Einführung in die Mikrocontroller Programmierung

Benjamin Reh & Sascha Tebben

Version 1.1

Inhaltsverzeichnis

1. Einleitung **5**

2. Grundlagen: Elektrizitäts-Lehre **7**
2.1. Was ist elektrischer Strom? . . . 7
2.2. Leitung in Metallen . . . 7
2.3. Physikalische Größen . . . 7
2.4. Wasser Analogon . . . 9
2.5. Schaltungen . . . 10
2.5.1. Vorwiderstände . . . 12

3. A Mikrocontroller **13**
3.1. Der Atmega168 . . . 13
3.2. Basis-Platine . . . 14
3.3. Aufgabe A.1: Zusammenbau . . . 15
3.4. Aufgabe A.2: Inbetriebnahme . . . 15

4. B Ein- und Ausgabe **17**
4.1. Pins & Ports . . . 17
4.2. Bit-Manipulation . . . 17
4.2.1. Bitshifting . . . 17
4.2.2. Bitweise Logik . . . 18
4.2.3. Ein- und Ausgabe . . . 19
4.3. LED anschließen . . . 20
4.3.1. Pins auf der Basis-Platine . . . 20
4.3.2. LEDs . . . 21
4.4. Aufgabe B.1: LED ansteuern . . . 22
4.5. Aufgabe B.2: Systemzeit . . . 22
4.6. Taster anschließen . . . 23
4.7. Aufgabe B.3: Taster einlesen . . . 24
4.8. Flankenerkennung . . . 24
4.8.1. Flanken . . . 24
4.8.2. Flankenerkennung . . . 24
4.9. Aufgabe B.4: Flankenerkennung . . . 25

5. C Zustandsautomat **26**
5.1. Automaten . . . 26
5.2. `enum` . . . 26
5.3. `switch...case` . . . 26
5.4. Aufgabe C.1: Implementierung einer Fußgängerampel . . . 27

6. D Serielle Schnittstelle **29**
6.1. Funktionsweise . . . 29
6.2. Verbindung zwischen PC und Mikrocontroller . . . 30
6.2.1. Terminalprogramm . . . 31
6.3. Aufgabe D.1: Ein- und Ausgabe . . . 31

7. E Analog-Digital-Wandler **33**
7.1. Analoge Signale . . . 33
7.1.1. Implementierung . . . 33
7.1.2. Drehpotentiometer . . . 33
7.2. Aufgabe E.1: Analoge Werte einlesen . . . 34
7.3. Schwellwertschalter (Trigger) . . . 34
7.3.1. Zweipunktregelung . . . 35
7.3.2. Photo-Widerstand . . . 36
7.4. Aufgabe E.2: Trigger . . . 36

8. F Pulsweiten-Modulation **37**
8.1. Pulsweitenmodulation (PWM) . . . 37
8.2. Timer und Counter . . . 37
8.3. Aufgabe F.1: Software PWM . . . 38
8.4. PWM-Counter im Mikrocontroller . . . 39
8.5. Aufgabe F.2: Hardware PWM der Basis . . . 40
8.6. Aufgabe F.3: 10-bit Hardware PWM . . . 40

9. G Datentypen **41**
9.1. Datentypen und nicht-assoziativität . . . 41
9.1.1. Datentypen . . . 41
9.1.2. Assoziativität . . . 42
9.2. Aufgabe G.1: Berechnungsdauer bestimmen . . . 43

10. H Interrupt-Programmierung **45**
10.1. Interrupts . . . 45
10.2. Aufgabe H.1: UART-Interrupt . . . 46

11. I Servoansteuerung **48**
11.1. Servos . . . 48
11.1.1. Anschluß an die Basis-Platine . . . 49
11.2. Aufgabe I.1: Servo ansteuern . . . 49

12. J (Projekt) RGB-LED und -Sensor **51**
12.1. Farbräume . . . 51
12.2. Aufgabe J.1: HSV-Spezialisierung . . . 53
12.3. RGB-LED . . . 53
12.4. Aufgabe J.2: Ansteuerung . . . 53
12.5. Farbsensor . . . 54
12.6. Aufgabe J.3: Farbsensor . . . 55

13. K (Projekt) Thermometer **56**
13.1. Temperatursensor . . . 56
13.2. Aufgabe K.1: Temperatur auslesen . . . 56
13.3. Gleitender Mittelwert . . . 57
13.4. Aufgabe K.2: Mittellung . . . 57
13.5. Siebensegmentanzeige . . . 57
13.5.1. Verschaltung . . . 58
13.5.2. Transistoren . . . 58
13.6. Aufgabe K.3: Siebensegmentanzeige . . . 59

14. L (Projekt) Scara-Roboter **61**
14.1. Kinematik . . . 61
14.1.1. Vorwärtskinematik . . . 61
14.1.2. Inverse Kinematik . . . 62
14.2. Scara-Roboter . . . 63
14.2.1. Kinematik eines 2D-Scara-Roboters . . . 63
14.2.2. Vorwärtskinematik . . . 63
14.3. Aufgabe L.1: Inverse Kinematik . . . 64
14.4. Aufgabe L.2: Ansteuerung . . . 65

15. M (Projekt) Synthesizer **67**
15.1. Aufbau Tonleiter . . . 67
15.1.1. Tonerzeugung mit einer Saite . . . 67
15.1.2. Tonleiter . . . 67
15.2. Tonerzeugung auf dem Mikrocontroller . . . 69
15.2.1. Analoge Ausgabe . . . 69
15.2.2. Sampling . . . 71
15.3. Aufgabe M.1: Tonausgabe . . . 72
15.4. Aufgabe M.2: Synthesizer . . . 75

A. Anhang **77**
A.1. Farbtabelle . . . 78

1. Einleitung

Dieses Praktikum hat das Ziel, einen Überblick über die Möglichkeiten und Grenzen der Programmierung von Mikrocontrollern zu vermitteln. Konkret wird dabei Software für den Atmega168 Mikrocontroller der Firma AVR programmiert. Das Absolvieren dieses Grundlagenpraktikums ermöglicht die Bearbeitung weiterer Projekte, bei denen auf das hier vermittelte Wissen zurückgegriffen werden kann.

Aufbau des Praktikums

Das Praktikum gliedert sich in thematische aufbauende Einheiten, die mit großen Buchstaben bezeichnet sind. Die Längen der Einheiten sind danach ausgerichtet, daß im Schnitt eine Einheit pro Woche mit ein bis zwei Nachmittagen abgeschlossen werden kann.

Die letzten Einheiten des Praktikums sind als Projekt-Einheiten gekennzeichnet. Diese sind vom Zeitaufwand und Anspruch etwas umfangreicher und bauen nicht mehr aufeinander auf. Die Bearbeitungsreihenfolge ist bei Projekt-Einheiten beliebig.

Die letzte Projekteinheit, der Synthesizer in Kapitel 15, ist etwas anspruchsvoller als die anderen Projekte. Die Bearbeitung ist freiwillig. Es kann dafür eine andere Projekteinheit ausgelassen werden.

Abschließen einer Einheit

Das erfolgreiche Abschließen einer Einheit erfolgt durch Vorführung des Ergebnisses und Beantwortung eventuell in der Einheit vorkommender Fragen. Dafür stehen die Betreuer an einem festen Termin einmal pro Woche zur Verfügung. Jedes Programmierergebnis, das vorgeführt werden soll, ist mit dem Vermerk

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

gekennzeichnet. Es ist also nicht erforderlich, Zwischenschritte oder Unteraufgaben zu sichern.

Dokumentation und Präsentation

Wie bei anderen Robotikpraktika auch, ist eine Dokumentation und eine Präsentation durchzuführen. Das Thema ist eines der als Projekt gekennzeichneten Einheiten. Es wird empfohlen, sich vor Beginn der ersten Projekteinheiten eines für die Dokumentation zu wählen, damit die Bearbeitung selbst gut dokumentiert werden kann.

Die Dokumentation ist in Form eines PDFs abzugeben. Andere Formate (`.doc(x)`, `.rtf` oder HTML) werden nicht akzeptiert.

Materialien

Zusätzliche Materialien wie Datenblätter, Codevorlagen oder auch dieses Dokument können unter http://joanna.iwr.uni-heidelberg.de/resources/ aus dem Uni Netz abgerufen werden.

2. Grundlagen: Elektrizitäts-Lehre

Dieses Kapitel dient als Grundlagenwissen und zum Nachschlagen während des Praktikums.

Das Arbeiten mit Mikrocontrollern findet im spannenden Grenzgebiet zwischen Hard- und Software statt. Zum Verständnis der Hardware-Seite ist es nötig, sich die Grundlagen der Elektrizitäts-Lehre wieder ins Gedächtnis zu rufen.
In diesem Kapitel wird weitgehend auf längere Herleitungen verzichtet und stattdessen auf das intuitive Verständnis der grundlegenden Zusammenhänge gesetzt.

2.1. Was ist elektrischer Strom?

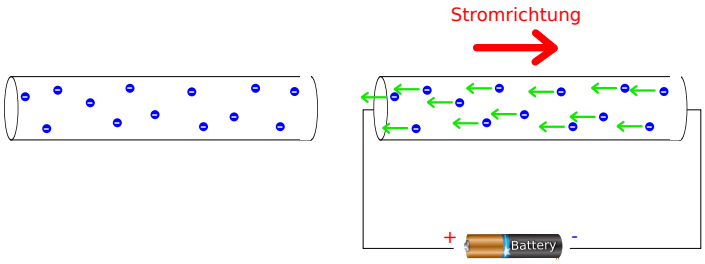
Elektrischer Strom ist allgemein die Bewegung elektrischer Ladungsträger. Damit sich Ladungsträger bewegen, bedarf es zweier Voraussetzungen:
Zum einen müssen die Ladungsträger beweglich, d.h. nicht ortsfest sein. In Metallen beispielsweise sind die Elektronen im Leitungsband frei beweglich. Deshalb sind Metalle Leiter; im Gegensatz zu den meisten Kunststoffen oder Glas, die wegen fehlender freier Ladungsträger Isolatoren sind.
Zum anderen muß auf die Ladungsträger eine Kraft wirken, d.h. sie müssen eine Potentialdifferenz erfahren. Dies kann eine unterschiedliche Konzentration von Ladungsträgern sein, die durch einen Stromfluß ausgeglichen wird.

2.2. Leitung in Metallen

Wie bereits erwähnt sind in Metallen einige der Elektronen frei beweglich. Im feldfreien Zustand sind diese gleichmäßig verteilt (Abb. 2.1(a)).
Legt man nun eine elektrische Spannung an, so werden die negativ geladenen Elektronen von dem positiven Potential angezogen (bzw. vom negativen abgestoßen). Aufgrund der daraus resultieren Kraft wandern die Elektronen nun von $-$ nach $+$.
Etwas verwirrend ist die Tatsache, daß der elektrischer Strom hingegen von $+$ nach $-$, also andersherum als die Elektronen fließt. Das liegt in der Tatsache begründet, daß man historisch den Strom als positive Ladungsträger definiert hat, Elektronen allerdings negativ geladen sind (Abb. 2.1(b)).

2.3. Physikalische Größen

Im Folgenden werden die physikalischen Größen so definiert, wie sie aus der Elektrizitäts-Lehre bekannt sein sollten. Eine Veranschaulichung findet dann in Kapitel 2.4 statt.



(a) Feldfreier Zustand (b) Stromfluß

Abbildung 2.1.: Metallische Leiter

Stromstärke

Naheliegend nach den vorangegangenen Betrachtungen ist es, eine Größe einzuführen, die die Menge der elektrischen Ladungen definiert, die pro Zeiteinheit fließt. Diese Größe ist die Stromstärke I, die in Ampere (A) gemessen wird.

Die elektrische Stromstärke I ist definiert als Ladungsmenge Q, die pro Zeit t durch einen Leiter fließt:

$$I = \frac{Ladung}{Zeit} = \frac{dQ}{dt} \tag{2.1}$$

$$1\,\mathrm{A} = 1\frac{\mathrm{Coulomb}}{\mathrm{Sekunde}} = 1\frac{\mathrm{C}}{\mathrm{s}} \tag{2.2}$$

$$1C \approx 6.24 \cdot 10^{19} Elektronen \tag{2.3}$$

Spannung

Als Pendant zur Stromstärke definieren wir die elektrische Spannung U. Diese Größe beschreibt die Potentialdifferenz/Energie, die die Ladungsträger erfahren. Die Spannung ist nicht ganz so intuitiv zu verstehen, wird aber im folgenden Abschnitt noch etwas veranschaulicht.

$$U = \frac{Energie}{Ladung} = \frac{dE}{dQ} \tag{2.4}$$

Gemessen wird die Spannung in Volt (V).

$$1\,\mathrm{V} = 1\,\frac{\mathrm{Joule}}{\mathrm{Coulomb}} = 1\,\frac{\mathrm{J}}{\mathrm{C}} \tag{2.5}$$

Leistung

Das Produkt aus Spannung und Strom definiert man als elektrische Leistung

$$P = U \cdot I \tag{2.6}$$

Durch Einsetzen von Gleichungen 2.1 und 2.4 erhält man

$$P = U \cdot I = \frac{dE}{dQ} \cdot \frac{dQ}{dt} = \frac{dE}{dt} \tag{2.7}$$

Gemessen wird die Leistung in Watt (W).

$$1\,\mathrm{W} = 1\,\frac{\mathrm{J}}{\mathrm{C}} \cdot \frac{\mathrm{C}}{\mathrm{s}} = 1\,\frac{\mathrm{J}}{\mathrm{s}} \tag{2.8}$$

Die elektrische Leistung ist beispielsweise dann interessant, wenn man die Verlustleistung und damit die Wärmeentwicklung eines elektrischen Bauteils betrachtet. Auch die mechanische Leistung eines elektrischen Motors kann so abgeschätzt werden, wenn man ungefähr den Wirkungsgrad kennt.

Widerstand

Der elektrische Widerstand R ist der Quotient aus Spannung U und Stromstärke I.

$$R = \frac{U}{I} \Leftrightarrow U = R \cdot I \tag{2.9}$$

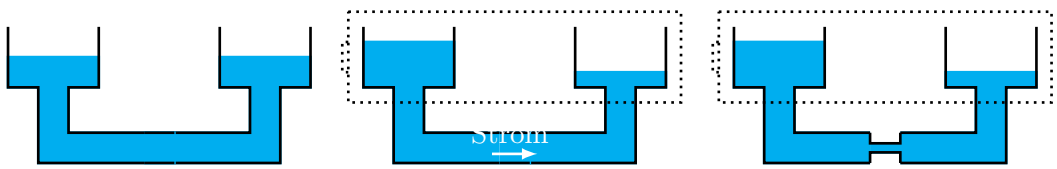
Gemessen wird der Widerstand in Ohm (Ω).

$$1\,\Omega = 1\,\frac{\mathrm{V}}{\mathrm{A}} \tag{2.10}$$

Der Widerstand ist nicht nur eine physikalische Größe; Widerstände sind auch diskrete Bauteile in Schaltungen.

2.4. Wasser Analogon

Abbildung 2.2(a) zeigt den Aufbau eines Gedankenexperiments, das der Veranschaulichung die elektrischen Größen dient. Abgebildet sind zwei nach oben hin geöffnete Behälter, die über eine gemeinsame Leitung miteinander verbunden sind. Die Apparatur ist so mit Wasser befüllt, daß der Wasserspiegel in beiden Behältern gleich hoch ist. Der Zustand ist statisch, es fließt kein Wasser.



(a) Ausgeglichenes Potential (b) Potentialunterschied (c) Widerstand

Abbildung 2.2.: Wasser Analogon

Ist der Wasserspiegel bei beiden Behältern nicht mehr auf gleicher Höhe, so entsteht ein Potentialunterschied (Abb. 2.2(b)). Um diesen auszugleichen, fließt Wasser durch die gemeinsame Leitung vom höheren Potential zum niedrigeren.

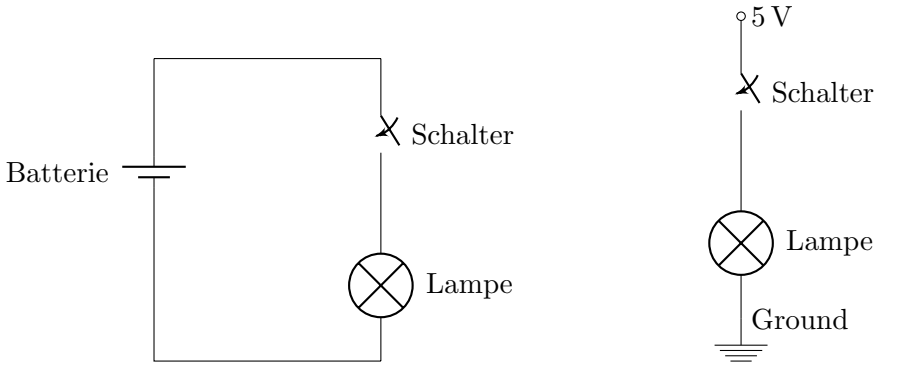
Ebenso fließt ein elektrischer Strom vom Ladungsüberschuß zum Ladungsmangel. Eine Batterie, oder allgemeiner eine Spannungsquelle, kann man als ein Bauteil auffassen, das eine konstante Potentialdifferenz (Spannung) herzustellen versucht. Im unserem Analogon entspricht dies also einer Pumpe, die immer genau so viel Wasser von einem Behälter in den anderen transportiert, daß die Wasserspiegel den gleichen Höhenunterschied beibehalten.

Eine Verengung der Leitung wie in Abbildung 2.2(c) führt zu einem reduzierten Durchfluß. Dies entspricht einem elektrischen Widerstand. Anschaulich wird klar, daß der Höhenunterschied darüber entscheidet, wie viel Wasser durch die Verengung strömt. Dieser Zusammenhang wird in der Elektrizitäts-Lehre durch die Größe des Widerstands $R = \frac{U}{I}$ beschrieben (vgl. Gleichung 2.9).

2.5. Schaltungen

Einfacher Stromkreis

Wie aus den vorherigen Abschnitten ersichtlich, müssen Stromkreise immer geschlossen sein damit ein Strom fließen kann. Abbildung (2.3(a)) zeigt einen einfachen Stromkreis aus Spannungsquelle (Batterie), einem Verbraucher (Lampe) und einem Schalter. Durch Öffnen und Schließen des Schalters kann der Stromkreis unterbrochen bzw. geschlossen werden.



(a) Vollständig

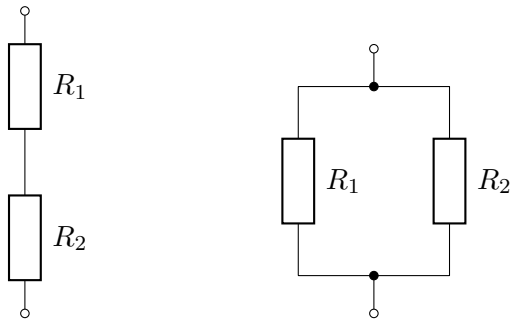
(b) Vereinfachte Darstellung

Abbildung 2.3.: Einfacher Stromkreis

Um Schaltpläne übersichtlicher zu gestalten, wird wie in Abb. 2.3(b) der Teil mit der Spannungsversorgung nicht mehr explizit dargestellt. Stattdessen wählt man ein gemeinsames Referenzpotential für alle Spannungen. Dieses wird *Masse* oder *Ground* genannt. Alle Spannungsangaben, wie die 5 V in Abb. 2.3(b), beziehen auf dieses Potential

Reihenschaltung

Abbildung 2.4(a) zeigt die Reihenschaltung zweier Widerstände R_1 und R_2. Mit ein paar einfachen Überlegungen ist es möglich, sich den resultierenden Gesamtwiderstand R_{ges} herzuleiten.



(a) Reihenschaltung

(b) Parallelschaltung

- *Ladungserhaltung:* Wir gehen davon aus, daß keine Ladung entsteht oder vernichtet wird. Folglich müssen zu jedem Zeitpunkt gleich viele Ladungsträger und damit auch gleich viel Strom durch R_1 wie durch R_2 fließen.
  $\Rightarrow I_1 = I_2 = I_{ges}$
- *Energieerhaltung:* Die ist trivial und gerade deshalb nicht unbedingt leicht verständlich. Die Energie, die durch den Gesamtwiderstand verbraucht wird, ist die Summe der Energien der einzelnen Widerstände
  $\Rightarrow U_{ges} = U_1 + U_2$

Zusammengesetzt erhalten wir

$$\Rightarrow R_{ges} = \frac{U_{ges}}{I_{ges}} = \frac{U_1}{I_1} + \frac{U_2}{I_2} = R_1 + R_2 \tag{2.11}$$

Zwei Widerstände in Reihe addieren sich.

Parallelschaltung

Für die Parallelschaltung in Abbildung 2.4(b) läßt sich der Gesamtwiderstand mit Hilfe der gleichen Erhaltungssätze herleiten.

- *Ladungserhaltung:* Da keine Ladung verloren geht oder erzeugt wird, teilt sie der Gesamtstrom auf beide Widerstände auf
  $\Rightarrow I_{ges} = I_1 + I_2$
- An beiden Widerständen liegt die gleiche Spannung an
  $\Rightarrow U_{ges} = U_1 = U_2$

Zum Berechnen des Gesamtwiderstands hilft es, den Kehrwert des Gesamtwiderstands zu betrachten.

$$\Rightarrow \frac{1}{R_{ges}} = \frac{I_{ges}}{U_{ges}} = \frac{I_1 + I_2}{U_{ges}} = \frac{I_1}{U_{ges}} + \frac{I_2}{U_{ges}} = \frac{1}{R_1} + \frac{1}{R_2} \tag{2.12}$$

Diese Gleichung läßt sich nun stürzen und nach R_{ges} auflösen

$$\Rightarrow R_{ges} = \frac{1}{\left(\frac{1}{R_1} + \frac{1}{R_2}\right)} = \frac{R_1 \cdot R_2}{R_1 + R_2} \tag{2.13}$$

Die Kehrwerte der Einzelwiderstände addieren sich zum Kehrwert des Gesamtwiderstands.

2.5.1. Vorwiderstände

Nachdem nun die Grundprinzipien einfacher Schaltungen mit ohmschen Widerständen geläufig sein sollten, folgt nun ein praxisnahes Beispiel, die Berechnung von Vorwiderständen.

Häufig werden LEDs[1] zur Anzeige und Ausgabe einfacher Signale verwendet. LEDs sind aber keine ohmschen Verbraucher wie Widerstände. Sie benötigen eine definierte Spannung und der durch sie fließende Strom ist ebenfalls vorgegeben. Die Diodenspannung ist unter anderem abhängig von der Farbe der Diode.

Um eine beliebige (aber natürlich bekannte) Diode an beispielsweise 5 V zu betreiben, wird ein Vorwiderstand benötigt (vgl. Abb. 2.4). Dieser Vorwiderstand soll nun berechnet werden:

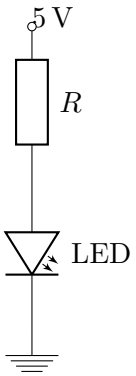


Abbildung 2.4.: Diode

Gleichung (2.11) ist aus der Reihenschaltung bekannt

$$\rightarrow \frac{U_{ges}}{I_{ges}} = R + \frac{U_{LED}}{I_{LED}} \tag{2.14}$$

Umformung nach R ergibt

$$\Rightarrow R = \frac{U_{ges}}{I_{ges}} - \frac{U_{LED}}{I_{LED}} = \frac{U_{ges} - U_{LED}}{I_{LED}} \tag{2.15}$$

Beispiel:

Betrieb einer grünen LED mit $U_{LED} = 2.1\,\mathrm{V}$ und $I_{LED} = 15\,\mathrm{mA}$ an $5\,\mathrm{V}$ Spannung. Vorwiderstand R ergibt sich durch Einsetzen der Werte

$$\Rightarrow R = \frac{5\,\mathrm{V} - 2.1\,\mathrm{V}}{15\,\mathrm{mA}} = \frac{2.9}{0.015}\,\Omega \approx 193\,\Omega \tag{2.16}$$

Widerstände sind als Bauteile nicht in jeder beliebigen Abstufungen erhältlich. Da alle Bauteile gewissen Fertigungstoleranzen unterliegen, ist eine exakte Wahl ohnehin nicht möglich. In der Praxis greift man in diesem Fall zu dem nächstgelegenen verfügbaren Widerstand. Der nächste passende Wert ist in unserem Fall $R = 220\,\Omega$.

[1] **L**ight **E**mitting **D**iode, Leuchtdiode

3. A Mikrocontroller

Ein Mikrocontroller vereint im Gegensatz zu einem reinen Prozessor zusätzlich Peripherie-Komponenten, wie serielle oder Bus-Protokolle zur Ein- und Ausgabe von Daten sowie meist auch einen Programm- und Datenspeicher. Sie benötigen dadurch weit weniger externe Komponenten und sind so vor allem im Embedded-Bereich verteten. Man kann sie weitgehend auch den SoC[1]-Prozessoren zuordnen.

Mikrocontroller sind aus der heutigen modernen Welt kaum wegzudenken, verrichten aber meist im Hintergrund ihre Tätigkeit. Sie werden überall dort eingesetzt, wo eine Steuer- oder Regeltätigkeit ausgeführt werden soll, deren Komplexität sich nicht durch eine einfache elektrische Schaltung realisieren läßt. Auch aus kostengründen wird der Einsatz eines Mikrocontrollers oft der einer Schaltung bevorzugt, da dieser je nach Ausführung bereits für wenige Euro zu haben ist.

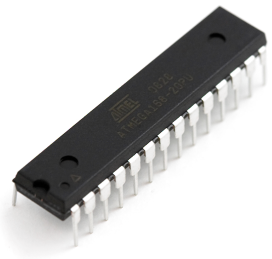


Abbildung 3.1.: Atmega168

Einsatzgebiete sind die Unterhaltungsindustrie mit CD-Playern, Radios und Spielzeug, im Haushalt in Weckern, Küchenherden und Thermostaten oder ganz prominent in Kraftfahrzeugen. Dort werden teilweise bis zu 80 Stück pro Fahrzeug gebaut. Dieser Trend ist allerdings rückläufig, da man zunehmend leistungsfähigere Hardware einsetzt und so die Funktionalität mehrerer Einheiten zusammenfaßt. Die in modernen Mobiltelefonen oder MP3-Spielern eingesetzten Prozessoren sind mittlerweile so leistungsfähig, daß man sie nur noch bedingt als Mikrocontroller bezeichnen kann.

3.1. Der Atmega168

Der in diesem Praktikum verwendeten Atmega 168 der Firma Atmel (Abb. 3.1) besteht aus zahlreichen Komponenten, von denen hier nur eine Auswahl gezeigt wird:

- 8-bit Prozessor mit max. 20 MHz Taktfrequenz
- Speicher
  - Programmspeicher (FLASH) mit 16 KBytes
  - Datenspeicher (SRAM) mit 1 KBytes
  - Nicht-flüchtiger Datenspeicher (EEPROM) mit 512 Bytes
- Integrierte Peripherie
  - Analog-Digital-Wandler

[1] System-on-Chip

  - Timer für Pulsweitenmodulation

- Zahlreiche Ein-/Ausgabe Schnittstellen
  - Pins
  - Serielle Schnittstelle (UART)

Für nähere Informationen sei auf das Datenblatt [4] verwiesen. Die aufgeführten Bestandteile mögen zum jetzigen Zeitpunkt noch nicht verständlich sein, es werden aber die meisten dieser Komponenten im Laufe dieses Praktikums behandelt werden.

3.2. Basis-Platine

Der nackte Controller, wie er in Abb. 3.1 gezeigt wird, ist als solcher ohne externe Beschaltung nicht sinnvoll einsetzbar. Die von uns entwickelte *Basis-Platine* (Abb. 3.2) enthält alle für den Betrieb notwendigen Komponenten und bietet viele Anschlußmöglichkeiten. Die Platine verfügt über folgende Komponenten:

- Einen Spannungsregler für die 5 V-Versorgung des Controllers.
- Eine Power-LED, die leuchtet, sobald Spannung anliegt.
- Glättung der Versorgungsspannung des Analog-Digital-Wandlers
- Einen 16 MHz-Quarz
- Eine 10-polige ISP-Programmierbuchse
- Einen Reset-Taster (ab Version 1.1)
- Zwei PMOS-Transistoren (optional, ab Version 1.1)

Neben den festen Komponenten, die für den Betrieb notwendig sind, bietet die Basis-Platine viele Pins zum freien Anschluß weiterer Komponenten.

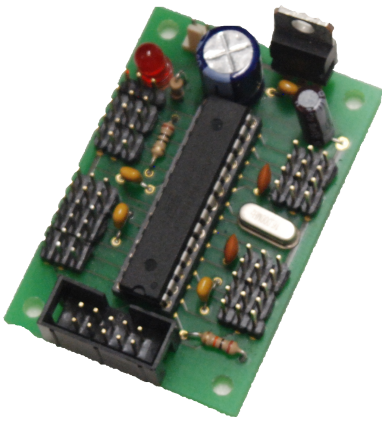


Abbildung 3.2.: Basis-Platine (Version 1.0)

3.3. Aufgabe A.1: Zusammenbau

Bauen Sie die Platine gemäß Lötanleitung zusammen.
http://joanna.iwr.uni-heidelberg.de/rlab/download/atmega-doku.pdf
Falls Sie noch nicht gelötet haben, lassen Sie es sich von Ihrer Betreuerin/Ihrem Betreuer zeigen, die/der Ihnen ebenfalls die dazu benötigten Komponenten aushändigen wird. Lassen Sie sich dabei Zeit, denn eine verläßlich funktionierende Platine erspart im Laufe des Praktikums viel Ärger bei der Fehlersuche!

3.4. Aufgabe A.2: Inbetriebnahme

a) Für die Programmierung wird ein Basis-Code angeboten, der die wichtigsten Funktionen bereits implementiert hat und über ein Makefile zum Kompilieren und Übertragen auf den Mikrocontroller verfügt.

Laden Sie sich den Quellcode mit Hilfe der Versionsverwaltung `git` herunter. Da der Code-Server nicht über ein Zertifikat verfügt, wird mit der ersten Zeile in folgendem Beispiel die Überprüfung abgeschaltet:

```
1 export GIT_SSL_NO_VERIFY=true
2 git clone https://joanna.iwr.uni-heidelberg.de/code/atmega/basis
```

Es wird ein Verzeichnis mit dem Namen `basis` erstellt.

Geben Sie `make` in diesem Verzeichnis ein. Sie sollten eine Ausgabe ähnlich der folgenden erhalten:

```
1  Compiling main.c ...
2  Compiling adc.c ...
3  Compiling timer.c ...
4  Compiling pwm.c ...
5  Compiling uart.c ...
6  Compiling servo.c ...
7  Linking basis.elf ...
8  Creating basis.hex ...
9  Creating basis.eep ...
10 avr-objcopy: --change-section-lma .eeprom=0x0000000000000000 never
       used
```

Durch Eingabe des Befehls `make` wird das Projekt in die Datei `basis.hex` kompilliert.

b) Bei Benutzung eines neuen Mikrocontrollers müssen bei der Erstinbetriebnahme die sogenannte FUSE-Bits gesetzt werden. Diese sind eine Konfiguration, die den Controller beispielsweise anweisen, den externen Quarz auf der Basis-Platine zu verwenden.

Dies kann zur Zeit nur mit den blauen AVR-Programmieradapter geschehen, die anderen sind dafür leider nicht geeignet.

Geben Sie dazu `make fuse` ein.

c) Um Ihren kompillierten Code zu übertragen, geben Sie `make prog` ein. Die Übertragung kann mit jedem Programmieradapter erfolgen, allerdings muß im Makefile der gewünschte Adapter aktiviert werden. Öffnen Sie die Datei `Makefile` mit in einem Texteditor.

```
1 #prog: prog-usb
2 #prog: prog-ser
3 #prog: prog-funk
4 prog: prog-diamexusb
```

Im obigen Beispiel sind die bunten Diamex-Programmieradapter aktiviert. Die blauen AVR-Adapter sind die in der ersten Zeile. Zeile 2-3 werden zur Zeit nicht verwendet.

4. B Ein- und Ausgabe

4.1. Pins & Ports

Die „Beinchen“ des Mikrocontroller-Chips werden als Pins bezeichnet. Während einige wenige Pins dedizierte Aufgaben haben, wie den Controller mit Spannung zu versorgen, stehen die meisten dem Anwender als programmierbare Pins zur Verfügung. Diese *Pins* wiederum werden zu Gruppen von acht zu sogenannten *Ports* zusammengefaßt. Ports werden mit großen Buchstaben bezeichnet während Pins mit Zahlen (0...7) durchnummeriert werden. So ist beispielsweise `B5` der sechste Pin an Port B.

Abbildung 4.1 zeigt den verwendeten Microcontroller mit beschrifteten Pins. Einige dieser Pins haben weitere Spezialfunktionen, diese sind in der Abbildung in Klammern hinter der Pinbezeichnung aufgeführt.

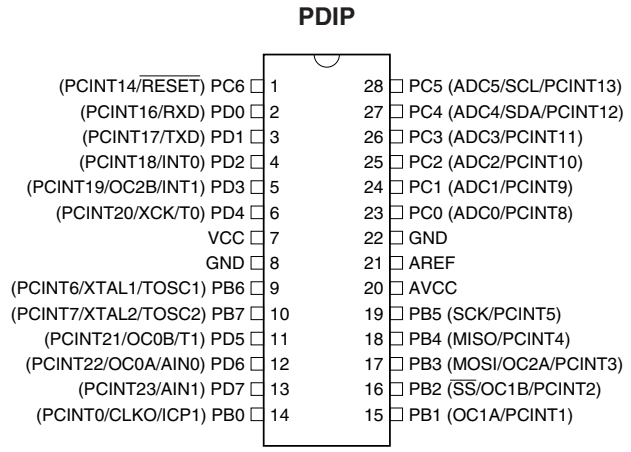


Abbildung 4.1.: Pinout des Atmega168 aus [4]

4.2. Bit-Manipulation

Die Ports stehen dem Programmierer als 8-bit Prozessor-Register zur Verfügung, wobei jedes Bit des Registers einem Pin entspricht. Daraus ergibt sich die Notwendigkeit, einzelne Bits einer Variablen bzw. eines Registers unabhängig manipulieren zu können. Dies wird in `C` mittels Bit-Shifting und bitweiser Logik durchgeführt.

4.2.1. Bitshifting

Die Bitshift-Operatoren sind `<<` und `>>`. Sie dürfen keinesfalls mit den Stream-Operatoren von `C++` verwechselt werden! Der Ausdruck `(a << b)` bedeutet, daß die binäre Repräsentation

von `a` um `b` Stellen nach links verschoben wird. Leere Stellen werden mit 0 aufgefüllt. Tabelle 4.1 zeigt einige Beispiele, die das Verständnis erleichtern sollen.

C-Code	Dezimal	Bit
`1`	1	00000001
`3`	3	00000011
`4`	4	00000100
`(1 << 2)`	4	00000100
`(1 << 3)`	8	00001000
`(3 << 2)`	12	00001100
`(4 >> 1)`	2	00000010
`(3 >> 1)`	1	00000001
`(4 >> 4)`	0	00000000

Tabelle 4.1.: Beispiele für das Bit-Shiften

Der Ausdruck `(1 << n)` bedeutet also eine binäre 1 an Stelle `n`.

4.2.2. Bitweise Logik

Bitweise Logik unterscheidet sich von Bool'scher Logik darin, daß die Logik-Operationen (AND, OR, XOR, NOT, ...) nicht auf den Ausdruck einer Variablen sondern auf jedes Bit einzeln angewendet werden. Tabelle 4.2 zeigt einen Vergleich der Darstellungsweise der Operatoren.

	Bool'sch	bit-weise
AND	`&&`	`&`
OR	`||`	`|`
XOR	`!=`	`^`
NOT	`!`	`~`

Tabelle 4.2.: Bool'sche und bit-Weise Logikoperatoren in `C`

Ein Bit setzen

Zum Setzen eines Bits wird der ODER-Operator (`|`) verwenden. Dabei wird ausgenutzt, das gilt:

$$0 \vee n = n$$
$$1 \vee n = 1$$

Im folgenden Beispiel wir das sechste Bit der Variablen `a` gesetzt. Tabelle 4.3 verdeutlicht dies in binärer Schreibweise.

Beispiel:

```
1 // Ausfuehrlich
2 a = a | (1 << 5);
3 // ...oder abgekuerzt
4 a |= (1 << 5);
```

$$\begin{array}{rr} & 10010010_2 \\ | & 00100000_2 \\ \hline = & 10110010_2 \end{array}$$

Tabelle 4.3.: Binäre Darstellung: Setzen eines Bits

Ein Bit löschen

Zun Löschen eines Bits wird der UND-Operator in Verbindung mit dem NICHT-Operator verwendet. Hierbei gilt:

$$0 \wedge n = 0 \Rightarrow \neg 1 \wedge n = 0$$
$$1 \wedge n = n \Rightarrow \neg 0 \wedge n = n$$

Beispiel:
Das folgende Beispiel löscht das sechste Bit in einer Variablen. Eine exemplarische Darstellung in Bit-Schreibweise ist in 4.4 abgebildet.

```
1 // Ausfuehrlich
2 a = a & ~(1 << 5);
3 // ...oder abgekuerzt
4 a &= ~(1 << 5);
```

$$\begin{array}{rr} & 10110010_2 \\ \& \quad \sim & 00100000_2 \\ \hline & 10110010_2 \\ \& & 11011111_2 \\ \hline = & 10010010_2 \end{array}$$

Tabelle 4.4.: Binäre Darstellung: Löschen eines Bits

4.2.3. Ein- und Ausgabe

Datenrichtung

Pins sind standardmäßig als Eingabepins konfiguriert. Um einen Pin auf Ausgang zu schalten, ist das entsprechende Bit im Data Direction Register (`DDR`x) zu setzen.

Beispiel:

```
1 //Pin C5 auf Ausgang setzen
2 DDRC |= (1<<5);
```

Ausgabe

Ist der Pin als Ausgang konfiguriert, so kann man ihn mit Hilfe der `PORT`x-Register entweder auf HIGH (5 V) oder auf LOW (GND) legen.

Beispiel:

```
1 //Pin C5 HIGH setzen
2 PORTC |= (1<<5);
3 //Pin C5 LOW setzen
4 PORTC &= ~(1<<5);
```

Eingabe

Ob der Pegel eines Pins HIGH (5 V) oder LOW (GND) ist, kann analog zur Ausgabe bitweise in den Registern `PIN`x ausgelesen werden.

Beispiel:

```
1 if (PINC & (1<<4)) { // C4 HIGH ?
2         // ist HIGH
3 } else {
4         // ist LOW
5 }
```

Wie in diesem Beispiel gezeigt, bietet es sich an, sich die Eigenschaft von `C` zu nutze zu machen, daß Ausdrücke, die $= 0$ als `false` und Ausdrücke $\neq 0$ als `true` interpretiert werden.

4.3. LED anschließen

4.3.1. Pins auf der Basis-Platine

Die Basis-Platine besitzt für jeden Mikrocontroller-Pin, der herausgeführt ist und daher in der Programmierung verwendet werden kann, eine Dreiergruppe aus Steckpins. Diese sind jeweils der herausgeführte Pin des Controllers mit einem 5 V-Pin in der Mitte sowie einem Ground-Pin kombiniert (siehe Abb. 4.2). Dadurch ist es möglich, Peripherie wie LEDs, Taster, Servos etc. mit einem Stecker direkt anzuschließen. Als einzige Ausnahme muß die Leiste, die in Abbildungen 4.2 violett dargestellt ist, manuell angeschlossen werden. Dies wird in Kapitel 11.1.1 beschrieben und ist hier vorerst nicht von Bedeutung.

Die Zuordnung zwischen MC-Pins und deren Verbindung zur Basis-Platine kann auf der Unterseite der Platine nachgesehen werden.

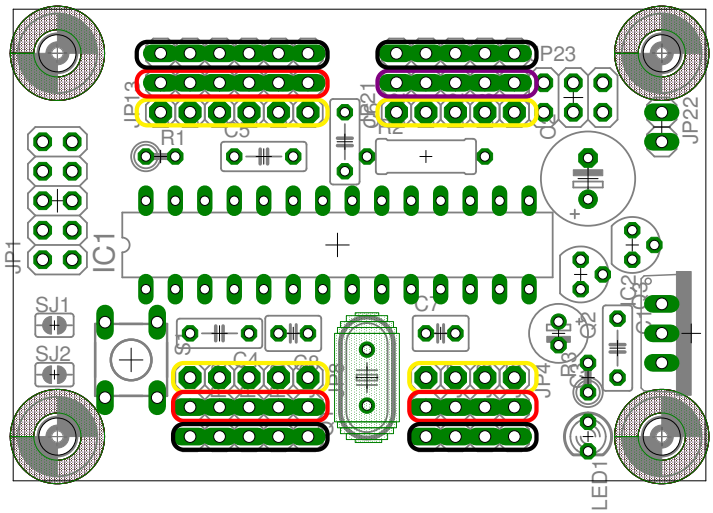


Abbildung 4.2.: Pin-Anordnung: Controller-Pin (gelb), 5 V (rot), Ground (Schwarz)

4.3.2. LEDs

Bei einer LED gilt es zuerst die Polaritität festzustellen. Im Allgemeinen ist das längere Beinchen die Anode (+) und das kürzere die Kathode (−). Auch ein Blick gegen das Licht hilft bei der Bestimmung (4.3).

Eine Verpolung übersteht eine Diode problemlos, in Sperrrichtungen fließt kein Strom und die Diode leuchtet nicht. Es ist aber darauf zu achten, daß Leuchtdioden nie ohne Vorwiderstand betrieben werden, weil sie so relativ schnell zerstört werden. Die in diesem Praktikum verwendeten Signaldioden haben bereits einen Vorwiderstand für 5 V integriert. Die Berechnung eines Vorwiderstands kann in Kapitel 2.5.1 nachgelesen werden.

Abbildung 4.3.: Polarität einer LED

4.4. Aufgabe B.1: LED ansteuern

a) Schließen Sie die Anode (+) an `PC5` des Mikrocontrollers und die Kathode (−) an Ground an. Damit die LED zu leuchten beginnt, muß das Programm folgende Dinge enthalten:

- Die Datenrichtung von `C5` muß auf Ausgang gestellt werden.
  Dazu wird Bit Nummer 5 in `DDRC` auf 1 gesetzt.
- Der Pin selbst muß `HIGH` gesetzt werden.
  In `PORTC` wird dafür ebenfalls das Bit Nummer 5 auf 1 gesetzt.
- Das Programm muß in einer Endlosschleife stehen bleiben.
  Dieser Punkt ist allgemein wichtig. Ein Mikrocontroller-Progamm sollte niemals die `main`-Funktion verlassen. Das hat einen Software-Neustart zur Folge, ohne daß dabei die Hardware zurückgesetzt wird. Dies kann unter Umständen zu unerwarteten Effekten führen. Daher wird stark empfohlen, jedes Programm mit einer Endlosschleife am Verlassen der `main`-Funktion zu hindern.

Kompilieren und übertragen Sie Ihr Programm mit `make prog`. Die LED sollte nun Leuchten. Tut Sie das nicht, so ist wahrscheinlich die LED verpolt oder sie wurde am falschen Pin angeschlossen.

b) Lassen Sie nun die LED blinken.

- Setzen Sie wie im vorangegangenen Aufgabenteil die Datenrichtung auf Ausgang.
- In einer Endlosschleife `while(1) {...}` aktivieren und deaktivieren Sie die LED abwechselnd, indem Sie Bit Nummer 5 abwechselnd setzen und löschen.
- Benutzen Sie dabei den Aufruf `_delay_ms(500)`, um zwischen den Aufrufen eine halbe Sekunde zu warten.

4.5. Aufgabe B.2: Systemzeit

Es ist allgemein kein guter Stil, delay-Funktionen zu verwenden, da diese den Controller blockieren und er in dieser Zeit für andere Aufgaben nicht zur Verfügung steht. Eine Ausnahme bilden Interrupts, die trotzdem währenddessen ausgeführt werden. Diese werden aber erst in Kapitel 10 behandelt.

Für zeitlich gesteuerte Aufgaben gibt es eine Art Systemzeit, die vom Basis-Code zur Verfügung gestellt wird. Die `getMsTimer()`-Funktion gibt einen 32-bit unsigned Integer mit der Zeit seit dem letzten Reset in Millisekunden zurück.

Folgendes Beispiel verwendet die Funktion für ein wiederkehrendes Ereignis:

```
1 const uint16_t delay=1000;
2 uint32_t nextEvent=getMsTimer()+delay;
3 while (1) {
4         if (nextEvent<getMsTimer()){
5                 nextEvent+=delay;
6                 // Einmal jede Sekunde
7         }
8         // Sonstige Aufgaben
9 }
```

a) Wie lange dauert es ungefähr, bis die 32-bit Variable überläuft? Warum wird in Zeile 4 des Beispiels nicht auf Gleichheit überprüft?

b) Implementieren Sie nun das Blinken mit Hilfe der Systemzeit. Verwenden Sie zum Umschalten der LED den XOR-Operator ($\underline{\vee}$):

$$n \underline{\vee} 1 = \neg n$$

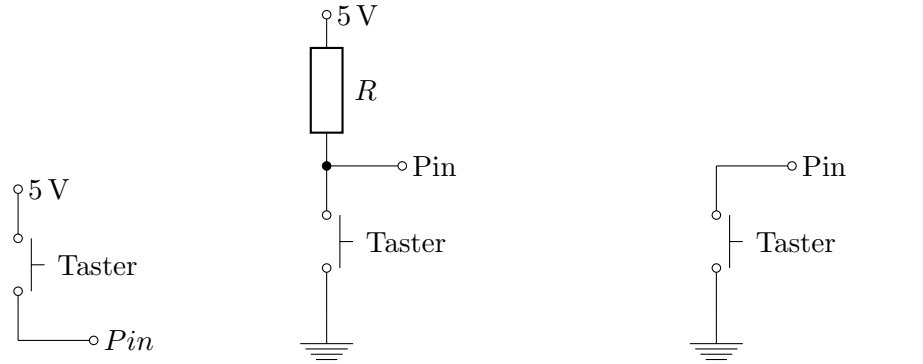
Siehe dazu Tabelle 4.2

4.6. Taster anschließen

Das Signal an einem Eingangs-Pin sollte jederzeit auf einem definierten Potential liegen. Ist das nicht der Fall, d.h. hängt der Pin ohne Verbindung „in der Luft“, ist der eingelesene Wert relativ zufällig.

In Abbildung 4.4(a) verbindet ein einfacher Taster (Schließer) beim Drücken die Leitung und am Pin liegen 5 V an. Ist der Taster hingegen nicht gedrückt, sind die Kontakte offen und der Pin bekommt kein sauberes Signal. Dieser Zustand führt zu einem zufälligen Ergebnis beim Einlesen des Signals. Deshalb sollte man nie einen Schalter auf diese Weise anschließen.

Ein Ausweg wäre die Verwendung eines Umschalters, mit dessen Hilfe man von einem Signal auf das andere Umschalten kann, also hier von 5 V auf 0 V. Dieser Aufbau führt aber zu einem teureren Taster und zu mehr Leitungen, daher strebt man im Allgemeinen eine andere Lösung an.



(a) Einfacher Taster, **falsch angeschlossen** (b) Pullup-Widerstand (c) Interner Pullup

Abbildung 4.4.: Taster am Eingang

Abbildung 4.4(b) zeigt einen anderen Weg; Widerstand R in dieser Schaltung *zieht* das Signal *nach oben* auf 5 V sobald der Schalter geöffnet ist. Daher kommt der Name *Pullup-Widerstand*. Wird der Schalter geschlossen, so verbindet er den Pin direkt mit Ground bzw. 0 V, der Widerstand ist, sofern groß genug gewählt, nicht mehr von Bedeutung. Der Widerstand sollte so groß gewählt werden, daß kein nennenswerter Strom von 5 V über den Widerstand R nach Ground fließt. Typische Werte sind im Bereich von 100 kΩ bis 1 MΩ.

Damit ist erreicht, daß das Signal am Eingang des Controllers in beiden Schaltzuständen sauber definiert ist.

Ein solcher Pullup-Widerstand läßt sich auch intern im Mikrocontroller dazu schalten. Dies geschieht, indem man im betreffenden `PORTx`-Register eine 1 schreibt, obwohl man den Pin im `DDRx`Register als Eingang konfiguriert hat.

Ein Taster läßt sich auf diese Weise wie in Abbildung 4.4(c) einfach zwischen Pin und Ground schalten.

Ein Nachteil hat diese Schaltweise allerdings: Das Drücken des Tasters führt zum Wechsel des Signals von `HIGH` auf `LOW`. Der Schalter ist also invertierend (*Low-Active*) angeschlossen.

4.7. Aufgabe B.3: Taster einlesen

Schreiben Sie ein Programm, bei dem die LED genau solange leuchtet, wie ein Taster, den Sie an `C3` anschließen, gedrückt ist. Schließen Sie dazu einen Taster zwischen dem Pin des Controllers und GND an. Denken Sie an den internen Pullup-Widerstand.

4.8. Flankenerkennung

4.8.1. Flanken

Eine Flanke ist der Wechsel eines Signals. Abbildung 4.5 zeigt die rotgestrichelten markierten Bereiche beider Flanken.

Den Wechsel `LOW` → `HIGH` wird als steigende Flanke (*rising edge*) bezeichnet. Umgekehrt ist der Wechsel `HIGH` → `LOW` eine fallende Flanke (*falling edge*).

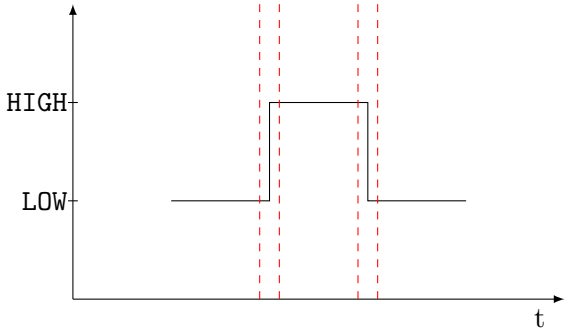


Abbildung 4.5.: Flanken gekennzeichnet

Die meisten Programme erfordern es, daß nur ein einziges mal beim Drücken des Taster eine Aktion ausgeführt wird. Das könnte z.B. das Weiterschalten eines Menüpunkts sein oder das Umschalten eines Schaltzustands. Fragt das Programm lediglich ab, ob eine Taste gedrückt ist, dann wird diese Aktion so oft ausgeführt, wie die Taste gedrückt ist. Mit einer Flankenerkennung erreicht man es nun, daß eine Aktion nur genau ein einziges Mal pro Tastendruck ausgeführt wird.

4.8.2. Flankenerkennung

Um nun eine Flanke d.h. eine Änderung in der Software zu erkennen, muß darauf zurückgegriffen werden, was der Wert des Signals beim vorhergegangenen Durchlauf der Schleife war.
Ein Flankenerkennung kann also wie folgt aussehen.

Beispiel:

```
1  uint8_t sw_alt=0;
2  while (1) {
3          uint8_t sw= ! (PINC & (1<<4));
4          if (sw_alt < sw) {
5             // steigende Flanke
6          } else if (sw_alt > sw) {
7             // fallende Flanke
8          } else {
9             // gar keine Flanke
10         }
11         sw_alt=sw;
12 }
```

In Zeile 3 dieses Codebeispiels wird der Wert des Schalters eingelesen. Da dieser aber Low-Active (vgl. 4.6) angeschlossen ist, wird der Wert invertiert. So enthält die Variable `sw` den tatsächlichen Zustand des Tasters. Die Variable `sw_alt` speichert den Wert von `sw` beim letzten Durchlauf der Schleife. Die Erkennung der Flanke selbst geschieht durch einen Vergleich dieser beiden Variablen.

4.9. Aufgabe B.4: Flankenerkennung

Implementieren Sie eine Flankenerkennung und Schalten Sie die LED bei jedem Tastendruck um. Verwenden Sie zum Umschalten wie auch davor den XOR-Operator.

Eventuell kann es vorkommen, daß der Zustand nicht zuverlässig gewechselt wird und relativ zufällig aussieht. In diesem Fall prellt der Taster, d.h. vor dem eigentlichen Kontakt beider Elektroden kommt es zu sehr kurzen Vorkontakten, ähnlich dem Prellen eines Basketballs. Die einfachste Methode, ein Prellen zu vermeiden, ist das Einfügen ein kleines Delays von einigen 10 ms bei Erkennung einer Flanke. Die Dauer ist abhängig von der Bauart und Qualität des Tasters.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

5. C Zustandsautomat

5.1. Automaten

Der Begriff des endlichen Zustandsautomaten bzw. finite state machine (FSM) sollte aus Vorlesungen der Informatik bekannt sein. In dieser Übung geht es darum, eine Fußgängerampel mit Hilfe eines Zustandsautomaten zu implementieren. Dabei sollen gewisse Anforderungen eingehalten werden.

- Der Automat selbst soll nicht-blockierend sein, d.h. delay-Funktionen für das Timing dürfen nicht verwendet werden.
- Die Implementation erfolgt mittels `enum`, sowie `switch...case`-Anweisungen.

5.2. `enum`

Ein `enum` ist äquivalent zu einem ein Integer, dessen Werte keine ganzen Zahlen sind, sondern selbst zu vergebende Bezeichnungen haben können, was die Lesbarkeit des Codes sehr verbessert. Folgender Ausschnitt verdeutlicht die Verwendung von `enum`s:

```
1 enum MeinEnumTyp {AUS, AN, DEFEKT} zustand;
2 enum MeinEnumTyp andererZustand;
3 if (zustand == DEFEKT) {
4         //reparieren
5         zustand=AUS;
6 }
```

Zeile 1 legt ein neues `enum` mit den drei Werten `AUS`, `AN` und `DEFEKT` an. Die Werte eines `enum`s sollten analog zu `define`s groß geschrieben werden, um eine Verwechslung mit Variablennamen zu vermeiden.

Zeile 2 legt einen zweiten Zustand des vorher definierten Typs `MeinEnumTyp` an. Die Zeilen 3 und 5 verdeutlichen die Lesbarkeit des Quelltexts unter Verwendung von `enum`s.

5.3. `switch...case`

Das Abhandeln verschiedener Werte einer Integervariablen läßt sich mit `switch...case` im Gegensatz zu `if...else if ...else` deutlich vereinfachen, wie im folgenden Beispiel gezeigt wird.

```
1  switch(z) {
2          case 0:
3                  // Fall z==0
4                  break;
5          case 1:
6          case 2:
7                  // Fall z==1 || z==2
8                  break;
9          default:
10                 // sonstige Werte
11                 break;
12 }
```

Abhängig des Wertes von `z` werden die entsprechenden Fälle aufgerufen. Die `break`-Anweisung in Zeile 4 bewirkt, daß an dieser Stelle die `switch...case`-Umgebung verlassen wird. Umgekehrt steht hinter Zeile 5 kein `break`, sodaß der Fall für `z==1` identisch ist mit dem von `z==2`. Ein beliebter Fehler ist das Vergessen des `break`-Anweisung, was zu einem unerwünschtem Verhalten des Codes führt, das auf den ersten Blick schwer nachvollziehbar erscheint.
Zeile 9 behandelt den Fall, daß keiner der vorherigen Fälle zutrifft.

Da, wie oben erwähnt, `enum`s vom Compiler auf Integer abgebildet werden, lassen sich diese sehr gut mit der `switch...case`-Anweisung behandeln. Der folgende Codeausschnitt demonstriert dies mit Hilfe der `enum`s aus obigen Beispiel.

```
1  switch(zustand) {
2          case AN:
3                  // Motor an
4                  break;
5          case AUS:
6                  // Motor aus
7                  break;
8  }
```

Ein weiterer Vorteil bei der Kombination von `enum`s mit `switch...case` sind spezifische Compiler-Warnungen. Im obigen Beispiel ist kein `default`-Fall vorhanden und es sind nicht alle Fälle aus dem `enum` abgehandelt. Der Compiler wirft daher die Warnung `warning: enumeration value 'DEFEKT' not handled in switch`. Diese Warnung hilft dabei, Fehler von vornherein auszuschließen.

5.4. Aufgabe C.1: Implementierung einer Fußgängerampel

Es soll eine Fußgängerampel programmiert werden. Eine dreifarbige Ampel für den rollenden Verkehr sowie eine zweifarbige Ampel für Fußgänger bestehend aus LEDs sollen angesteuert werden. Mit Hilfe eines Tasters soll die Grün-Phase für die Fußgänger angefordert werden können.

a) Überlegen Sie sich, ob sie hierfür einen Mealy- oder Moore-Automaten benötigen bzw. worin der Unterschied beider liegt. Was sind die Eingaben, was die Ausgaben der Zustände?

b) Schließen Sie nun die fünf LEDs für die Ampel sowie den Taster für die Grünanforderung

an wie sie es in Kapitel 4 gelernt haben. Schreiben Sie für jede Ampel je eine Funktion, die als Parameter die Farbe (codiert als `enum`) enthält, in der diese leuchten soll.

c) Implementieren Sie nun die Zustandslogik mit Hilfe von `enum`s und `switch...case`. Für den zeitabhängigen Zustandswechsel verwenden Sie den ebenfalls aus Kapitel 4.5 bekannten `getMsTimer()`.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

6. D Serielle Schnittstelle

Serielle Schnittstellen finden man an heutigen Computern immer weniger. In der Vergangenheit wurden daran zum Beispiel Mäuse, Analogmodems oder Plotter angeschlossen. Die Verbreitung des USB-Standards hat diese Funktion aber mittlerweile abgelöst. Nichtsdestoweniger spielt diese Schnittstelle weiterhin eine wichtige Rolle, gerade wegen der Einfachheit des Protokolls.

Der RS-232-Standard der seriellen Schnittstellen auf PC-Seite arbeitet mit Signalpegeln von bis zu ±15 V. Die Namen der seriellen Schnittstellen auf PCs sind meist `COM`x`:` unter DOS/Windows bzw. `/dev/ttyS`x unter Linux. USB-Seriell-Wandler heißen meist `/dev/ttyACM`x oder `/dev/ttyUSB`x. Dieser längst historische, aber bei PCs dennoch weiterhin gültige RS-232-Standard für die Serielle Schnittstelle aus den sechziger Jahren wird nicht auf Mikrocontrollern verwendet.

Die serielle Schnittstelle auf Mikrocontrollern folgt meist dem UART-Standard. UART steht dabei für **U**niversal **A**synchronous **R**eceiver **T**ransmitter. Dieser Standard umfaßt einen Spannungsbereich 0 ... 5 V und ist dadurch deutlich besser geeignet. Dieser unterscheidet sich vom RS-232-Standard nicht nur durch den anderen Spannunsbereich sondern auch darin, daß er LOW-Active ist. Möchte man einen Mikrocontroller mit UART and einen PC mit RS-232 anschließen, benötigt man einen Pegelwandler.

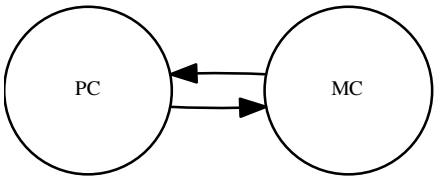


Abbildung 6.1.: Serielle Verbindung

6.1. Funktionsweise

Die serielle Schnittstelle wird asynchron und voll-duplex betrieben. Dazu steht für jede Kommunikationsrichtung je eine Leitung zur Verfügung (siehe Abb. 6.1). Asynchron bedeutet, daß es keinen gemeinsamen Takt für beide Leitungen gibt. Da die Leitungen unabhängig sind, kann gleichzeitig in beide Richtungen versendet und empfangen werden (voll-duplex). Eine dritte Leitung für Ground, die den beiden anderen Leitungen ein gemeinsames elektrisches Bezugspotential liefert, wird ebenfalls benötigt.

Übertragungsform

Die am häufigsten verwendete Übertragungsform wird als `8N1` bezeichnet. Dabei werden pro Paket 8 Nutzbits übertragen, nach einem ersten Bit als Start-Bit. `N` steht für *No Parity*, also keine Parität. Es ist möglich, zur Fehlererkennung ein Paritätsbit mit zu versenden. Dies wird in der Regel eher selten gemacht. Die 1 am Ende steht für 1 Stop-Bit.

Abbildung 6.2 zeigt beispielhaft eine serielle Übertragung im UART-Standard. Der Bus ist LOW-Active, d.h. eine logische 1 entspricht 0 V während 5 V einer logischen 0 ent-

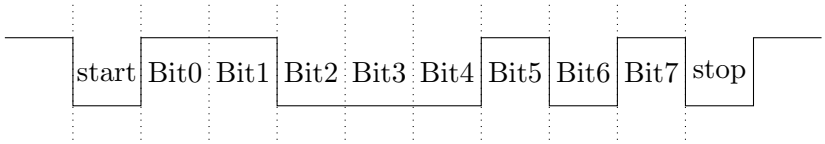


Abbildung 6.2.: Beispiel für eine UART-Übertragung

spricht. Zu sehen ist als erstes das notwendige Startbit. Danach folgen 8 Datenbits, gefolgt von einem Stop-Bit. Das zuerst übertragenen Bit ist dabei das höchstwertige Bit. Die Übertragung für ein Byte Nutzdaten benötigt also so lange, wie die Übertragung von 10 Bits.

Übertragungsraten

Die sogenannte Baud-Rate[1] beschreibt, wie viele Bits auf dem Bus pro Sekunde übertragen werden. Als Default wird oft 9600 als Baudrate verwendet, was genau 960 Bytes an Nutzdaten pro Sekunde entspricht.

Folgende Tabelle zeigt häufig verwendete Werte:

Baud in Bits/s	Nutzdaten in Bytes/s	
9600	960	
38400	3.840	
57200	5.720	(Nicht mit 16 MHz-Quarz)
115200	11.520	(Nicht mit 16 MHz-Quarz)

Die Taktfrequenz für die serielle Schnittstelle wird auf dem Mikrocontroller dadurch erzeugt, daß die vom verbauten Quarz vorgegebene Taktfrequenz (hier 16 MHz) durch einen sogenannten Frequenzteiler heruntergeteilt wird. Näheres zu Frequenzteilern findet sich in Kapitel 8. Es ist an dieser Stelle wichtig zu wissen, daß dieser Teiler nur ganzzahlige Werte annehmen kann. Das bedeutet wiederum, daß nicht jede Baudrate aus 16 MHz exakt eingestellt werden kann. Siehe dazu auch die Tabellen im Datenblatt [4] S. 192ff. Eine kleine Abweichung ist oft tolerierbar, während Baudraten wie 57200 oder 115200 zu große Fehler aufweisen, als daß diese bei der Übertragung zu einem PC verwendet werden könnten. Möchte man dieses Problem umgehen, so kann zum Beispiel ein Quarz mit 18.432 MHz verwendet werden. Dieser Quarz erreicht alle üblichen Baudraten korrekt, kann allerdings an anderer Stelle durch die sehr krumme Zahl zu Komplikationen führen.

6.2. Verbindung zwischen PC und Mikrocontroller

Am Mikrocontroller wird der `RX`-Pin (`D0`) zum Empfangen und der `TX`-Pin (`D1`) zum Senden verwendet. Zum Anschluß an einen PC, wird eine USB-UART-Schnittstelle verwendet. Diese sieht wie ein USB-Kabel mit Pin-Steckverbindern am Ende aus. Die Wandlung des 5 V-UART-Signals in einen emulierten seriellen Port findet durch eine Elektronik statt, die direkt im USB-Stecker des Adapters integriert ist.

[1] Benannt nach Jean-Maurice-Émile Baudot[9]

Beim Anschluß ist darauf zu achten, daß die Leitung überkreuzt werden, d.h. der orangene TXD-Stecker auf den RX-Pin (`PD0`) sowie der gelbe RXD-Stecker auf den TX-Pin (`PD1`) des Controllers gesteckt wird. Außerdem muß der schwarze GND-Stecker mit einem der Ground Pins des Controllerboards verbundern werden. Die weiteren Stecker für VCC, RTS und CTS werden nicht benötigt.

Die durch Einstecken des Wandlers erzeugte USB-Schnittstelle bekommt in der Regel den Gerätenamen `/dev/ttyUSB0`. Sollte diese bereits vorhanden sein, so erhöht sich jeweils die Zahl solange, bis ein Gerätenamen frei ist. Es kann bei einem unsauberen Trennen vorkommen, daß ein Name nicht vergeben wird, obwohl er frei zu sein scheint. Dies kann beispielsweise dann geschehen, wenn ein Programm die Schnittstelle geöffnet hat während man den Adapter abzieht. Steckt man ihn erneut ein, so bekommt er eine neue Nummer. Dies kann zu unerwarteten Fehlern führen und sollte beachtet werden.

6.2.1. Terminalprogramm

Ein Terminalprogramm wird dazu verwendet, Tastatureingaben direkt an die serielle Schnittstelle weitersenden bzw. empfangene Zeichen auf dem Bildschirm wiederzugeben. Das Programm übernimmt ebenfalls die Konfiguration der Schnittstelle wie das Setzen des richtigen Übertragungsmodus sowie der Baudrate.
Unabhängig von der verwendeten Software sind folgende Einstellungen vorzunehmen:

- Baudrate je nach Einstellung in `uart.h` auf dem Controller
  Default: 9600
- 8N1
  - 8 Bit
  - Keine Parität
  - 1 Stop Bit
- Kein Hardware Flow Control
- Kein Software Flow Control

Ein rein Kommandozeilenbasiertes Terminalprogramm ist `minicom`. Die Bedienung ist für maus-verwöhnte Benutzer etwas gewöhnungsbedürftig, bietet aber recht praktisch und kompakt alle Funktionen. Die Konfigurationsdialog beim Start des Programms läßt sich mittels `minicom -s` aufrufen.

Ein grafisches Terminalprogramm ist `gtkterm`. Dort lassen sich alle Einstellungen per Maus vornehmen.

6.3. Aufgabe D.1: Ein- und Ausgabe

a) Wie lautet der Wert des Bytes, das in Abbildung 6.2 übertragen wird?

b) Schreiben Sie ein Programm, das die Zeile „*Hallo Welt!*“ einmal die Sekunde über die serielle Schnittstelle ausgibt. Verwenden Sie dazu die `uart_puts(`s`)`-Funktion des Basiscodes. Die Default-Baudrate von 9600 behalten Sie einfach bei. Verbinden Sie sich mit einem Terminal-Programm auf die Schnittstelle und sehen Sie sich die

Ausgaben an.
Einen Zeilenumbruch erzeugt man durch Senden des Strings `"\n\r"`. Diese Konvention geht noch auf die Zeit zurück, als man Typenrad- und Nadeldrucker hatte, um seine Ausgabe zu erhalten. `\n` erzeugt einen Zeilenvorschub (*newline*) und `\r` ist ein *carriage return*, also ein Zurückfahren des Druckkopfes auf die Startposition links.

c) Schreiben Sie nun ein einfaches Echo-Programm. Das Zeichen, das der Controller vom PC empfängt, soll zurückgeschickt werden. Verwenden Sie dazu die `uart_putc(`c`)`- bzw. `uart_getc(`c`)`-Funktionen. Letztere ist eine blockierende Funktion, die bis zum Empfangen eines Zeichens wartet. Testen Sie das Programm in Ihrem Terminal. Jedes eingetippte Zeichen sollte nun auf dem Bildschirm erscheinen.

d) `uart_getc(`c`)` empfängt immer nur ein einzelnes Zeichen. Schreiben Sie nun ein Programm, das einen ganzen String einliest und anschließend wieder ausgibt.

In `C` ist ein String ein Array aus `char` in diesem Beispiel von fester Länge. Per Konvention hat das letzte Zeichen eines Strings immer den Wert `0x00` bzw. `\0`. So ist das Ende eines Strings im Speicher erkennbar.

Legen Sie als erstes ein `char`-Array mit fester Länge an. Füllen Sie dieses Array mittels `uart_getc(`c`)`-Aufrufen. Beachten Sie hierbei die Grenzen des Arrays, sowie das abschließende Zeichen `\0` am Ende der Eingabe.

e) Schreiben Sie einen kleinen Taschenrechner mit Integern. Das erste Zeichen, das eingegeben wird, soll das Rechenzeichen (`+-*/`) sein. Danach folgt die Eingabe eines Integers. Mit der Eingabetaste wird die Zahl beendet, die dann mittels `atoi()` (erfordert `stdlib.h`) in einen Integer gewandelt werden kann.

Geben Sie das Ergebnis aus.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

7. E Analog-Digital-Wandler

7.1. Analoge Signale

Digitale Signale haben diskrete, klar unterscheidbare Zustände. Im binären Fall 0 und 1. Ein Signal an einer digitalen Leitung wird deshalb immer als einer dieser Zustände interpretiert. Der Wert ist (von Übertragungsfehlern abgesehen) immer exakt. Analoge Signale zeichnen sich hingegen durch einen Stufen- und unterbrechungsfreien, stetigen Verlauf aus. Es existieren exakte Signale nur insoweit, wie es die Meßgenauigkeit zuläßt.

Um analoge Signale digital zu verarbeiten gibt es die sogenannten **A**nalog **D**igital **C**onverter (ADC). Im verwendeten Mikrocontroller wandelt der ADC eine analoge Spannung im Bereich $0 \ldots 5\,\mathrm{V}$ in einen 10 bit-Integer $(0 \ldots 1023)$ um. Der Bereich der Eingangsspannung läßt sich auch auf einen Wert zwischen $0 \ldots 1.1\,\mathrm{V}$ herabsenken, um so die Auflösung kleinerer Signale zu verbessern. Dies wird in Abschnitt 13 Bei der Programmierung eines Thermometers durchgeführt.

7.1.1. Implementierung

Im verwendeten Atmega168 gibt es 6 Pins, an denen eine analoge Spannung digitalisiert werden kann (`C0 ... C5`). Der Controller besitzt allerdings nur einen einzigen ADC, der dann über einen Multiplexer zwischen diesen sechs Pins umgeschaltet werden kann. Der Vorgang des Digitalisierens eines analogen Signals dauert abhängig von der eingestellten Genauigkeit eine gewisse Zeit.

Die in dem Basiscode implementierte Funktion `getADCValue(`k`)` schaltet den Multiplexer auf den gewünschten Kanal k, löst eine Wandlung aus und wartet auf ihren Abschluß. Der Rückgabewert ist ein 10 bit-Zahl.

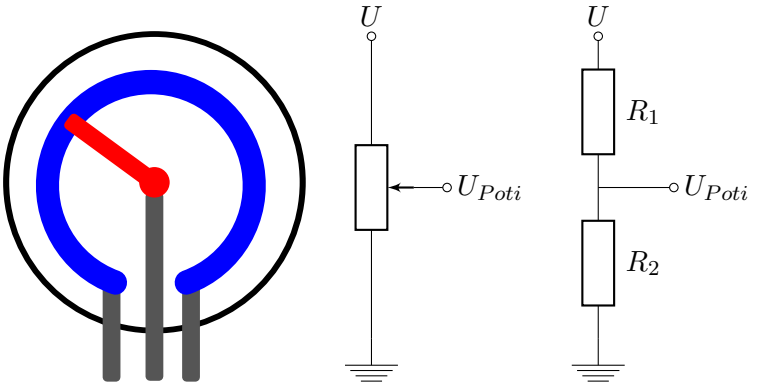
7.1.2. Drehpotentiometer

Ein Potentiometer, oder kurz: Poti, ist ein einstellbarer Widerstand. Am häufigsten werden Drehpotentiometer verwendet. Ein Schleifkontakt bewegt sich dabei auf einem Ring aus Material mit einem gewissen elektrischen Widerstand. Dieser Schleifkontakt kann je nach Ausführung mit einer Achse oder Schraubendreher von oben gedreht werden, sodaß er an einer einstellbaren Stelle die Spannung am Widerstandsmaterial abgreifen kann (Abb. 7.1(a)).

Das Schaltsymbol ist in Abbildung 7.1(b) dargestellt. Betrachtet man nun das Ersatzschaltbild in 7.1(c) mit der Voraussetzung, daß $R_1 + R_2 = R_{Poti}$ ist, so läßt sich ein einfaches Verhältnis feststellen.

Unter Zuhilfenahme des Wissens aus Kapitel 2 folgt hier ohne Beweis

$$\frac{R_2}{R_1 + R_2} = \frac{R_2}{R} = \frac{U_{Poti}}{U} \tag{7.1}$$



(a) Aufbau Poti (b) Schaltsymbol (c) Ersatzschaltbild

Abbildung 7.1.: Potentiometer

Das Verhältnis $\frac{R_2}{R}$ beschreibt, wie weit das Poti gedreht wurde, was dem Verhältnis der Ausgangsspannung zur Eingangsspannung entspricht.
Das Poti verhält sich wie ein einstellbarer *Spannungsteiler*, solange der Ausgang nicht belastet wird, d.h. kein nennenswerter Strom durch ihn fließt.

7.2. Aufgabe E.1: Analoge Werte einlesen

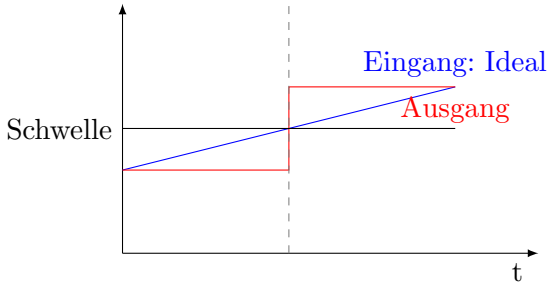
Schließen Sie ein Drehpotentiometer an einen der ADC-Eingänge an. Die beiden äußeren Anschlüsse legen Sie auf die 5 V des Controllers bzw. GND. Am mittleren Pin können sie nun gemäß Gleichung (7.1) eine einstellbare Zwischenspannung abgreifen, die sie auf einen der ADC-Pins des Controllers legen.
Lassen sie sich die ADC-Werte über die serielle Schnittstelle anzeigen.

7.3. Schwellwertschalter (Trigger)

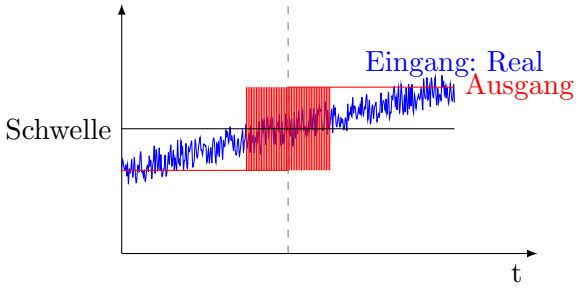
In vielen Anwendung ist es erforderlich, ab einem gewissen Schwellwert eines Analogsignals ein digitales Signal zu schalten. Beispielsweise bei einem Nachtlicht, bei dem beim Unterschreiten einer gewissen Helligkeit das Licht eingeschaltet werden soll. Oder bei einer einfachen Heizung, die sich nicht regulieren läßt, sondern nur ein- und ausschalten.

Bei der Verwendung nur einer Schaltschwelle, die zwischen beiden Ausgangszuständen umschaltet, ergeben sich einige Schwierigkeiten, die im Folgenden dargelegt werden. Wie in Abbildung 7.2(a) erkennbar, schaltet ab einem gewissen Wert des Eingangssignals der Ausgang sauber um.

In der Realität ist aber jede Messung von Störeinflüssen wie dem Rauschen behaftet. Deshalb mißt man nie eine ideale, monotone Kurve. Das führt dazu, daß Schwankungen im Bereich des Schwellwerts den Ausgang mehrfach hin- und her schalten lassen. Dieses Verhalten ist natürlich unerwünscht, da es das Stellglied, also die Heizung, die Lampe etc. beschädigen kann.



(a) Idealer Verlauf

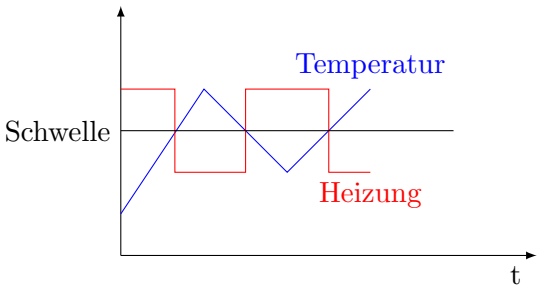


(b) Tatsächlicher Verlauf

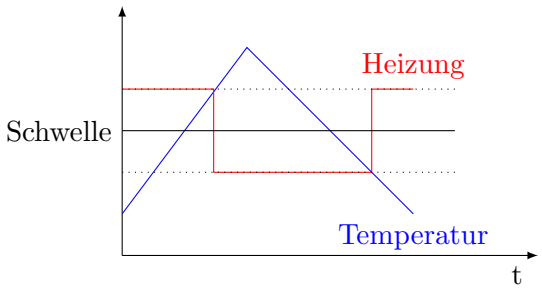
Abbildung 7.2.: Trigger

7.3.1. Zweipunktregelung

Noch deutlich schwerwiegender ist dieser Effekt bei rückgekoppelten Regelungssystemen. Am Beispiel einer Heizung, die nur entweder voll eingeschaltet oder ausgeschaltet werden kann, soll dies im folgenden erläutert werden.



(a) Viele Schaltzyklen



(b) Mit Hysterese

Abbildung 7.3.: Beispiel Heizung

Angenommen der Raum sei sehr kalt und wird nun von der Heizung erwärmt. Erreicht der Raum die Zieltemperatur, schaltet die Heizung ab. Der Heizkörper ist noch etwas warm aber schon bald kühlt sich der Raum wieder auf eine Temperatur kurz unterhalb der Schwelltemperatur ab, woraufhin sich sich die Heizung wieder einschaltet. Umgekehrt schaltet sie sich darauf erneut schnell ab. Es treten unnötig viele Schaltzyklen auf. Dieses Szenario ist in Abbildung 7.3(a) zu sehen.

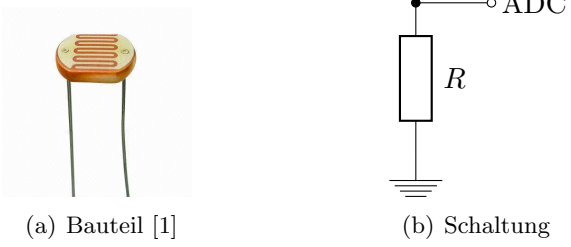
Um das zu vermeiden, ersetzt man den einen Schaltpunkt durch zwei. Diese wählt man idealer Weise etwas oberhalb und etwas unterhalb der eigentlich erwünschten Schwelle. Ein solches Verhalten nennt man *Hysterese*. Dadurch verringern sich die Schaltzyklen. Als Nachteil ist die Regelung allerdings weniger präzise (vgl. Abb. 7.3(b)).

7.3.2. Photo-Widerstand

Ein Photo-Widerstand ist ein elektronisches Bauteil, das sich wie ein Widerstand verhält, dessen Wert mit steigendem Lichteinfall sinkt. Dieses Verhalten basiert auf dem sogenannten Photo-Effekt, für dessen Deutung Albert Einstein 1921 den Nobelpreis erhielt.

Abbildung 7.4(b) zeigt einen Schaltplan, in der ein Photo-Widerstand in einem einem Spannungsteiler eingesetzt wird. Dies ist die übliche Schaltung, in der Photo-Widerstände verwendet werden. R beträgt typischer Weise ca. $1.2\,\mathrm{k}\Omega$.

Es spricht theoretisch nichts dagegen, die Widerstände R und R_{Photo} zu vertauschen. Bei dieser Anordnung allerdings steigt die Spannung am PIN wenn der Lichteinfall größer wird, was intuitiver erscheint.



(a) Bauteil [1] (b) Schaltung

Abbildung 7.4.: Photo-Widerstand

7.4. Aufgabe E.2: Trigger

a) Schließen Sie den Photo-Widerstand gemäß Abbildung 7.4(b) als Widerstandsteiler an. Implementieren Sie einen einfachen Schwellwertschalter (ohne Hysterese), sodaß bei Unterschreiten eines Werts eine LED angeschaltet wird. Durch geschicktes Verschatten des Sensors sollte es nun möglich sein, die LED zum flackern zu bringen.

b) Erweitern Sie nun den Schwellwertschalter um eine Hysterese. Wählen Sie einen geeigneten Wert, sodaß einerseits die LED nicht mehr flackert, andererseits die zwei Schaltpunkte nicht zu weit auseinander sind.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

8. F Pulsweiten-Modulation

8.1. Pulsweitenmodulation (PWM)

Ein PWM-Signal ist ein digitales Rechtecksignal, das sich zeitlich periodisch wiederholt. Das Verhältnis zwischen der Zeit, in der das Signal auf 1 liegt, zur Gesamtperiodendauer nennt man *duty cycle*.

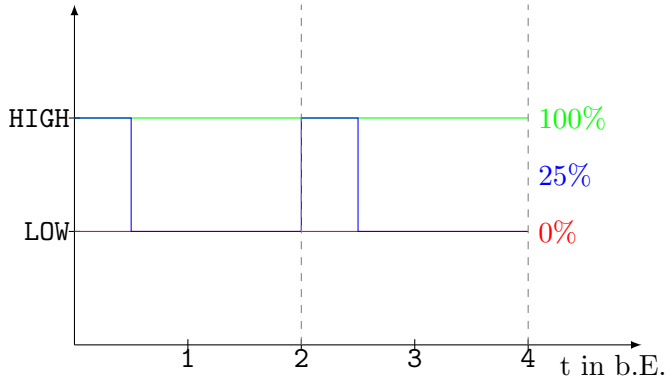


Abbildung 8.1.: Verschiedene Pulsweitensignale

Beispielsweise ist bei einem Signal mit 25 % duty cycle das erste Viertel der Periodendauer auf 1 und drei Viertel auf 0. Danach wiederholt sich das Signal (vgl. Abb. 8.1).

Dieses Signal ist in der Tat ein echtes digitales Signal. Schließt man allerdings ein solches Signal an ein hinreichend träges System an, so kann man davon sprechen, daß sich das System ähnlich wie bei Ansteuerung mit einem analogen Signal verhält. Ein träges System wäre zum Beispiel eine Glühlampe. Schon bei vergleichsweise geringen Frequenzen, d.h. hohen Periodendauern, ist die Glühwendel so träge, daß man die pulsierende Ansteuerung nicht erkennt. Bei einer Leuchtdiode hingegen sind das Auge bzw. die Netzhaut sowie die Nervenleitungen, die trägen Elemente, die das Ein-und Abschalten der LED nicht wahrnehmen. Dieser Effekt ist unter dem Namen *Trägheit des Auges* oder *Persistence of Vision (POV)* bekannt.

8.2. Timer und Counter

Die Atmega-Mikrocontroller besitzten je nach Modell mehrere Zählregister (Counter). Diese Register sind entweder 8 oder 16 bit groß und zählen Ereignisse, die in der Regel vom Systemtakt gegeben werden. Um die Zählrate zu festzulegen, wird ein sogenannter prescaler verwendet, der den Systemtakt herunter teilt. Abbildung 8.2 veranschaulicht, wie beispielsweise aus einem 16 MHz Systemtakt durch einen Frequenzteiler mit Faktor $1/64$

eine Zählfrequenz von 250 KHz wird. Ein 16bit Counter würde in diesem Beispiel dann ungefähr mit 3.8 Hz überlaufen, d.h. alle $(3.8\,\text{Hz})^{-1} \approx 262\,\text{ms}$.

$$\text{Quarz} \xrightarrow[62.5\text{ns}]{16\,\text{MHz}} \text{prescaler: 64} \xrightarrow[4\mu\text{s}]{250\,\text{KHz}} \text{16-bit Counter} \xrightarrow[\approx 262\text{ms}]{\approx 3{,}8\,\text{Hz}} \text{Overflow}$$

Abbildung 8.2.: Frequenzteilung in Countern

Zur Erzeugung eines PWM-Signals werden diese Counter eingesetzt. Sie werden dazu so konfiguriert, daß sie mit einer durch den Prescaler bestimmen Frequenz zählen. Bei Erreichen eines Maximalswerts werden diese dann auf 0 zurückgesetzt. In der Regel ist der Maximalwert einfach die Registergröße, also beispielsweise 255 bei einem 8 bit-Counter. Bei manchen Countern läßt sich der Maximalwert frei bestimmen, ab dem der Counter zurückgesetzt wird. Dieser Wert definiert dann die Frequenz bzw. Periodendauer des PWM-Signals.

Es gibt diverse Modi, in denen ein PWM-Generator verwendet werden kann. Hier sei der Einfachheit halber zuerst der sog. Fast-PWM-Modus erklärt. Ein Vergleichsregister bestimmt den duty cycle: erreicht der Wert des Counters sein Maximum und wird zurückgesetzt, so wird ein Ausgangs-Pin des Mikrocontrollers auf logisch 1 gesetzt. Bei Erreichen des Wertes, der im Vergleichsregister hinterlegt wird, wir der Pin wieder auf 0 gesetzt. Abbildung 8.3 veranschaulicht dies.

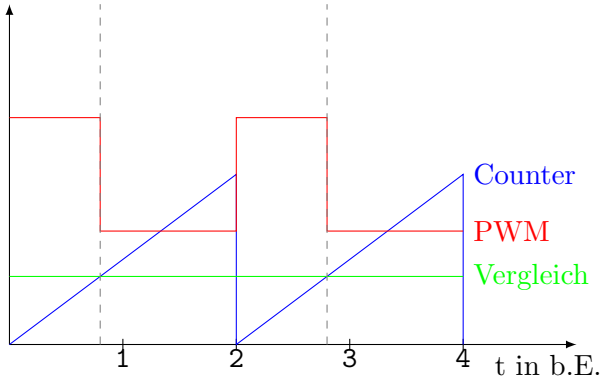


Abbildung 8.3.: Erzeugung eines Fast-PWM-Signals mittels Counter

8.3. Aufgabe F.1: Software PWM

Implementieren Sie zunächst eine Software-Pulsweitenmodulation mit einer 8-bit Variablen als Counter in einer Schleife, sowie den Schaltbedingungen des Ausgabepins. Zur Verzögerung zwischen den Schleifendurchläufen können Sie die `_delay_us(`v`)`-Funktion verwenden, wobei v die Dauer in Microsekunden[1] ist. Die Pulsweite soll die Helligkeit einer LED steuern.

Verwenden Sie ein Drehpotentiometer an einem ADC (siehe Kapitel 7) zur Steuerung der Helligkeit.

[1] $1\,\mu\text{s} = 10^{-6}\,\text{s}$

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

8.4. PWM-Counter im Mikrocontroller

Die Funktionsweise von PWM-Countern in Hardware soll anhand der bereits im Basiscode implementierten `setPWM(`v`)`-Funktion erläutert werden. Folgender Code-Ausschnitt zeigt die Datei `pwm.c` in etwas kompakterer Darstellung:

```
1 void PWMInit() {
2         TCCR0A = (1<<WGM00)|(1<<COM0A1);
3         TCCR0B = (1<<CS01) | (1<<CS00);
4 }
5 void setPWM(uint8_t pwmWert) {
6         OCR0A = pwmWert;
7 }
```

In den Zeilen 2 und 3 konfiguriert die Initialisierungsfunktion die beiden Timer-Counter-Configuration-Register (TCRR) für den Counter mit der Nummer 0.

Als erstes wird dabei das `WGM00`-Bit gesetzt. WGM ist kurz für Waveform Generation Mode. Die erste 0 in der Bezeichnung steht dabei wieder für den Counter0, der hier verwendet wird. Die beiden weiteren Bits `WGM01` und `WGM02` werden nicht gesetzt, sind also auf 0. Wie man im Datenblatt [4] S. 101, Tabelle 13-8 nachlesen kann, bedeutet diese Konfiguration eine Phasenkorrekte PWM. Dies bedeutet, daß der Counter von 0 bis 255 hochzählt und danach wieder bis 0 herunter zählt. Das Verhalten dieser phasenkorrekten PWM unterscheidet sich von dem oben beschriebenen Fast-PWM, wie im folgenden näher beschrieben wird.

Desweiteren wird das `COM0A1`-Bit gesetzt. COM steht hier für Compare Output Match. Diese Bits konfigurieren, wie sich der Pin des Mikrocontrollers bei Erreichen des Werts, der im Vergleichsregisters `OCR0A` gespeichert ist, verhalten soll. In Tabelle 13-4 auf Seite 100 des Datenblatts kann man nachlesen, daß dadurch der Ausgabe-Pin `D5` bei jedem Erreichen des Vergleichsregisters getoggelt (umgeschaltet) wird. Abbildung 8.4 illustriert dieses Verhalten. Der Hauptunterschied zu dem oben beschriebenen Fast-PWM ist die halbe Frequenz bzw. doppelte Periodendauer des Signals, da der Ausgang nicht beim Überlauf sondern nur bei Erreichen des Vergleichsregisters umgeschaltet wird.

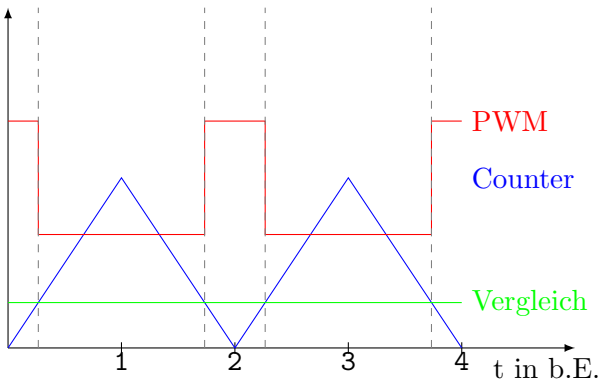


Abbildung 8.4.: Erzeugung eines Phasenkorrekten-PWM-Signals mittels Counter

8.5. Aufgabe F.2: Hardware PWM der Basis

a) Steuern Sie nun eine LED mit der schon vorbereiteten `setPWM(`v`)`-Funktion der Basis an. Verwenden Sie diese Funktion zunächst als Blackbox; die Funktionsweise wird im nächsten Aufgabenteil behandelt. Der Pin, der für die Ausgabe verwendet wird, ist `D6`. Schließen Sie dort die LED an.

b) Erzeugen Sie nun ein pulsierendes Licht, wie es oft als Anzeige Laptops im Standby verwendet wird. Setzen Sie dazu den duty cycle der PWM mit Hilfe einer abwechselnd steigenden und fallenden Rampe.

c) Erweitern Sie die Basis so, daß Sie nun auch eine PWM auf `D5` über `OCR0B` ausgeben können. Ersetzen Sie dabei nicht die bereits implementierte Funktionalität der `setPWM(`v`)`-Funktion. Schließen Sie dort die LED an und lassen diese ebenfalls pulsieren. Dazu ist es nötig, in der Initialisierung entsprechende Bits in der Registern zu setzen, was Sie im Datenblatt nachlesen können.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

8.6. Aufgabe F.3: 10-bit Hardware PWM

a) Implementieren Sie nun selbständig eine PWM-Ausgabe auf dem 10-bit Counter1. Stellen Sie dabei gleich zu Anfang sicher, daß die Servo-Initialisierung in Ihrem Code deaktiviert ist, da diese die gleichen Register verwenden. Setzen Sie dabei die folgende Konfiguration, die Sie dem Datenblatt [4] entnehmen.

- PWM, Phase Correct, 10-bit
- Set OC1A/OC1B on Compare Match when upcounting. Clear OC1A/OC1B on Compare Match when downcounting.
- prescaler 8

Achten Sie dabei sorgfältig darauf, daß Sie die Bits in die korrekten Register schreiben. Die Register, in denen sich die einzelnen Steuerungsbits befinden, sind nicht notwendiger Weise übertragbar von einem Counter auf einen anderen. Ein Bug auf dieser Ebene ist erfahrungsgemäß schwer zu finden.
Am Ende dieser Aufgabe sollten Sie zwei 10-bit PWMs zur Verfügung haben, deren Funktionsfähigkeit Sie über das Pulsieren einer LED überprüfen können. Beachten Sie hierbei, daß die PWM-Pins des Counter1 andere sind als bei Counter0, Sie also die LED umstecken müssen.
Dieser Code wird Ihnen als Grundlage für die Ansteuerung einer RGB-LED in Kapitel 12.3 dienen.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

9. G Datentypen

9.1. Datentypen und nicht-assoziativität

In diesem Abschnitt werden die zur Verfügung stehenden Datentypen erläutert sowie auf interessante Phänomene hingewiesen, die bei der bei der Verwendung dieser auftreten können.

9.1.1. Datentypen

Die Wahl des passenden Datentyps für verwendete Variablen ist ein wichtiger Faktor der Mikrocontroller-Programmierung. Zum einen besitzt der Controller nur einen vergleichsweise kleinen Arbeitsspeicher, sodaß die Größe eine wichtige Rolle spielt. Zum anderen handelt es sich bei den verwendeten Controllern um 8-bit Controller, d.h. die Größe der meisten Prozessorregister beträgt 1 Byte. Größere Datentypen können (mit einigen Ausnahmen) nur in mehreren Taktzyklen berechnet werden. Ein wichtiger Faktor ist auch, daß neben den begrenzten Ressourcen der Controller keine Gleitkomma-Recheneinheit besitzt und diese Berechnungen dann vom Compiler auf Rechnungen mit der Integer-Einheit abgebildet werden. Es empfiehlt sich daher, sofern möglich, auf Gleitkomma-Berechnungen zu verzichten.

Der `avr-gcc` compiler bietet die meisten, aus dem Standard `C`-Repertoire bekannten Datentypen. Dazu gehören Integer verschiedener Größe mit und ohne Vorzeichen, wie sie durch das Einbinden der `<inttypes.h>` zur Verfügung gestellt werden, sowie der Datentyp `float`. Der Datentyp `double` steht nicht zu Verfügung, dieser wird per default auf ein `float` abgebildet. Auch darf man nicht erwarten, daß ein `float` auf dem Mikrocontroller die gleiche Genauigkeit besitzt wie sein Pendant auf dem PC. Eine Übersicht über die Datentypen zeigt Tabelle 9.1.

`char`	signed integer	1 Byte
`int`	signed integer	2 Bytes
`float`	Gleitkommazahl	4 Bytes
`double`	wie `float`	4 Bytes
`uint8_t`	unsigned integer	1 Byte
`int8_t`	signed integer	1 Byte
`uint16_t`	unsigned integer	2 Byte
`int16_t`	signed integer	2 Byte
`uint32_t`	unsigned integer	4 Byte
`int32_t`	signed integer	4 Byte

Details: https://gcc.gnu.org/wiki/avr-gcc

Tabelle 9.1.: Datentypen in `avr-gcc` sowie `inttypes.h`

9.1.2. Assoziativität

Aus mathematischer Sicht sind die Addition ($+$) und die Multiplikation ($\cdot$) in $\mathbb{R}$ assoziativ, es gilt

$$(a+b)+c = a+(b+c) \tag{9.1}$$

sowie

$$\begin{aligned} (a \cdot b) \cdot c &= a \cdot (b \cdot c) \\ \left(\frac{1}{d} \cdot \frac{1}{e}\right) \cdot \frac{1}{f} &= \frac{1}{d} \cdot \left(\frac{1}{e} \cdot \frac{1}{f}\right) \end{aligned} \tag{9.2}$$

Dies bei der Verwendung von Integern sowie Gleitkomma-Zahlen mit endlicher Genauigkeit nicht mehr der Fall.

Gleitkommazahlen

Da jeder `float` eine endliche Genauigkeit hat, kann es bei der Addition zweier Zahlen unterschiedlicher Größenordnungen dazu kommen, daß die Genauigkeit der größeren Zahl die der kleineren nicht enthält.

Beispiel:

$$\begin{aligned} \left(10^{20} + (-10^{20})\right) + 1 &\neq 10^{20} + \left((-10^{20}) + 1\right) \\ 0 + 1 &\neq 10^{20} + (-10^{20}) \\ 1 &\neq 0 \end{aligned}$$

Integer

Die Multiplikation bzw. Division von Ganzzahlen kann durch zwei Effekte beeinflußt werden. Zum einen kann der Wertebereich durch die Multiplikation überschritten (Überlauf) werden. Zum anderen wird bei einer Division immer auf die nächstkleinere ganz Zahl gerundet, d.h. die Nachkommastelle abgeschnitten.

Beispiel:

$$\begin{aligned} \left(5 \cdot 3\right) \div 2 &\neq 5 \cdot \left(3 \div 2\right) \\ 15 \div 2 &\neq 5 \cdot 1 \\ 7 &\neq 5 \end{aligned}$$

Es bietet sich also an, bei längeren Termen mit Integer-Operationen durch Klammern an der Richtigen Stelle die Genauigkeit zu erhöhen. Allgemein kann man als Regeln festhalten, erst zu multiplizieren und dann zu dividieren. Wichtig ist dabei zu beachten, daß bei der Multiplikation mehrerer Zahlen der Wertebereich nicht überläuft. Gegebenenfalls ist vorher auf einen größeren Datentyp zu casten.

Casts

Das Umwandeln einer Variable eines Datentyps in einen anderen bezeichnet man als Cast. Dies wird erreicht, indem man den neuen Typ in Klammern vor die zu castende Variable schreibt. (Da wir in `C` und nicht `C++` programmieren, verwenden wir diese Form des Casts.) Schreibt man den Cast nicht explizit hin, so kann der Compiler oft trotzdem implizit casten.

Beispiel:

```
1 uint16_t a = 42;
2 int8_t b = 7;
3 float f=1.;
4 f = (float) a; // expliziter Cast
5 a = b; // impliziter Cast
```

In manchen Fällen sollte man sich aber nicht auf den impliziten Cast verlassen. Der `avr-gcc` hat beispielsweise in manchen Versionen einen Bug, sodaß er `float`s und `integer` nicht immer korrekt implizit casted.
Folgendes Beispiel zeigt die Kombination aus Casts und richtiger Klammersetzung zur genaueren Mittelwertsberechung:

Beispiel:

```
1 int8_t a = 122
2 int8_t b = 96;
3 int8_t c = -2;
4 int8_t average = ( (int16_t) a + (int16_t) b + (int16_t) c )/3;
```

9.2. Aufgabe G.1: Berechnungsdauer bestimmen

a) Schreiben Sie ein Programm, das die Dauer für die Berechnung von $z = (z + 1) \cdot 2$ für die Datentypen `uint8_t`, `uint16_t`, `uint32_t` und `float` mißt.
Gehen Sie dabei so vor, daß diese Rechnung so oft in einer Schleife hintereinander ausgeführt werden, daß die Dauer mittels `getMsTimer()` meßbar wird.

Beachten Sie hierbei, daß der Compiler die Berechnung in der Schleife wegoptimiert (entfernt), sofern das Ergebnis hinterher nicht mehr benötigt wird. Um zu erzwingen, daß die Variable nicht entfernt wird, legen Sie sie als `volatile` an.

Beispiel:

```
1 volatile float z=0.;
```

Geben Sie das Ergebnis über die serielle Schnittstelle aus.

b) Die so ermittelten Zeiten enthalten noch die Dauer, die das Programm benötigt, um die Schleife selbst zu berechnen. Damit diese von der ursprünglichen Zeit abgezogen

werden kann, muß die Dauer einer Schleife ohne der Berechnung von z ermittelt werden. Da der Compiler aber leere Schleifen wegoptimiert, kann man in die Schleife einen Assembler-Befehl schreiben, der den Prozessor anweist, einen Takt nichts zu unternehmen. `nop` steht dabei für No Operation. Selbstverständlich muß für korrekte Ergebnisse der Zeitmessung dieser Befehl ebenfalls in Schleifen mit Berechnungen aufgenommen werden.

```
1 uint16_t i=0;
2 for (i=0; i < GrosseZahl; i++) {
3         asm volatile ("nop");
4         // Ev. Berechnung
5 }
```

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

c) Interpretieren Sie die Ergebnisse. Wie verhalten sich die korrigierten Rechenzeiten zum Datentyp und warum ist dies so?

10. H Interrupt-Programmierung

10.1. Interrupts

Interrupts sind ein sehr hardwarenahes Konzept der Mikrocontroller-Programmierung. Die Grundidee ist dabei, daß ein Ereignis, meistens ausgelöst durch eine Hardware-Komponente, den normalen Programmablauf unterbricht und der Prozessor in eine spezielle Interrupt Service Routine (ISR) springt. Nachdem diese beendet ist, führt der Prozessor das Programm an der ursprünglichen Stelle fort.

Interrupts ermöglichen das asynchrone Abarbeiten von Ereignissen, ohne daß diese ständig gepollt werden müssen.
Beim Einsatz von Interrupts müssen folgende Dinge beachtet werden:

- Eine ISR-Funktion muß korrekt benannt existieren. Ist das nicht der Fall, so startet der Controller einfach beim Auslösen eines Interrupts neu (der Programmzeiger springt an den Anfang)
- Der Interrupt für das gewünschte Ereignis muß im entsprechenden Register aktiviert sein
- Global müssen Interrupts aktiviert sein (`sei()`)
- Auf Datenkonsistenz muß geachtet werden, ähnlich zur Multithread-Programmierung am PC
- Globale Variablen, die zwischen einer ISR und anderen Programmteilen ausgetauscht werden, sollten immer als `volatile` gekennzeichnet werden, damit der Compiler sie nicht wegoptimiert.

Am Besten lassen sich Interrupts an folgendem Beispiel erklären.

Beispiel:
Im Basis-Code wird die Systemzeit über ein Interrupt erzeugt.

```
1        TCNT2  = 0x00;
2        TCCR2A = 1<<WGM21;
3        TCCR2B = (1<<CS22) | (1<<CS20);
4        OCR2A = OCR2A_VAL;
5        TIMSK2  |= 1<<OCIE2A;
6        sei();
```

In der Initialisierung des Timers, werden in den Zeilen 1 - 4 Konfigurationen am Counter vorgenommen, sodaß der Vergleichswert alle 1 ms erreicht wird. Schlagen Sie Details dazu im Datenblatt [4] bzw. in Kapitel 8 nach. In Zeile 5 wird der Interupt aktiviert, der bei jedem Erreichen des Vergleichswerts `OCR2A` ausgelöst wird. Der Befehl in Zeile 6 aktiviert die globale Interrupt-Behandlung.

Bei jedem Auslösen des Interrupts, springt der Prozessor in folgende ISR:

```
1 ISR (TIMER2_COMPA_vect) {
2         ms_timer++;
3 }
```

Der in Zeile 1 gezeigte Parameter des ISR-Defines bezeichnet das Ereignis, für das diese Routine zuständig ist. In Zeile 2 wird die globale Variable `ms_timer` inkrementiert.

Datenkonsistenz

Damit es nicht zu Dateninkosistenzen kommt, darf im Hauptprogramm nicht auf die Variable `ms_timer` direkt zugegriffen werden. Die Variable ist eine 32-Bit Variable und kann deshalb im 8-Bit Prozessor nicht in einem Takt verarbeitet werden. Würde im Hauptprogramm direkt auf diese Variable zugegriffen werden, so kann es passieren, daß der timer-Interrupt während einer Berechnung mit der Variablen auftritt und sich somit während der Berechnung der Wert ändert. Das Ergebnis wäre somit falsch.

Um dies zu vermeiden wird eine Funktion verwendet, die gesichert eine lokale Kopie anlegt:

```
1 uint32_t getMsTimer() {
2         uint32_t ret;
3         cli();
4         ret = ms_timer;
5         sei();
6         return ret;
7 }
```

In Zeile 3 werden global die Interrupts deaktiviert, in Zeile 5 wieder reaktiviert. Dazwischen springt der Prozessor in keine ISRs. Interrupts, die in dieser Zeit auftreten, werden vorgemerkt. Es gehen also keine Ereignisse verloren, solange nicht das gleiche Ereignis mehrfach auftritt. Um das zu vermeiden, sollte der Bereich zwischen `cli()` und `sei()` wie hier möglichst kurz sein.

10.2. Aufgabe H.1: UART-Interrupt

a) Das einfache Echo-Programm aus Kapitel 6 soll nun in eine Interrupt-Routine verlegt werden. Der Interrupt soll ausgelöst werden, sobald ein Zeichen empfangen wurde.

Hierzu soll der `USART_RX_vect`-Interruptvektor verwendet werden. Der Interrupt-Handler hat also die Form:

```
1 ISR (USART_RX_vect) {
2 }
```

Schreiben Sie dort das Echo-Programm hinein.

Aktivieren Sie anschließend den Interrupt durch Setzen des *RX Complete Interrupt Enable*-Flags (`RXCIE0`) im `UCSR0B`-Register.

b) In diesem Aufgabenteil wird der Datenaustausch zwischen ISR und dem Hauptprogramm implementiert. Ihr Programm soll alle 20 Zeichen, die als Echo zurückgeschickt wurden, einen Zeilenvorschub auslösen. Dies soll allerdings nicht in der ISR sondern im Hauptprogramm stattfinden. Definieren Sie sich eine globale `volatile`-Variable

zum Austausch zwischen ISR und Hauptschleife, die die Anzahl der verschickten Zeichen enthält.

Variablen, die sowohl im Hauptprogramm als auch in einer ISR verwendet werden, sollten prinzipiell als `volatile` gekennzeichnet sein. Dadurch weiß der Compiler, daß sich diese Variable jederzeit ändern kann, was ungewünschte Codeoptimierungen verhindert.

Der Programmteil, der den Zeilenvorschub erzeugt sowie die Austauschvariable zurücksetzt, sollte über `cli()` und `sei()` geschützt werden.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

11. I Servoansteuerung

11.1. Servos

Abbildung 11.1.: Geöffnetes Servo[10]

Servos sind Stellmotoren, die eine Sollposition über ein PWM-Signal übermittelt bekommen. Im Innern eines Servos befindet sich eine Regelelektronik, die über ein Potentiometer die Ist-Position mißt und den Elektromotor so ansteuert, daß dieser auf die Soll-Position fährt (siehe Abbildung 11.1).

Die Dauer des `HIGH`-Signals liegt zwischen $1 - 2\,\text{ms}$, die Periodendauer bei ca. $20\,\text{ms}$. Die zweite Millisekunde des `HIGH`-Signals ist direkt Proportional zum Soll-Winkel des Servos. Mit anderen Worten fährt das Servo bei einem `HIGH`-Signal von $1\,\text{ms}$ auf einen der beiden Anschläge und bei $2\,\text{ms}$ auf den anderen. Die Werte dazwischen entsprechen den Winkelwerten dazwischen (siehe Abbildung 11.2).

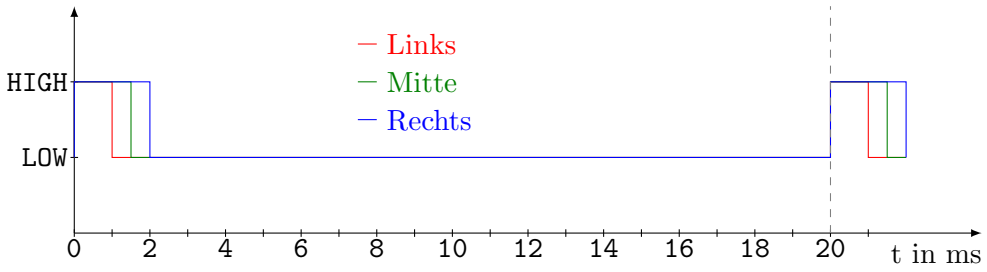


Abbildung 11.2.: PWM-Signal für Servos

Da das Signal nur zum Servo übertragen wird, aber keine Daten zurück, ist es von außen nicht möglich zu erkennen, ob das Servo seine Zielposition tatsächlich erreicht hat bzw. wo es gerade steht.

11.1.1. Anschluß an die Basis-Platine

Der verwendete Mikrocontroller bietet an `B1` und `B2` die Pulsweitenausgänge von `Timer1`. Diese werden durch den Basis-Code für die Ansteuerung der Servos verwendet.

Beachten Sie, daß die Spannungsversorgung der mittleren Pinleiste bei den Pins `B1` bis `B5` *nicht* wie bei den anderen Pins auf 5 V liegt! Die Spannungsversorgung muß auf dieser Leiste erst angeschlossen werden. Dazu gibt es zwei Möglichkeiten:

- Das Überbrücken der zwei Pins (gelb und rot) zum Beispiel mittels eines Jumpers. So liegt auf der Leiste die gleichen 5 V an wie bei den anderen Pinleisten auch.
- An den Versorgungspin (gelb) kann eine externe Spannungsquele angeschlossen werden. Dies hat vor allem beim Betrieb von Servos zwei Vorteile
  - Servos können mit einer höheren Spannung von modellabhängig 6 V – 7.2 V betrieben werden. Dies erhöht das maximale Drehmoment und die Geschwindigkeit gegenüber der Ansteuerung mit 5 V.
  - Der Hauptgrund ist, daß Servos vergleichsweise hohe Ströme ziehen können, die den Spannungsregler nicht belasten sollen, da sich dieser sonst erwärmt. Auch können beim Anfahren des Servos Stromspitzen entstehen, die von der auf der Basisplatine verbauten Spannungsversorgung nicht abgefangen werden können. Dadurch kann es zu einem Absturz des Controllers kommen.

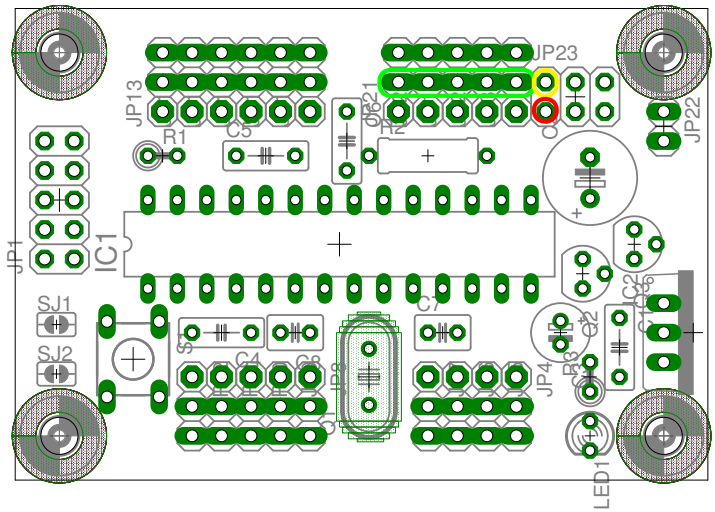


Abbildung 11.3.: Leiste (grün), Versorgungspin (gelb), 5 V (rot)

11.2. Aufgabe I.1: Servo ansteuern

a) Schließen Sie ein Servo an `B1` an. Die braune oder schwarze Ader im Servokabel zeigt Ground an und muß folglich von der Platinenmitte wegzeigen. Schließen Sie

den Versorgungspin (gelb in Abb. 11.3) direkt an Ihr Netzteil an, das sie vorher auf 6 V einstellen.

Aktivieren Sie die Servo-Initialisierung, indem die das Kommentarzeichen vor dem Aufruf der `servoInit()`-Funktion entfernen. Steuern Sie nun das Servo mit Hilfe der Funktion `setServo(`*nr*`,`*pos*`)` an. *nr* ist in diesem Fall 0, da der Servo an `B1` angeschlossen ist. *pos* bewegt sich im Bereich von $1000 - 2000$ und beschreibt somit die Länge des Pulses in μs (vgl. Abb. 11.2).

Lassen Sie zur Übung das Servo zwischen zwei Positionen hin und her fahren, analog zum Blinken der LED. Beachten Sie hierbei, daß das Servo im Winkelbereich durch mechanische Anschläge begrenzt ist. Sollte ein Steuersignal über diesen Anschlag hinaus gehen, so erwärmt sich das Servo relativ schnell und kann durchbrennen. Ein blockieren des Servos über längere Zeit sollte vermieden werden.

b) Das Servo fährt seine Endposition immer mit maximaler Geschwindigkeit an, die stark von der Last des Servos abhängt. Zur gezielteren Steuerung verändert man den Soll-Wert des Servos nicht sprunghaft wie im vorherigen Abschnitt. Stattdessen ändert man den an das Servo übermittelten Wert kontinuierlich, wodurch das Servo den Positionsvorgaben langsam folgt. Dabei wählt man die Servogeschwindigkeit, das heißt die Positionsinkremente pro Zeit, derart, daß das Servo bei jeder angenommenen Last noch ausreichend den Vorgaben folgen kann. Die Abschätzung ist in der Regel rein experimentell zu bestimmen.

   Lassen Sie das Servo wie zuvor zwischen zwei Positionen wechseln, nur diesmal mit langsamerer Geschwindigkeit, indem sich in einer Schleife der Wert des Servos linear ändert.

c) Die im vorherigen Aufgabenteil implementierte Funktionalität, des gezielten Fahrens soll nun nebenläufig in Form einer Interrupt-Funktion implementiert werden. Es bietet sich an, den Überlauf des `Timer1` dafür zu verwenden, da ohnehin nur eine Änderung der Pulsdauer zu Beginn einer neuen Periode sinnvoll ist. Gehen Sie dabei wie folgt vor:

   - Legen Sie zwei globale `volatile`-Variablen an, eine für die Zielposition und eine für das Inkrement. Diese Variablen dienen dem Austausch zwischen der Interrupt-Routine und dem Hauptprogramm.
   - Legen Sie eine ISR-Funktion mit dem Vektor `TIMER1_OVF_vect` an. Dafür ist die `avr/interrupt.h` einzubinden.
   - Aktivieren Sie den Interrupt nach der Initialisierung der Servos, indem das `TOIE1`-Bit im `TIMSK1`-Register gesetzt wird.
   - Berechnen Sie in der ISR die Soll-Position des Servos anhand des Inkrements und der Zielposition. (Speichern Sie dazu ebenfalls die aktuelle Soll-Position, also den Wert, der `setServo()` übergeben wird.) Durch ein sprunghaftes Umsetzen des Soll-Wertes in der Hauptschleife bewegt sich das Servo nun trotzdem langsam zur Zielposition.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

12. J (Projekt) RGB-LED und -Sensor

Dieses Projekt besteht aus zwei Teilen. Im ersten Teil wird eine dreifarbige LED angesteuert. Im zweiten Teil wird mit Hilfe dieser LED und einem Phototransistor ein Farbsensor gebaut. In beiden Teilen wird die Konvertierung zwischen dem RGB und dem HSV-Farbraum implementiert.

12.1. Farbräume

Der am häufigsten verwendete Farbraum ist der RGB-Farbraum. Die Farben rot, grün und blau werden zu unterschiedlichen Anteilen zusammen gemischt, um eine beliebige (sichtbare) Farbe zu erhalten. Licht, das aus allen drei Fraben in gleichem Anteil besteht, wird als weiß wahrgenommen. Diese Form der Mischung nennt man additive Farbmischung. So besteht bei den meisten Bildschirmen jeder Pixel aus drei Subpixeln, in eben diesen drei Grundfarben.

Drucker arbeiten meist im CMY(K)-Farbraum, bei dem die Farbe aus den Grundfarben *C*yan, *M*agenta und Gelb (*Y*ellow) gemischt werden. Alle drei Farbpigmente zusammen ergeben schwarz, es handelt sich also um subtraktive Farbmischung. Damit in Druckern aber Schwarz nicht aus den drei Grundfarben gemischt werden muß, gibt es meist noch schwarze Farbe (Blac*K*) einzeln.

Diese Farbräume sind für Aufgaben wie beispielsweise die Bilderkennung eher ungeeignet. Dort verwendet man bevorzugt Farbräume, in denen die Farbinformation nicht durch die Farbanteile repräsentiert werden. Im folgenden wird der HSV-Farbraum erläutert, der benutzt werden soll, um später eine dreifarbige LED anzusteuern.

In diesem Farbraum wird eine Farbe durch Farbwert (Hue), Farbsättigung (Saturation) und den Hellwert (Value) definiert.

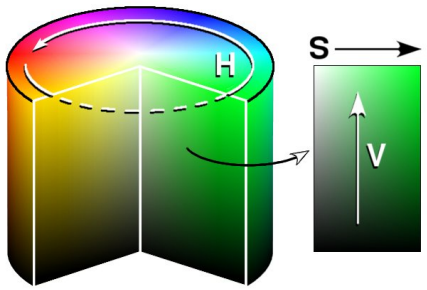


Abbildung 12.1.: HSV-Farbraum veranschaulicht [2]

Abbildung 12.1 zeigt die Darstellung des HSV-Farbraums als Zylinder. Entlang der radialen Achse ist der Sättigungswert S aufgetragen. Er erstreckt sich von farblosem Grau

bis zur reinen Farbe. Entlang der Hoch-Achse ist der V-Wert aufgetragen, der dunkel (Schwarz) bis zu hell (Weiß/Farbe) abträgt. Entlang des Winkels ist der Farbwert Hue abgetragen, den man gelegentlich auch als Farbwinkel bezeichnet. Analog zu einem geometrischen Winkel weist dieser Wert auch eine Periodizität auf, welche allerdings implementationsabhängig ist. Wir definieren die Bereiche wie folgt:

$$H \in [0°; 360°], S \in [0; 1], V \in [0; 1] \tag{12.1}$$
$$R \in [0; 1], G \in [0; 1], B \in [0; 1] \tag{12.2}$$
$$\tag{12.3}$$

HSV→RGB

Die Umrechnung einer Farbe aus dem HSV-Raum in den RGB-Raum berechnet sich wie folgt:

$$h_\mathrm{i} := \left\lfloor \frac{H}{60°} \right\rfloor \tag{12.4}$$
$$f := \left(\frac{H}{60°} - h_\mathrm{i} \right) \tag{12.5}$$
$$p := V \cdot (1 - S); \quad q := V \cdot (1 - S \cdot f); \quad t := V \cdot (1 - S \cdot (1 - f)) \tag{12.6}$$
$$(R, G, B) := \begin{cases} (V, t, p), & \text{falls } h_\mathrm{i} \in \{0, 6\} \\ (q, V, p), & \text{falls } h_\mathrm{i} = 1 \\ (p, V, t), & \text{falls } h_\mathrm{i} = 2 \\ (p, q, V), & \text{falls } h_\mathrm{i} = 3 \\ (t, p, V), & \text{falls } h_\mathrm{i} = 4 \\ (V, p, q), & \text{falls } h_\mathrm{i} = 5 \end{cases} \tag{12.7}$$

Die in Gleichung (12.4) verwendete Gauß-Klammer ($\lfloor\rfloor$) bedeutet ein ganzzahliges Abrunden entsprechend einer Konvertierung zu einem Integer.

RGB→HSV

Die umgekehrte Transformation von RGB nach HSV erfolgt nach folgendem Algorithmus:

$$MAX := \max(R, G, B),\ MIN := \min(R, G, B) \tag{12.8}$$
$$\tag{12.9}$$

$$H := \begin{cases} 0, & \text{falls } MAX = MIN \Leftrightarrow R = G = B \\ 60° \cdot \left(0 + \frac{G-B}{MAX-MIN}\right), & \text{falls } MAX = R \\ 60° \cdot \left(2 + \frac{B-R}{MAX-MIN}\right), & \text{falls } MAX = G \\ 60° \cdot \left(4 + \frac{R-G}{MAX-MIN}\right), & \text{falls } MAX = B \end{cases} \tag{12.10}$$

$$\text{falls } H < 0^\circ \text{ dann } H := H + 360^\circ \tag{12.11}$$

$$S := \begin{cases} 0, & \text{falls } MAX = 0 \Leftrightarrow R = G = B = 0 \\ \frac{MAX-MIN}{MAX}, & \text{sonst} \end{cases} \tag{12.12}$$

$$V := MAX \tag{12.13}$$

$$\tag{12.14}$$

Zu Beachten sind die Spezialfälle von Grau ($R = G = B$) bei dem per Definition $H := 0$ ist. Zusätzlich definiert man bei Schwarz ($R = G = B = 0$) $S := 0$.

12.2. Aufgabe J.1: HSV-Spezialisierung

a) Die LED soll später mit voller Helligkeit und Sättigung angesteuert werden. Vereinfachen Sie die Umrechnung von HSV nach RGB für den Spezialfall von $S = 1$ und $V = 1$.

Abbildung 12.2.: Farbskala für Hue [3]

12.3. RGB-LED

Eine sogenannte RGB-LED vereint in einem Gehäuse drei LEDs mit den Grundfarben rot, grün und blau. Sie besitzt in der Regel 4 Pins, davon sind drei für die einzelnen LEDs und ein Pin allen gemeinsam. Man unterscheidet zwischen LEDs mit gemeinsamer Anode (+) und LEDs mit gemeinsamer Kathode (−). Der gemeinsame Pin ist in der Regel durch eine unterschiedliche Länge im Vergleich zu den anderen gekennzeichnet. Abbildung 12.3 zeigt die Verschaltung der verwendeten Variante mit gemeinsamer Kathode.

12.4. Aufgabe J.2: Ansteuerung

a) Schließen Sie die LED an ihren Mikrocontroller an. Verwenden Sie dabei die zwei 10-bit PWM-Ausgänge für die Farben grün und blau und einen 8-Bit-PWM für rot. Der Grund dafür wird in den nächsten Schritten ersichtlich. Den gemeinsamen Pin, d.h. die gemeinsame Kathode (−) legen Sie auf GND.

b) Schreiben Sie eine Funktion, die mit Hilfe der drei PWM-Ausgängen die Helligkeiten der drei Farben setzt. Verwenden Sie dazu bitte folgendes Template:

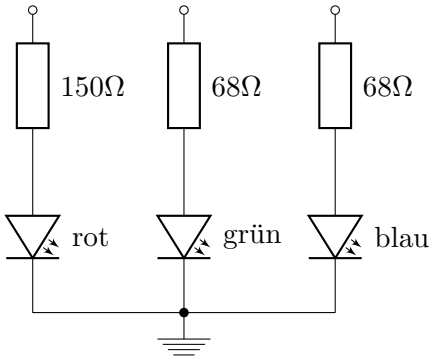


Abbildung 12.3.: Schaltung der RGB-LEDs

```
1 void setRGB(uint8_t r, uint8_t g , uint8_t b) {}
```

Da die Variablen `r,g,b` 8-bit Länge haben sollen, müssen für die 16 bit-PWMs die Werte entsprechend hochskaliert werden.

c) Die drei verbauten LEDs besitzen jeweils eine unterschiedliche Leuchtkraft. Damit bei `setRGB(0xff,0xff,0xff)` ungefähr ein Weißton entsteht, müssen die grüne sowie vor allem die blaue LED etwas gedimmt werden. Skalieren Sie dazu die Eingangswerte bei der Weitergabe an die PWM-Erzeuger herunter. Führen Sie diese Skalierung nicht mit Gleitkommazahlen durch, sondern durch ein geschicktes Multiplizieren und Teilen mit Ganzzahlen.

Warum war es sinnvoll, Rot auf den 8-bit-PWM zu legen? Die Farbe der LED ist sehr abhängig vom Betrachtungswinkel. Verwenden Sie nicht zu viel Zeit darauf, den perfekten Weißton zu finden. Ein Papiertaschentuch erweist sich als geeigneter Diffusor.

d) Implementieren Sie die oben berechnete Konvertierung von HSV nach RGB für den Spezialfall $S = 1$ und $V = 1$. Lassen Sie den H-Wert in einer Schleife kontinuierlich durchlaufen und setzen Sie dabei die RGB-LED auf den berechneten Farbton.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

12.5. Farbsensor

Nun soll mit Hilfe einer RGB-LED und einem Phototransistor ein Farbsensor gebaut werden. Die dafür vorgesehene Platine vereint bereits beide Komponenten mit einem zusätzlichen Schutz vor Fremdlicht. Die LED wechselt zwischen den vier Zuständen (aus, rot, grün und blau). Dabei wird jeweils die Helligkeit des Lichts, das von dem zu untersuchenden Objekts zurückgeworfenen wird, mittels Phototransistor und ADC gemessen. Aus den drei Komponenten des Lichts, die um das Umgebungslicht korrigiert werden, läßt sich über eine Transformation in den HSV-Raum die Farbe bestimmen.

12.6. Aufgabe J.3: Farbsensor

a) **Anschluß** Die RGB-LED des Sensors kann an beliebige Pins des Controllers angeschlossen werden, da hier keine Pulsweitenmodulation eingesetzt wird. Es gibt zwei Stecker mit jeweils drei Polen. Der Stecker mit dem braunen Kabel in der Mitte und dem roten Kabel außen ist der Stecker für die RGB-LED. Alle drei Leitungen dieses Steckers gehen zu den Anoden (+) der drei Farben.

Der zweite Stecker in Standardkonfiguration (rot in der Mitte, braun außen) ist für die Spannungsversorgung und den Sensor. Er wird auf einen ADC-Pin gesteckt, schwarz gehört (`GND`) dabei nach außen.

b) **RAW-Werte** Der Controller soll nun alle vier Zustände (aus, rot, grün und blau) nacheinander durchschalten und die Helligkeitswerte messen. Zwischen Wechsel der LED-Farbe und Messung sollte ein kleiner Delay eingebaut werden. Obwohl die Elektronik sehr schnell ist, ist so das gemessene Signal auf jeden Fall stabil.

Lassen Sie sich alle vier Werte über die serielle Schnittstelle ausgeben.

Damit die Werte reproduzierbar sind, sollte der Sensor direkt auf dem zu messenden Papier aufgelegt werden. Verwenden Sie als Referenz die Farbtabelle in Anhang A.1.

c) **Korrektur und Normierung** Obwohl durch das Gehäuse Fremdlicht nahezu vollkommen abgehalten wird, sollte es trotzdem berücksichtigt werden. Die Messung mit abgeschalteter LED liefert einen Wert für das Fremdlicht.

Betrachten Sie die Werte für Weiß. Durch die unterschiedlichen Leuchtstärken der LEDs sowie die nicht konstante spektrale Empfindlichkeit des Phototransistors (siehe Datenblatt) sind die Meßwerte der drei Farben nicht gleich. Für die Wandlung in den HSV-Farbraum sollen die Farbwerte für rot, grün und blau auf das Intervall $[0:1]$ normiert werden. Da die Wandlung nicht zeitkritisch ist, können alle Berechnung als Gleitkommazahlen durchgeführt werden.

Korrigieren Sie die gemessenen RGB-Werte um das Fremdlicht und normieren Sie diese anschließend.

d) **HSV-Konvertierung** Implementieren Sie nach Abschnitt 12.1 die Wandlung der normierten RGB-Werte in den HSV-Farbraum. Beachten Sie dabei die genannten Spezialfälle.

e) **Farb-Bestimmung** Ihr Programm soll nun die acht auf dem Blatt (A.1) abgedruckten Farben unterscheiden können und diese dann über die serielle Schnittstelle ausgeben. Diese Farben sind: Weiß, Grau/Schwarz, Rot, Grün, Blau, Cyan, Magenta, Gelb. Zwischen Grau und Schwarz ist eine Unterscheidung kaum möglich, da selbst ein schwarzer Druck immer noch ein Licht reflektiert.

Theoretisch lassen sich die bunten Farben durch eine Unterteilung des Hue-Werts in sechs äquidistante Intervalle bestimmen. Warum klappt das hier nicht so ganz?

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

13. K (Projekt) Thermometer

In dieser Übung soll ein Thermometer gebaut werden, das die Temperatur in °C misst und über zwei Siebensegmentanzeigen ausgibt.

13.1. Temperatursensor

Der verwendete Sensor ist ein LM35-Sensor im TO-92–Gehäuse. Dieser Sensor ist direkt in °C kalibriert und liefert ein Analogsignal proportional zur Temperatur. Es gilt dabei der Zusammenhang zwischen Spannung U und Temperatur T:

$$U = \frac{10\,\mathrm{mV}}{^\circ\mathrm{C}} T \tag{13.1}$$

Laut Datenblatt [5] ist der Sensor für einen Temperaturbereich von maximal 150 °C ausgelegt. Dies entspricht einer Spannung von 1.5 V. Bei Verwendung einer Referenzspannung von 5 V für den 10-bit ADC ergäbe sich so eine Auflösung von

$$\frac{5\,\mathrm{V}}{1023} \cdot \frac{1\,^\circ\mathrm{C}}{10\,\mathrm{mV}} \approx 0.5\,^\circ\mathrm{C} \tag{13.2}$$

Realistisch muß man aber von einer Meßgenauigkeit des ADC von ± 2 LSB[1] ausgehen, sodaß man realistischer Weise maximal auf ±2.0 °C genau messen kann.

Um die Genauigkeit zu verbessern, kann der ADC auf eine interne Referenzspannung von 1.1 V umgeschaltet werden. Die 10 bit des ADC sind damit auf einen kleineren Spannungsbereich verteilt. Als Einschränkungen können jetzt Temperaturen von maximal 110 °C gemessen werden.

13.2. Aufgabe K.1: Temperatur auslesen

a) Schließen Sie den Sensor gemäß Datenblatt [5] an. V_{out} wird dabei an einen der ADC-Pins des Microcontrollers angeschlossen. Vergewissern Sie sich, daß der Sensor funktioniert, in dem Sie die ADC-Werte über die serielle Schnittstelle herausschreiben.

b) Setzen Sie nun die ADC Referenzspannung auf die internen 1.1 V. Dies ist im Datenblatt [4] unter S. 250 beschrieben. Lesen Sie die Werte aus.
Beachten Sie, daß beim Umsetzen der Referenzspannung die ersten ADC-Werte falsch sind. Setzen Sie daher die Referenzspannung direkt bei der Initialisierung des ADC in der Datei `adc.c`, damit dieses Problem nicht auftritt.

[1] Least Significant Bit

c) Um Gleitkommaberechnungen zu vermeiden, berechnen und geben Sie die Temperatur in 0.1 °C aus. Die Werte werden noch stark schwanken, weshalb im nächsten Schritt eine Mittelung durchgeführt wird.

13.3. Gleitender Mittelwert

Betrachte man eine Messreihe mit N Messungen x_i mit $i = 0, 1, \ldots, N$. Der Mittelwert $\bar{x}_j$ der letzten n Messungen berechnet sich dann als

$$\bar{x}_N = \frac{1}{n} \sum_{i=N-n}^{N} x_i \tag{13.3}$$

wobei wir hier $n < N$ annehmen. Diesen Mittelwert über ein Fenster nennt man *gleitenden Mittelwert*.

Die Berechnung von $\bar{x}_N$ anhand der Formulierung in Gleichung (13.3) erfordert in jedem Schritt eine Summation über die letzten x_i. Durch Verwenden des Mittelwerts aus dem vorherigen Schritt $\bar{x}_{N-1}$ läßt sich diese Berechnung vereinfachen:

$$\bar{x}_N = \frac{1}{n}\Big(\sum_{i=N-n-1}^{N-1} x_i - x_{N-n-1} + x_N \Big) = \bar{x}_{N-1} + \frac{1}{n}\big(x_N - x_{N-n-1}\big) \tag{13.4}$$

Zur Erhöhung der Rechengenauigkeit schreibt man oft

$$S_N = S_{N-1} + x_N - x_{N-n-1} \tag{13.5}$$

$$\bar{x}_N = \frac{S_N}{n} \tag{13.6}$$

In jedem Schritt muß nun nur der letzte Messwert abgezogen und der neue Messwert zur letzten Summe S hinzuaddiert werden.

13.4. Aufgabe K.2: Mittellung

a) Erstellen Sie ein Array von fester Größe, um die ADC-Meßwerte x_i zwischenzuspeichern. Verwenden Sie dieses Array als Ringpuffer. Implementieren Sie eine Mittelwertsberechnung nach Gleichungen (13.5) und (13.6).

   Initialisieren Sie die Werte des Arrays vor, damit nicht auf zufälligem Speicherinhalt gerechnet wird. Welcher Effekt tritt auf, wenn die x_i sowie S mit 0 vorinitialisiert werden? Wie müßte besser vorinitialisiert werden?

b) Suchen Sie ein Optimum für n sowie ein Delay zwischen den Messungen, sodaß die Werte nicht mehr allzusehr springen.

13.5. Siebensegmentanzeige

Siebensegmentanzeigen werden im allgemeinen dazu verwendet, Ziffern von 0...9 darzustellen. Unter Verwendung von Groß- und Kleinschreibung lassen sich ebenfalls die Buch-

staben A. . . H anzeigen, womit es möglich wird, ein Nibble (halbes Byte) in hexadezimaler Schreibweise darzustellen.

13.5.1. Verschaltung

Um eine einzelne Siebensegmentanzeige anzusteuern, bedarf es 7 (mit Dezimalpunkt 8) einzeln steuerbarer Pins. Bei zwei Anzeigen wären es demnach bereits 16 usw. Um diese Anzahl möglichst gering zu halten, erfolgt die Ansteuerung mehrere Ziffern oft im Zeitmultiplexverfahren. In Abbildung 13.1 ist eine Schaltung für eine solche Ansteuerung dargestellt.

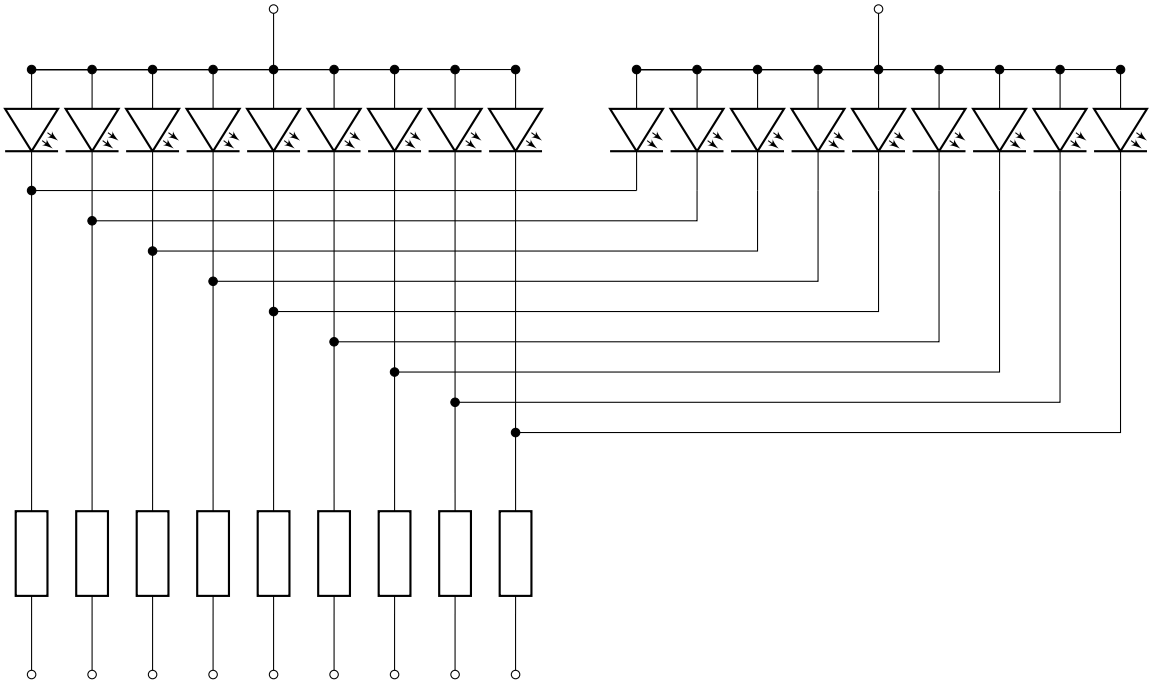


Abbildung 13.1.: Verschaltung der zwei Siebensegmentanzeigen

Die zwei oberen Anschlüsse sind mit der gemeinsamen Anode (+) jeweils einer der Siebensegmentanzeigen verbunden. Mit ihnen lassen sich die Anzeigen getrennt voneinander an- und abschalten. Die unteren 8 Anschlüsse sind mit den Kathoden (−) der einzelnen Segmente verbunden, jeweils die zwei gleichen Segmente auf beiden Anzeigen.

Zum Darstellen von Ziffern werden beide Anzeigen zeitlich schnell wechselnd alternierend umgeschaltet. Die Trägheit des Auges sorgt auch hier dafür, daß es so aussieht, als würden beide Anzeigen gleichzeitig leuchten.

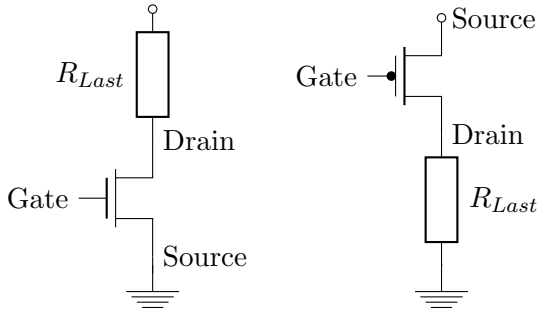
13.5.2. Transistoren

Die zwei gemeinsamen Anoden müssen relativ große Ströme liefern, d.h. jeweils maximal $8 \times 20\,\text{mA}$. Dies ist deutlich mehr, als ein Pin des Mikrocontrollers liefern kann (siehe [4], S. 100). Aus diesem Grund werden hier die zwei Transistoren der Basis-Platine verwendet.

Der Begriff Transistor ist ein Kunstwort und steht für Transfer Resistor, also einstellbarer Widerstand. Im Allgemeinen unterscheidet man zwischen Bipolartransistoren (z. B.

PNP, NPN) und Feldeffekttransistoren. Erstere Schalten mit Hilfe eines kleinen Steuerstroms einen größeren Strom, es handelt sich also um stromgesteuerte Stomquellen. Feldeffekttransistoren (FETs) steuern durch die angelegte Spannung (bzw. genauer: die Ladung auf dem Transistor) ihre Leitfähigkeit. Sehr häufig verwendet man sogenannte MOSFETs (Metal-Oxid-Field-Effect-Transistor).

Im folgenden werden Transistoren in ihrer Eigenschaft als binäre Schalter betrachtet. P-MOS-Transistoren verwendet man, um Lasten nahe der Versogungsspannung zu schalten. N-MOS-Transistoren verwendet man zum Schalten nahe Ground. (Siehe Abb. 13.2). P-MOS-Transistoren sind invertierend.



(a) N-MOS (b) P-MOS

Abbildung 13.2.: MOSFETs als Schalter

Auf der Basis-Platine handelt es sich um P-MOS Transistoren des Typs BS250 mit einem Maximalstrom von jeweils 230 mA. Zwei Pins können als Stromquellen, wie in Abbildung 13.3 gezeigt, durch die Transistoren geschaltet werden. Die Gates („Schalteingänge“) der Transistoren sind mit den Pins `PB1` und `PB2` des Mikrocontrollers verbunden.

13.6. Aufgabe K.3: Siebensegmentanzeige

a) Bringen Sie alle Segmente zum Leuchten. Verbinden Sie dazu die zwei Anoden-Anschlüsse (+) mit 5 V durch Stecken des Steckers auf die mittlere Pinleiste (vgl. Abb. 4.2). Der zweipolige Stecker der Anoden zeichnet sich dadurch aus, daß er im Kabel einen Knoten hat.

   Legen Sie die Kathoden (zweimal dreipolig und einmal zweipolig) auf Ground. Alle 16 Segmente sollten nun leuchten.

b) Legen Sie nun die 8 Kathodenpins auf freie Pins Ihrer Wahl am Controller. Schreiben Sie nun eine Funktion, die die Ziffern 0...9 durch Ansteuerung der betreffenden Segmente ausgibt. Die Anode bleibt vorerst fest auf 5 V, sodaß beide Anzeigen die gleiche Ziffer ausgeben. Beachten Sie, daß die LEDs nur dann leuchten, wenn der Controller-Pin auf Ground liegt (LOW-Active).

c) Schließen Sie nun die Anoden auf die zwei Pins gemäß Abbildung 13.3 an. Durch wechselseitiges Aktivieren von `PB1` und `PB2` lassen sich nun die Anzeigen getrennt Schalten. Beachten Sie, daß die Transistoren invertieren.

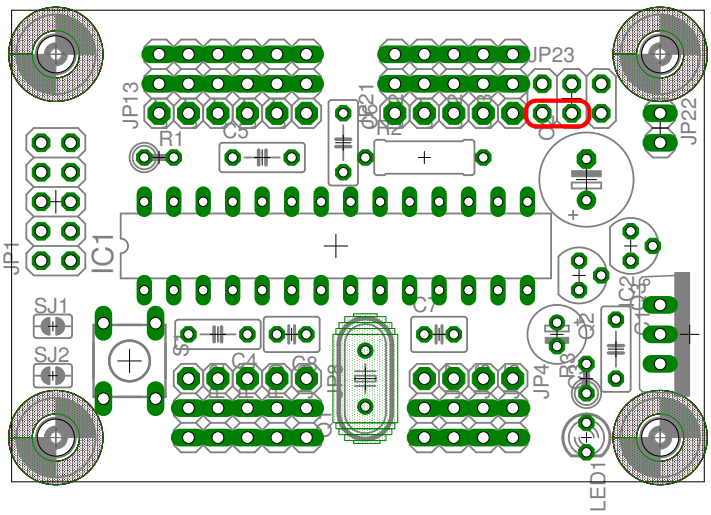


Abbildung 13.3.: Ausgangspins der Transistoren (rot)

Hinweis: Sollten sich die einzelnen Anzeigen nicht ganz abschalten lassen, sondern zwischen hell und mittelhell wechseln, so sind eventuell die BS250 Transistoren in der Basis-Platine falsch herum verlötet. Ziehen Sie zur Korrektur die überarbeitete Lötanleitung zu Rate.

d) Das Umschalten zwischen den Anzeigen und das Ausgeben der in für diesen Zeitabschnitt gültigen Ziffer soll nun in einem Interrupt geschehen. Erweitern Sie dazu den Interruptvektor `TIMER2_COMPA_vect`, der bereits in der `timer.c` definiert ist. Der auszugebende Wert soll wie üblich in einer globalen `volatile` Variable abgelegt werden.

e) Geben Sie nun die aktuelle Temperatur auf den zwei Anzeigen aus. Der Dezimalpunkt der Einer-Stelle soll dabei leuchten.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

14. L (Projekt) Scara-Roboter

In diesem Projekt geht es darum, die Begriffe der (inversen) Kinematik zu verstehen und diese anhand eines konkreten Beispiels einen zweiachsigen Roboters nachzurechnen und anzusteuern.

14.1. Kinematik

Die Kinematik ist der Unterbereich der Mechanik, der sich mit der Beschreibung der Bewegungen von Körpern im Raum beschäftigt. Die Ursachen für die Bewegungen, wie Kräfte oder Drehmomente, werden dabei nicht betrachtet. Die kinematischen Größen sind typischerweise Positionen, Geschwindigkeiten und Beschleunigungen bei translatorischen, bzw. deren Pendants Winkel, Winkelgeschwindigkeiten und Winkelbeschleunigungen bei rotatorischen Bewegungen.

Eine kinematische Kette besteht aus einer Aneinanderreihung von Gelenken mit einem oder mehreren Freiheitsgraden, zwischen denen sich Glieder von bestimmter Länge befinden. Beispielhaft ist dies in Abbildung 14.1 an einem Roboter der Firma Kuka dargestellt.

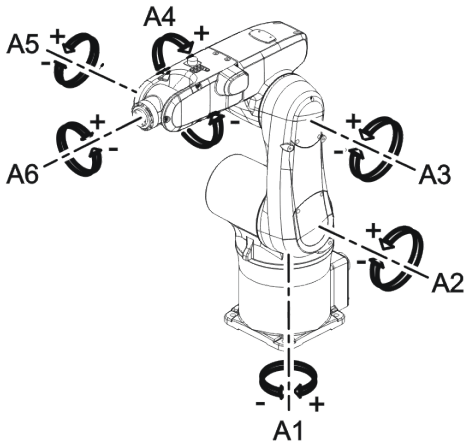


Abbildung 14.1.: Anordnung der Achsen beim KR5 sixx R650 [7] der Firma Kuka.

14.1.1. Vorwärtskinematik

Mit der Kinematik eines Roboter wird häufig die Beschreibung der Position eines ausgezeichneten Punkts am Roboter in Abhängigkeit der Roboterfreiheitsgrade bezeichnet.

Die Freiheitsgrade eines Roboters sind meist Drehgelenke mit der Möglichkeit, entlang einer Achse zu rotieren. Seltener sind Drehgelenke mit zwei oder drei Drehachsen anzutreffen, die dann Kardan- bzw. Kugelgelenk genannt werden. Lineare Freiheitsgrade mit

einer, zwei oder gar drei Bewegungsrichtungen existieren ebenfalls, werden aber eher selten verbaut.

Vereinfacht wird im folgenden ein Roboter mit n Drehachsen angenommen. Zur einfacheren Darstellung seien diese Achswinkel Θ_i zu einem Vektor

$$\vec{\Theta} = (\Theta_1, \Theta_2, \dots \Theta_n)^T \tag{14.1}$$

zusammengefaßt.

Ist $\vec{\Theta}$ bekannt, so läßt sich immer eindeutig der sogenannte *Tool Center Point* (TCP) oder Endeffektorpunkt berechnen. Der TCP ist ein ausgezeichneter Punkt eines Werkzeugs, das am Flansch des Roboters angebracht ist. Dies könnte beispielsweise die Elektrode einer Schweißzange oder die Düse einer Lackieranlage sein.

Obwohl es sich bei dem TCP um einen Punkt im kartesischen Raum mit drei translatorischen Freiheitsgraden (x, y, z) handelt, so ordnet man ihm trotzdem drei Winkel (a, b, c) zu. Diese Winkel beschreiben, wohin ein räumlich ausgedehntes Werkzeug zeigen würde, und geben somit die Orientierung und die Stoßrichtung vor.
Die sechs Freiheitsgrade des TCP werden durch

$$\vec{x} = (x, y, z, a, b, c)^T \tag{14.2}$$

dargestellt. Die Position und Ausrichtung des TCP kann nun als Funktion k der Achswinkel $\vec{a}$ dargestellt werden:

$$\vec{x} = k(\vec{\Theta}) \tag{14.3}$$

Diese sogenannte *Vorwärtskinematik* ist bei bekannten Segmentlängen immer eindeutig, d.h. jede Achswinkelkonfiguration hat genau einen möglichen Endeffektorpunkt. Berechnet wird die Vorwärtskinematik durch abwechselnde Aneinanderreihung von Rotationen um die Achswinkel und Translationen um die jeweiligen Segmentlängen. Dabei muß natürlich beachtet werden, daß Rotationen und Translationen im Allgemeinen nicht kommutieren.

14.1.2. Inverse Kinematik

Die Umkehrung k^{-1}, die so genannte *inverse Kinematik*, ist nicht immer eindeutig. Es können zwischen 0 und unendlich vielen Lösungen für $\vec{\Theta} = k^{-1}(\vec{x})$ existieren, abhängig davon, wie $\vec{x}$ gewählt wird und wie das kinematische System aufgebaut ist.
Man kann zwischen folgenden Anzahlen an Lösungen unterscheiden:

- Keine Lösung: Punkt ist für den Roboter nicht erreichbar
- Eine Lösung: Idealfall, der Punkt kann auf nur eine Weise erreicht werden, es existiert genau ein $\vec{\Theta}$.
- Endlich viele Lösungen: Es gibt eine Mehrdeutigkeit der Lösungen. Alle möglichen $\vec{\Theta}$, die zu $\vec{x}$ führen sind diskret unterscheidbar. Streng genommen ist der Fall mit nur einer Lösung ein Spezialfall von diesem Fall.
- Unendlich viele Lösungen: Die $\vec{\Theta}$ bilden ein Kontinuum. Dies ist meist dann der Fall, wenn zwei Achsen eines Roboters koaxial stehen. Die Drehung der einen Achse kann durch Drehung der anderen revidiert werden.

Abbildung 14.2 zeigt, wie zwei unterschiedliche Achswinkelkonfigurationen zur gleichen Endeffektorposition führen können.

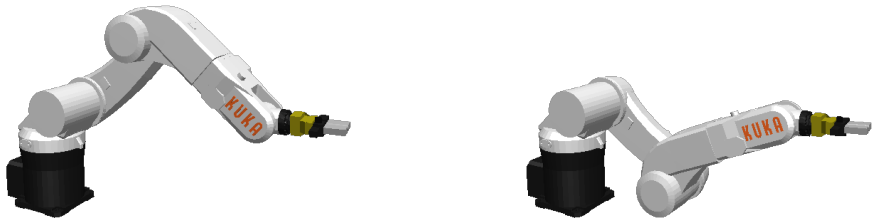


Abbildung 14.2.: Zwei Lösungen am Kuka KR sixx R850

14.2. Scara-Roboter

Der Begriff Scara steht für *Selective Compliance Assembly Robot Arm* oder *Selective Compliance Articulated Robot Arm* und bezeichnet einen Roboteraufbau, wie er in der Industrie vornehmlich für sogenannte Pick-and-Place–Aufgaben eingesetzt wird. Ein Scara-Roboter besitzt zwei parallele, senkrechte Drehachsen.

Abbildung 14.3.: Kuka KR10 SCARA [6]

14.2.1. Kinematik eines 2D-Scara-Roboters

Im folgenden soll die Kinematik eines Scara-Roboters im zweidimensionalen Fall, also ohne Berücksichtigung der Höhe, geometrisch betrachtet werden. Zunächst wird die erste Drehachse mit Winkel Θ_1 in den Ursprung eines kartesischen Koordinatensystems gelegt. Θ_1 beschreibt den Winkel zwischen x-Achse und dem ersten Glied der Länge l_1. Am Ende dieses ersten Glieds befindet sich eine weitere Drehachse mit Θ_2, die den Winkel zwischen dem ersten und dem zweiten Glied beschreibt. Am Ende des zweiten Glieds mit der Länge l_2 befindet sich der Endeffektor mit den kartesischen Koordinaten (x, y).

14.2.2. Vorwärtskinematik

Die Beschreibung des Endeffektorpunkts $\vec{x}$ in Abhängigkeit der zwei Gelenkstellungen Θ_1 und Θ_2 ist vergleichsweise einfach. Dazu genügt es, die Projektion der jeweiligen Glieder auf die x sowie y-Achse zu betrachten.

$$x = l_1 \cos \Theta_1 + l_2 \cos(\Theta_1 + \Theta_2) \tag{14.4}$$
$$y = l_1 \sin \Theta_1 + l_2 \sin(\Theta_1 + \Theta_2) \tag{14.5}$$

14.3. Aufgabe L.1: Inverse Kinematik

Ziel dieser Aufgabe ist es, eine möglichst einfache Formulierung für Θ_1 und Θ_2 in Abhängigkeit des Endeffektorpunkts (x, y) zu erhalten. Im Prinzip könnte man so vorgehen, daß man die Gleichungen (14.4) und (14.5) versucht nach den Winkeln aufzulösen. Man merkt relativ schnell, daß dies trivial nicht möglich ist. Aus diesem Grund ist es hilfreich, den Endeffektor nicht durch kartesische sondern durch Radialkoordinaten (r, φ) zu beschreiben.

$$r^2 = x^2 + y^2 \tag{14.6}$$
$$\tan \varphi = \frac{y}{x} \tag{14.7}$$

Durch Einzeichnen des Radius r unter dem Winkel φ (blau) ergibt sich so ein Dreieck mit den Seitenlängen l_1, l_2 und r. Der Winkel gegenüber l_2 sei mit α und der gegenüber r mit β bezeichnet.

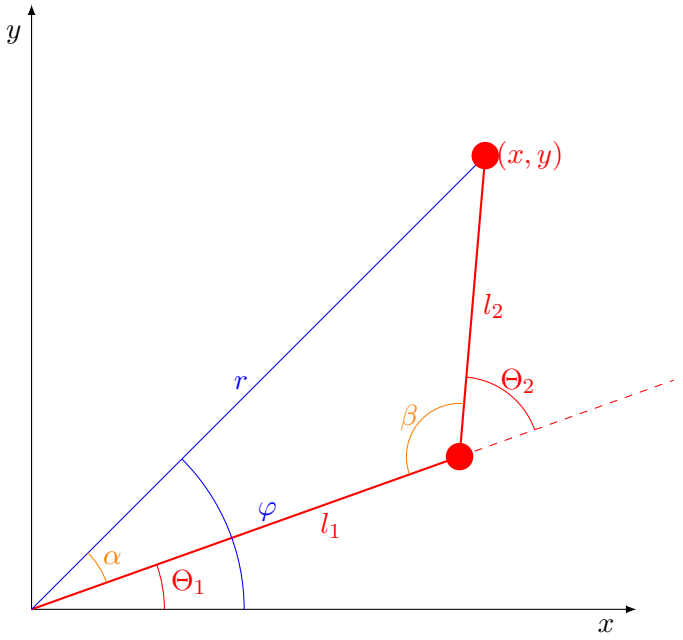


Abbildung 14.4.: Kinematik eines Scara-Roboters

a) Drücken Sie α und β jeweils durch $\Theta_{1,2}$ sowie φ aus.

b) Ist durch die Kongruenzsätze ein Dreieck mit drei bekannten Seitenlängen wie in diesem Fall eindeutig bestimmt? Wie steht dieses Ergebnis mit der Mehrdeutigkeit der inversen Kinematik in Zusammenhang?

c) Nutzen Sie nun den Kosinus-Satz, um α und β anhand der Seitenlängen zu bestimmen. Drücken Sie zuletzt Θ_1 und Θ_2 in Abhängigkeit der Radialkoordinaten und Gliedlängen aus.

d) Betrachten Sie nun die Mehrdeutigkeit der inversen Kinematik. An welchen Stellen der Berechnung gibt es mehr als nur eine Lösung?

14.4. Aufgabe L.2: Ansteuerung

a) **Kalibrierung** Zunächst müssen die Servos kalibriert werden, sodaß diese Positionen in Form von Winkeln im Bogenmaß anfahren. Im Folgenden wird dies beispielhaft für den ersten Servo beschrieben, der Vorgang muß für den zweiten Servo wiederholt werden.

Es ist davon auszugehen, daß der angefahrene Winkel proportional zum PWM-Signal ist. Dazu kommt noch ein Offset, der den Nullpunkt eines jeden Servos festsetzt. Man kann daher den Servo-Winkel α als Funktion des PWM-High-Signals t ausdrücken:

$$\alpha = a \cdot t + b \tag{14.8}$$

Die beiden Parameter a und b lassen sich mit Hilfe zweier Wertepaare (α_1, t_1) sowie (α_2, t_2) berechnen, die später experimentell bestimmt werden. Durch Einsetzen beider Paare in Gleichung (14.8) ergibt sich so ein lineares Gleichungssystem aus zwei Gleichungen. Lösen Sie dieses nach a und b auf, sodaß Sie direkt aus zwei bekannten Wertepaaren die Parameter für die Kalibrierung der Servos berechnen können.

Zur Bestimmung der Wertepaare gehen Sie wie folgt vor:

1. Lassen Sie dazu das Servo auf einen gut bestimmbaren Winkel α_1 (z.B. 0, $\frac{\Pi}{2}$) fahren und notieren Sie sich die dazugehörige PWM-Angabe t_1.

2. Bestimmen Sie auf diese Weise ein zweites Wertepaar (α_2, t_2).
3. Setzen Sie diese Werte in das von Ihnen gelöste LGS ein, um a und b für dieses Servo zu erhalten.

Schreiben Sie nun eine Funktion, die mit Hilfe des Zusammenhangs aus Gleichung (14.8) die Servos nach Winkeln im Bogenmaß ansteuert:

```
1 void setServoAngle(uint8_t nr, float alpha) { }
```

b) **Radialkoordinaten** Implementieren Sie nun die inverse Kinematik für die Radialkoordinaten (r, φ) nach folgender Vorlage

```
1 void gotoRadial(float r, float phi) { }
```

Das Quadrieren einer Variablen geschieht im Gegensatz zur Verwendung der `pow()`-Funktion am effizientesten durch eine Multiplikationen der Variablen mit sich selbst.

Testen Sie diese Funktion und damit die Korrektheit der inversen Kinematik, indem Sie den Endeffektor entlang einer geraden fahren lassen. Da die `setServo`-Funktion

der Basis ungültige Werte ignoriert, kann es geschehen, daß bei einem Fehler mindestens einer der Servos nicht verfährt. Neben einem Fehler in der Implementierung kann auch das Anfahren nicht erreichbarer Werte zu einem Fehler führen. Um dies in der Diagnose auszuschließen, sollten Sie den Endeffektor mit $r = 12 \ldots 20$ sowie $\varphi = \frac{\pi}{2}$ verfahren, da diese Werte bekanntermaßen erreichbar sind.

c) **Kartesische Koordinaten** Zur Ansteuerung des Roboters soll nun die Angabe in kartesischen Koordinaten (x, y) möglich sein. Implementieren Sie dafür folgende Funktion, welche die kartesischen Koordinaten in Radialkoordinaten umwandelt und den Roboter mittels der vorher implementierten `gotoRadial(...)`-Funktion ansteuert.

```
1 void gotoXY(float x, float y) { }
```

Verwenden Sie zur Berechnung von φ die `atan2(`Δy`,`Δx`)` Funktion, die gegenüber dem normalen `atan(`$\frac{\Delta y}{\Delta x}$`)` den Vorteil hat, daß das Ergebnis im richtigen Quadranten liegt.

d) **Ansteuerung** Lassen Sie den Endeffektor nun eine Linie, ein Rechteck und einen Kreis abfahren. Geben Sie dafür die sich in Abhängigkeit zur Zeit ändernden kartesischen Koordinaten vor. Es wird auffallen, daß die Länge des Zeitintervalls für jeden Schritt nach unten hin durch die vergleichsweise lange Rechendauer der inversen Kinematik begrenzt ist. Ebenfalls fällt auf, daß trotz nahezu beliebig kleiner Schritte ein gewisses Ruckeln unvermeidbar ist. Dies liegt an der begrenzten Rechengenauigkeit der Gleitkommaberechnung auf dem Controller (vgl. Kapitel 9). Optimieren Sie die Schrittweite dahingehend, daß die Bewegung möglichst glatt und ruckelfrei aussieht.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

15. M (Projekt) Synthesizer

Dieses Projekts ist umfangreicher und daher freiwillig. Sollten Sie sich für die Bearbeitung entscheiden, so können Sie eines der anderen Projekte auslassen.

In diesem Projekt soll ein Musiksynthesizer gebaut werden. Dieser besteht aus einer analogen Ausgabeeinheit an die Aktiv-Lautsprecher angeschlossen werden können, sowie einer Schnittstelle zur Ansteuerung der Noten. Die Töne selbst werden vorberechnet und im Flash-Speicher des Controllers abgelegt, von wo aus sie dann in Echtzeit ausgegeben werden.

15.1. Aufbau Tonleiter

Im folgenden wird kurz motiviert, wie es aus physikalischer Sicht zum Aufbau der verwendeten Tonleiter kommt.

15.1.1. Tonerzeugung mit einer Saite

Betrachtet man eine schwingende Saite zur Tonerzeugung, die einmal angeregt frei schwingt, so erzeugt diese einen charakteristischen Ton, der von den physikalischen Eigenschaften der Saite und des Resonanzkörpers anhängt. Zunächst wird bei der folgenden Betrachtung die Rolle des Resonanzkörpers vernachlässigt. Die Saite erzeugt ein charakteristisches Spektrum an Schwingungen, das durch die Länge sowie den physikalischen Eigenschaften, wie Zugspannung und Elastizität, der Saite bestimmt ist. Eine jede, in diesem Spektrum vorhandene Schwingung muß der Bedingung unterliegen, daß Anfang und Ende der Saite sich nicht bewegen können, da sie eingespannt sind. Siehe dazu Abb. 15.1. Der Grundton ist dabei die Schwingung mit nur einem Bauch und der tiefsten Frequenz (rot). Die erste Oberschwingung hat zwei Bäuche sowie einen Knotenpunkte in der Mitte der Saite und erzeugt einen Ton mit doppelter Frequenz zum Grundton (blau). Die zweite Oberschwingung hat demnach drei Bäuche und zwei Knotenpunkte (violett) usw. Mit höherer Ordnung der Schwingung nimmt ihre Amplitude ab. Die tatsächlich hörbare Lautstärke der einzelnen Frequenzanteile eines Tons wird maßgeblich vom Klangkörper und dessen Resonanzen bestimmt.

15.1.2. Tonleiter

Wird die Länge der Saite exakt halbiert, zum Beispiel indem man sie wie bei einer Gitarre mit einem Finger herunterdrückt und am Schwingen hindert, so ergibt sich ein neuer Ton. Der Grundton dieses neuen Tons entspricht der ersten Oberschwingung des alten. Somit ist der neue Ton nicht wirklich neu, lediglich hat die Grundschwingung die doppelte Frequenz. Musikalisch betrachtet entspricht die Halbierung der Saite demnach einer Oktave. Je nach

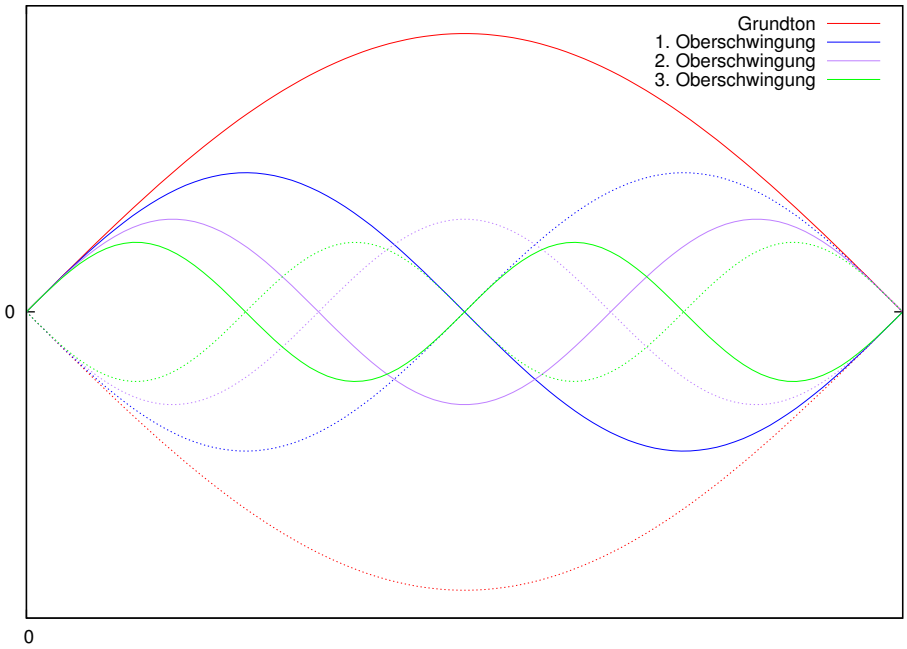


Abbildung 15.1.: Modi einer schwingenden Saite

Stimmung des Instruments ergeben sich dann die Töne zwischen einer Oktave durch Teilen der Saitenlägen in bestimmten Längenverhältnissen.

In diesem Praktikum verwenden wir die gleichstufige oder gleichtemperierte Stimmung, bei der eine Oktave aus 12 Halbtönen besteht, deren Abstände ein gleiches Frequenzverhältnis zueinander haben. Es ergibt sich durch die Aufteilung von 12 Halbtonschritten auf eine Verdoppelung der Frequenz ein Frequenzverhältnis zwischen zwei benachbarten Tönen von

$$\sqrt[12]{2} \approx 1.059463094 \tag{15.1}$$

Die Frequenzverhältnisse untereinander sind somit definiert, es benötigt aber noch einen absoluten Anker, den man meist mit dem Kammerton A von 440 Hz angibt.

Numeriert man nun die Noten mit C beginnend als 0 durch, so ist A Ton Nr. 10. Die Frequenz f des Tons Nr. i ergibt sich somit zu

$$f_i = 440\,\text{Hz} \cdot 2^{\frac{i-10}{12}} \approx 246.94\,\text{Hz} \cdot 2^{\frac{i}{12}} \tag{15.2}$$

Es gibt weitere Stimmungen wie zum Beispiel die reinen Stimmungen, die darauf basieren, daß bestimmte Töne ganzzahlige Frequenzverhältnisse (z.B. Quinte: $3/2$) zueinander haben. Da der in diesem Projekt gebaute Synthesizer aufgrund verschiedener technischer Beschränkungen ohnehin nicht exakte die Töne wird treffen können, halten sich die Unterschiede der Stimmung stark in Grenzen.

15.2. Tonerzeugung auf dem Mikrocontroller

In diesem Abschnitt wird die Erzeugung von Tönen auf dem Mikrocontroller dargelegt. Als erstes folgt die Ausgabe analoger Werte durch ein geglättetes PWM-Signal, anschließend die Erzeugung eines Sinus-Tons.

15.2.1. Analoge Ausgabe

Die Ausgabe von Tonsignalen erfordert eine Ausgabe von Analogen Werten, d.h. einem Signal, das mehr Werte als nur die binären 0 oder 1 ausgeben kann. Dafür werden Digital-Analog-Wandler (DAC) verwendet, die ein digitales Signal in eine Analoge Spannung wandeln. Im Bereich der Verbraucher-Elektronik werden dafür DAC mit 24 bit Auflösung verwendet.

In diesem Projekt hingegen werden wir ein digitales PWM-Signal mit Hilfe eines RC-Tiefpaßes glätten. Wie in Kapitel 8 gesehen, kann bei einem ausreichend trägen System eine PWM wie eine analoge Ausgabe angesehen werden. Ein Tiefpaß ist genau so ein träges System auf elektronischer Basis.

Tiefpaß

Da die Theorie des Tiefpaßes keine zentrale Rolle beim Synthesizer besitzt, wird hier nur sehr oberflächlich seine Funktionsweise erläutert. Ein RC-Tiefpaß besteht aus einem Widerstand R und einem Kondensator C, die wie in Abb. 15.2 verschaltet sind.

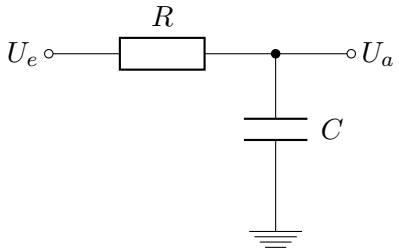


Abbildung 15.2.: RC-Tiefpaß

Ein Kondensator ist ein Bauteil, das solange Ladung Q aufnimmt bzw. abgibt, bis seine Spannung U der von außen angelegten Spannung entspricht. Das Verhältnis C

$$C = \frac{Q}{U} \tag{15.3}$$

bezeichnet man als die Kapazität eines Kondensators.

Ist nun wie in Abbildung 15.2 ein Widerstand in Reihe zum Kondensator geschaltet, so kann sich dieser nicht augenblicklich aufladen, denn der zum Aufladen fließende Strom wird durch R begrenzt. Nach einem Sprung in der Eingangsspannung U_e benötigt der TP eine gewisse Zeit, bis die Ausgangsspannung U_a auf gleicher Höhe ist.

Ist U_e nun beispielsweise eine sinusförmige Wechselspannung mit kleiner Frequenz, so ändert sich U_e nur langsam mit der Zeit. Der träge Tiefpaß hat also vergleichsweise viel Zeit, dem Eingangssignal zu folgen. Bei hochfrequenter Wechselspannung hingegen kann

die Ausgangsspannung des TP nicht mehr dem Eingangssignal folgen. Hochfrequente Anteile eines Signals werden demnach gedämpft, niedrigfrequente Anteile nahezu ungedämpft durchgelassen, daher der Name Tiefpaß.

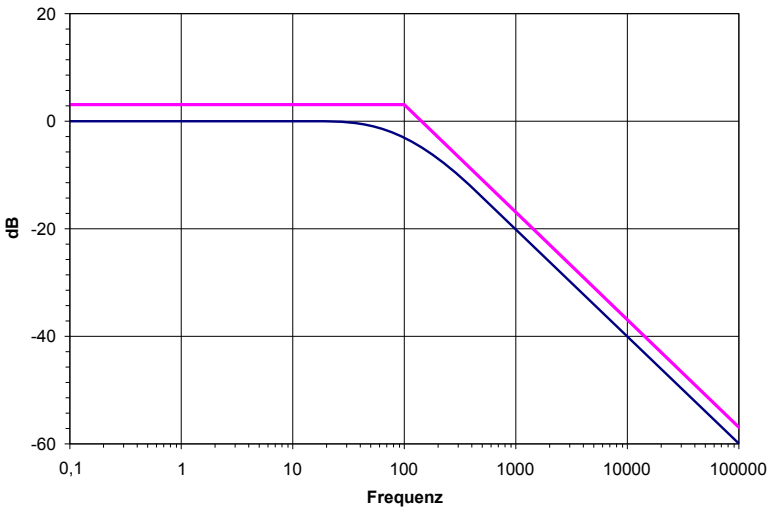


Abbildung 15.3.: Frequenzgang Tiefpaß

Abbildung 15.3 zeigt den Frequenzgang eines Tiefpaßfilters. Die Dämpfung ist in Abhängigkeit der Eingangsfrequenz dargestellt. Es ist zu beachten, daß dies eine Doppelt-logarithmische Darstellung ist, denn die Einheit dB beschreibt die logarithmische Abschwächung der Amplitude. In blau ist der tatsächliche Frequenzgang eingezeichnet, der sich über zwei Geraden approximieren läßt.

Vereinfacht kann man sagen, daß Frequenzen bis zu einer Grenzfrequenz f_g ungehindert durchgelassen werden, während höhere gedämpft werden. Die Grenzfrequenz ergibt sich als:

$$f_g = \frac{1}{2\pi RC} \tag{15.4}$$

Auslegung

Um eine maximale PWM-Frequenz zu erreichen, wird `Timer1` mit prescaler 1 im 8-bit Modus verwendet. Die PWM-Frequenz beträgt somit

$$f_{PWM} = \frac{16\,\text{MHz}}{256} = 62.5\,\text{kHz} \tag{15.5}$$

Die Obergrenze des menschlichen Gehörs liegt altersabhängig zwischen $10 \ldots 30\,\text{kHz}$. Der eingesetzte Tiefpaß hat einen Widerstand von $330\,\Omega$ und einen Kondensator mit $470\,\text{nF}$. Es ergibt sich dadurch eine Grenzfrequenz von $f_g \approx 1.0\,\text{kHz}$. Damit ist eine relativ gute Glättung des PWM-Signals möglich und es wird nur ein kleiner Teil des hörbaren Spektrums gedämpft, der vom zu bauenden Synthesizer ohnehin nicht verwendet wird.

Die analoge Spannung am Ausgang U_a des Tiefpaßes ist nun proportional zum Duty-Cycle des Pulsweitensignals.

15.2.2. Sampling

Der zu erzeugende Ton ist ein zeitabhängiges Spannungssignal. Die Spannung wird dabei auf 8-bit, d.h. 256 Werte diskretisiert und über den eben besprochenen Tiefpaß als geglättete PWM ausgegeben.

Die Zeitkomponente wird ebenfalls diskretisiert, das heißt, daß eine Aktualisierung der Spannungswerte mit einer festen Abtast- oder Sampling-Frequenz erfolgt. Audio-CDs verwenden eine Abtastrate von 44.1 kHz, was als sehr gut angesehen werden kann, denn nach dem Abtast-Theorem kann eine Frequenz von maximal der halben Samplingfrequenz dargestellt werden.

Bei der Erzeugung eines Tonsignals auf dem Mikrocontroller spielt die Rechengeschwindigkeit die entscheidende Rolle. Die Erzeugung einer Sinus-Welle mit Gleitkomma-Berechnung in Echtzeit würde eine deutlich zu kleine Samplingfrequenz zur Folge haben. Folglich wird das Projekt mit Lookup-Tabellen auf vorberechneten Funktionswerten aufbauen, da der Controller 16 KB an Flash-/Programmspeicher hat, in welchem genug Platz ist, vorberechnete Wellenformen abzuspeichern. Die zugrundeliegende Idee hierbei ist es, eine Periode eines jeden Tons vorzuberechnen und im Flash-Speicher des Controllers abzulegen, sodaß zur Laufzeit dann nur die Werte geladen und hintereinander ausgegeben werden müssen.

Wie in Gleichung (15.2) gezeigt, läßt sich zu jedem Ton i die passende Grundtonfrequenz f_i berechnen. Zunächst wird angenommen, bei dem zu erzeugenden Ton handele es sich um einen Sinus mit Frequenz f_i. Damit die Töne nahtlos aneinander gereiht werden können, müssen die Knoten des Sinus auf einen Samplingzeitpunkt fallen, wie in Abb. 15.4 dargestellt ist. Dies bedeutet allerdings, daß die Wahl der Frequenz f für einen Ton dadurch eingeschränkt ist, daß die Anzahl des Samples n_{sample} ganzzahlig sein muß.

$$f = \frac{f_{sample}}{n_{sample}}, n_{sample} \in \mathbb{Z} \tag{15.6}$$

Beispiel:
Wie später noch dargelegt werden wird, arbeitet der zu programmierende Synthesizer mit einer Samplingfrequenz von $f_{sample} = 10\,\text{kHz}$. Bei Erzeugung des Kammertons A von eigentlich 440 Hz ergibt sich

$$n_{sample} = 23 \approx 22.727 \tag{15.7}$$

Daraus folgt dann eine tatsächliche Frequenz f von

$$f = \frac{10\,\text{kHz}}{23} \approx 434.78\,\text{Hz} \tag{15.8}$$

Diese Abweichung kann sehr deutlich gehört werden. Je höher der Ton desto größer wird der Fehler: Für das A'' zwei Oktaven höher wäre die Frequenz $f \approx 1666.66\,\text{Hz}$ anstatt der theoretischen $f_i = 1760\,\text{Hz}$. Das direkt darunterliegende G'' wird dabei sogar auf die gleiche Frequenz abgebildet werden Die Töne sind nicht nur falsch sondern auch ununterscheidbar.

Erhöhung der Genauigkeit

Um dieses Problem zu umgehen, geht man dazu über, nicht nur eine Periode eines Tons zu samplen sondern mehrere. Dadurch folgt die weniger strikte Forderung, daß der Anfangs-

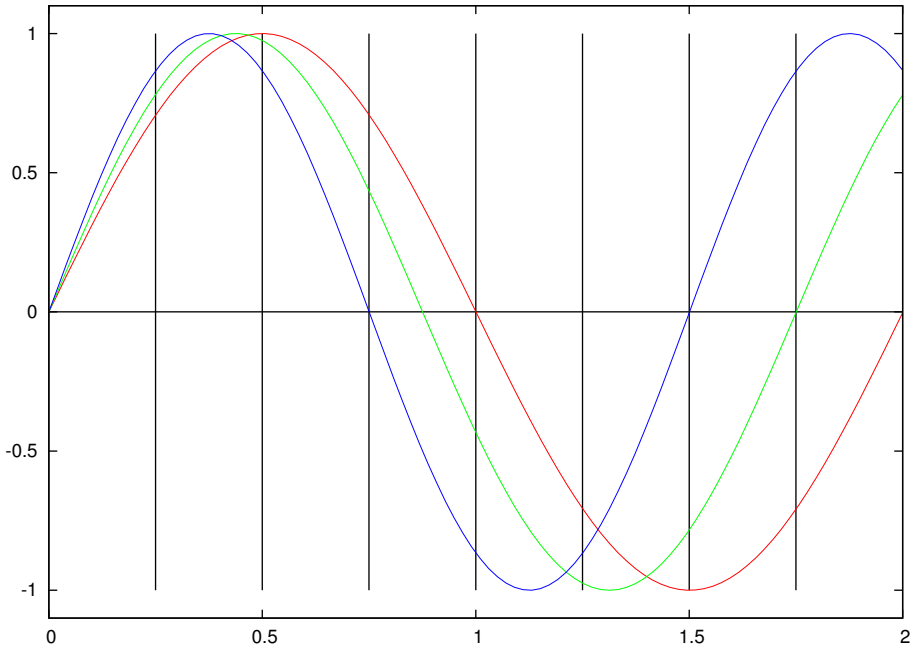


Abbildung 15.4.: Erlaubte Frequenzen

sowie der Endknotenpunkte nach m Perioden auf dem Samplingraster liegen muß.

Beispiel:
Fordert man einen Richtwert von ungefähr 100 Samples pro Ton, so ergibt sich für Kammerton A $m = 4$ und damit $n_{sample} = 91 \approx 90.909$. Die tatsächliche Frequenz beträgt nun

$$f = 10\,\text{kHz}\frac{4}{91} \approx 439.56\,\text{Hz} \tag{15.9}$$

und ist damit deutlich besser. Für A'' ergibt sich eine Frequenz von $f = 1752.58\,\text{Hz}$ und für G'' $f = 1666.67\,\text{Hz}$ anstatt der theoretischen 1661.21 Hz. Obwohl nach wie vor ein nicht verleugnender Fehler existiert, so ist doch auf diese Weise ein recht präzises Sampling möglich.

Wie in der Übung selbst beschrieben ist, wird ein Python-Skript zur Verfügung gestellt, das das Sampling übernimmt und die Werte in eine Header-Datei schreibt, die dann in das C-Projekt auf dem Mikrocontroller übernommen werden kann.

15.3. Aufgabe M.1: Tonausgabe

a) Schließen Sie den Tiefpaß-Eingang an Pin `B1` des Mikrocontrollers an, der Ausgang für das von `timer1` generierten PWM-Signals ist.
Konfigurieren Sie den Timer auf `Fast PWM, 8-bit` mit einem prescaler von 1 durch setzen der entsprechenden Bits, die Sie im Datenblatt [4] nachlesen können. Aktivieren Sie ebenfalls den `CompareMatch1`, damit das Signal auch am Pin ankommt. Schließen Sie die Aktiv-Lautsprecher am Ausgang des Tiefpasses an.

Schreiben Sie zunächst zum Testen einige Codezeilen, die ein vorerst per Gleitkomma-Berechnung erzeugtes Sinus-Signal als Duty-Cycle ausgibt. Beachten Sie hierbei, daß die Sinus-Funktion Werte im Bereich von $[-1.0, 1.0]$ ausgibt, das 8-bit-PWM-Vergleichsregister aber Werte von $[0 : 255]$ erwartet. Der Nullpunkt wird also um 127 nach oben verschoben. Es sollte nun ein Ton zu hören sein, wenn Sie einen $\sin(x)$ in Schritten von $\delta x = 0.2$ ohne explizites Delay durchlaufen. Die Verzögerung zwischen den Samples wird alleine durch die Rechendauer erzeugt.

Ab diesem Schritt ist es sehr hilfreich ein Oszilloskop anzuschließen, um die Ausgabe zu visualisieren. Sind Sie mit dem Gerät noch nicht vertraut, zeigt Ihnen der Betreuer den Umgang damit.

b) Nachdem nun die Analogausgabe funktioniert, soll ein erstes Signal aus der Lookup-Table erzeugt werden. Laden Sie das Python-Script `generator.py` herunter [8] und führen Sie es mittels `python generator.py` aus. Das Skript erzeugt eine Datei namens `tone.h`, die Sie in ihrem Projekt mit einem einfachen `#include`-Statement einbinden können. Diese Datei enthält drei Arrays

- `tones` Dieses Array ist im Programmspeicher hinterlegt, und deshalb nicht direkt zugreifbar wie übliche Variablen im RAM. Hier sind alle Werte für alle Töne hinterlegt.
- `toneOffset` Dieses Array enthält die Indizes, ab denen die Töne, die in `tones` gespeichert sind, beginnen.
- `toneLength` Dieses Array enthält die Anzahl des Samples, die zu einem Ton gehören.

Folgendes Beispiel verdeutlicht die Benutzung:

```
1 uint8_t tone_nr=10; // Note A
2 uint16_t offset=toneOffset[tone_nr];
3 uint16_t t;
4 for (t=0; t < toneLength[tone_nr]; t++) {
5         uint8_t  pwm=pgm_read_byte(&tones[t+offset]);
6         // PWM Ausgeben
7 }
```

Adaptieren Sie das obige Beispiel und geben Sie einen Ton aus. Die Lookuptable ist für eine bestimmte Samplingfrequenz angelegt worden, die hier erstmal nicht von Bedeutung ist. Ein einfaches Delay von ungefähr 100 μs zwischen den Samples sollte einen hörbaren Ton erzeugen.

c) Die Wahl der besten Samplingfrequenz kann eine lange Zeit in Anspruch nehmen, deshalb folgt hier eine Empfehlung. Der Compare-Match Interrupt von Timer2 wird zum Setzen des nächsten Sampling-Werts verwendet. `Timer2` wird dazu mit folgender Konfiguration betrieben:

- CTC-Modus
- prescaler = 64
- Vergleichswert OCR2A = 25
- TIMER2 Output Compare Match A Interrupt (OCIE2A) an

Es ergibt sich somit eine Samplingfrequenz von

$$f_{sample} = \frac{16\,\text{MHz}}{64 \cdot 25} = 10\,\text{kHz} \tag{15.10}$$

bei der ein Interrupt ausgelöst wird. Der dazugehörige Handler wird mit folgendem Namen bezeichnet und muß, damit er nicht doppelt belegt ist, aus der `timer.c` entfernt werden.

```
1 SIGNAL (TIMER2_COMPA_vect) { }
```

Schreiben Sie nun die Ausgabe aus dem vorherigen Teil so um, daß bei jedem Aufruf der ISR ein Sample ausgegeben wird. Die `for(t=...)`-Schleife wird aufgelöst. Bei jedem Aufruf der ISR, was durch den Timer-Interrupt geschieht, wird das Sample zum Wert von `t` ausgegeben und `t` um eins erhöht bzw. auf 0 gesetzt, wenn es den Maximalwert überschreitet.

d) Die `main()`-Funktion soll nun den zu spielenden Ton durchwechseln. Die Anzahl der Töne ist in dem define `NO_TONES` in der generierten `tone.h` definiert. Folgendes Beispiel erläutert die für diese Aufgabe geplante Trennung zwischen ISR und `main`-Funktion:

```
1  volatile uint8_t tone_nr=0;
2  volatile uint16_t t=0 ;
3
4  main() {
5  ...
6
7          uint8_t i;
8          for (i=0; i < NO_TONES; i++) {
9                          tone_nr =i;
10                         _delay_ms(10); //Achtung, siehe Hinweis!!!
11         }
12
13 ...
14 }
15 SIGNAL ( TIMER2_COMPA_vect ) {
16         if (t>=toneLength [ tone_nr ]) {
17                 t=0;
18         }
19         uint16_t offset = toneOffset [ tone_nr ];
20         uint8_t pwm = pgm_read_byte (& tones [ t + offset ]) ;
21         // PWM Ausgeben
22         t++;
23 }
```

Hinweis:
Die `_delay_ms` und `_delay_us` Funktionen sind so programmiert, daß sie genau so viel CPU-Zeit verbrauchen, wie angegeben ist. Die implementierte Interrupt-Routine wird allerdings sehr häufig aufgerufen und verbraucht dafür vergleichsweise viel Rechenzeit. Der Prozessor ist also vorwiegend mit der ISR beschäftigt und kann das Delay kaum abarbeiten. Folglich sind die tatsächlichen Delay-Zeiten in diesem Beispiel etwa 10-100 mal so groß wie angegeben. Das Delay muß in diesem Aufgabenteil also entsprechend implementationsabhängig experimentell bestimmt werden.

15.4. Aufgabe M.2: Synthesizer

Bei der folgenden Programmierung gilt es nun sehr effizient zu programmieren und gegebenenfalls ein wenig die Lesbarkeit des Codes zu opfern. Bei einer Interrupt-Frequenz von 10 kHz und einem Prozessor-Takt 16 MHz bleiben 1600 Takte für die Berechnung. Insbesondere sollten Funktionsaufrufe vermieden werden. Bei jedem Aufruf werden die Prozessorregister gesichert, damit nach der Rückkehr aus der Funktion exakt mit dem Kontext weitergerechnet werden kann, von dem aus die Funktion aufgerufen wurde. Das Sichern auf den Stack sowie das Wiederherstellen kann damit ungefähr 60-100 Prozessor-Zyklen in Anspruch nehmen.

a) Nun soll der Synthesizer mehrere Töne gleichzeitig spielen können. Dazu müssen die erzeugten Töne zusammengemischt werden. Ein einfaches Aufaddieren der Amplituden kann den erlaubten Wertebereich überschreiten, sodaß es dann zu einem sehr unschönen clipping-Effekt kommt. Es existieren einige Theorien darüber, wie das Abmischen durchgeführt werden kann, ohne dabei den Höreindruck zu mindern. Der Einfachheit halber und auch aufgrund der begrenzten Rechenzeit wird in diesem Synthesizer einfach über die Amplituden summiert und im Anschluß durch die Anzahl der aktiven Töne geteilt.

   Implementieren Sie nun die Möglichkeit, mehrere Töne gleichzeitig spielen zu können. Sie benötigen dafür

   1. Flags, die darüber entscheiden, ob der Ton an oder aus ist.
   2. für jeden Ton eine eigene Zählvariable für den Sampleindex
      Der Sampleindex gibt die Position des aktuell gespielten Samples an. Im obigen Beispiel ist dies die Variable `t`.

   Achten Sie darauf, daß Töne nur dann deaktiviert werden, wenn ihre Samples vollständig abgespielt worden sind. Werden Sie vorher deaktiviert, kommt es zu einem Knacken durch die Unstetigkeit im Ausgangssignal.

   Aktivieren und Deaktivieren Sie in ihrer Hauptschleife alle Töne nacheinander.

b) Quinten gelten mit einem Frequenzverhältnis von 3 : 2 als konsonante Intervalle. Fünf Halbtonschritte nach der verwendeten gleichstufigen Stimmung ergeben

$$\left(\sqrt[12]{2}\right)^5 \approx 1.33483 \approx \frac{3}{2} \tag{15.11}$$

   Spielen Sie in der Hauptschleife nun wieder einen Ton sowie den fünf Halbtöne darüberliegenden Ton gleichzeitig. Je höher die Töne werden, desto deutlicher hört man Fehler des Synthesizers durch die von der theoretischen Stimmung abweichenden Frequenzen.

c) Damit nun sinnvoll Musik gespielt werden kann, soll der Synthesizer an den PC angeschlossen werden, der dann per serieller Verbindung die zu spielenden Noten verschickt.

   Dabei soll ein Protokoll verwendet werden, bei dem der PC angelehnt an die von der MIDI[1]-Schnittstelle verschickten Ereignisse dem Controller Anweisungen zum

[1] Musical Instrument Digital Interface

Aktivieren und Deaktivieren einer Note gibt.

Jedes vom PC an den Controller verschickte Byte ist ein Befehl. Das höchstwertige Bit gibt dabei an, ob die Note aktiviert oder deaktiviert werden soll, die unteren 7 Bits bestimmen die Notennummer beginnend von C. Folgendes Code-Beispiel veranschaulicht die effiziente Implementation der Protokolls.

```
1 c=uart_getc();
2 nr=c & ~(1<<7); // Hoechstwertiges Bit ignorieren;
3 tone_on_off[nr]=c & (1<<7); // Nur hoechstwertiges Bit ansehen
```

Als Sonderfall soll zusätzlich das Verschicken einer 255 alle Noten deaktivieren.

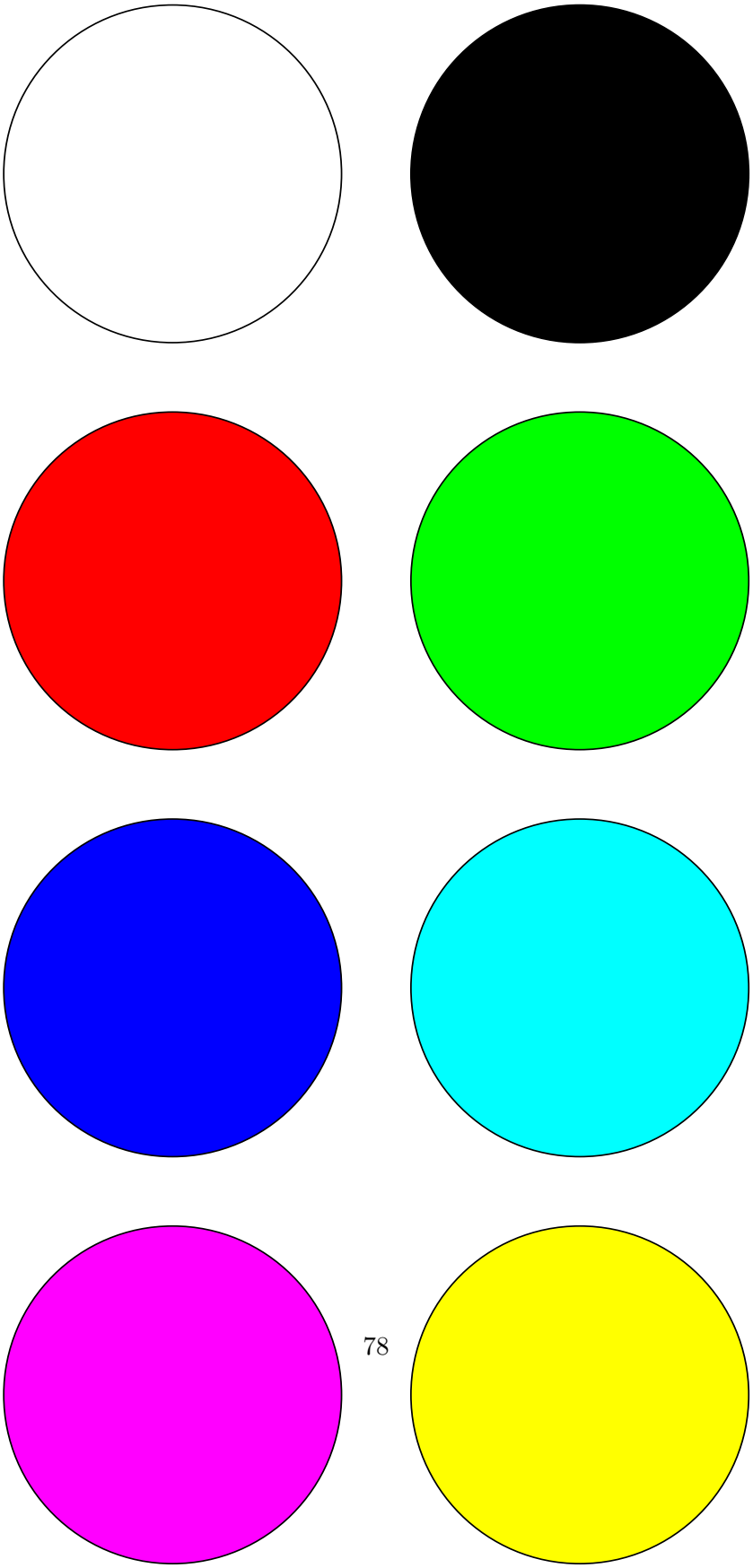
Dieses können Sie testen, indem Sie das Python-Skript `synth.py` sowie das Midi-File `elise2.mid` herunterladen. Starten Sie das Synthesizer-Skript. Es schickt die Befehle zum Spielen von Beethovens „Für Elise“ im o.g. Protokoll über die serielle Schnittstelle. Eventuell können müssen sie die Bezeichnung der seriellen Schnittstelle im Skript anpassen.

Theoretisch können beliebige Midi-Files abgespielt werden, von denen es eine ganze Reihe im Internet gibt. Die verwendete Python-Bibliothek ist allerdings nicht vollständig kompatibel zum kompletten Midi-Standard, sodaß es Einschränkungen gibt.

Speichern Sie diesen Zustand ab, um ihn später vorführen zu können.

A. Anhang

A.1. Farbtabelle



Literaturverzeichnis

[1] 2013. http://p.globalsources.com/IMAGES/PDT/B1059413054/CDS-Photoresistor-LDR.jpg.

[2] Wikimedia Commons. en:hsv color space visualization; hsv as a cylindrical object. created by wapcaplet in the en:gimp.

[3] Wikimedia Commons. Hue scale 0—360.the hue unit is degree.

[4] Atmel Corporation. Datasheet to atmega48/88/168, 2005. http://joanna.iwr.uni-heidelberg.de/resources/datasheets/atmega168.pdf.

[5] Atmel Corporation. Datasheet to atmega48/88/168, 2005. http://joanna.iwr.uni-heidelberg.de/resources/datasheets/atmega168.pdf.

[6] Germany for KUKA Roboter GmbH Jo Teichmann, Augsburg. Kuka industrial robot kr10 scara, 2007. [Online; Stand 28. September 2015].

[7] KUKA Roboter GmbH. *Betriebsanleitung KR 5 sixx R650, R850*, 2010. Version: BA KR 5 sixx V5 de, Stand: 14.01.2010.

[8] Benjamin Reh. Python skript zur erzeugung von waveform-lookuptablen, 2015. http://joanna.iwr.uni-heidelberg.de/resources/synthesizer/generator.py.

[9] Wikipedia. Baud, die freie enzyklopädie, 2013. http://de.wikipedia.org/wiki/Baud.

[10] Wikipedia. Servo — wikipedia, die freie enzyklopädie, 2013. [Online; Stand 16. Dezember 2013].

Version

This document was created from git `commit 949de46fe99ada13ca06efbccb211f38c13fe41d`