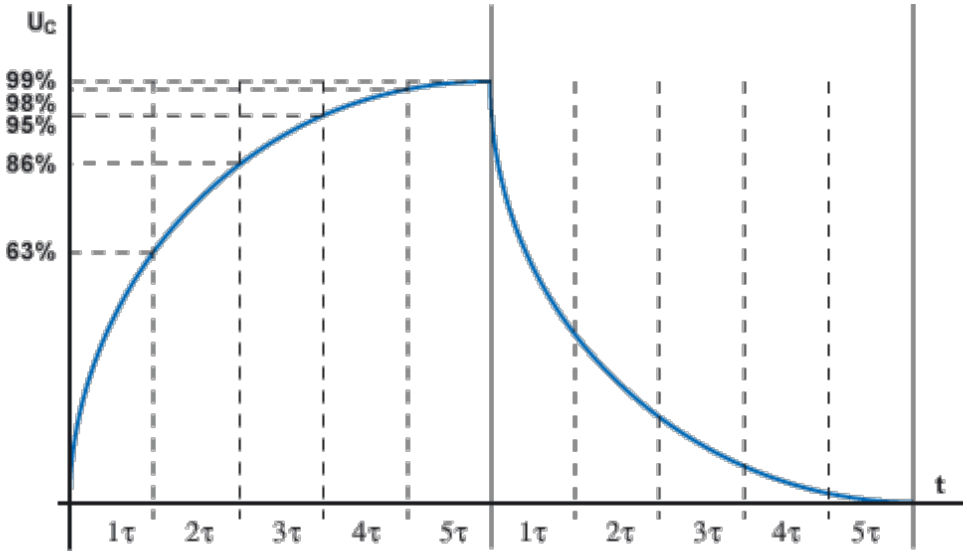

Abbildung 10: Lade- und Entladekurve eines Kondensator [3]

Spannung am Kondesator nach dem er 1τ lang, siehe Abbildung 10, geladen wurde.

7.5 Zusammenfassung

8 Aufgabe 5 - Ansteuerung eines Lautsprechers

In dieser Aufgabe wird ein 4 Bit großes Signal, mittels einer Schaltung, in ein analoges Signal umgewandelt. Dieses Signal soll dann auf einem Lautsprecher ausgegeben werden.

Die Umwandlung von einem digitalen Signal zu einem Analogen geschieht mithilfe eines sogenannten R2R-Netzwerk. Dieses addiert das Signal eines Bits proportional zu dessen Position zum anaolgen Signal. Das resultierende Signal wird dann durch einen Operationsverstärker soweit verstärkt, dass es hörbar an einem Lautsprecher ausgegeben werden kann.

Ein R2R-Netzwerk ist eine Schaltung aus Widerständen welche jeweils die Größe R und $2R$ besitzen. Ein Beispiel dieser Schaltung befindet sich im Abschnitt 8.2 und wird auch in der Praktikumsaufgabe implementiert.

8.1 Materialien

Tabelle 10: Aufgabe 5 - Verwendete Materialien

Bezeichnung	Eigenschaften	Menge
Widerstand	$100k\Omega$	5
	Braun - Schwarz - Gelb - Gold	
Widerstand	$51k\Omega$	3
	Grün - Braun - Orange - Gold	
Widerstand	$3k\Omega$	1
	Orange - Schwarz - Rot - Gold	
Kondensator	$10\mu F$ 1	
Operationsverstärker	MCP6241	1
Lautsprecher	k.A.	1
Mikrocontroller	Arduino Uno R3	1

8.2 Vorbereitung

8.2.1 Aufgabe 1

In Abbildung 11 ist das gegebene R2R-Netzwerk zu sehen.

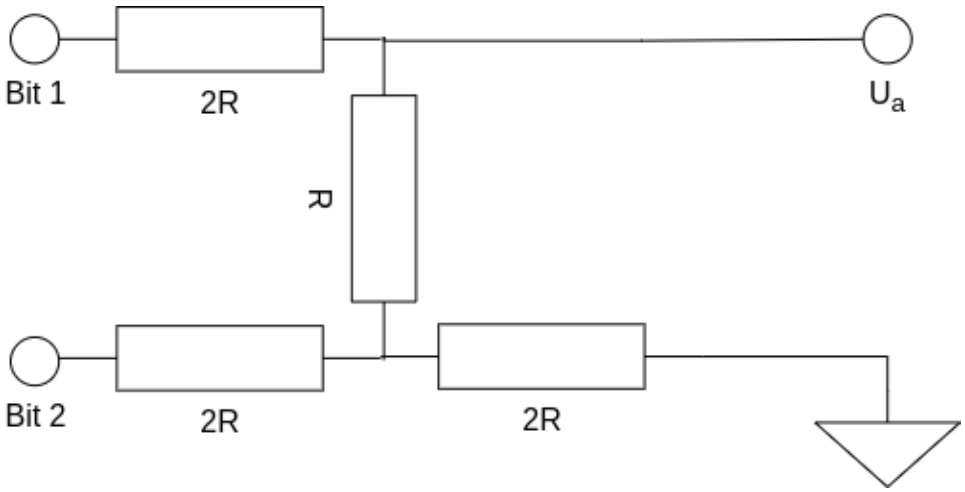

Abbildung 11: Gegebene R2R-Schaltung

a,b) Es sind die Werte für U_a allgemein zu berechnen, unter der Annahme der folgenden Werte für U_{Bit1} und U_{Bit2}.

U_{Bit1}	U_{Bit2}
$0V$	$0V$
$0V$	U_{Bit}
U_{Bit}	$0V$
U_{Bit}	U_{Bit}

Um die Werte für U_a zu berechnen, ist es am besten, zuerst die allgemeine Formel, die in **b)** gesucht wird zu finden, um dann einzusetzen. Da U_{Bit1} und U_{Bit2} als Spannungsquellen betrachtet werden können, kann mittels dem Überlagerungssatz von Helmholz eine allgemeine Formel gefunden werden.

Fall 1 - U_{Bit2} wird kurzgeschlossen:

Es resultiert das Schaltbild von Abbildung 12.

Die mit Blau markierten Komponenten können zu einer Schaltung mit einer Ersatzspannungsquelle mit Innenwiderstand zusammengefasst werden. Der Innenwiderstand beträgt demnach:

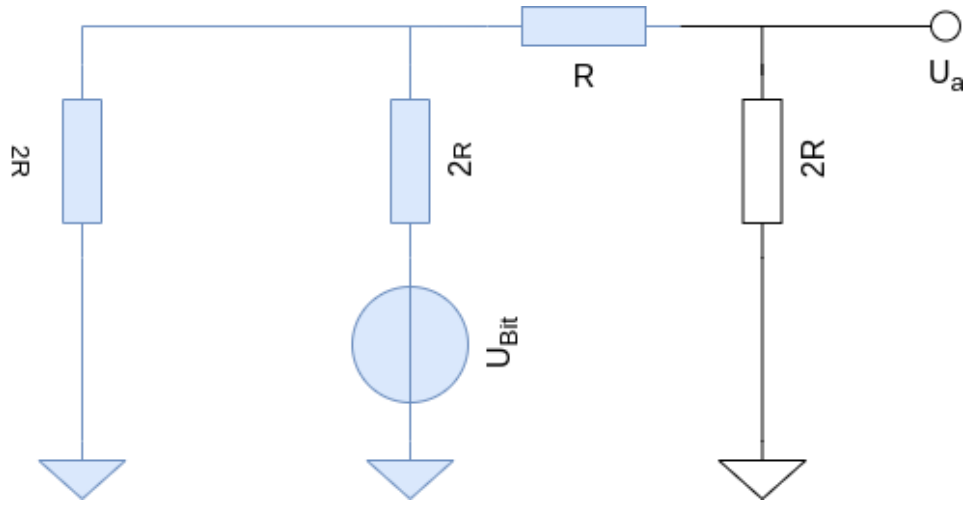

Abbildung 12: Fall 1

$$R_i = 2R||2R + R \tag{48}$$
$$= 2R \tag{49}$$

Da bei eine Leerlaufspannung die Klemmen offen sind, wird der Widerstand R effektiv kurzegschlossen. Der Leerlaufwiderstand is daher der Spannungsabfall über den Widerstand $2R$:

$$U_L = U_{2R} \tag{50}$$
$$= I * 2R \tag{51}$$
$$= \frac{U_{Bit} * 2R}{R_g} \tag{52}$$
$$= \frac{U_{Bit} * 2R}{2R + 2R} \tag{53}$$
$$= \frac{U_{Bit}}{2} \tag{54}$$
$$\tag{55}$$

Die Resultierende Spannung U_{a1} ist dann gleich der Spannung über den nicht in der Ersatzspannungsquelle inkludierten Widerstands $2R$:

$$U_{a1} = I * 2R \tag{56}$$
$$= \frac{U_L * 2R}{R_i + 2R} \tag{57}$$
$$= \frac{\frac{U_{Bit}}{2} * 2R}{2R + 2R} \tag{58}$$
$$= \frac{U_{Bit}R}{4R} \Rightarrow \tag{59}$$
$$\underline{\underline{U_{a1} = \frac{U_{Bit}}{4}}} \tag{60}$$

Fall 2 - U_{Bit1} wird kurzgeschlossen:

Es resultiert das Schaltbild von Abbildung 13.

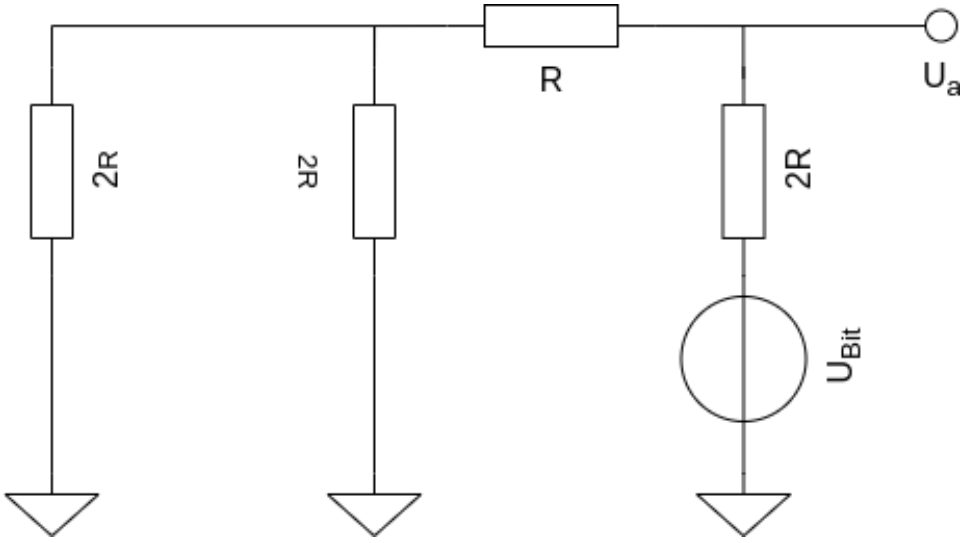

Abbildung 13: Fall 2

Die linken Widerstände können zusammengefasst werden zu $2R||2R = R$ Es folgt:

$$U_{a2} = U_{Bit} - U_{2R} \tag{61}$$
$$= U_{Bit} - \frac{U_{Bit}}{R_g} * U_{2R} \tag{62}$$
$$= U_{Bit} - \frac{U_{Bit} * 2R}{4R} \tag{63}$$
$$= \frac{U_{Bit}}{2} \tag{64}$$

Aus Fall 1 und Fall 2 folgt nun die Lösung für **b)**:

$$U_a = U_{a1} + U_{a2} \tag{65}$$
$$= \frac{U_{Bit}}{2} + \frac{U_{Bit}}{4} \Rightarrow \tag{66}$$

Man kann erkennen, dass die Spannung für jedes Bit halbiert wird. D.h., die Gewichtung eines Bits ist $\frac{Bit_n}{Bit_{n-1}} = \frac{1}{2}$ Daraus folgt die Formel:

$$U_{a(1...n)} = \sum_{k=1}^{n} \frac{U_{Bit}}{2^n} \tag{67}$$

Es folgt daraus die folgende Wertetabelle für **a)**

U_{Bit1}	U_{Bit2}	U_a
$0V$	$0V$	$0V$
$0V$	U_{Bit}	$\frac{U_{Bit}}{4}$
U_{Bit}	$0V$	$\frac{U_{Bit}}{2}$
U_{Bit}	U_{Bit}	$\frac{3}{4}U_{Bit}$

c) Es ist die Berechnungsvorschrift auf ein 4-Bit R2R Netzwerk anzuwenden:

$$U_{a(1_4)} = \frac{U_{Bit}}{2} + \frac{U_{Bit}}{4} + \frac{U_{Bit}}{8} + \frac{U_{Bit}}{16} \tag{68}$$
$$= \frac{8U_{Bit}}{16} + \frac{4U_{Bit}}{16} + \frac{2U_{Bit}}{16} + \frac{U_{Bit}}{16} \tag{69}$$
$$= \frac{15U_{Bit}}{16} \tag{70}$$

8.2.2 Aufgabe 2

Es ist zu Berechnen, welche Werte ausgegeben werden, damit ein gegebenes Ausganssignal erreicht wird. Für die Ausgabe gibt es 16 verschiedene Werte. Je niedriger der Wert, desto niedriger ist die resultierende Ausgangsspannung. Zur Berechnung wird daher folgende Formel verwendet. Da eine Ganzzahl benötigt wird werden die Ergebnisse gerundet.

$$n = 5/16 * U_{sin} \tag{71}$$

Tabelle 11: Zuordnung der Spannungswerte

U_{sin}	Dezimal	Hexadezimal
$2,3$	7	1
$3,0$	9	5
$3,7$	11	3
$4,2$	13	7
$4,6$	14	F
$4,7$	15	7
$4,6$	14	F
$4,2$	13	7
$3,7$	11	3
$3,0$	9	5
$2,3$	7	1
$1,6$	5	5
$1,0$	3	3
$0,4$	1	1
$0,1$	0	0
$0,0$	0	0
$0,1$	0	0
$0,4$	1	1
$1,0$	3	3
$1,6$	5	5

8.2.3 Aufgabe 3

Es soll für bestimmte Frequenzen berechnet werden, wielange ein Bitmuster anliegen muss. Dafür ist die Frequenz in Hz gegeben und es werden laut

Angabe 20 Bitmuster pro Signalperiode benötigt. Da die Frequenz der Kehrwert der Zeit pro Signalperiode ist, kann folgende Formel zu berechnung der gesuchten Zeit verwendet werden.

$$n = \frac{\frac{1}{frequenz}}{20} \tag{72}$$

Damit ergiben sich folgende Werte für bestimmte Frequenzen.

Tabelle 12: Zeit pro Bitmuster

Ton	Frequenz (Hz)	Zeit pro Bitmuster (μs)
c'	262	$190,84$
d'	294	$170,068$
e'	330	$151,\overline{51}$
f'	349	$143,266$
g'	392	$127,5\overline{51}$
a'	440	$113,\overline{63}$
h'	494	$101,215$

8.3 Praktikumsaufgabe

Es wurde die Schaltung implementiert aus Abbildung 14 implementiert. Nach dem besprochenen R2R-Netzwerk, welches die Pins 7, 6, 5 und 4 als Eingabe hat, folgt die Verstärkung des Signals über einen Operationsverstärker. Da der Operationsverstärker Signale zwischen 0V und 5V erzeugt, der Lautsprecher jedoch eine Spannung von $-2,5V \leq U \leq 2,5V$ benötigt folgt ein Kondensator, welche die Spannung zentiert. Der darauf folgende Widerstand regelt die Lautstärke des Tons. Der Widerstand kann daher beliebig gewählt werden, je nach Vorlieben für verschiedene Lautstärken.

Das Programm soll weiters ein Klavier implementieren. D.h., bei Eingabe der Tasten "a", "s", "d", "f", "g", "h", "j" sollen jeweils die Töne "c"', "d"', "e"', "f"', "g"', "a"', "h"' ausgegeben werden. Um die benötigte Geschwindigkeit beim Schalten der Pins zu erreichen, werden die Register die diese verwalten direkt angesprochen.

Im nachfolgenden Text wird der Programmcode der Aufgabe erläutert. Einzelne Teile des Codes werden ausgewählt und beschrieben. Am Ende dieser

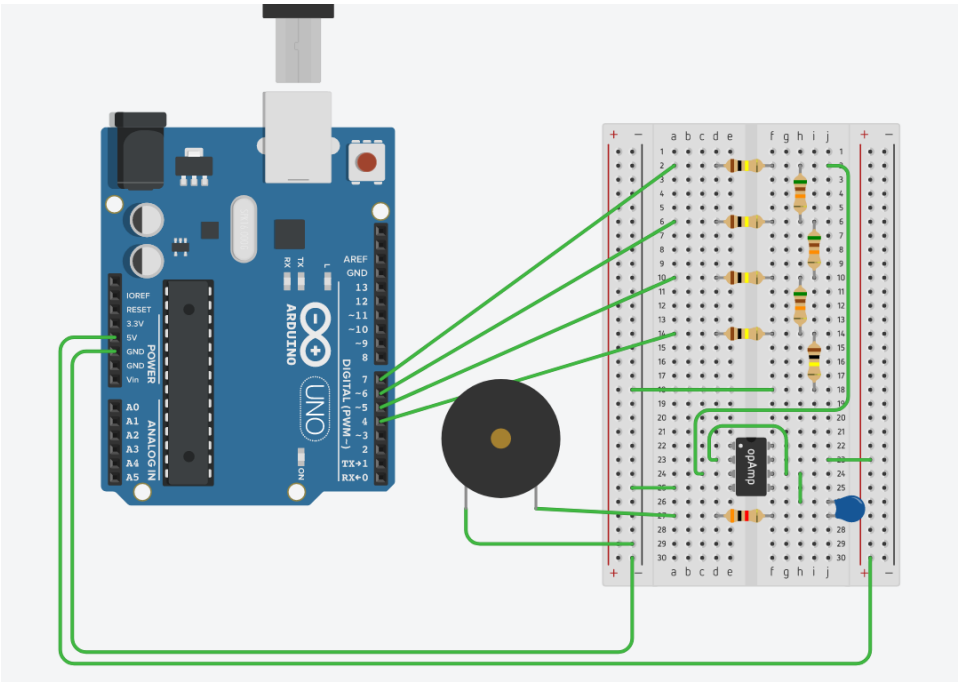

Abbildung 14: Implementierte Schaltung aus Aufgabe 5

Sektion befindet sich der vollständige Programmcode.

```
1 void setup()
2 {
3     Serial.begin(9600);
4
5     DDRD = DDRD | B11111100;
6 }
```

Listing 30: Anlegen der Pins über Register

In Listing 30 werden die benötigten Pins konfiguriert. Das Register "DDRD" verwaltet die Pins 0 bis 7. Durch das übergebene Bitmuster, werden die Pins 7 bis 2 als Ausgabepins konfiguriert. Konfiguration über die Methode *pinMode*() ist nicht mehr nötig. Des weiteren ist das Einstellen der Pins 2 und 3 ein Artifakt des Debuggens des Programms und ist nicht notwendig.

```
1 void loop()
2 {
3     double time = (double)millis() / 1000;
4
```

```
5       if (Serial.available() > 0) {
6           last_read = Serial.read();
7       }
8       ...
9   }
```

Listing 31: Lesen von Eingaben

Listing 31 zeigt, wie Eingaben des Benutzers eingelesen werden können. Durch das Aufrufen von *Serial.available*() wird abgefragt, ob eine Eingabe getätigt wurde. *Serial.read*() speichert die Eingabe dann in einer globalen Variable.

```
1       switch(last_read) {
2           case A:
3               play_frequency(NOTE_C, time);
4           break;
5           case S:
6               play_frequency(NOTE_D, time);
7           break;
8           case D:
9               play_frequency(NOTE_E, time);
10          break;
11          case F:
12              play_frequency(NOTE_F, time);
13          break;
14          case G:
15              play_frequency(NOTE_G, time);
16          break;
17          case H:
18              play_frequency(NOTE_A, time);
19          break;
20          case J:
21              play_frequency(NOTE_H, time);
22          break;
23          default:
24              PORTD = B00000000;
25          break;
26      }
27  }
```

Listing 32: Prüfen von Eingaben

Benutzereingaben von Listing 31 wird hier, in Listing 32, überprüft. Die Konstanten die hier verwendet werden, sind den ASCII-Werten der jeweiligen Buchstaben zugeordnet. Dies hat zur Folge, dass nur Kleinbuchstaben als korrekte Eingaben erkannt werden. Wird eine Eingabe nicht erkannt,

werden die Pins mittels dem Register $PORTD$ ausgeschaltet. $PORTD$ ist ein Register, ähnlich wie $DDRD$, welches die Ausgabe der Pins von 7 bis 0 steuert.

```
void play_frequency(int frequency, double time) {
    //sin is from -1 to 1 so +1 to start at 0
    double y = sin(frequency * time) + 1;
    byte value = y / 2 * 16;
    PORTD = value << 2;
}
```

Listing 33: Abspielen von Frequenzen

Die Methode $playFrequency(\dots)$ aus Listing 33 spielt schließlich den Ton ab. Sie übernimmt eine gewünschte Frequenz, sowie die Zeit, welche seit dem Programmstart vergangen ist. Durch das Aufrufen von sin(...) mit den genannten Variablen wird ein Wert für das Sinussignal berechnet. Dieser Wert ist $-1 \leq n \leq 1$ und muss daher mittels der Addition von 1 auf $0 \leq n \leq 2$ gebracht werden, um ihn anschließend auf $0 \leq n \leq 1$ zu normalisieren. Mittels der Multiplikation mit 16 wird der Wert errechnet, welcher laut Vorbereitung benötigt wird. Durch das Bitshifting in der letzten Zeile, um zwei Bitstellen, werden die entsprechenden Pins geschaltet.

```
#include <math.h>

const int PIN_1 = 7;
const int PIN_2 = 6;
const int PIN_3 = 5;
const int PIN_4 = 4;

const int FREQUENCY = 440;

const int NOTE_C = 262;
const int NOTE_D = 294;
const int NOTE_E = 330;
const int NOTE_F = 349;
const int NOTE_G = 392;
const int NOTE_A = 440;
const int NOTE_H = 494;

const int A = 'a';
const int S = 's';
const int D = 'd';
const int F = 'f';
const int G = 'g';
const int H = 'h';
const int J = 'j';
```

```
25
26 double previous_time = 0;
27
28 int last_read = 97;
29
30 void setup()
31 {
32   Serial.begin(9600);
33
34   DDRD = DDRD | B11111100;
35 }
36
37 void loop()
38 {
39   double time = (double)millis() / 1000;
40
41   if (Serial.available() > 0) {
42     last_read = Serial.read();
43   }
44
45   switch(last_read) {
46     case A:
47       play_frequency(NOTE_C, time);
48     break;
49     case S:
50       play_frequency(NOTE_D, time);
51     break;
52     case D:
53       play_frequency(NOTE_E, time);
54     break;
55     case F:
56       play_frequency(NOTE_F, time);
57     break;
58     case G:
59       play_frequency(NOTE_G, time);
60     break;
61     case H:
62       play_frequency(NOTE_A, time);
63     break;
64     case J:
65       play_frequency(NOTE_H, time);
66     break;
67     default:
68       PORTD = B00000000;
69     break;
70   }
71 }
72
73 void play_frequency(int frequency, double time) {
```

```
74
75    //sin is from -1 to 1 so +1 to start at 0
76    double y = sin(frequency * time) + 1;
77    byte value = y / 2 * 16;
78    PORTD = value << 2;
79  }
```

Listing 34: Vollständiger Programmcode der Aufgabe 5

8.4 Fehlerdiskussion

In der Methode $play_frequency$ werden die falschen Pins angesprochen. Die Bits der *value* Variable müssten um 4 Stellen verschoben werden anstelle von 2. Die Werte von 0 bis 16 werden mittels 4 Bits abgebildet, während in dem Register 8 Bits vorhanden sind. In der derzeitigen Implementierung werden maximal die Pins 5 und 4 angesprochen, nicht jedoch 7 und 6. Mittels eines Shifts um 4 anstelle von 2 können alle benötigten Pins angesprochen werden.

8.5 Zusammenfassung

9 Aufgabe 6 - Motorsteuerung mit Nähreungssensor

In dieser Aufgabe soll mithilfe ein Motor angesteuert werden. Um die Geschwindigkeit des Motors zu steuern, wird eine Regulierung mittels PWM-Signal realisiert. Der Motor soll weiters auf einen Ultraschallsensor reagieren, welche die Entfernung zu Objekten misst. Nähert sich ein Objekt auf weniger als 30cm, so soll der Motor ausgeschaltet, bzw. seine Geschwindigkeit auf 0 reduziert, werden. Eine 7-Segment Anzeige soll die derzeitige Geschwindigkeitsstufe des Motors anzeigen und mittels Schieberegister gesteuert werden.

Zwei Taster sollen einerseits die Laufrichtung des Motors und andererseits die Geschwindigkeit ändern. Wird Taster A betätigt, soll sich die Geschwindigkeit, mit der sich der Motor gegen den Uhrzeigersinn dreht, um eins erhöht werden. Dreht sich der Motor bei Betätigung des Tasters im Uhrzeigersinn, so wird die Laufrichtung und Geschwindikeit entsprechend geändert. Wird Taster B gedrückt, soll das selbe Verhalten für die andere Laufrichtung implementiert werden.

Die Geschwindikeiten sollen hierbei zwischengespeichert und wiederhergestellt werden. Es soll daher das folgende Beispiel möglich sein.

- Motor dreht sich mit Geschwindigkeit 3 im Uhrzeigersinn
- Taster A wird betätigt
- Motor dreht sich mit Geschwindigkeit 1 gegen den Uhrzeigersinn
- Taster A wird betätigt
- Motor dreht sich mit Geschwindigkeit 2 gegen den Uhrzeigersinn
- Taster B wird betätigt
- Motor dreht sich mit Geschwindigkeit 4 im Uhrzeigersinn
- Taster B wird betätigt
- Motor dreht sich mit Geschwindigkeit 5 im Uhrzeigersinn
- Taster A wird betätigt
- Motor dreht sich mit Geschwindigkeit 3 gegen den Uhrzeigersinn

Würde die Geschwindigkeit mittels Tastendruck auf > 9 gebracht werden wird sie wieder auf 0 zurückgesetzt.

9.1 Materialien

Tabelle 13: Aufgabe 6 - Verwendete Materialien

Bezeichnung	Eigenschaften	Menge
7-Segement Anzeige		1
Ultraschallsensor	HC-SR04	1
Schieberegister	74HC595	1
Taster	4 Polig	2
Motortreiber	L293D	1
LED	Rot	1
LED	Grün	1
Widerstand	150Ω	10
	Braun - Grün - Braun - Gold	
Widerstand	$10k\Omega$	2
	Braun - Schwarz - Orange - Gold	
Diode		2
DC-Motor		1
Mikrocontroller	Arduino Uno R3	1

9.2 Vorbereitung

9.2.1 Aufgabe 1

Als erste Aufgabe ist zu Überlegen, wie die Entfernungsmessung vorgenommen werden kann. Dafür wird die Beschreibung des HC-SR04 Moduls von microcontroller.net verwendet [2].

Es gibt für die Messung hierbei zwei Dinge zu berücksichtigen Erstens, ein Messzyklus wird mit einer fallenden Flanke am Triggereingang gestartet. Zweitens, das Ziel ist es, so oft wie möglich zu messen, da eine Sicherheitt gewährleistet werden soll. D.h., der Motor soll ausschalten, da eine Gefährdung durch Nähe angenommen werden soll. Daher muss ermittelt werden, wieviel ein Zeit eine Messung benötigt, um die Messabstände so kurz wie möglich zu machen.

Tabelle 14: Aufgabe 6.2

	A	B	C	D	E	F	G	DP
0	1	1	1	1	1	1	0	0
1	0	1	1	0	0	0	0	0
2	1	1	0	1	1	0	1	0
3	1	1	1	1	0	0	1	0
4	0	1	1	0	0	1	1	0
5	1	0	1	1	0	1	1	0
6	1	0	1	1	1	1	1	0
7	1	1	1	0	0	0	0	0
8	1	1	1	1	1	1	1	0
9	1	1	1	1	0	1	1	0

Laut microcontroller.net kann ein 50Hz Rechtecksignal angelegt werden, um eine kontinuierliche Messung zu erreichen. Es muss also alle 20ms eine fallende Flanke angelegt werden. Eine fallende Flanke am Echo-Ausgang gibt dann an, dass das Echo des Ultraschallsignals gemessen wurde. Mit der Dauer t zwischen dem Start und Ende der Messung, sowie der Schallgeschwindigkeit in Luft $c_{Luft} = 343,5\frac{m}{s}$, kann dann die Entfernung d gemessen Werden. Diese Dauer muss dann noch halbiert werden, da die gemessene Dauer den Hin- **und** Rückweg beinhaltet.

$$d = \frac{v * t}{2} \tag{73}$$

9.2.2 Aufgabe 2

Es sollen die Bytes gefunden werden, welche an das Schieberegister übertragen werden müssen, um die Zahlen von 1 bis 9 anzuzeigen.

9.2.3 Aufgabe 3

Wie in Sektion 4.2 muss ein geeigneter Widerstand für die LEDs gefunden wernde. Die Berechnung und des Widerstands und die reale Auswahl aus der E6-Reihe erfolgt wie in Sektion 4.2. D.h., es wurden 200Ω berechnet und der 150Ω Widerstand aus der E6-Reihe ausgewählt.

9.3 Praktikumsaufgabe

Es wurde die Schaltung aus Abbildung 15 implementiert. Für die Taster zur Geschwindigkeitssteuerung wurden die Pins 2 und 3 verwendet. Des weiteren wurden sie, wie in Abschnitt 5 als Pullup-Eingang konfiguriert und geschalten.

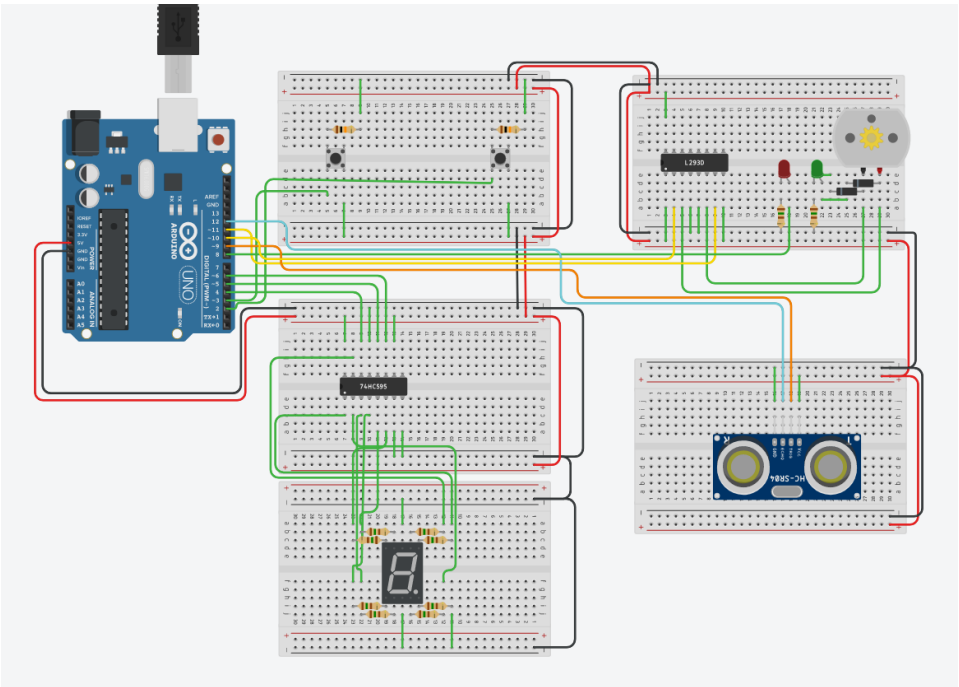

Abbildung 15: Implementierte Schaltung aus Aufgabe 6

Die Pins 4, 5 und 6 dienen der Steuerung des Schieberegisters. Pin 5 dient dazu, den Schreibvorgang zu starten. Ein *LOW* Signal an diesem Pin führt dazu, dass Daten in das Schieberegister geschoben werden können. Pin 4 legt das Datensignal an und Pin 6 dient zum synchronisieren des Datensignals mit den weiterschieben im Register. D.h., an Pin 6 liegt das Clock-Signal an.

Die Ausgänge 1 bis 7 des Schieberegisters sind an den Eingängen A bis G der 7-Segment Anzeige angeschlossen. Der Ausgang 8 sow der eingang *DP* der 7-Segment Anzeige werden nicht benötigt und werden folglich an Masse angeschlossen. Da die Anzeige aus LEDs besteht wird vor jedem Eingang

ein Vorwiderstand von 150Ω angeschlossen.

Die Pins 12 und 9 sind dem Ultraschallmodul angeschlossen. Der Trigger Eingang wird von Pin 12 gesteuert. Das Echo-Signal wird dann von Pin 9 empfangen.

Die Pins 11 und 10 werden für die Steuerung des Motors verwendet. Um die Geschwindigkeit des Motors zu Steuern wird ein PWM-Signal verwendet, daher müssen auch die Pins ein PWM-Signal unterstüzen. Der Motortreiber schaltet die Versogungsspannung dann entsprechend. Je nach eingeschalteten Pin, 11 oder 10, dreht sich der Motor entweder nach Links oder nach Rechts.

Die Ausgänge des Motortreibers sind am Motor angeschlossen. Eine Anforderung war, dass eine grüne LED leuchtet, wenn der Motor in Bewegung ist und eine Rote wenn er stoppt. Um die Komplexität des Programms zu reduzieren, wurde die Funktion der grünen LED als Schaltung implementiert. Zwei Dioden verbinden jeweils den Plus- und Minuspol des Motors mit der LED. Die Dioden werden benötigt um einen Strom in Richtung Motor zu unterbinden. Die LED leuchtet dann immer, wenn eine Spannung am Motor anliegt. Da der Motor mittels PWM gesteuert wird, leuchtet die LED stärker, je schneller der Motor läuft. Die rote LED wird mittel Pin 8 gesteuert.

Im nachfolgenden Text wird der Programmcode der Aufgabe erläutert. Einzelne Teile des Codes werden ausgewählt und beschrieben. Am Ende dieser Sektion befindet sich der vollständige Programmcode.

```
1  const int SEGMENT_COUNT = 10;
2
3  const int DELAY_TRIGGER = 20;
4  const int DELAY_READ = 550;
5
6  const byte MAX_MOTOR_SPEED = 255;
7  const byte SEGMENT_DISPLAY[] = {
8  B11111100,
9  B01100000,
10 B11011010,
11 B11110010,
12 B01100110,
13 B10110110,
14 B10111110,
15 B11100000,
16 B11111110,
17 B11110110
```

```
};
```

Listing 35: Wichtige Konstanten

In Listing 35 sind wichtige Konstanten enthalten. $SEGMENT\_DISPLAY$ ist ein Array, welches die Bytes enthält, die zum Anzeigen der Zahlen von 0 bis 9 benötigt wird. $SEGMENT\_COUNT$ beinhaltet die Länge des Arrays. Die Werte für $SEGMENT\_DISPLAY$ wurde aus Aufgabe 2 der Sektion 9.2 entnommen. Der Wert $DELAY\_TRIGGER$ wird verwendet, um die Steuerung des Ultraschallsensors zu timen. Um die Geschwindigkeit pro Geschwindigkeitsstufe zu errechnen, wird später im Programm die Konstante MAX_MOTOR_SPEED verwendet.

```
volatile int segment_index_left = 0;
volatile int segment_index_right = 0;

/*
true => motor turns right,
false => motor turns left
*/
volatile bool turn_direction = true;
```

Listing 36: Wichtige Variablen

Die in Listing 36 gezeigten Variablen *segment_index_left* und *segment_index_right* werden verwendet, um die Geschwindigkeit nach Links, bzw. nach Rechts, des Motors zu speichern. *turn_direction* gibt an, ob sich der Motor nach Rechts oder nach Links derehen soll. Ist die Variable auf *true* gesetzt, soll sich der Motor nach Rechts drehen, ansonsten nach Links. Alle drei Variablen besitzen das *volatile* Schlüsselwort, da sie von Interrupts beschrieben und gelesen werden.

```
setup() {
    ...

    attachInterrupt(
    digitalPinToInterrupt(PIN_BUTTON_LEFT),
    on_button_left_rising,
    RISING);

    attachInterrupt(
    digitalPinToInterrupt(PIN_BUTTON_RIGHT),
    on_button_right_rising,
    RISING);

    ...
```

```
}
```

Listing 37: Setup der Taster

In Listing 37 werden die Taster mit Interrupts verknüpft. Anders als in Abschnitt 5, wird nur ein Interrupt beim Drücken des Tasters benötigt, nicht jedoch beim los lassen. Daher wird der $attachInterrupt(\dots)$ Methode lediglich $RISING$ anstatt $CHANGE$ mitgegeben.

```
void on_button_left_rising() {
    segment_index_left++;

    if (segment_index_left >= SEGMENT_COUNT) {
        segment_index_left = 0;
    }
    turn_direction = false;
}

void on_button_right_rising() {
    segment_index_right++;

    if (segment_index_right >= SEGMENT_COUNT) {
        segment_index_right = 0;
    }
    turn_direction = true;
}
```

Listing 38: Interrupts der Taster

Die verknüpften Interrupts sind in Listing 38 zu sehen. Beide sind analog zueinander im Bezug auf die Drehrichtung des Motors. Wird ein Taster betätigt, wird zuerst die $segment\_index\_*$ Variable erhöht. Übersteigt diese die Anzahl an Indices des $SEGMENT\_DISPLAY$ Arrays, wird sie auf 0 zurückgesetzt. Abschließend wird die Variable zur Steuerung der Drehrichtung entsprechend gesetzt.

```
void update_display(byte value)
{
    digitalWrite(PIN_LATCH, LOW);
    shiftOut(PIN_DATA, PIN_CLOCK, LSBFIRST, value);
    digitalWrite(PIN_LATCH, HIGH);
}
```

Listing 39: Aktualisierung der 7-Segment-Anzeige

In Listing 39 wird die 7-Segment-Anzeige auf den übergebenen Wert aktualisiert. Dafür wird zuerst Pin 5, welcher den Schreibzugriff regelt, auf LOW

geschalten. Damit ist das Schieberegister bereit zum beschrieben werden. Die Methode $shiftOut(\dots)$ schreibt dann, unter verwendung des Daten- und Clockpins, den Wert in das Register. Anschließend wird mit dem Setzen auf $HIGH$ des Pin 5 das Schreiben beendet.

```
bool is_distance_safe() {
    // Clear trigger
    digitalWrite(PIN_ULTRASONIC_TRIGGER, LOW);
    delayMicroseconds(DELAY_TRIGGER);

    // Send High Signal
    digitalWrite(PIN_ULTRASONIC_TRIGGER, HIGH);
    delayMicroseconds(DELAY_TRIGGER);
    digitalWrite(PIN_ULTRASONIC_TRIGGER, LOW);

    double duration = pulseIn(PIN_ULTRASONIC_ECHO, HIGH);
    double distance= duration*0.034/2;

    return distance > 30;
}
```

Listing 40: Distanz Messung

Die Messung der Distanz wird in Listing 40 durchgeführt. Der Trigger wird dazu zuerst auf für 20ms auf LOW geschalten um einen konsistenten Zustand zu erreichen. Dannach wird der Messvorgang gestartet, indem der Trigger für 20ms auf $HIGH$ geschalten wird und anschließend wieder auf LOW. Durch $pulseIn(\dots)$ blockiert das Programm, bis ein Signal am Echo-Pin anliegt und gibt dann die vergangene Zeit, in Mikrosekunden, zurück. Die Distanz beträgt dann:

$$d = \frac{t[\mu s] * 343\frac{m}{\mathbf{s}}}{2} \tag{74}$$

$$= \frac{t[\mu s] * 0.000343\frac{m}{\mu \mathbf{s}}}{2} \tag{75}$$

$$= \frac{t[\mu s] * 0.0343\frac{\mathbf{cm}}{\mu s}}{2} \tag{76}$$

Da der Sicherheitsabstand 30cm beträgt, wird wahr, bzw. $true$, zurückgegeben, wenn der Abstand $> 30cm$ ist, ansonsten falsch, bzw. $false$.

```
void loop()
{
```

```
    int segment_index = 0;

    if (turn_direction) {
    segment_index = segment_index_right;
    } else {
    segment_index = segment_index_left;
    }

    update_display(SEGMENT_DISPLAY[segment_index]);

    ...
```

Listing 41: Einstellen der 7-Segment Anzeige

Zuerst wird in Listing 41, je nach Drehrichtung, die enstprechende *segment_index_** Variable in *segment_index* gespeichert. Der zugehörige Wert des *SEGMENT_DISPLAY* Arrays wird dann der *update_display* Methode übergeben. Dadurch wird die aktuelle Geschwindigkeit des Motors angezeigt.

```
    ...

    byte motor_speed = MAX_MOTOR_SPEED
        / SEGMENT_COUNT
        * segment_index;

    if (!is_distance_safe()) {
        motor_speed = 0;
    }

    ...
```

Listing 42: Berechnung des PWM-Signals

Die "Stärke" des PWM-Signals wird nun in Listing 42 berechnet. Eine Division der *MAX_MOTOR_SPEED* Konstanten durch die Anzahl an Geschwindigkeitsstufen wird durchgeführt und dann mit der aktuellen Geschwindigkeit multipliziert. Dadurch ergibt sich der Wert, auf den das PWM-Signal eingestellt werden muss. Sollte es sich herausstellen, dass der Sicherheitsabstand unterschritten wird, wird der Wert auf 0 gesetzt.

```
    ...

    if (turn_direction) {
        analogWrite(PIN_MOTOR_RIGHT, motor_speed);
        analogWrite(PIN_MOTOR_LEFT, 0);
    } else {
        analogWrite(PIN_MOTOR_LEFT, motor_speed);
        analogWrite(PIN_MOTOR_RIGHT, 0);
```

```
    }

    if (motor_speed > 0) {
        digitalWrite(PIN_STOP_LED, LOW);
    } else {
        digitalWrite(PIN_STOP_LED, HIGH);
    }

    ...
```

Listing 43: Ansteuerung des Motors

Unter Verwendung der *analogWrite*(...) Methode wird nun das PWM-Signal ausgegeben. Dieser Vorgang ist in Listing 43 abgebildet. Je nach Drehrichtung wird das Signal entweder an Pin 12 oder 10 angelegt. Der jeweils andere Pin wird ausgeschaltet. Abschließend wird die rote LED ausgeschaltet, wenn die Geschwindigkeit > 0 beträgt, oder eingeschalten, wenn dies nicht der Fall ist.

```
/*
Adafruit Arduino - Lesson 4. 8 LEDs and a Shift Register
*/

const int PIN_LATCH = 5;
const int PIN_CLOCK = 6;
const int PIN_DATA = 4;
const int PIN_MOTOR_LEFT = 11;
const int PIN_MOTOR_RIGHT = 10;
const int PIN_STOP_LED = 8;

const int PIN_BUTTON_LEFT = 3;
const int PIN_BUTTON_RIGHT = 2;
const int PIN_ULTRASONIC_TRIGGER = 9;
const int PIN_ULTRASONIC_ECHO = 12;

const int SEGMENT_COUNT = 10;

const int DELAY_TRIGGER = 20;
const int DELAY_READ = 550;

const byte MAX_MOTOR_SPEED = 255;
const byte SEGMENT_DISPLAY[] = {
  B11111100,
  B01100000,
  B11011010,
  B11110010,
  B01100110,
  B10110110,
```

```
  B10111110,
  B11100000,
  B11111110,
  B11110110
};


volatile int segment_index_left = 0;
volatile int segment_index_right = 0;

/*
true => motor turns right,
false => motor turns left
*/
volatile bool turn_direction = true;

unsigned long previous_time = 0;

void setup()
{
  Serial.begin(115000);

  pinMode(PIN_LATCH, OUTPUT);
  pinMode(PIN_DATA, OUTPUT);
  pinMode(PIN_CLOCK, OUTPUT);
  pinMode(PIN_MOTOR_LEFT, OUTPUT);
  pinMode(PIN_MOTOR_RIGHT, OUTPUT);
  pinMode(PIN_STOP_LED, OUTPUT);

  pinMode(PIN_ULTRASONIC_TRIGGER, OUTPUT);
  pinMode(PIN_ULTRASONIC_ECHO, INPUT);

  pinMode(PIN_BUTTON_LEFT, INPUT_PULLUP);
  pinMode(PIN_BUTTON_RIGHT, INPUT_PULLUP);

  attachInterrupt(
    digitalPinToInterrupt(PIN_BUTTON_LEFT),
    on_button_left_rising,
    RISING);

  attachInterrupt(
    digitalPinToInterrupt(PIN_BUTTON_RIGHT),
    on_button_right_rising,
    RISING);

  digitalWrite(PIN_ULTRASONIC_TRIGGER, HIGH);
}

void loop()
```

```
{
  int segment_index = 0;

  if (turn_direction) {
    segment_index = segment_index_right;
  } else {
    segment_index = segment_index_left;
  }

  update_display(SEGMENT_DISPLAY[segment_index]);

  byte motor_speed = MAX_MOTOR_SPEED
    / SEGMENT_COUNT
    * segment_index;

  if (!is_distance_safe()) {
    motor_speed = 0;
  }

  if (turn_direction) {
    analogWrite(PIN_MOTOR_RIGHT, motor_speed);
    analogWrite(PIN_MOTOR_LEFT, 0);
  } else {
    analogWrite(PIN_MOTOR_LEFT, motor_speed);
    analogWrite(PIN_MOTOR_RIGHT, 0);
  }

  if (motor_speed > 0) {
    digitalWrite(PIN_STOP_LED, LOW);
  } else {
    digitalWrite(PIN_STOP_LED, HIGH);
  }
}

bool is_distance_safe() {
  // Clear trigger
  digitalWrite(PIN_ULTRASONIC_TRIGGER, LOW);
  delayMicroseconds(DELAY_TRIGGER);

  // Send High Signal
  digitalWrite(PIN_ULTRASONIC_TRIGGER, HIGH);
  delayMicroseconds(DELAY_TRIGGER);
  digitalWrite(PIN_ULTRASONIC_TRIGGER, LOW);

  double duration = pulseIn(PIN_ULTRASONIC_ECHO, HIGH);
  double distance= duration*0.034/2;

  return distance > 30;
}
```

```
128
129 void update_display(byte value)
130 {
131    digitalWrite(PIN_LATCH, LOW);
132    shiftOut(PIN_DATA, PIN_CLOCK, LSBFIRST, value);
133    digitalWrite(PIN_LATCH, HIGH);
134 }
135
136 void on_button_left_rising() {
137   segment_index_left++;
138
139   if (segment_index_left >= SEGMENT_COUNT) {
140     segment_index_left = 0;
141   }
142   turn_direction = false;
143 }
144
145 void on_button_right_rising() {
146   segment_index_right++;
147
148   if (segment_index_right >= SEGMENT_COUNT) {
149     segment_index_right = 0;
150   }
151   turn_direction = true;
152 }
```

Listing 44: Vollständiger Programmcode der Aufgabe 6

9.4 Fehlerdiskussion

Die Konstante $DELAY\_READ$ wird nicht verwendet und ist ein Artefakt aus der Entwicklung des Programms. Sie kann ohne Konsequenzen gelöscht werden.

Der Strom durch das Schieberegister bei eingeschalteter Anzeige ist zu hoch. Eine andere Wahl der Widerstände würde dies verhinder. Nach Experimentieren an der Schaltung stellt sich heraus, dass ein Widerstand von 500Ω zu akzeptablen Resultaten führt.

9.5 Zusammenfassung

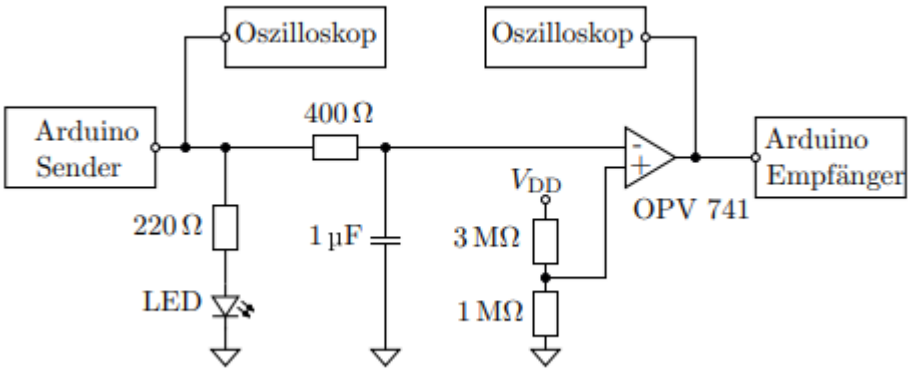

Abbildung 16: Simulation eines Infrarotsignals

10 Aufgabe 7 - Datenübertragung mittels Infra-Rot (IR)

In dieser Aufgabe soll eine Datenübertragung mittels Infrarotsignal implementiert werden. Um ein solches Signal in der verwendeten Simulationssoftware zu simulieren, wurde die Schaltung aus Abbildung 16 verwendet.

Für diese Aufgabe, soll ein 38kHz Signal von einem Sender-Mikrocontroller and einen Empfänger-Mikrocontroller gesendet werden. Eine Übertragung von Text soll mithilfe des Sendens von Morsezeichen implementiert werden. Es soll dem Benutzer dann ermöglicht werden, über die serielle Schnittstelle Zeichen an den Sender zu übergeben. Dieser sendet die Zeichen dann an den Empfänger, welche sie wieder ausgibt.

10.1 Materialien

10.2 Vorbereitung

10.2.1 Aufgabe 1

Zur Codierung von Buchstaben in Morsecode soll recherchiert werden, wie diese Codierung erfolgt. Es werden sogenannte *dit*s und *dah*s verwendet [6]. Ein *dit* wird als Punkt und ein *dah* als Strich symbolisiert. Des weiteren

Tabelle 15: Aufgabe 7 - Verwendete Materialien

Bezeichnung	Eigenschaften	Menge
LED	Rot	1
Widerstand	220Ω Rot - Rot - Braun - Gold	1
Widerstand	400Ω Gelb - Schwarz - Braun - Gold	1
Widerstand	$1M\Omega$ Braun - Schwarz - Grün - Gold	1
Widerstand	$3M\Omega$ Orange - Schwarz - Grün - Gold	1
Kondensator	$10\mu F$ - 1	
Operationsverstärker		1
(optional) Oszilloskop		2
Mikrocontroller	Arduino Uno R3	2

wird nicht zwischen Groß- und Kleinbuchstaben unterschieden. In Tabelle 16 werden die Zuordnung zwischen Buchstaben und Morsecode gezeigt.

Tabelle 16: Morsecode von Buchstaben

Buchstabe	Code	Buchstabe	Code	Buchstabe	Code
A	* -	J	* - - -	S	* * *
B	- * * *	K	- * -	T	-
C	- * - *	L	* - * *	U	* * -
D	- * *	M	- -	V	* * * -
E	*	N	- *	W	* - -
F	* * - *	O	- - -	X	- * * -
G	- - *	P	* - - *	Y	- * - -
H	* * * *	Q	- - * -	Z	- - * *
I	* *	R	* - *		

10.2.2 Aufgabe 2

Es soll überlegt werden, wie das Einlesen von Zeichen und die Codierung dieser mittels eines Mikrocontroller efolgen könnte.

Zuerst kann ein Buchstabe mittels *Serial.read*() eingelesen werden. Durch die Verwendung einer Matrix, welche die Kodierung abbildet, kann ein Buch-

stabe kodiert werden. Man kann dann ein Signal für die Dauer t an einen Ausgangspin anlegen um ein dit zu senden. Um ein dah zu senden, kann das Signal für die Dauer $3t$ anliegen.

10.2.3 Aufgabe 3

Es soll der benötigte Vorwiderstand der Infrarot-LED berechnet werden. Dazu sind folgende Werte gegeben:

$$U = 5V \tag{77}$$
$$I_D = 130mA \tag{78}$$
$$U_D = 1,6V \tag{79}$$
$$R_{on} = 14\Omega \tag{80}$$

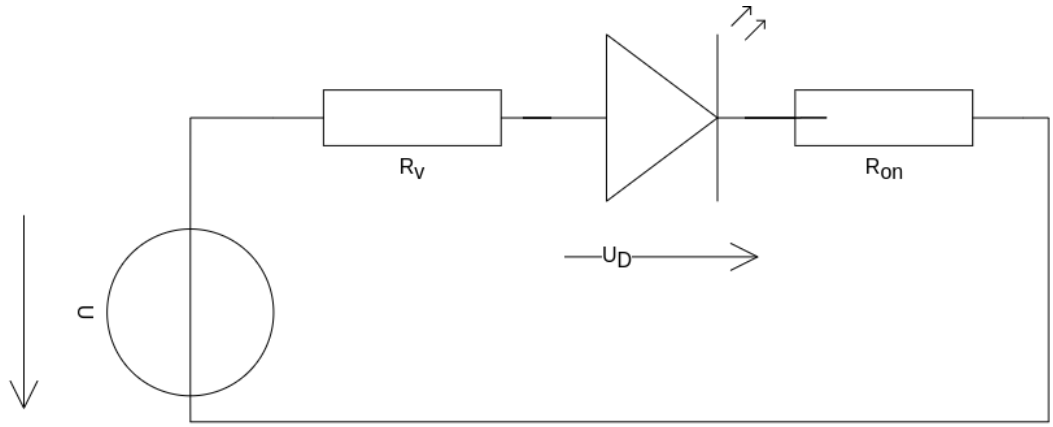

Abbildung 17: Angenommene Schaltung

Gesucht ist der Widerstand R_v. Zur Berechnung wurde die Schaltung aus Abbildung 17 angenommen.

$$U_T = R_{on} * I_D \tag{81}$$
$$= 14\Omega * 0,13A \tag{82}$$
$$= 1,82V \tag{83}$$

$$U_{RV} = U - U_T - U_D \tag{84}$$
$$= 5V - 1,82V - 1,6V \tag{85}$$
$$= 1,58V \tag{86}$$

$$R_v = \frac{U_{RV}}{I_D} \tag{87}$$
$$= \frac{1,58V}{0,13A} \tag{88}$$
$$\underline{\underline{= 12,154\Omega}} \tag{89}$$

Der errechnete Vorwiderstand beträgt $12,154\Omega$, daher wird der 10Ω Widerstand aus der E6-Reihe verwendet.

10.3 Praktikumsaufgabe

In Abbildung 15 ist die implementierte Schaltung abgebildet. Links ist der sendende Mikrocontroller, rechts der Empfangende platziert. Der Pin 11 wurde bei beidem als Aus- bzw. Eingang verwendet, da dieser PWM-Modulation unterstützt. Die PWM-Unterstützung ist hier nicht notwendig, mehr dazu in der Fehlerdiskussion.

Im nachfolgenden Text wird der Programmcode der Aufgabe erläutert. Einzelne Teile des Codes werden ausgewählt und beschrieben. Am Ende der jeweiligen Sektionen befinden sich die vollständigen Programmcodes des Senders und des Empfängers.

10.3.1 Sender

```
1 const unsigned long DURATION = 25;
2
3 const int PIN_SENDER = 11;
4
5 /*
6 1 - short
7 3 - long
```

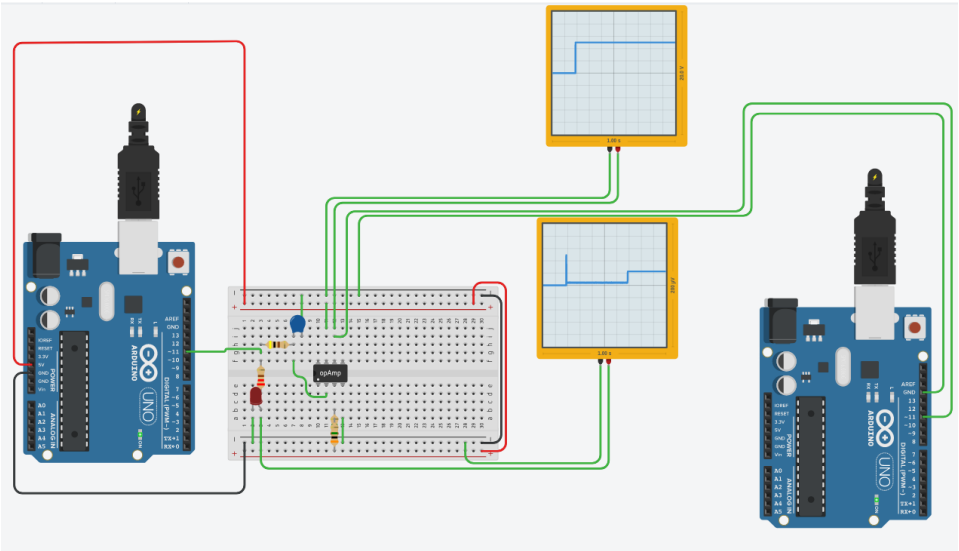

Abbildung 18: Implementierte Schaltung für Aufgabe 7

```
-1 - NOP

NOP is needed since it's complex to work with jagged arrays in
    C
*/
const int CODE[26][4] = {
  {1, 3, -1, -1},    // A
  {3, 1, 1, 1},    // B
  {3, 1, 3, 1},    // C
  {3, 1, 1, -1},   // D
  {1, -1, -1, -1},   // E
  {1, 1, 3, 1},    // F
  {3, 3, 1, -1},   // G
  {1, 1, 1, 1},    // H
  {1, 1, -1, -1},    // I
  {1, 3, 3, 3},    // J
  {3, 1, 3, -1},   // K
  {1, 3, 1, 1},    // L
  {3, 3, -1, -1}, // M
  {3, 1, -1, -1},    // N
  {3, 3, 3, -1},   // O
  {1, 3, 3, 1},    // P
  {3, 3, 1, 3},    // Q
  {1, 3, 1, -1},   // R
  {1, 1, 1, -1},   // S
  {3, -1, -1, -1},   // T
```

```
  {1, 1, 3, -1},  // U
  {1, 1, 1, 3},   // V
  {1, 3, 3, -1},  // W
  {3, 1, 1, 3},   // X
  {3, 1, 3, 3},   // Y
  {3, 3, 1, 1},   // Z
};

const int OFFSET = 'A';
```

Listing 45: Wichtige Konstanten des Senders

In Listing 45 werden die verwendeten Konstanten gezeigt. Die Konstante *DURATION* wird verwendet, um die Zeit für dits und dahs zu berechnen. In der *CODE* Matrix ist die Kodierung von Buchstaben zu Morsecode abgebildet. Dabei werden dits mit 1 und dahs mit 3 beschrieben. In C ist ein jagged Array, d.h. ein Array, bei dem die Zeilen unterschiedliche Längen haben, nicht trivial zu erstellen. Daher werden nicht verwendete Positionen mit -1 angegeben. Die Konstante *OFFSET* speichert den ASCII-Wert des Buchstaben 'A'.

```

void loop()
{
  if(Serial.available()) {
    send_letter(Serial.read());
  }
}

bool send_letter(char letter) {
  if (letter >= 'a' && letter <= 'z') {
    // If letter is lower case, set to uppercase
    letter -= 32;
  } else if (letter < 'A' || letter > 'Z') {
    return false;
  }

  Serial.println(letter);
  // Normalize to 0 - 25;
  int index = letter - 'A';
  for (int i = 0; i < 4; i++) {
    int multiplier = CODE[index][i];

    if (CODE[index][i] > -1) {
      digitalWrite(PIN_SENDER, HIGH);
      delay(DURATION * multiplier);
      digitalWrite(PIN_SENDER, LOW);
      delay(DURATION);
```

```
    } else {
      digitalWrite(PIN_SENDER, HIGH);
      delay(DURATION * 4);
      digitalWrite(PIN_SENDER, LOW);
      delay(DURATION);
      // Exit loop after end of code
      break;
    }
  }

  digitalWrite(PIN_SENDER, LOW);
  return true;
}
```

Listing 46: Programmschleife des Senders

Die Programmschleife samt Hilfsmethode ist in Listing 46 dargestellt. Die Schleife selbst liest nur einen Buchstaben vom seriellen Interface und übergibt diesen an die Funktion *send_letter*.

Im ersten Teil der Hilfsmethode wird zuerst überprüft, ob der Buchstabe ein Kleinbuchstabe ist. Ist dies der Fall, wird er zu einem Großbuchstaben umgewandelt. Die Umwandlung wird durch Subtraktion des Unterschieds zwischen dem ASCII-Wert von 'a' und 'A' durchgeführt. Sollte der Buchstabe weder ein Klein- noch ein Großbuchstabe sein, wird die Routine beendet.

Danach wird der benötigte Index der Kodierungsmatrix berechnet, indem vom ASCII-Wert des übergeben Buchstaben der Wert von 'A' abgezogen wird. Dies normalisiert den Wert auf den Bereich $0 \leq index < 26$. Die Zeile am entsprechenden Index wird sodann durchiteriert. Ist der gespeicherte Wert an der Stelle $[index][i]$ größer als der definierte Platzhalter von -1, wird ein Morsecode gesendet. Der Wert der Matrix dient dazu als Multiplikator der Zeitspanne, für die ein $HIGH$-Signal an Pin 11 angelegt wird. Wird das Ende der Zeile, d.h. -1, erreicht, wird ein $HIGH$-Signal für die Zeit von $4 * DURATIOn$ angegelegt, um das Ende des Signals zu markieren. Abschließend wird die Schleife beendet und Pin 11 auf LOW geschaltet.

```
const unsigned long DURATION = 25;

const int PIN_SENDER = 11;

/*
1 - short
3 - long
-1 - NOP

```

```
NOP is needed since it's complex to work with jagged arrays in
    C
*/
const int CODE[26][4] = {
  {1, 3, -1, -1},   // A
  {3, 1, 1, 1},   // B
  {3, 1, 3, 1},   // C
  {3, 1, 1, -1},  // D
  {1, -1, -1, -1},  // E
  {1, 1, 3, 1},   // F
  {3, 3, 1, -1},  // G
  {1, 1, 1, 1},   // H
  {1, 1, -1, -1},   // I
  {1, 3, 3, 3},   // J
  {3, 1, 3, -1},  // K
  {1, 3, 1, 1},   // L
  {3, 3, -1, -1}, // M
  {3, 1, -1, -1},   // N
  {3, 3, 3, -1},  // O
  {1, 3, 3, 1},   // P
  {3, 3, 1, 3},   // Q
  {1, 3, 1, -1},  // R
  {1, 1, 1, -1},  // S
  {3, -1, -1, -1},  // T
  {1, 1, 3, -1},  // U
  {1, 1, 1, 3},   // V
  {1, 3, 3, -1},  // W
  {3, 1, 1, 3},   // X
  {3, 1, 3, 3},   // Y
  {3, 3, 1, 1},   // Z
};

const int OFFSET = 'A';

void setup()
{
  Serial.begin(115000);
  pinMode(PIN_SENDER, OUTPUT);
}

void loop()
{
  if(Serial.available()) {
    send_letter(Serial.read());
  }
}

bool send_letter(char letter) {
  if (letter >= 'a' && letter <= 'z') {
```

```
    // If letter is lower case, set to uppercase
    letter -= 32;
  } else if (letter < 'A' || letter > 'Z') {
    return false;
  }

  Serial.println(letter);
  // Normalize to 0 - 25;
  int index = letter - 'A';
  for (int i = 0; i < 4; i++) {
    int multiplier = CODE[index][i];

    if (CODE[index][i] > -1) {
      digitalWrite(PIN_SENDER, HIGH);
      delay(DURATION * multiplier);
      digitalWrite(PIN_SENDER, LOW);
      delay(DURATION);
    } else {
      digitalWrite(PIN_SENDER, HIGH);
      delay(DURATION * 4);
      digitalWrite(PIN_SENDER, LOW);
      delay(DURATION);
      // Exit loop after end of code
      break;
    }
  }

  digitalWrite(PIN_SENDER, LOW);
  return true;
}
```

Listing 47: Vollständiger Programmcode des Senders der Aufgabe 7

10.3.2 Empfänger

```
void loop()
{
    int value = digitalRead(PIN_RECEIVER);
    if(value == LOW && start_time <= 0) {
        start_time = millis();
    } else if (value == HIGH && start_time > 0) {
        unsigned long duration = millis() - start_time;
        int decoded = decode(millis() - start_time);
        start_time = 0;
        if (decoded > -1) {
            Serial.println((char)(decoded + OFFSET));
        }
```

```
    }
}
```

Listing 48: Programmschleife des Empfängers

In Listing 48 ist die Programmschleife des Empfängers abgebildet. Zuerst wird der Zustand des Pin 11 gelesen. Ist der Zustand *LOW*, wird ein Timer gestartet. Ändert sich der Zustand, wird die vergangene Zeit gemessen und dekodiert. Der Timer wird zurückgesetzt und, sollte das Dekodieren erfolgreich gewesen sein, wird der empfangenen Buchstabe auf der seriellen Schnittstelle ausgegeben.

```
int decode(unsigned long duration) {
  int result = -1;
  if (duration > DURATION*4 - DURATION_THRESHOLD &&
      duration < DURATION*4 + DURATION_THRESHOLD) {
    current_code[current_code_index] = -1;
    for(int i = 0; i < 26; i++) {
      for(int j = 0; j < 4; j++) {
        if(CODE[i][j] == current_code[j]) {
          if(CODE[i][j] == -1) {
            // Set result if codes match until end
            result = i;
          }
        } else {
          break;
        }
      }
    }

    // Reset code
    for(int i = 0; i < 4; i++) {
      current_code[i] = -1;
    }
    current_code_index = -1;
  } else if (duration > DURATION*3 - DURATION_THRESHOLD &&
      duration < DURATION*3 + DURATION_THRESHOLD) {
    current_code[current_code_index] = 3;
  } else if (duration > DURATION - DURATION_THRESHOLD &&
      duration < DURATION + DURATION_THRESHOLD) {
    current_code[current_code_index] = 1;
  }
  current_code_index++;

  // Return -1 if encoding not possible
  return result;
```

```
}
```

Listing 49: Dekodieren empfangenen Daten

Das eigentliche Dekdoieren wird in Listing 49 vorgenommen. Ist die Zeit zwischen einen Zustandswechsel vier mal $DURATION$, d.h. 100ms, wird der Buchstabe als Empfangen angenommen. Durch vergleichen der Empfangenen dits und dahs wird der Buchstabe aus der Kodierungsmatrix gesucht. Sollte das Ergebnis dekodiert werden können, so wird dieser in der Variable *result* gespeichert. Danach wird die Variable, die die empfangene Dits und Dahs zwischenspeichert, zurückgesetzt. Ist die Zeit zwischen den Zustandswechseln $3 * DURATION$ oder $DURATION$, wird jeweils ein Dah oder Dit in die Variable $current_code$ zwischengespeichert. Schlussendlich wird das Ergebnis zurückgegeben, oder -1 wenn das Dekodieren nocht nicht oder nicht möglich ist.

```
const unsigned long DURATION = 25;
const unsigned long DURATION_THRESHOLD = 5;

const int PIN_RECEIVER = 11;

/*
1 - short
2 - long
-1 - NOP

NOP is needed since it's complex to work with jagged arrays in
    C
*/
const int CODE[26][4] = {
  {1, 3, -1, -1},   // A
  {3, 1, 1, 1},   // B
  {3, 1, 3, 1},   // C
  {3, 1, 1, -1},  // D
  {1, -1, -1, -1},  // E
  {1, 1, 3, 1},   // F
  {3, 3, 1, -1},  // G
  {1, 1, 1, 1},   // H
  {1, 1, -1, -1},   // I
  {1, 3, 3, 3},   // J
  {3, 1, 3, -1},  // K
  {1, 3, 1, 1},   // L
  {3, 3, -1, -1}, // M
  {3, 1, -1, -1},   // N
  {3, 3, 3, -1},  // O
  {1, 3, 3, 1},   // P
  {3, 3, 1, 3},   // Q
```

```
  {1, 3, 1, -1},  // R
  {1, 1, 1, -1},  // S
  {3, -1, -1, -1},  // T
  {1, 1, 3, -1},  // U
  {1, 1, 1, 3},   // V
  {1, 3, 3, -1},  // W
  {3, 1, 1, 3},   // X
  {3, 1, 3, 3},   // Y
  {3, 3, 1, 1},   // Z
};
int current_code[4] = {};
int current_code_index = 0;

const int OFFSET = 65;

int start_time = 0;

void setup()
{
  Serial.begin(115000);
  pinMode(PIN_RECEIVER, INPUT);
}

void loop()
{
  int value = digitalRead(PIN_RECEIVER);
  if(value == LOW && start_time <= 0) {
    start_time = millis();
  } else if (value == HIGH && start_time > 0) {
    unsigned long duration = millis() - start_time;
    int decoded = decode(millis() - start_time);
    start_time = 0;
    if (decoded > -1) {
      Serial.println((char)(decoded + OFFSET));
    }
  }
}

int decode(unsigned long duration) {
  int result = -1;
  if (duration > DURATION*4 - DURATION_THRESHOLD &&
      duration < DURATION*4 + DURATION_THRESHOLD) {
    current_code[current_code_index] = -1;
    for(int i = 0; i < 26; i++) {
      for(int j = 0; j < 4; j++) {
        if(CODE[i][j] == current_code[j]) {
          if(CODE[i][j] == -1) {
            // Set result if codes match until end
            result = i;
```

```
80            }
81          } else {
82            break;
83          }
84        }
85      }
86
87      // Reset code
88      for(int i = 0; i < 4; i++) {
89        current_code[i] = -1;
90      }
91      current_code_index = -1;
92    } else if (duration > DURATION*3 - DURATION_THRESHOLD &&
93        duration < DURATION*3 + DURATION_THRESHOLD) {
94      current_code[current_code_index] = 3;
95    } else if (duration > DURATION - DURATION_THRESHOLD &&
96        duration < DURATION + DURATION_THRESHOLD) {
97      current_code[current_code_index] = 1;
98    }
99    current_code_index++;
100
101   // Return -1 if encoding not possible
102   return result;
103 }
```

Listing 50: Vollständiger Programmcode des Empfängers der Aufgabe 7

10.4 Fehlerdiskussion

Es hätte zur Übertragung von Daten ein analoger Pin verwendet werden sollen. Die Verwendung eines PWM-Pins führt zu fehlerhaften Übertragungen und Implementationsfehlern.

Des weiteren konnte ein sinnvolles Debugging des Codes nicht durchgeführt werden. Beim Simulieren von zwei Mikrocontrollern dauert das Simulieren von 100ms, in der verwendeten Software, mehrere Sekunden. Eine manuelle bzw. visuelle Überprüfung der Funktionalität ist dabei praktisch nicht mehr durchführbar.

Schlussendlich wäre eine interessante Erweiterung oder Alternative dieser Aufgabe, die Verwendung einer Huffman Kodierung anstelle des Morsecodes. Beim Morsecode wird ein Endzeichen benötigt, um Zeichen zu Unterscheiden. Bei der Huffman-Kodierung ist kein Codewort der Prefix eines anderen. Damit kann eine Dekodierung ohne Endwort durchgeführt werden,

da es Eindeutig ist, wann ein Wort endet [4].

10.5 Zusammenfassung

Literatur

[1] Arduino. *Arduino Uno Rev 3*. https://store.arduino.cc/arduino-uno-rev3.

[2] KT-Elektronic. *Ultraschall Messmodul HC-SR04*. https://www.mikrocontroller.net/attachment/218122/HC-SR04_ultraschallmodul_beschreibung_3.pdf.

[3] Elektronik Kompenidum. *Kondensator im Gleichstromkreis*. https://www.elektronik-kompendium.de/sites/grd/0205301.htm.

[4] Wikipedia. *Huffman*. https://en.wikipedia.org/wiki/Huffman_coding.

[5] Wikipedia. *Leuchtdiode - Elektrische Eigenschaften*. https://de.wikipedia.org/wiki/Leuchtdiode#Elektrische_Eigenschaften.

[6] Wikipedia. *Morse code*. https://en.wikipedia.org/wiki/Morse_code.