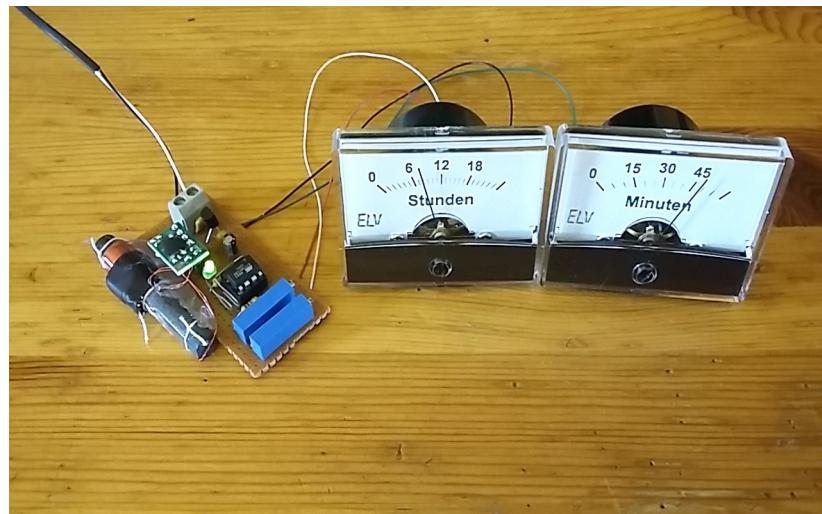


Einstieg in die Elektronik mit Mikrocontrollern

Band 3
von Stefan Frings



Downloaded von <http://stefanfrings.de>

Inhaltsverzeichnis

1 Einleitung.....	4
2 Kommandozeile.....	5
2.1 Grundschaltung.....	5
2.2 Avrdude.....	6
2.3 C-Compiler.....	8
2.4 Makefile.....	9
3 Eieruhr.....	15
3.1 Informationen zur Bauteil-Wahl.....	15
3.2 Schaltung.....	16
3.3 Hardware Konfiguration.....	16
3.4 Piepen.....	17
3.5 Zeitmessung.....	18
3.6 Stromaufnahme Reduzieren.....	19
4 Quiz-Buzzer.....	23
4.1 Schaltung.....	24
4.2 Programm.....	25
5 Fahrrad-Blinker.....	29
5.1 Schaltung.....	29
5.2 Programm.....	30
5.3 Aufbau in groß.....	42
6 Verkehrsampel.....	44
6.1 Minimal Ampel.....	44
6.2 Fußgänger Überweg.....	47
6.3 Exkurs: Leuchtdioden.....	48
6.4 Kreuzung.....	50
6.5 Exkurs: Spannungsregler.....	53
6.6 Bedarfsampel.....	54
6.7 Exkurs: Entstörung.....	55
6.8 Reset Pin doppelt belegen.....	56
7 Servos.....	60
7.1 Exkurs: Arduino Nano.....	60
7.2 Probe-Aufbau.....	62
7.3 Steuersignal erzeugen.....	63
7.4 Exkurs: Dioden in Stromversorgung.....	69
7.5 Tee Roboter.....	70
7.6 Zahnpulz-Uhr.....	72
7.7 Exkurs: Variablen in Interruptroutinen.....	74
7.8 Vampirkatze.....	77
8 Code Scanner.....	82
8.1 Exkurs: Zeitmessung mit Überlauf.....	87
8.2 Anregungen zum Erweitern.....	88
9 Sieben-Segment Anzeigen.....	89
9.1 Ansteuerung.....	90
9.2 Stoppuhr.....	107
9.3 Torzähler.....	112
9.4 Batterie Meßgerät.....	114
9.5 Tachometer.....	117
9.6 Thermometer.....	124
10 Matrix Anzeigen.....	128
10.1 Zeichensätze.....	134

11	LCD Anzeigen.....	139
11.1	Anschlussbelegung.....	139
11.2	Timing.....	140
11.3	Befehlssatz.....	142
11.4	LCD Library.....	143
11.5	Text Ausgeben.....	147
11.6	Zeichensatz Generator.....	148
11.7	LCD an 3,3V.....	150
12	Analoge Funkuhr.....	153
12.1	Analoge Einbauinstrumente.....	153
12.2	DCF Empfänger.....	155
12.3	DCF Signal.....	157
12.4	Schaltung.....	158
12.5	Programm.....	159
12.6	Inbetriebnahme.....	162
12.7	Fehlerbehebung.....	163
13	Experimente mit WLAN.....	164
13.1	Das ESP-01 Modul erforschen.....	165
13.2	Das Internet der Dinge.....	171
13.3	Per WLAN Fernsteuern.....	178
14	Wie geht es weiter?.....	183
15	Material-Liste.....	184

1 Einleitung

Du hast durch die vorherigen Bände dieses Buches gelernt, dein erstes C Programm zu schreiben und auf dem Mikrocontroller auszuführen. In Band zwei habe ich dir außerdem einen groben Überblick über interessante Bauteile rund um Mikrocontroller vermittelt.

Kommen wir nun zu den Experimenten.

In diesem Band 3 möchte ich dir an konkreten Beispielen zeigen, wie man die Bauteile anwendet und dazu passende Programme schreibt.

Einige einfache Schaltpläne habe ich aus Text-Zeichen zusammengesetzt (ASCII-Art), um dich auf das Lesen solcher Pläne in Diskussionsforen vorzubereiten.

Ich bitte dich dringend darum, die Datenblätter der verwendeten Bauteile zu lesen, auch wenn dein Englisch noch nicht gut genug ist, um sie komplett zu verstehen. Nur dann wirst du meine Erklärungen nachvollziehen können. Elektronik Lernen bedeutet Lesen, Lesen und nochmal Lesen. Dann ein bisschen Rechnen und dann erst aus dem Talent, Große Dinge aus kleinen Teilen zu bauen.

Führe die Experimente in der Reihenfolge durch, wie sie im Buch stehen, oder lies zumindest die Kapitel. Denn sie bauen aufeinander auf und vermitteln Fachwissen, dass ich nicht wiederhole.

Die Downloads zum Buch findest du auf der Seite

http://stefanfrings.de/mikrocontroller_buch/index.html.

Für Fragen zu den Schaltungen, wende dich bitte an den Autor des Buches stefan@stefanfrings.de oder besuche das Forum <http://mikrocontroller.net>.

Jedes Hauptkapitel beginnt mit einer Material-Liste für das jeweilige Kapitel. Zusätzlich findest du ganz am Ende des Buches eine vollständige Material-Liste.

Ein wichtiger Hinweis:

In diesem Buch werden wir den AVR Mikrocontroller teilweise mit einem 5 V Netzteil betreiben. Falls du dazu ein altes Handy-Ladegerät verwenden möchtest, überprüfe vorher dessen Ausgangsspannung. Manche Ladegeräte liefern nämlich im Leerlauf weit mehr als die aufgedruckten 5 V. Und bei mehr als 6 V wird dein Mikrocontroller kaputt gehen! In vielen Fällen kann man die Spannung unter 6 V bekommen, indem man das Netzteil zusätzlich mit einem 220 Ohm Widerstand belastet.

2 Kommandozeile

Kommandozeile – auch bekannt als: Eingabeaufforderung, CMD-Fenster, Konsole, Shell, Terminalfenster.

In diesem Experiment zeige ich, wie man auf der Kommandozeile arbeitet, also unabhängig vom AVR Studio. Alle Experimente dieses Buches bauen darauf auf.

Das Nutzen der Kommandozeile bedeutet vor allem, dass du lernst, Makefiles zu verwenden anstatt die Projekte in irgendwelchen grafischen Dialogen zu konfigurieren.

Ich möchte dir den Umgang mit der Kommandozeile beibringen, weil alle professionellen Entwickler sie ebenfalls nutzen. Die Kommandozeile bietet drei wesentliche Vorteile gegenüber dem AVR Studio:

- **Reproduzierbare Ergebnisse.**

Alle nötigen Einstellungen sind in Textdateien festgelegt, so dass Fehlbedienung ausgeschlossen wird. Man kann das Projekt beliebig oft compilieren und hochladen, wobei sichergestellt ist, dass das Ergebnis stets das Selbe ist.

- **Unabhängigkeit von der Entwicklungsumgebung.**

In den 10 Jahren, wo ich AVR Mikrocontroller programmiere, wurde die Entwicklungsumgebung von Atmel bereits zwei mal so sehr verändert, dass sie die alten Projekte nicht mehr öffnen konnte. Durch Verwendung von Makefiles kann man hingegen jeden beliebigen Texteditor verwenden (natürlich auch das AVR Studio) und nach Lust und Laune hin und her wechseln.

- **Kompatibilität zu anderen Betriebssystemen.**

Die Software von Atmel läuft nur unter Windows, aber wenn du Makefiles verwendest, kannst du auch mit Linux und Mac OS arbeiten. Linux wird oft empfohlen, um Probleme mit Windows Treibern zu umgehen.

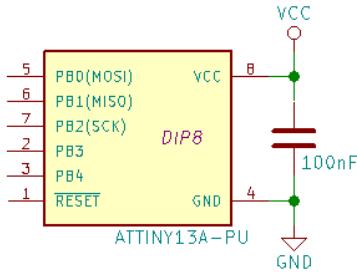
2.1 Grundschaltung

Beginnen wir mit der Grundschaltung. Unser erstes Ziel besteht darin, einen ATtiny13 Mikrocontroller mit Strom zu versorgen und mit dem Programmieradapter zu verbinden, so dass wir ihn programmieren können.

Material:

- 1 Mikrocontroller ATtiny13A-PU (alternativ ATtiny25, 45 oder 85)
- 1 Kondensator 100nF
- 1 Steckbrett und Kabel
- 1 ISP Programmieradapter
- 1 Batteriekasten mit 3 Zellen (ca. 3,6V)

Die minimale Beschaltung des ATtiny13 kennst du schon:



Der Kondensator ist notwendig, um die Spannungsversorgung zu stabilisieren. Er sollte möglichst nahe zum Mikrocontroller platziert werden. Diese Minimalschaltung genügt, um den Programmieradapter zu benutzen. Er wird an die Leitungen VCC, GND, MISO, MOSI, SCK und Reset angeschlossen.

2.2 Avrdude

Bisher hast du den Programmieradapter mit einem grafischen Programm bedient. Dieses mal werden wir stattdessen das Programm avrdude verwenden. Avrdude ist das beste Tool zur Benutzung von Programmieradapters, denn es unterstützt alle Geräte und alle Funktionen. Was Avrdude nicht kann, können andere Programme auch nicht. Und wo andere Programme versagen (insbesondere bei wenig hilfreichen Fehlermeldungen), kommt man mit avrdude oft doch noch weiter.

Jetzt muss du aber herausfinden, welche Parameter avrdude benötigt, um deinen ISP Programmieradapter anzusprechen. Diese Information findest du im Internet oder in der Bedienungsanleitung deines Gerätes.

Für zwei Geräte zeige ich dir nun, wie das geht. Bei anderen Programmieradapters wird es sehr ähnlich gehen.

2.2.1 Diamex USB-ISP

Der Diamex USB-ISP Programmieradapter wird vom Betriebssystem als virtuelles serielles Gerät erkannt. Das Betriebssystem legt also einen virtuellen seriellen Port an und weist ihn dem Stick zu.

Unter Windows kann man dies im Gerätemanager sehen (siehe Band 1 Kapitel „Bedienung ISP Programmer“).

Unter Linux kannst du direkt nach dem Einstecken den Befehl „dmesg“ eingeben, um den Namen der virtuellen seriellen Schnittstelle zu erfahren. Das ist meistens /dev/ttyUSB0.

Unter Windows:

```
avrdude -P \\.\com3 -c stk500v2 -B20 -p attiny13
```

Unter Linux:

```
sudo avrdude -P /dev/ttyUSB0 -c stk500v2 -B20 -p attiny13
```

Wobei du hinter -P den richtigen virtuellen seriellen Port angeben musst.

2.2.2 Atmel AVR ISP mkII

Der Atmel AVR ISP mkII wird nicht als serielles Gerät erkannt, sondern direkt über USB angesteuert.

Unter Windows:

```
avrdude -P usb -c avrispmkII -B20 -p attiny13
```

Unter Linux:

```
sudo avrdude -P usb -c avrispmkII -B20 -p attiny13
```

2.2.3 USBASP

Auch USBASP Programmieradapter haben keinen seriellen Port. Bei ihnen lautet der Befehl unter Windows:

```
avrdude -P usb -c usbasp -p attiny13
```

Unter Linux:

```
sudo avrdude -P usb -c usbasp -p attiny13
```

Der Parameter **-B** entfällt bei USBASP Geräten, weil sie sich je nach Modell automatisch einstellen oder über einen Jumper zur Auswahl zwischen „schnell“ und „langsam“ verfügen.

Für alle Programmieradapter gilt:

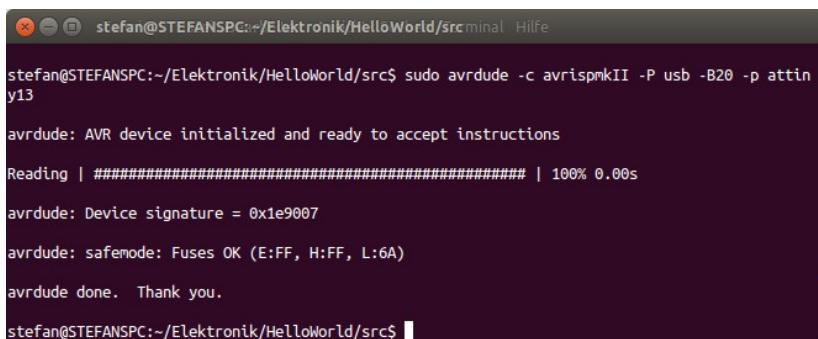
Mit **-P** gibst du den Anschluss oder den virtuellen seriellen Port an, wo der Programmieradapter an deinen Computer angeschlossen ist.

Mit **-c** gibst du an, welchen Typ vom Programmieradapter verwendet wird. Viele Programmieradapter sind zu irgendeinem „Original“ von Atmel kompatibel, wie der oben genannte Diamex, so dass man dann den Namen des entsprechenden Produktes von Atmel angibt.

Mit **-B20** stellst du die Geschwindigkeit ein, mit der dein Programmieradapter zum Mikrocontroller spricht. Größere Zahlen bewirken langsamere Kommunikation. Solange du keinen guten Grund hast, solltest du immer **-B20** angeben. Manche Programmieradapter ermitteln die optimale Geschwindigkeit automatisch und funktionieren daher auch gut ohne diesen Parameter.

Mit dem Parameter **-p attiny13** gibst du den Typ des Mikrocontrollers an.

Du solltest jetzt soweit sein, dass das Programm **avrdude** erfolgreich mit dem Programmieradapter kommuniziert und dieser wiederum erfolgreich mit dem Mikrocontroller kommuniziert. Die Bildschirmausgabe sollte ungefähr so aussehen:



```
stefan@STEFANSPC:~/Elektronik/Helloworld/src$ sudo avrdude -c avrispmkII -P usb -B20 -p attiny13
avrdude: AVR device initialized and ready to accept instructions
Reading | #####| 100% 0.00s
avrdude: Device signature = 0x1e9007
avrdude: safemode: Fuses OK (E:FF, H:FF, L:6A)
avrdude done. Thank you.
stefan@STEFANSPC:~/Elektronik/Helloworld/src$
```

Damit hast du noch kein Programm in den Mikrocontroller hochgeladen, sondern lediglich die Verbindung und Kommunikation geprüft. Falls das nicht funktioniert, kann man den Parameter **-v** verwenden, um mehr Details zu sehen:

```

stefan@STEFANSPC: ~/Elektronik/Helloworld/src

Programmer Type : STK500V2
Description      : Atmel AVR ISP mkII
Programmer Model: AVRISP mkII
Hardware Version: 1
Firmware Version Master : 1.23
Vtarget          : 4.3 V
SCK period       : 20.12 us

avrdude: AVR device initialized and ready to accept instructions

Reading | ##### | 100% 0.00s

avrdude: Device signature = 0x1e9007
avrdude: safemode: lfuse reads as 6A
avrdude: safemode: hfuse reads as FF

avrdude: safemode: lfuse reads as 6A
avrdude: safemode: hfuse reads as FF
avrdude: safemode: Fuses OK (E:FF, H:FF, L:6A)

avrdude done. Thank you.

stefan@STEFANSPC:~/Elektronik/Helloworld/src$ 

```

Hier sehen wir unter anderem die interessante Ausgabe „Vtarget: 4.3V“. Meine Batterien haben also überraschenderweise deutlich mehr als 3,6 Volt. Wobei man berücksichtigen sollte, dass diese Spannungsmessung nur ein grobes Schätzzeisen ist. Manche Programmieradapter können die Spannung gar nicht messen.

Beide Bildschirmfotos zeigen, dass die Verbindung zum Mikrocontroller funktioniert hat. Es wird keine Fehlermeldung angezeigt.

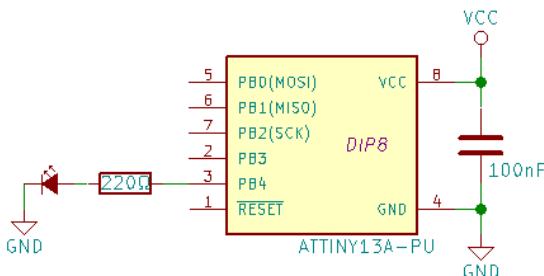
Falls du hier schon auf unerwartete Probleme gestoßen bist, lass dir helfen. Es ist wirklich wichtig, dass du diesen Teil ans Laufen bekommst, sonst kannst du kein einziges Experiment aus den folgenden Kapiteln nachvollziehen.

2.3 C-Compiler

Bisher hast du den C-Compiler nicht direkt benutzt, sondern mit der Entwicklungsumgebung AVR Studio gearbeitet. Das AVR Studio benutzt den C-Compiler im Hintergrund, um deinen Quelltext in Maschinencode zu übersetzen, den der Mikrocontroller wiederum ausführen kann.

Jetzt zeige ich dir, wie man den C-Compiler „zu Fuß“ verwendet.

Schließe eine LED an PB4 an:



Erstelle mit irgendeinem Text-Editor ein kleines Testprogramm, dass diese LED zum Leuchten bringt. Unter Windows eignet sich dazu das mitgelieferte Notepad, aber es gibt auch schönere und kostenlose Programme im Internet, zum Beispiel das Programmers Notepad, Notepad++ und Editpad. Unter Linux benutze ich gerne gedit oder kedit.

Speichere das folgende Programm unter dem Namen test.c ab:

```

#include <avr/io.h>

int main(void)
{
    DDRB |= (1<<PB4);
}

```

```
    PORTB |= (1<<PB4);  
}
```

Jetzt öffnest du ein Kommandozeilen-Fenster und gehst mit dem cd Befehl in das Verzeichnis, wo du die Textdatei abgespeichert hast. Dann gibst du ein:

```
avr-gcc -mmcu=attiny13 -c test.c
```

Als Ergebnis erhältst du eine sogenannte Objekt-Datei mit dem Namen test.o. Diese Datei enthält den Maschinencode. Wenn dein Projekt aus mehreren *.c Dateien besteht, musst du jede einzeln compilieren und erhältst daher auch viele *.o Dateien.

Als Nächstes muss man die ganzen *.o Dateien zu einem Programm zusammen führen. Diesen Vorgang nennt man „Das Linken“:

```
avr-gcc -mmcu=attiny13 -o test.elf test.o
```

Hinter -o gibt man den namen der Elf-Datei an, das ist das Resultat des Linkens. Dahinter gibt man eine oder mehrere *.o Dateien an, das sind die Dateien, die zusammen gelinkt werden sollen.

Nun haben wir eine elf Datei. Unter Linux wäre das schon das fertige ausführbare Programm. Entsprechend einer exe Datei unter Windows.

Wir müssen das Programm aber noch in ein anderes Format umwandeln, welches der Programmieradapter verarbeiten kann. Das geht mit folgendem Befehl:

```
avr-objcopy -j .text -j .data -O ihex test.elf test.hex
```

Der Parameter -j bestimmt, welche Teile aus der elf Datei verwendet werden sollen. Dabei steht .text für den Ausführbaren Machinencode und .data sind die Daten (Texte, Zahlen, etc). Der Parameter -O gibt an, dass wir eine Umwandlung in das Intel-Hex Format haben wollen. Dieses Format versteht avrdude. Dahinter kommen zwei Dateienamen, nämlich die elf Datei als Quelle und der Name der hex Datei als Ziel.

Um das Programm nun in den Mikrocontroller hochzuladen, benötigen wir avrdude. Zum Beispiel:

```
avrdude -P \.\com3 -c stk500v2 -B20 -p attiny13 -U flash:w:test.hex
```

Denke dran, bei Linux „sudo“ davor zu schreiben. Nach den bereits bekannten Parametern habe wir einen neuen:

```
-U flash:w:test.hex
```

Damit befehlen wir dem avrdude, dass er die Datei test.hex in den Flash Speicher des Mikrocontrollers übertragen soll.

Das Ganze war jetzt ziemlich mühsam und du wirst diese Befehle nur ungern immer wieder neu eintippen. Vor allem nicht bei großen Projekten, die aus vielen Dateien bestehen. An dieser Stelle kommen wir zum Makefile.

2.4 Makefile

Das Makefile ist die Konfigurationsdatei für das Programm make, welches die obigen manuellen Vorgänge automatisiert. Ich zeige hier mal das komplette Makefile aus meiner „HelloWorld“ Vorlage.

```

# Makefile for this AVR project

# make code      Compiles the source code
# make fuses     Program fuses
# make program   Program flash and eeprom

# make list      Create generated code listing
# make clean     Delete all generated files

# Programmer hardware settings for avrdude
# AVRDUDE_HW = -c avr910 -P /dev/ttyUSB0 -b 115200
# AVRDUDE_HW = -c arduino -P COM3 -b 57600
AVRDUDE_HW = -c avrispmkII -P usb -B16

# Name of the program without extension
PRG = HelloTiny

# Microcontroller type and clock frequency
MCU = attiny13
F_CPU = 1200000

# Optional fuse settings, for example
# LFUSE = 0x6A
# HFUSE = 0xFF

# Objects to compile (*.c and *.cpp files, but with suffix .o)
OBJ = driver/serialconsole.o driver/systemtimer.o main.o

#####
# You possibly do not need to change settings below this marker
#####

# Binaries to be used
# You may add the path to them if they are not in the PATH variable.
CC      = avr-gcc
OBJCOPY = avr-objcopy
OBJDUMP = avr-objdump
AVRDUDE = avrdude
AVR_SIZE = avr-size

# Do we need to write Eeprom? (yes/no)
EEPROM = no

# Extra Libraries
#LIBS = -L path/to/libraries -llibrary1 -llibrary2

# Extra Includes
#INCLUDES = -I path/to/include_files

# Compiler flags for C and C++
FLAGS = -Wall -O1 -fshort-enums -mmcu=$(MCU) -DF_CPU=$(F_CPU)

```

```

# Additional compiler flags for C only
CFLAGS = -std=c99

# Additional compiler flags for C++ only
CPPFLAGS = -fno-exceptions

# Linker flags
LDFLAGS =

# Enable creation of a map file
# LDFLAGS += -Wl,-Map,$(PRG).map

# Enable floating-point support in printf
#LDFLAGS += -Wl,-u,vfprintf -lprintf_flt -lm

# Enable floating-point support in scanf
#LDFLAGS += -Wl,-u,vscanf -lscanf_flt -lm

# Enable automatic removal of unused code
FLAGS += -ffunction-sections -fdata-sections
LDFLAGS += -Wl,--gc-sections

# Collect fuse operations for avrdude
ifdef FUSE
    FUSES += -U fuse:w:$(FUSE):m
endif
ifdef LFUSE
    FUSES += -U lfuse:w:$(LFUSE):m
endif
ifdef HFUSE
    FUSES += -U hfuse:w:$(HFUSE):m
endif
ifdef EFUSE
    FUSES += -U efuse:w:$(EFUSE):m
endif
ifdef FUSE0
    FUSES += -U fuse0:w:$(FUSE0):m
endif
ifdef FUSE1
    FUSES += -U fuse1:w:$(FUSE1):m
endif
ifdef FUSE2
    FUSES += -U fuse2:w:$(FUSE2):m
endif
ifdef FUSE3
    FUSES += -U fuse3:w:$(FUSE3):m
endif
ifdef FUSE4
    FUSES += -U fuse4:w:$(FUSE4):m
endif
ifdef FUSE5
    FUSES += -U fuse5:w:$(FUSE5):m
endif
ifdef FUSE6
    FUSES += -U fuse6:w:$(FUSE6):m

```

```

endif
ifdef FUSE7
    FUSES += -U fuse7:w:$(FUSE7):m
endif

# Workaround: avrdude does not support attiny13a
ifeq ($(MCU),attiny13a)
    AVRDUDE MCU = attiny13
else
    AVRDUDE MCU = $(MCU)
endif

# Default sections
ifeq ($(EEPROM),yes)
all: code eeprom
else
all: code
endif

# Program code
code: $(PRG).hex

# Eeprom content
eeprom: $(PRG)_eeprom.hex

# Generated code listing
list: $(PRG).lst

# Remove all generated files
clean:
    rm -f $(OBJ) $(PRG).hex $(PRG).elf $(PRG).lst $(PRG).map \
$(PRG)_eeprom.hex

# Program flash memory with or without eeprom
ifeq ($(EEPROM),yes)
program: code eeprom
    $(AVRDUDE) -p $(AVRDUDE MCU) $(AVRDUDE HW) -U flash:w:$(PRG).hex:i -U
eeprom:w:$(PRG)_eeprom.hex:i
else
program: code
    $(AVRDUDE) -p $(AVRDUDE MCU) $(AVRDUDE HW) -U flash:w:$(PRG).hex:i
endif

# Program fuses
fuses:
    $(AVRDUDE) -p $(AVRDUDE MCU) $(AVRDUDE HW) $(FUSES)

%.o : %.c
    $(CC) -c $(FLAGS) $(CFLAGS) $(INCLUDES) $(LIBS) -o $@ $<

%.o : %.cpp
    $(CC) -c $(FLAGS) $(CPPFLAGS) $(INCLUDES) $(LIBS) -o $@ $<

```

```

$(PRG).elf: $(OBJ)
    $(CC) $(FLAGS) $(LDFLAGS) -o $@ $^
    $(AVR_SIZE) $(PRG).elf

$(PRG).lst: $(PRG).elf
    $(OBJDUMP) -h -S $< > $@

$(PRG).hex: $(PRG).elf
    $(OBJCOPY) -j .text -j .data -O ihex $< $@

$(PRG)_eeprom.hex: $(PRG).elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O ihex $< $@

```

Öffne die Vorlage in einem Texteditor. Achte beim Editieren darauf, dass du die Tabulator-Zeichen vor den eingerückten Zeilen nicht aus versehen durch Leerzeichen ersetzt. Denn dann funktioniert das Makefile nicht mehr.

Ganz oben trägst du die richtigen Parameter für avrdude ein. Also genau die Parameter, mit denen du eben noch avrdude manuell aufgerufen hast.

Hinter MCU gehört der Name des Mikrocontrollers, also in unserem Fall attiny13. Und eine Zeile darunter trägst du hinter F_CPU die Taktfrequenz ein, mit der der Mikrocontroller läuft. Der ATtiny13 läuft standardmäßig mit 1200000 Hertz. Die ATtiny25, 45 und 85 laufen standardmäßig mit 1000000 Hertz.

Die Werte für die Fuses sollen auskommentiert sein, weil du die Fuses deiner Mikrocontroller nicht verändern sollst. Alle Projekte in diesem Buch funktionieren mit den vom Hersteller vorgegebenen Standardeinstellungen.

Nun gibt es nur noch eine Zeile, die wir anpassen müssen, und zwar die Liste der Objekt-Dateien. Jede *.c Datei wird in eine Objekt-Datei (*.o) compiliert. Wir haben nur eine, also geben wir ein:

```
OBJ = test.o
```

Damit ist die Konfiguration von Make abgeschlossen. Du kannst jetzt folgende Befehle eingeben, um Make aufzurufen:

- **make clean**

Löscht alle vom Compiler und Linker erzeugten Dateien. Du musst dieses Kommando aufrufen, wenn du das Makefile oder eine *.h Datei verändert hast. Veränderte *.c Dateien erkennt der Make hingegen automatisch.

- **make code**

Compiliert und Linkt das Programm zu einer hex Datei.

- **make list**

Erzeugt ein Assembler-Listing (*.lst Datei), das ist die für menschen lesbare Darstellung des Maschinen-Codes. Braucht man gelegentlich zur Fehleranalyse.

- **make program**

Lädt das Programm in den Mikrocontroller hoch.

- **make fuses**

Verändert die Fuses des Mikrocontrollers entsprechend den Vorgaben im Makefile. Geht nur über ISP.

Um ein Programm in den Mikrocontroller zu laden, gebe ich immer „make clean program“ ein. Dass dazu auch ein Kompilier-Vorgang nötig ist, weiß make schon, weil das im Makefile drin steht. Aber du kannst auch gerne „make clean code program“ eingeben, wenn es dir lieber ist.

Makefiles sind keine exotische Sonderlocke, sondern ein uralter und dennoch immer noch aktueller Standard in der Softwareentwicklung. Sogar der Linux Kernel wird immer noch mit einem Makefile compiliert.

Du hast nun gelernt, wie man AVR Mikrocontroller Projekte ohne grafische Entwicklungsumgebung erstellt und benutzt. Im Internet wirst du Anleitungen finden, wie man Makefiles mit aktuellen Entwicklungsumgebungen verwendet. In diesem Buch beschränken wir uns allerdings auf die Kommandozeile und den Texteditor deines Vertrauens.

3 Eieruhr

Am Beispiel einer batteriebetriebenen Eieruhr lernst du die Energiesparfunktionen der AVR Mikrocontroller kennen.

Die Eieruhr ist ein Zeitmesser mit fester Vorgabe von 3 Minuten. Er eignet sich prima als kleines Geschenk für Mütter und Omas. Natürlich gehören die Teile dann fest verlötet in ein schönes handliches Gehäuse.

Material:

- 1 Mikrocontroller ATtiny13A-PU (alternativ ATtiny25, 45 oder 85)
- 1 Kondensator 100nF
- 1 Piezokeramischer Schallwandler mit Gehäuse
- 1 Leuchtdiode in rot
- 1 Widerstand 220 Ohm ¼ Watt
- 1 Kurzhubtaster
- 1 ISP Programmieradapter
- 1 Steckbrett und Kabel
- 1 Batteriekasten mit 3 Zellen (ca. 3,6V)
oder 3V Knopfzelle mit Halter, z.B. CR2032

3.1 Informationen zur Bauteil-Wahl

Piezo-Keramische Schallwandler sehen so aus:



Die Variante mit Gehäuse ist zu bevorzugen, sonst musst du selbst noch einen Resonanzkörper drumherum basteln. Denn das nackte Metallplättchen mit Kristall wird ansonsten nur sehr leise Geräusche erzeugen.

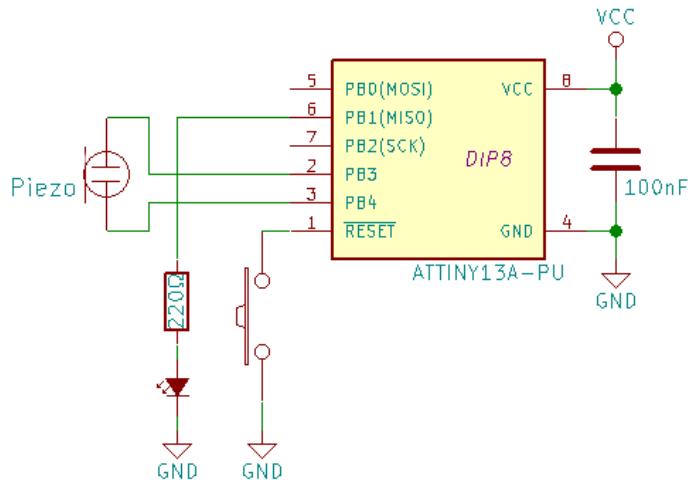
Bei der Wahl der Leuchtdiode muss die Batteriespannung beachtet werden. Denn die Batteriespannung muss auf jeden Fall deutlich höher sein, als die Spannung der LED, sonst leuchtet sie nicht.

Weisse Leuchtdioden brauchen zum Beispiel in der Regel mehr als 3 Volt. Rote Leuchtdioden leuchten schon ab 1,6 Volt, während die grünen wenigstens 2 Volt benötigen. Für den Betrieb an einer 3V Knopfzelle ist die rote LED also die beste Wahl.

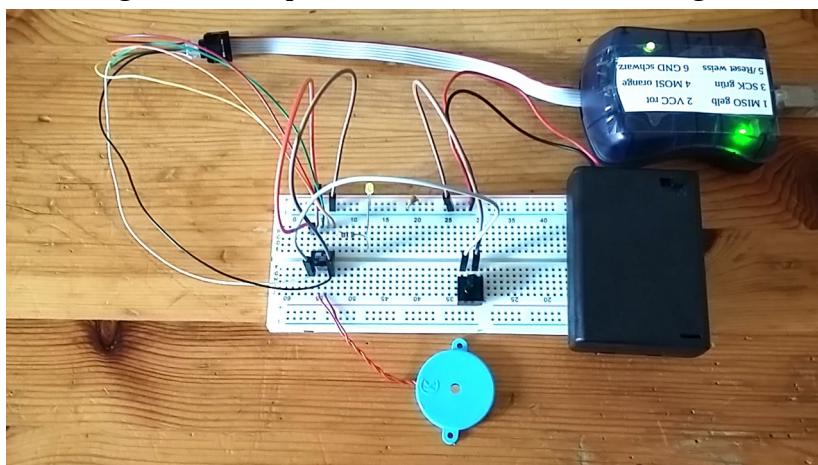
Die Größe der Leuchtdiode ist hingegen egal, weil weder Helligkeit noch Betriebsspannung von der Größe abhängen.

3.2 Schaltung

Stecke die Bauteile gemäß dem folgenden Schaltplan zusammen.



Um den Mikrocontroller zu programmieren, kannst du wahlweise die Platine aus Band 1 verwenden oder du verbindest den Programmieradapter mit losen Litzen wie im folgenden Foto.



Der Taster ist mit dem Reset-Eingang verbunden, er wird also ohne eine Zeile Programmcode bereits funktionieren.

3.3 Hardware Konfiguration

Lade dir meine „Hello World“ Vorlage für Attiny Mikrocontroller von meiner Homepage herunter. In dem Makefile trägst du die Einstellung deines Programmieradapters ein, sowie den Mikrocontroller Typ. Wir werden die Fuses wieder so belassen, wie der Chip verkauft wurde. Die Taktfrequenz des ATTtiny13 ist daher 1,2Mhz (bzw. 1Mhz beim ATTiny 25, 45 oder 85).

```
# Programmer hardware settings for avrdude
AVRDUDE_HW = -c avrispmkII -P usb -B16

# Name of the program without extension
PRG = Eieruhr

# Microcontroller type and clock frequency
MCU = attiny13
F_CPU = 1200000
```

In der Liste der Objekte sollst du die serielle Konsole entfernen, die brauchen wir nicht:

```
# Objects to compile
OBJ = driver/serialconsole.o driver/systemtimer.o main.o
```

Lösche die Datei hardware.h, für die Eieruhr wird sie nicht benötigt.

3.4 Piepen

Zuerst schreiben wir eine kleine Funktion, die den Schallwandler zum Piepen anregt. Damit er möglichst laut wird, verwenden wir den sogenannten Brücken-Betrieb. Dazu brauchen wir zwei I/O Pins.

Um die Membran nach oben zu bewegen, legen wir den linken Anschluss auf High Pegel und den rechten Anschluss auf Low Pegel:

```
PORTB |= (1<<PB3);
PORTB &= ~(1<<PB4);
```

```
High o-----[____]-----o Low
```

Stromrichtung: →

Um die Membran nach unten zu bewegen, legen wir den linken Anschluss auf Low Pegel und den rechten Anschluss auf High Pegel:

```
PORTB &= ~(1<<PB3);
PORTB |= (1<<PB4);
```

```
Low o-----[____]-----o High
```

Stromrichtung: ←

Da sich Piezo Kristalle elektrisch gesehen wie Kondensatoren verhalten, fließt der Strom immer nur für einen kurzen Moment. Wir können die beiden I/O Pins nach den Tonausgabe einfach in beliebigem Zustand belassen. Unabhängig vom Pegel der I/O Pins wird langfristig kein Strom mehr fließen.

Jetzt brauchen wir Stück Programm, das den Schallwandler zu Schwingungen anregt. Leider hat der ATtiny13 nur einen Timer, und der ist bereits belegt. Wir müssen die Töne daher quasi „Zu Fuß“ erzeugen, indem unser Programm die I/O Pins direkt ansteuert. Und zwar so (ganze Datei main.c):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/systemtimer.h"

#define FREQUENZ 4000

void piepen(void)
```

```

{
    cli();
    for (int i=0; i<500; i++)
    {
        PORTB |= (1<<PB3);
        PORTB &= ~(1<<PB4);
        _delay_us(1000000/2/FREQUENZ);

        PORTB &= ~(1<<PB3);
        PORTB |= (1<<PB4);
        _delay_us(1000000/2/FREQUENZ);
    }
    sei();
}

int main(void)
{
    initSystemTimer();

    // Ausgänge einstellen
    DDRB |= (1<<PB1)+(1<<PB3)+(1<<PB4);

    for (uint8_t i=0; i<20; i++)
    {
        piepen();
        _delay_ms(200);
    }
}

```

Der Befehl cli() deaktiviert störende Unterbrechungen durch den Systemtimer. Der Befehl sei() reaktiviert den Timer anschließend wieder. Du kannst ja mal den Unterschied ausprobieren, wie der Ton ohne diese beiden Befehle klingt.

3.5 Zeitmessung

Die Eieruhr soll nach dem Start drei Minuten lang blinken und dann das Ende der Zeit durch Piepen melden.

Der Systemtimer arbeitet normalerweise mit 16 Bit Zahlen. Er kann daher bis zu 65536 Millisekunden, also kaum mehr als eine Minute messen. Ändere in der Datei systemtimer.h die Definition von timer_t auf einen 32 Bit Integer, dann können wir sehr viel längere Zeiten messen:

```

// Size of the timer counter.
// You may change this to 32bit, if needed.
#define timer_t uint32_t

```

Als nächstes tausche die main() Funktion durch folgende aus:

```

int main(void)
{
    initSystemTimer();

    // Ausgänge einstellen
    DDRB |= (1<<PB1)+(1<<PB3)+(1<<PB4);

    piepen();

    // Drei Minuten warten
    while(milliseconds()<180000)
    {
        // LED Blinken lassen
    }
}

```

```

PORTB |= (1<<PB1);
_delay_ms(100);

PORTB &= ~(1<<PB1);
_delay_ms(1900);
}

// 20 mal Piepen
for (uint8_t i=0; i<20; i++)
{
    piepen();
    _delay_ms(200);
}
}

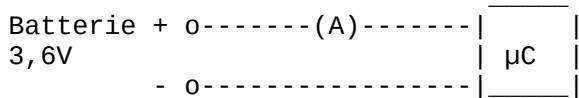
```

Gebe jetzt „make clean code“ ein, um das ganze Programm neu zu compilieren. Dadurch erzwingst du, dass der Quelltext vom Systemtimer neu compiliert wird, was nach der Modifikation von systemtimer.h unbedingt notwendig ist. Lade das Programm dann mit „make program“ in den Mikrocontroller.

Wenn du nun das Programm durch Druck auf den Reset Taster startest, wird der Schallwandler einmal piepen. Dann wird die Leuchtdiode 180 Sekunden lang blinken, und dann wird der Piepser 20x ertönen, um zu signalisieren, dass die Zeit abgelaufen ist.

3.6 Stromaufnahme Reduzieren

Stecke nun den ISP Programmieradapter ab und messe die Stromaufnahme mit deinem digital Multimeter.



Das Symbol (A) soll dein Multimeter darstellen. Stelle es auf den 20 mA Messbereich und kontrolliere die Stromaufnahme der Schaltung. Sie müsste ungefähr 1 mA betragen, wenn die LED aus ist und deutlich mehr, wenn die LED leuchtet. Der größte Stromverbraucher ist die LED, deswegen lassen wir sie immer nur kurzzeitig leuchten. Nach Ablauf der 180 Sekunden bleibt die Stromaufnahme konstant bei ungefähr 1 mA.

Eine CR2032 Knopfzelle würde somit etwa eine Woche lang halten. Ein Satz Mignon Batterien würde ungefähr drei Monate halten. Nicht gerade zufriedenstellend, denn andere Küchen-Timer laufen mit einem Satz Batterien mehrere Jahre.

Das kannst du auch, ich helfe dir dabei. Füge folgende Zeilen am Anfang der Datei main.c ein:

```
#include <avr/power.h>
#include <avr/sleep.h>
```

Diese Dateien enthalten Funktionen, um diverse Stromspar-Optionen zu aktivieren. Die nächsten Kapitel benutzen sie.

3.6.1 Unbenutzte Eingänge

Unbenutzte Eingänge neigen dazu, auf Radiowellen zu reagieren. Sie flattern dann mit hoher Frequenz wild hin und her, was die Stromaufnahme unnötig in die Höhe treibt. Aktiviere für alle unbenutzten I/O Pins die internen Pull-Up Widerstände, dadurch werden sie zuverlässig auf High Pegel gehalten. Füge dazu eine Zeile ein:

```
// Ausgänge einstellen
DDRB |= (1<<PB1)+(1<<PB3)+(1<<PB4);
```

```
PORTB |= (1<<PB0)+(1<<PB2);
```

Ich hatte dadurch schon einmal etwa 1 mA eingespart. Je nach Betriebsumgebung kann es allerdings auch passieren, dass der Spar-Effekt nicht so groß ist. Trotzdem solltest du die Pull-Up Widerstände vorsichtshalber aktivieren.

3.6.2 ADC Deaktivieren

Beim ATtiny13A gibt es ein neues Register mit dem Namen PRR, womit man den ADC und den Timer0 komplett abschalten kann. Damit der Compiler das neue Register erkennt und die entsprechenden Zugriffsfunktionen bereit stellt, musst du im Makefile den genauen Mikrocontroller-Typ „attiny13a“ (nicht „attiny13“) eintragen. Denn der alte ATtiny13 (ohne „A“) hatte diese Funktion noch nicht.

Den Timer brauchen wir, aber den ADC können wir wirklich abschalten, weil wir ihn ohnehin nicht benutzen. Füge dazu ganz am Anfang der main() Funktion diese Zeile ein:

```
power_adc_disable();
```

Dadurch sinkt die Stromaufnahme laut Datenblatt um ungefähr 0,2 mA.

3.6.3 Power-Down Modus

Wenn du deiner Eieruhr keinen an/aus Schalter spendieren möchtest, musst du dafür sorgen, dass sie sich nach Ablauf der 3 Minuten von selbst komplett abschaltet und somit keinen Strom mehr verbraucht.

Alle AVR Mikrocontroller können das, du musst sie nur in den Power-Down Modus versetzen. Dadurch wird der Taktgeber angehalten, das Programm bleibt stehen und der Mikrocontroller beginnt einen ewigen Dornröschen-Schlaf.

Es gibt nur drei Möglichkeiten, den Mikrocontroller aus dem Power-Down Modus zu erwecken, und zwar durch

- Ein externes Interrupt-Signal (was nicht passieren wird, weil wir keinen externen Interrupt aktiviert haben).
- Durch einen Watchdog-Interrupt (was nicht passieren wird, weil wir den Watchdog nicht aktiviert haben).
- Durch Druck auf den Reset Taster.

Dazu fügen wir ganz am Ende der main() Funktion diese zwei Zeilen ein:

```
int main(void)
{
    ...
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    sleep_mode();
}
```

Am Ende des Programmes, wenn die Uhr abgelaufen ist, legt sich der Mikrocontroller schlafen und verbraucht fast keinen Strom mehr. Die Batterie vergammelt von selbst schneller, als dass der Mikrocontroller sie entlädt. Ein ein/aus Schalter ist nun wirklich nicht mehr notwendig.

3.6.4 Taktfrequenz Ändern

Während die 3 Minuten ablaufen, nimmt der Mikrocontroller ungefähr 1 mA Strom auf. Dazu kommt noch die Stromaufnahme der blinkenden Leuchtdiode.

In dieser Zeit macht der Mikrocontroller praktisch nicht viel mehr, als mit 1,2 Millionen Schritten pro Sekunde im Kreis zu laufen. Das ist pure Energieverschwendungen. Durch Senkung der Taktfrequenz lässt sich die Stromaufnahme in dieser Zeit erheblich reduzieren.

Wir benutzen den Mikrocontroller zur Zeit mit 1/8 der Oszillatorkreisfrequenz. Per Software kann der Clock-Prescaler (Takt-Vorteiler) jedoch auch auf andere Werte gestellt werden, unter anderem auf 256.

Dann haben wir eine effektive Taktfrequenz von nur noch $9,6 \text{ MHz} : 256 = 37,5 \text{ kHz}$. Das ist ziemlich wenig, dementsprechend gering wird die Stromaufnahme sein.

Zum Ändern der Taktfrequenz dient die Funktion `clock_prescale_set()`, die auf das CLKPR Register zugreift. Lies dir dazu das Kapitel „System Clock Prescaler“ im Datenblatt des ATtiny13 durch.

Füge folgende Befehle vor und hinter der 3-Minuten Warteschleife ein:

```
clock_prescale_set(clock_div_256);
while(milliseconds()<180000)
{
    // LED Blinken lassen
    PORTB |= (1<<PB1);
    _delay_ms(100);

    PORTB &= ~(1<<PB1);
    _delay_ms(1900);
}
clock_prescale_set(clock_div_8);
```

Somit wird die Taktfrequenz des Mikrocontrollers vorübergehend auf den niedrigste mögliche Frequenz von 37,5 kHz gestellt und nach der Warteschleife wieder zurück auf die normalen 1,2 MHz.

Probiere das Programm aus. Du wirst bemerken, dass die LED jetzt viel zu langsam blinkt und die 3 Minuten sind ebenfalls viel länger geworden. Ist ja auch logisch, wir haben den Prescaler von 8 auf 256 geändert, also läuft nun alles entsprechend 32 mal langsamer ab.

Wir korrigieren die Zeiten einfach direkt im Quelltext, indem wir die Zahlen durch 32 teilen:

```
clock_prescale_set(clock_div_256);
while(milliseconds()<180000/32)
{
    // LED Blinken lassen
    PORTB |= (1<<PB1);
    _delay_ms(100/32);

    PORTB &= ~(1<<PB1);
    _delay_ms(1900/32);
}
clock_prescale_set(clock_div_8);
```

Probiere das Programm aus. Dabei stößt du wahrscheinlich auf ein unerwartetes Problem: Die Kommunikation zwischen ISP Programmieradapter und Mikrocontroller funktioniert nicht mehr. Und zwar deswegen, weil der Programmieradapter angesichts der reduzierten Taktfrequenz nun zu schnell ist. Dazu habe ich zwei Lösungsvorschläge:

1. Verbinde den Reset Pin durch eine Drahtbrücke mit GND. Schalte die Stromversorgung erst danach ein. Dadurch verhindert du, dass dein Programm startet und somit wird die Taktfrequenz nicht herab gesetzt. Lade das Programm in den Mikrocontroller, danach kannst du die Drahtbrücke wieder entfernen.

- Alternativ kannst du im Makefile bei AVRDUDE_HW den Parameter -B 200 einstellen. Das macht den ISP Programmieradapter so langsam, dass die Kommunikation trotz niedriger Taktfrequenz funktioniert.

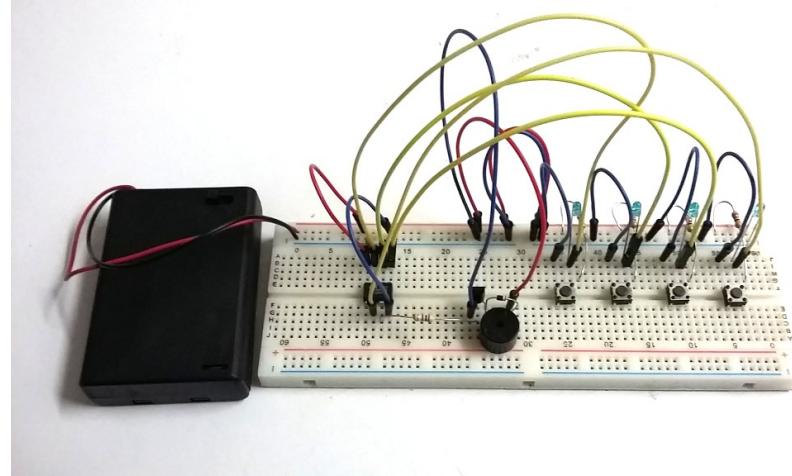
Jetzt kannst du das Programm erneut hochladen, dieses mal wird es klappen. Du wirst sehen, dass das Timing jetzt dank unserer Änderung im Quelltext wieder stimmt.

Entferne den ISP Programmieradapter und kontrolliere nochmal die Stromaufnahme der Schaltung. Sie beträgt jetzt in den Phasen, wo die LED dunkel ist, nur noch etwa 0,1 mA.

Wenn du die Eieruhr jetzt noch auf eine kleine Platine lötest und in ein schönes Gehäuse einbaust, hast du ein passendes Geschenk für den nächsten Muttertag.

4 Quiz-Buzzer

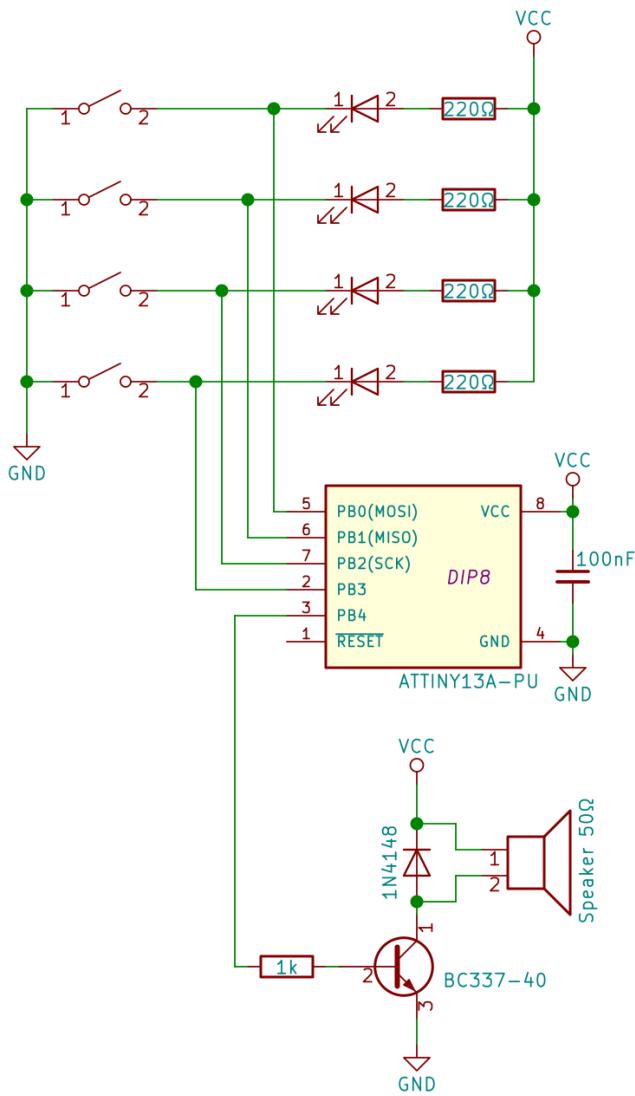
In diesem Kapitel bauen wir einen Buzzer, wie man ihn aus Quiz-Shows im Fernseher kennt. Jeder der vier Kandidaten hat einen Knopf, den er schnell drücken muss, wenn er eine Frage beantworten kann. Dann ertönt ein Summer und Leuchtdioden zeigen an, welcher Kandidat als erster gedrückt hat.



Material:

- 1 Mikrocontroller ATtiny13A-PU (alternativ ATtiny25, 45 oder 85)
- 1 Kondensator 100nF
- 4 Kurzhubtaster
- 4 Leuchtdioden
- 4 Widerstände 220 Ohm ¼ Watt
- 1 Widerstand 1k Ohm ¼ Watt
- 1 Transistor BC337-40
- 1 Diode 1N4148
- 1 Signalgeber ohne Ansteuerung mit mindestens 32 Ohm
(z.B. AL-60P06 oder AL-60P12)
- 1 Batteriekasten mit 3 Zellen (ca. 3,6V)

4.1 Schaltung



Die Ports PB0 bis PB3 werden zunächst als Eingang verwendet, um die Taster abzufragen. Sobald ein Taster gedrückt wurde, wird der entsprechende Pin als Ausgang mit Low Pegel umgeschaltet, damit die LED dauerhaft leuchtet.

Dieses mal verwenden wir einen elektromagnetischen Signalgeber anstatt einen Piezo Kristall. In seinem Innern befindet sich eine federnd befestigte Metallplatte, die von einem Elektromagneten angezogen wird. Er ist deutlich lauter als ein Piezo Schallwandler, verbraucht aber auch viel mehr Strom. Der Ausgang des Mikrocontrollers wäre nicht stark genug, um den benötigten Strom zu liefern. Daher muss der Signalgeber über einen Transistor angesteuert werden. Das Programm wird den Transistor 2400 mal pro Sekunde ein und aus schalten, um einen lauten Signalton zu erzeugen. Die Freilaufdiode am Signalgeber beschützt den Transistor vor Überspannung, welche die Spule des Signalgebers beim Abschalten des Stromes erzeugen würde.

Wenn dein Batteriekasten keinen ein/aus Schalter hat, möchtest du eventuell einen Schalter in die Zuleitung einbauen oder einen Reset-Taster hinzufügen, um die Anzeige für die nächste Quizfrage zurücksetzen zu können.

4.2 Programm

Ich schlage vor, dass du zunächst dieses Programm abschreibst. Es besteht nur aus einer einzigen Datei namens main.c:

```
#include <stdint.h>
#include <util/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/power.h>
#include <avr/sleep.h>

// Leere Interrupt-Routine
ISR(PCINT0_vect)
{
}

// Den Summer ansteuern
void summen(void)
{
    // Schwingung mit 2,5kHz erzeugen
    for (int i=0; i<3000; i++)
    {
        PORTB ^= (1<<PB4);
        _delay_us(200);
    }

    // Summer-Pin auf Low schalten = Strom aus.
    PORTB &= ~(1<<PB4);
}

int main(void)
{
    // PB4 ist ein Ausgang (für den Summer)
    DDRB = 0b00010000;

    // Bei den Eingängen PB0 .. PB3 den Pull-Up Widerstand einschalten
    PORTB = 0b00001111;

    // Den ADC deaktivieren, brauchen wir nicht
    power_adc_disable();

    // Pin-Change Interrupt für die 4 Taster erlauben
    GIMSK = (1<<PCIE);
    PCMSK = 0b00001111;
    sei();

    while(1)
    {
        // Schlafen, bis eine Taste gedrückt wird
        set_sleep_mode(SLEEP_MODE_PWR_DOWN);
        sleep_mode();

        // Tasten abfragen
        uint8_t tasten=PINB & 0b00001111;

        // Wenn irgendeine Taste gedrückt wurde...
        if (tasten != 0b00001111)
        {
    }
```

```

// Wenn Taster 0 gedrückt wurde,
// schalte dessen LED ein
if ((tasten & (1<<PB0)) == 0)
{
    PORTB &= ~(1<<PB0);
    DDRB |= (1<<PB0);
}

// Wenn Taster 1 gedrückt wurde,
// schalte dessen LED ein
if ((tasten & (1<<PB1)) == 0)
{
    PORTB &= ~(1<<PB1);
    DDRB |= (1<<PB1);
}

// Wenn Taster 2 gedrückt wurde,
// schalte dessen LED ein
if ((tasten & (1<<PB2)) == 0)
{
    PORTB &= ~(1<<PB2);
    DDRB |= (1<<PB2);
}

// Wenn Taster 3 gedrückt wurde,
// schalte dessen LED ein
if ((tasten & (1<<PB3)) == 0)
{
    PORTB &= ~(1<<PB3);
    DDRB |= (1<<PB3);
}

// Und nun einmal schön laut summen
summen();

// Jetzt das Programm anhalten, damit
// keiner mehr drücken kann
while (1);

}
}
}

```

Das Programm konfiguriert zuerst den Pin PB4 als Ausgang, damit wir den Summer über den Transistor ansteuern können. Hier ist Vorsicht geboten: Wenn der Summer zu lange Dauer-Strom erhält, brennt er durch. Standardmäßig ist der Pin auf Low Pegel, so dass kein Strom fließt.

```
DDRB = 0b00010000;
```

Bei den vier Eingängen werden die internen Pull-Up Widerstände des Mikrocontrollers aktiviert, damit sie bei nicht gedrücktem Taster eindeutig auf High Pegel liegen. Wenn jemand den Taster drückt, wird der Eingang auf Low gehen.

```
PORTB = 0b00001111;
```

Als Nächstes wird der Analog-zu-digital-Konverter (ADC) ausgeschaltet, was für lange Haltbarkeit der Batterie vorteilhaft ist. Wir brauchen den ADC nicht.

```
power_adc_disable();
```

Danach aktivieren wir den Pin-Change Interrupt für die vier Eingänge mit den Tastern. Der Pin-Change Interrupt wird den Mikrocontroller aus dem Tiefschlaf erwecken.

```
GIMSK = (1<<PCIE);  
PCMSK = 0b00001111;  
sei();
```

Die Interrupt-Routine wird immer dann aufgerufen, wenn sich der Signalpegel an einem der vier Eingänge verändert - also wenn jemand seinen Taster drückt oder loslässt. Wir werten die Taster allerdings im Hauptprogramm aus, deswegen darf die Interrupt-Routine leer bleiben. Einfach weglassen wäre jedoch falsch, denn dann würde der Mikrocontroller im Falle des Interrupts irgend etwas undefiniertes tun.

```
ISR(PCINT0_vect)  
{  
}
```

Wir brauchen den Pin-Change Interrupt lediglich dazu, um den Mikrocontroller aus seinem Tiefschlaf zu erwecken.

Nach dieser Initialisierung wird der Mikrocontroller schlafen gelegt. Er verbraucht nun fast keinen Strom mehr.

```
set_sleep_mode(SLEEP_MODE_PWR_DOWN);  
sleep_mode();
```

An dieser Stelle hält die Programmausführung an, bis jemand eine Taste drückt. Dann führt der Mikrocontroller die leere Interrupt-Routine aus und dann setzt er das Programm fort. Wir lesen zuerst den aktuellen Zustand der Tasten ein:

```
uint8_t tasten=PINB & 0b00001111;
```

und schalten danach die LED ein, deren zugehörige Taste gedrückt wurde. Der gleiche Code wird vier mal für die Ports PB0 bis PB3 wiederholt:

```
if ((tasten & (1<<PB0)) == 0)  
{  
    PORTB &= ~(1<<PB0);  
    DDRB |= (1<<PB0);  
}
```

Anschließend wird der Signalgeber angesteuert, damit er lautstark Piepst:

```
void summen(void)  
{  
    // Schwingung mit ca. 2,4kHz erzeugen  
    for (int i=0; i<3000; i++)  
    {  
        PORTB ^= (1<<PB4);  
        _delay_us(200);  
    }  
  
    // Summer-Pin auf Low schalten = Strom aus.  
    PORTB &= ~(1<<PB4);  
}
```

Der gelb markierte Befehl schaltet bei jedem Schleifendurchlauf den Port PB4 um, so das er immer abwechselnd zwischen High und Low wechselt. Zum Schluss wird der Ausgang auf Low geschaltet, damit der Signalgeber nicht durchbrennt. Wenn die Schleife eine gerade Anzahl von Wiederholungen hat (was mit 3000 hier der Fall ist), kann man den letzten Befehl auch weglassen. Aber sicher ist sicher, ein Befehl mehr schadet nicht.

Nach dem Signalton wird das Programm durch eine leere Endlosschleife angehalten:

```
while (1);
```

Dadurch wird es alle weiteren Tastendrücke ignorieren. Der erste Kandidat, der deinen Taster drückt, dessen LED wird aufleuchten. Wer zu spät drückt, der wird ignoriert. Um das Gerät zurück zu setzen, schaltest du kurz den Batteriekasten aus oder drückst den Reset-Taster.

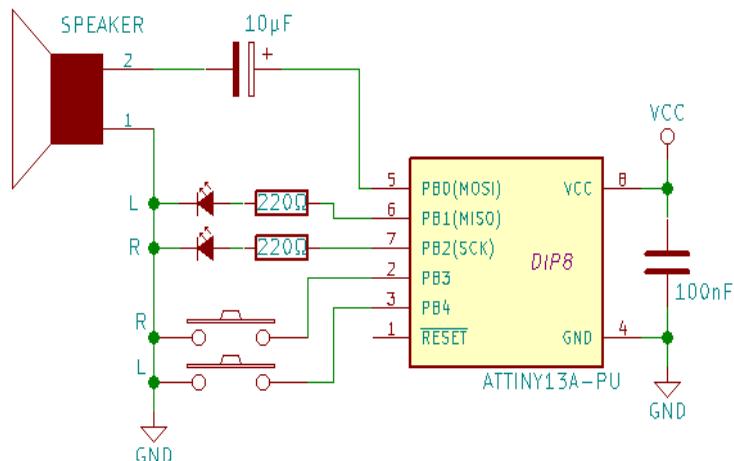
5 Fahrrad-Blinker

Wir werden eine Modell von einem Fahrrad-Blinker bauen. Dazu brauchst du folgende Bauteile:

- 1 Mikrocontroller ATtiny13A (alternativ ATtiny25, 45 oder 85)
- 1 Kondensator 100 nF
- 1 Kondensator 10 µF
- 2 Leuchtdioden gelb
- 2 Widerstände 220 Ohm ¼ Watt
- 2 Kurzhubtaster
- 1 Kleiner Lautsprecher, möglichst mit Gehäuse
- 1 Steckbrett und Kabel
- 1 Batteriekasten mit 3 Zellen (ca. 3,6 V)

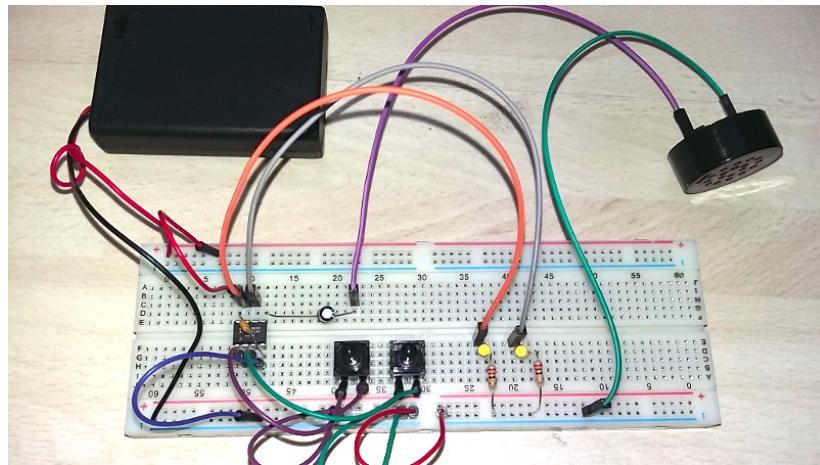
5.1 Schaltung

Die Schaltung wird zwei Taster bekommen, mit denen man den linken und den rechten Blinker einschalten kann. Zwei Leuchtdioden dienen als Blinker und ein Lautsprecher soll dabei Klick-Klack machen.



Die Leuchtdioden sind so angeschlossen, dass sie an gehen, wenn der Mikrocontroller ein High-Signal ausgibt. Die Eingänge mit den Tastern werden wir so konfigurieren, dass sie von den internen Pull-Up Widerständen hoch gezogen werden. Durch Druck auf eine Taste geben wir hingegen ein Low-Signal auf den Eingang.

Auf dem Steckbrett aufgebaut sieht der Aufbau so aus:



In diesem Plan habe ich wieder die Leitungen zum Programmieradapter ausgelassen.

Wir verwenden dieses mal einen kleinen Lautsprecher, weil er sich besser dazu eignet, die Klick-Klack Geräusche wiederzugeben. Mit dem Piezo ginge es auch, aber der klingt nicht so gut. Der Mikrocontroller wird mit 1 oder 1,2Mhz Taktfrequenz betrieben. Das ist die Standardvorgabe, also brauchen wir die Fuses nicht zu verändern.

5.2 Programm

Beginne mit der Hello-World Vorlage für die kleinen ATtiny Mikrocontroller, die du auf meiner Homepage findest.

Im Makefile stelle den Mikrocontroller-Typ und seine Taktfrequenz (1200000 Hz für den ATtiny13 oder 1000000 für den ATtiny 25, 45 und 85) ein. Dann entfernst du die Datei driver/serialconsole.c weil sie nicht benötigt wird. In der Datei main.c entfernst du den entsprechenden include Befehl von der serielle Konsole. In die Datei hardware.h trägst du Makros ein, mit denen das Programm später auf die Eingänge und Ausgänge zugreifen wird:

```
#ifndef _HARDWARE_H_
#define _HARDWARE_H_

#include <avr/io.h>

// Initialisierung der Ausgänge
#define CONFIG_OUTPUTS { DDRB |= (1<<PB0) + (1<<PB1) + (1<<PB2); }

// Linke LED an PB2
#define LED_LEFT_ON    { PORTB |= (1<<PB2); }
#define LED_LEFT_OFF   { PORTB &= ~(1<<PB2); }

// Rechte LED an PB1
#define LED_RIGHT_ON   { PORTB |= (1<<PB1); }
#define LED_RIGHT_OFF  { PORTB &= ~(1<<PB1); }

// Lautsprecher an PB0
#define SPEAKER_HIGH   { PORTB |= (1<<PB0); }
#define SPEAKER_LOW    { PORTB &= ~(1<<PB0); }

// Initialisierung der Eingänge
#define CONFIG_INPUTS  { PORTB |= (1<<PB3) + (1<<PB4); }

// Linker Taster an PB3
#define BUTTON_LEFT_PRESSED ((PINB & (1<<PB3))==0)
```

```

// Rechter Taster an PB4
#define BUTTON_RIGHT_PRESSED ((PINB & (1<<PB4))==0)

#endif // _HARDWARE_H_

```

5.2.1 Leuchtdioden einschalten

Für den Anfang begnügen wir uns damit, die beiden LED's einzuschalten.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/systemtimer.h"
#include "hardware.h"

int main(void)
{
    CONFIG_OUTPUTS;

    LED_LEFT_ON;
    LED_RIGHT_ON;
}

```

Die ganzen Includes stammen aus der Hello-World Vorlage. Du brauchst sie momentan noch gar nicht alle aber, aber sie schaden auch nicht.

Versuche, das Programm in den Mikrocontroller zu laden. Wahrscheinlich wird es fehlschlagen, weil der Lautsprecher an der MOSI Leitung stört. Er belastet die Leitung zu stark. Stecke ihn also zum Programmieren ab.

Wenn der Programmervorgang nun immer noch fehlschlägt, könnte es unter Umständen sein, dass die beiden Leuchtdioden auch raus müssen. Dann hast du allerdings einen ziemlich schlechten Programmieradapter erwischt. Gute Programmieradapter lassen sich von Leuchtdioden mit 220 Ohm Vorwiderstand nicht stören.

Diese beiden Probleme habe ich hier absichtlich provoziert, damit du sie frühzeitig kennen und umgehen lernst. Sorge nun dafür, dass das Programm läuft. Die beiden Leuchtdioden müssen leuchten.

5.2.2 Leuchtdioden blinken

In der nächsten Ausbaustufe soll das Programm je nach Tastendruck die linke oder die rechte Leuchtdiode einige male blinken lassen. Mein Vorschlag dazu:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/systemtimer.h"
#include "hardware.h"

#define ANZAHL_BLINKEN 5

void blinken_links(void)

```

```

{
    for (uint8_t i=0; i<ANZAHL_BLINKEN; i++)
    {
        LED_LEFT_ON;
        _delay_ms(500);
        LED_LEFT_OFF;
        _delay_ms(500);
    }
}

void blinken_rechts(void)
{
    for (uint8_t i=0; i<ANZAHL_BLINKEN; i++)
    {
        LED_RIGHT_ON;
        _delay_ms(500);
        LED_RIGHT_OFF;
        _delay_ms(500);
    }
}

int main(void)
{
    CONFIG_OUTPUTS;
    CONFIG_INPUTS;

    while (1)
    {
        if (BUTTON_LEFT_PRESSED)
        {
            blinken_links();
        }

        if (BUTTON_RIGHT_PRESSED)
        {
            blinken_rechts();
        }
    }
}

```

Probiere das Programm aus. Wenn du auf die linke Taste drückst wird die linke Leuchtdiode fünf mal blinken. Wenn du auf die rechte Taste drückst wird die rechte Leuchtdiode fünf mal blinken.

5.2.3 Blinker mit Abbruch-Funktion

Es fehlt aber noch eine Möglichkeit, den Blinker frühzeitig wieder aus zu schalten. Und wenn man versehentlich auf die falsche Taste gedrückt hat, sollte man seine Wahl durch Druck auf die andere Taste korrigieren können.

Ich zeige dir nun, wie man das mit Hilfe des Systemtimers machen kann, der in der Kopiervorlage „Hello-World“ enthalten ist.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/systemtimer.h"
#include "hardware.h"

#define ANZAHL_BLINKEN 10

```

```

// Wartet bis beide Tasten mindestens 100ms lang losgelassen wurden
void warte_auf_loslassen(void)
{
    timer_t letzterImpuls=milliseconds();
    do
    {
        if (BUTTON_LEFT_PRESSED || BUTTON_RIGHT_PRESSED)
        {
            // Wartezeit verlängern
            letzterImpuls=milliseconds();
        }
    } while (milliseconds()-letzterImpuls < 100);
}

// Lässt die linke LED blinken
void blinken_links(void)
{
    timer_t warteSeit;
    for (uint8_t i=0; i<ANZAHL_BLINKEN; i++)
    {
        LED_LEFT_ON;

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_LEFT_PRESSED)
            {
                // Blinken abbrechen
                LED_LEFT_OFF;
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blinken abbrechen
                LED_LEFT_OFF;
                return;
            }
        }

        LED_LEFT_ON;

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_LEFT_PRESSED)
            {
                // Blinken abbrechen
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blinken abbrechen
                return;
            }
        }
    }
}

```

```

        }
    }

// Lässt die rechte LED blinken
void blinken_rechts(void)
{
    timer_t warteSeit;
    for (uint8_t i=0; i<ANZAHL_BLINKEN; i++)
    {
        LED_RIGHT_ON;

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blinken abbrechen
                LED_RIGHT_OFF;
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_LEFT_PRESSED)
            {
                // Blinken abbrechen
                LED_RIGHT_OFF;
                return;
            }
        }
        LED_RIGHT_OFF;

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blinken abbrechen
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_LEFT_PRESSED)
            {
                // Blinken abbrechen
                return;
            }
        }
    }
}

int main(void)
{
    CONFIG_OUTPUTS;
    CONFIG_INPUTS;
    initSystemTimer();

    while (1)
    {

```

```

if (BUTTON_LEFT_PRESSED)
{
    warte_auf_loslassen();
    blinken_links();
}

if (BUTTON_RIGHT_PRESSED)
{
    warte_auf_loslassen();
    blinken_rechts();
}
}
}

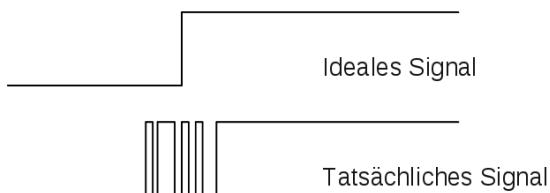
```

Probiere das Programm aus. Durch Druck auf eine Taste schaltest du den Blinker ein. Durch erneuten Druck geht er wieder aus – oder auch automatisch nach 10 mal Blinken. Falls du versehentlich auf der falschen Seite blinkst, kannst du durch Druck auf die andere Taste zur anderen Seite wechseln.

Bin diesem Programm findest du zwei wesentliche Änderungen:

Erstens wird in main() nach einem erkannten Tastendruck darauf gewartet, dass die Taste wieder losgelassen wird. Somit kann später beim Blinken erkannt werden, ob die Taste erneut gedrückt wurde.

Die Prozedur `warten_auf_loslassen()` ist etwas komplizierter, als du möglicherweise erwartet hast. Das hat mit dem Prellen der Taster zu tun. Prellen bedeutet, dass bei einem Tastendruck (und auch beim Loslassen) das elektrische Signal mehrmals zwischen Low und High hin und her wechselt.



Dieser Effekt wird von den mechanischen Eigenschaften des Tasters verursacht. Die Prozedur `warten_auf_loslassen()` wartet daher ein bisschen länger. Erst wenn beide Taster 100ms lang losgelassen sind (und nicht mehr prellen) wird die Funktion beendet.

Die zweite Änderung ist, dass ich die Aufrufe von `_delay_ms(500)` durch Warteschleifen ersetzt habe. Die Warteschleifen ermitteln die verstrichene Zeit mit Hilfe des Systemtimers. In den Warteschleifen habe ich dann Befehle eingefügt, die auf erneutes Drücken der Tasten reagieren.

5.2.4 Blinker mit Klick-Klack

Jetzt wollen wir das Programm noch so erweitern, dass es den Lautsprecher ansteuert. Dazu möchte ich noch ein Detail des Schaltplanes erklären.

Zwischen Mikrocontroller und Lautsprecher befindet sich ein $10 \mu\text{F}$ Kondensator. Wenn der Mikrocontroller seinen Lautsprecherausgang auf High ($=3,6 \text{ V}$) legt, wird für einen kurzen Moment Strom durch den Kondensator und den Lautsprecher fließen. Dabei lädt sich der Kondensator auf. Wenn er voll ist, fließt kein Strom mehr. Damit beschützen wir den Lautsprecher vor Dauerstrom – was ihn kaputt machen würde.

Wenn jetzt der Ausgang des Mikrocontrollers auf Low geht, entlädt sich der Kondensator wieder über den Lautsprecher. Der Strom fließt also anders herum durch den Lautsprecher.

Indem wir den Ausgang schnell zwischen High und Low wechseln lassen, können wir Geräusche erzeugen. Die Membran des Lautsprechers wird zusammen mit dem Stromfluss hin und her schwingen.

Den Klick-Ton erzeugen wir, indem wir den Lautsprecher 10 mal mit 100µS Zeitabstand hin und her schwingen lassen. Den Klack-Ton erzeugen wir, indem wir den Lautsprecher 10 mal mit 200µS Zeitabstand hin und her schwingen lassen.

Das ganze Programm sieht so aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/systemtimer.h"
#include "hardware.h"

#define ANZAHL_BLINKEN 10

// Hoher Ton
void klick(void)
{
    for (int i=0; i<10; i++)
    {
        SPEAKER_HIGH;
        _delay_us(100);
        SPEAKER_LOW;
        _delay_us(100);
    }
}

// Tiefer Ton
void klack(void)
{
    for (int i=0; i<10; i++)
    {
        SPEAKER_HIGH;
        _delay_us(200);
        SPEAKER_LOW;
        _delay_us(200);
    }
}

// Wartet bis beide Tasten mindestens 100ms lang losgelassen wurden
void warte_auf_loslassen(void)
{
    timer_t letzterImpuls=milliseconds();
    do
    {
        if (BUTTON_LEFT_PRESSED || BUTTON_RIGHT_PRESSED)
        {
            // Wartezeit verlängern
            letzterImpuls=milliseconds();
        }
    }
    while (milliseconds()-letzterImpuls < 100);
}

// Lässt die linke LED blinken
```

```

void blitzen_links(void)
{
    timer_t warteSeit;
    for (uint8_t i=0; i<ANZAHL_BLINKEN; i++)
    {
        LED_LEFT_ON;
        klick();

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_LEFT_PRESSED)
            {
                // Blitzen abbrechen
                LED_LEFT_OFF;
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blitzen abbrechen
                LED_LEFT_OFF;
                return;
            }
        }
        LED_LEFT_OFF;
        klack();

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_LEFT_PRESSED)
            {
                // Blitzen abbrechen
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blitzen abbrechen
                return;
            }
        }
    }
}

// Lässt die rechte LED blitzen
void blitzen_rechts(void)
{
    timer_t warteSeit;
    for (uint8_t i=0; i<ANZAHL_BLINKEN; i++)
    {
        LED_RIGHT_ON;
        klick();

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)

```

```

    {
        if (BUTTON_RIGHT_PRESSED)
        {
            // Blinken abbrechen
            LED_RIGHT_OFF;
            warte_auf_loslassen();
            return;
        }
        if (BUTTON_LEFT_PRESSED)
        {
            // Blinken abbrechen
            LED_RIGHT_OFF;
            return;
        }
    }

    LED_RIGHT_OFF;
    klack();

    // Warte 500ms
    warteSeit=milliseconds();
    while (milliseconds()-warteSeit < 500)
    {
        if (BUTTON_RIGHT_PRESSED)
        {
            // Blinken abbrechen
            warte_auf_loslassen();
            return;
        }
        if (BUTTON_LEFT_PRESSED)
        {
            // Blinken abbrechen
            return;
        }
    }
}

int main(void)
{
    CONFIG_OUTPUTS;
    CONFIG_INPUTS;
    initSystemTimer();

    while (1)
    {
        if (BUTTON_LEFT_PRESSED)
        {
            warte_auf_loslassen();
            blinken_links();
        }

        if (BUTTON_RIGHT_PRESSED)
        {
            warte_auf_loslassen();
            blinken_rechts();
        }
    }
}

```

Probiere das Programm wieder aus. Vergiss nicht, den Lautsprecher vor dem Übertragen des Programms abzustecken und danach wieder anzuschließen.

5.2.5 Strom-sparender Blinker

Da der Blinker mit Batterie versorgt wird, soll er im Ruhezustand möglichst wenig Strom verbrauchen. Tatsächlich können wir die Stromaufnahme des ATtiny13 so weit reduzieren, dass wir auf einen Ein/Aus Schalte verzichten kommen. Denn der Chip verbraucht im Power-Down Modus weniger als 1 Mikroampere.

Mit der Funktion set_sleep_mode(SLEEP_MODE_PWR_DOWN) stellen wir ein, dass wir diesen Power-Down Modus verwenden wollen. Anschließen können wir mit sleep_mode() den sparsamen Ruhezustand aktivieren.

Allerdings sollten wir uns auch Gedanken darüber machen, wie wir den Mikrocontroller wieder aus seinen Schlaf erwecken wollen. Dazu verwenden wir den Pin-Change-Interrupt. Wenn eine Taste gedrückt wird, ändert sich die Spannung an dem jeweiligen Eingang. Und das nutzen wir als Aufweck-Signal. Die folgenden Zeilen sind dazu nötig:

```
PCMSK |= (1<<PCINT3) + (1<<PCINT4);  
GIMSK |= (1<<PCIE);
```

Im PCMSK Register stellen wir ein, welche Pins zum Aufwecken führen sollen. Im Register GIMSK schalten wir diese Funktion ein. Da nun Interrupts aktiviert sind, müssen wir auch eine Interrupt-Routine programmieren. Das ist eine spezielle Prozedur, die beim Auftreten eines Interruptes ausgeführt wird. In unserem Fall genügt eine leere Prozedur. Sie muss aber existieren!

Das entsprechende erweiterte Programm sieht nun so aus:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <stdint.h>  
#include <util/delay.h>  
#include <avr/pgmspace.h>  
#include <avr/interrupt.h>  
#include <avr/power.h>  
#include <avr/sleep.h>  
#include "driver/systemtimer.h"  
#include "hardware.h"  
  
#define ANZAHL_BLINKEN 10  
  
// Hoher Ton  
void klick(void)  
{  
    for (int i=0; i<10; i++)  
    {  
        SPEAKER_HIGH;  
        _delay_us(100);  
        SPEAKER_LOW;  
        _delay_us(100);  
    }  
}  
  
// Tiefer Ton  
void klack(void)  
{  
    for (int i=0; i<10; i++)
```

```

    {
        SPEAKER_HIGH;
        _delay_us(200);
        SPEAKER_LOW;
        _delay_us(200);
    }
}

// Wartet bis beide Tasten mindestens 100ms lang losgelassen wurden
void warte_auf_loslassen(void)
{
    timer_t letzterImpuls=milliseconds();
    do
    {
        if (BUTTON_LEFT_PRESSED || BUTTON_RIGHT_PRESSED)
        {
            // Wartezeit verlängern
            letzterImpuls=milliseconds();
        }
    }
    while (milliseconds()-letzterImpuls < 100);
}

// Lässt die linke LED blinken
void blinken_links(void)
{
    timer_t warteSeit;
    for (uint8_t i=0; i<ANZAHL_BLINKEN; i++)
    {
        LED_LEFT_ON;
        klick();

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_LEFT_PRESSED)
            {
                // Blinken abbrechen
                LED_LEFT_OFF;
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blinken abbrechen
                LED_LEFT_OFF;
                return;
            }
        }
        LED_LEFT_OFF;
        klack();

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_LEFT_PRESSED)
            {

```

```

        // Blinken abbrechen
        warte_auf_loslassen();
        return;
    }
    if (BUTTON_RIGHT_PRESSED)
    {
        // Blinken abbrechen
        return;
    }
}
}

// Lässt die rechte LED blinken
void blinken_rechts(void)
{
    timer_t warteSeit;
    for (uint8_t i=0; i<ANZAHL_BLINKEN; i++)
    {
        LED_RIGHT_ON;
        klick();

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blinken abbrechen
                LED_RIGHT_OFF;
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_LEFT_PRESSED)
            {
                // Blinken abbrechen
                LED_RIGHT_OFF;
                return;
            }
        }
        LED_RIGHT_OFF;
        klack();

        // Warte 500ms
        warteSeit=milliseconds();
        while (milliseconds()-warteSeit < 500)
        {
            if (BUTTON_RIGHT_PRESSED)
            {
                // Blinken abbrechen
                warte_auf_loslassen();
                return;
            }
            if (BUTTON_LEFT_PRESSED)
            {
                // Blinken abbrechen
                return;
            }
        }
    }
}

```

```

}

// Leere Interrupt-Routine
ISR(PCINT0_vect)
{
    return;
}

int main(void)
{
    CONFIG_OUTPUTS;
    CONFIG_INPUTS;
    initSystemTimer();

    // Strom-sparen einrichten
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);

    // nur bei Attiny13A:
    power_adc_disable();

    // Pin-Change Interrupts zum Aufwachen erlauben
    PCMSK |= (1<<PCINT3) + (1<<PCINT4);
    GIMSK |= (1<<PCIE);

    while (1)
    {
        if (BUTTON_LEFT_PRESSED)
        {
            warte_auf_loslassen();
            blinken_links();
        }

        if (BUTTON_RIGHT_PRESSED)
        {
            warte_auf_loslassen();
            blinken_rechts();
        }

        // Nix zu tun, aus schalten
        sleep_mode();
    }
}

```

5.3 Aufbau in groß

Falls du den Blinker nun in groß aufbauen möchtest, um ihn an dein Fahrrad zu montieren, möchte ich dir einen Ratschlag geben.

Im Motorrad Zubehör Handel findest du schöne wetterfeste LED_Blinker. Die musst du allerdings umbauen, damit sie mit weniger als 12V funktionieren. Aber immerhin kommst du so an brauchbare Gehäuse heran.

Falls du die Blinker selbst bauen möchtest, dann suche dir im Elektronik Katalog Leuchtdioden heraus, die besonders hell sind. Verwende andere Vorwiderstände, so dass die Leuchtdioden mit mehr Strom (aber nicht zu viel) betrieben werden. Dann sind sie am hellsten.

Verstärke die Ausgänge des Mikrocontrollers mit NPN Transistoren (zum Beispiel BC337-40), um mehr als 40 mA Strom für die Leuchtdioden liefern zu können. Vergiss nicht, vor die Basis des Transistors einen Vorwiderstand zu schalten (siehe Band 2).

Die Zuleitungen zu den Tastern reagieren sehr empfindlich auf elektromagnetische Störfelder. Damit dein Blinker nicht versehentlich von alleine an geht, setze zusätzliche 100nF Kondensatoren ein:



In einem späteren Kapitel erkläre ich diese Entstör-Maßnahme.

6 Verkehrsampel

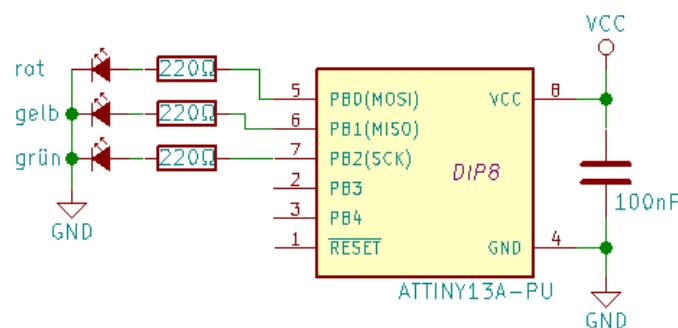
Ich werde dir in diesem Kapitel zeigen, wie man ein Modell von einer Verkehrsampel baut. Wir werden klein anfangen und sie schrittweise weiter ausbauen.

Material:

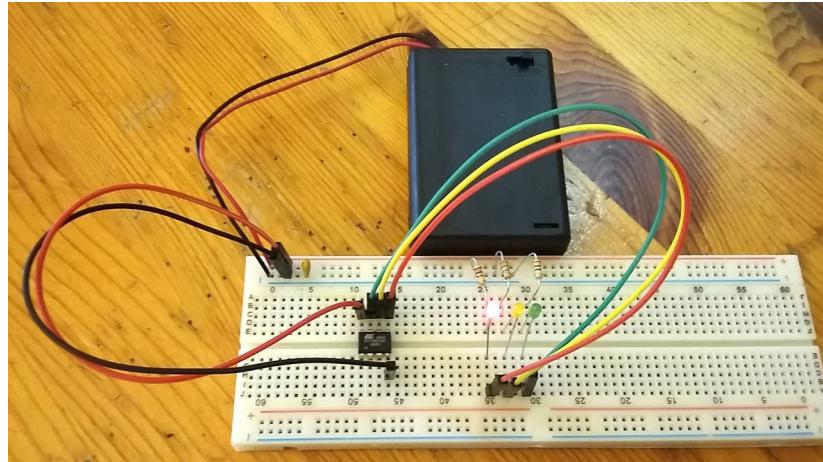
- 1 Mikrocontroller ATtiny13A-PU (alternativ ATtiny25, 45 oder 85)
- 3 Kondensatoren 100 nF
- 2 Leuchtdioden rot
- 2 Leuchtdioden gelb
- 2 Leuchtdioden grün
- 3 Dioden 1N4148
- 3 Widerstände 220 Ohm ¼ Watt
- 1 Widerstand 4,7 k Ohm ¼ Watt
- 1 Widerstand 47 k Ohm ¼ Watt
- 1 Kurzhubtaster
- 1 Spannungsregler LF33CV
- 1 Kondensator 2,2 µF
- 1 Steckbrett und Kabel
- 1 Batteriekasten mit 3 Zellen (ca. 3,6 V)
- 1 Netzteil 5 V mindestens 500 mA

6.1 Minimal Ampel

Wir schließen an den Mikrocontroller nur den 100nF Kondensator und drei Leuchtdioden an.



Die Leuchtdioden leuchten bei High Pegel. Mein Aufbau sieht so aus:



Beginne wieder mit der HelloWorld Vorlage, aber entferne dort die „serialconsole“ und den „systemtimer“. Ich möchte dass du dieses mal die Datei hardware.h löscht.

Bevor wir das Programm schreiben, sollten wir uns dessen Funktion überlegen und aufschreiben. Die Verkehrsampeln in Deutschland durchlaufen folgendes Leuchtmuster:

- 1.** rot
- 2.** rot+gelb
- 3.** grün
- 4.** gelb

(und dann wiederholen)

Die gelben Phasen sind kürzer als rot und grün. Am besten wird es wohl sein, wenn wir die Zeit für jede Phase einzeln festlegen können. Wie immer gibt es viele Möglichkeiten zum Ziel. Ich schlage zunächst folgenden geradlinigen Weg vor:

```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    // Konfiguriere drei I/O Pins als Ausgang
    DDRB = 0b00000111;

    // Endlosschleife
    while (1)
    {
        PORTB = 0b00000001; // rot
        _delay_ms(8000);
        PORTB = 0b00000011; // rot+gelb
        _delay_ms(2000);
        PORTB = 0b00000100; // grün
        _delay_ms(8000);
        PORTB = 0b00000010; // gelb
        _delay_ms(2000);
    }
}
```

Da die Leuchtdioden an den selben I/O Pins hängen, wie der Programmieradapter, flackern sie mit, während das Programm in den Mikrocontroller hochgeladen wird.

Wenn dir die Binärzahlen nicht gefallen, kannst du den Quelltext so umschreiben:

```
#include <avr/io.h>
```

```

#include <util/delay.h>

int main(void)
{
    // Konfiguriere drei I/O Pins als Ausgang
    DDRB = (1<<PB2) + (1<<PB1) + (1<<PB0);

    // Endlosschleife
    while (1)
    {
        PORTB = (1<<PB0); // rot
        _delay_ms(8000);
        PORTB = (1<<PB0)+(1<<PB1); // rot+gelb
        _delay_ms(2000);
        PORTB = (1<<PB2); // grün
        _delay_ms(8000);
        PORTB = (1<<PB1); // gelb
        _delay_ms(2000);
    }
}

```

Jetzt kann man besser erkennen, welche Ausgänge aktiviert werden. Doch ansonsten ist der Programmcode nicht wirklich übersichtlicher geworden. Wir brauchen immer noch Kommentare, um klarzustellen, wann welche Farben leuchten. Deswegen schlage ich vor, die Anschlussbelegung in Makros auszulagern.

```

#include <avr/io.h>
#include <util/delay.h>

// Drei Leuchtdioden an Port B
#define ROT    (1<<PB0)
#define GELB   (1<<PB1)
#define GRUEN  (1<<PB2)

int main(void)
{
    // Konfiguriere drei I/O Pins als Ausgang
    DDRB = ROT+GELB+GRUEN;

    // Endlosschleife
    while (1)
    {
        PORTB = ROT;
        _delay_ms(8000);
        PORTB = ROT+GELB;
        _delay_ms(2000);
        PORTB = GRUEN;
        _delay_ms(8000);
        PORTB = GELB;
        _delay_ms(2000);
    }
}

```

Das sieht besser aus, nicht wahr? Makros sind einfache Text-Ersetzungen. Der Compiler ersetzt ROT durch (1<<PB0) bevor er diese Zeile in Maschinencode übersetzt. Wenn man die Verwendung von Makros nicht übertriebt, können sie dabei helfen, denn Quelltext besser lesbar zu machen. Und man kann die Zuordnung von I/O-Pin zu externer Hardware in gewissen Grenzen konfigurierbar machen.

Du könntest diese Ampel in Groß bauen, mit richtigen 230V Lampen und Relais, um die Ausgänge des Mikrocontrollers zu verstärken. Doch eine einsame Ampel mitten auf der Straße wäre sinnlos. Dieser Gedanke führt zur nächsten Ausbaustufe.

6.2 Fußgänger Überweg

Für einen Fußgänger-Überweg brauchen wir mindestens zwei Ampeln. Nämlich eine für die Autos, und eine für die Fußgänger. Wir sollten und wieder vorher überlegen, welche Leuchtmuster notwendig sind.

Phase	Autos	Fußgänger
1.	rot	grün
2.	rot+gelb	gelb
3.	grün	rot
4.	gelb	rot+gelb

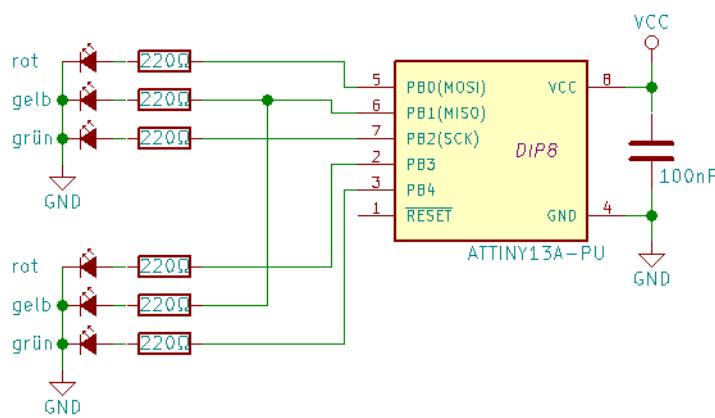
(und dann wiederholen)

Fangen wir wieder mit der Hardware an. Den Pin 1 können wir nicht verwenden, denn das ist unser Reset Eingang.

Zwar kann man ihn mit einer Fuse zu PB5 umwandeln, aber dann haben wir keinen Reset Pin mehr. Die braucht aber der Programmieradapter. Ohne Reset-Leitung können wir den Chip nicht mehr umprogrammieren.

Wir haben also nur noch zwei freie I/O Pins (PB3 und PB4) für drei zusätzliche Leuchtdioden. Das ist einer zu wenig. Doch wenn wir es geschickt anstellen, kommen wir immer noch mit dem kleinen ATtiny13 aus.

Achte mal auf die gelben Lichter. Laut obiger Tabelle leuchten die gelben Lichter beider Ampeln immer gleichzeitig. Also besteht keine Notwendigkeit, die einzeln anzusteuern. Wir können sie einfach parallel ansteuern.



Doch Vorsicht: Bei der Beschaltung des Mikrocontrollers dürfen wir den Programmieradapter nicht vergessen.

Manche Programmieradapter sind zu schwach, um mehr als eine LED anzusteuern. Die Doppelbelastung durch zwei gelbe LED's könnte dann zu Fehlfunktionen führen. Wir haben hier allerdings Glück, denn die gelben LED's hängen an der MISO Leitung. Diese Leitung überträgt Daten vom AVR Mikrocontroller zum Programmieradapter. Hier zählt also die Belastbarkeit des AVR, nicht die des Programmieradapters. Also alles ist gut, wir können anfangen, das Programm anzupassen.

```

#include <avr/io.h>
#include <util/delay.h>

// Drei Leuchtdioden an Port B
#define AUTO_ROT      (1<<PB0)
#define BEIDE_GELB    (1<<PB1)
#define AUTO_GRUEN    (1<<PB2)
#define PASSANTEN_ROT (1<<PB3)
#define PASSANTEN_GRUEN (1<<PB4)

int main(void)
{
    // Konfiguriere fünf I/O Pins als Ausgang
    DDRB = AUTO_ROT+BEIDE_GELB+AUTO_GRUEN+
           PASSANTEN_ROT+PASSANTEN_GRUEN;

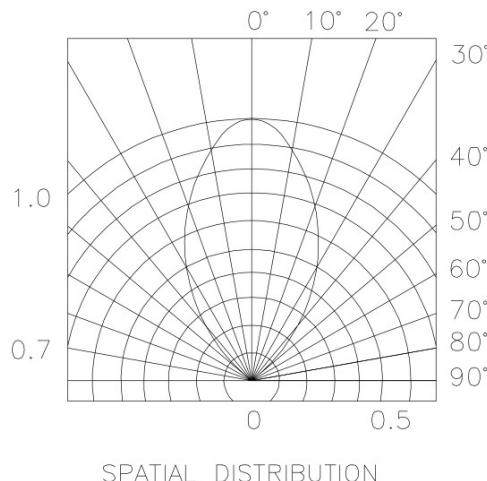
    // Endlosschleife
    while (1)
    {
        PORTB = AUTO_ROT+PASSANTEN_GRUEN;
        _delay_ms(8000);
        PORTB = AUTO_ROT+BEIDE_GELB;
        _delay_ms(2000);
        PORTB = AUTO_GRUEN+PASSANTEN_ROT;
        _delay_ms(8000);
        PORTB = BEIDE_GELB+PASSANTEN_ROT;
        _delay_ms(2000);
    }
}

```

6.3 Exkurs: Leuchtdioden

Bis Ende der 90er Jahre hatte man Leuchtdioden immer mit dem maximal zulässigen Strom betrieben, weil sie ziemlich wenig Licht abgaben. Sie waren ungefähr genau so ineffizient für Glühlampen und auch noch um ein Vielfaches teurer. Doch das hat sich bis heute grundlegend geändert.

In Katalogen wird die Helligkeit von Leuchtdioden in der Einheit mcd angegeben und der Abstrahlwinkel in Grad. Was bedeutet das? Schaue dir das folgende Diagramm an, dass ich aus einem Datenblatt des Herstellers Kingbright kopiert habe:



Stelle dir vor, dass unten im schwarzen Nullpunkt die Leuchtdiode liegt. Sie strahlt senkrecht nach oben in die Richtung, wo 0° steht. Das Ei in der Mitte des Diagramms zeigt an, wie hell die Leuchtdiode wirkt, wenn man sie aus unterschiedlichen Winkeln betrachtet.

Am rechten Rand sind die Betrachtungswinkel angegeben, sie gelten für die geraden strahlenförmigen Linien.

Am linken Rand und unten sind die relativen Helligkeiten angegeben, sie gelten für die kreisrunden Bögen.

Um zu erfahren, wie hell die Leuchtdiode bei senkrechtem Blick von oben ist, folgt man der senkrechten 0° Linie nach unten, bis man beim Ei angekommen ist. Dann folgt man dem Bogen nach links bis zur mcd Skala und kommt bei 1 raus. Das bedeutet, wir sehen aus diesem Blickwinkel die maximale Helligkeit.

Jetzt wollen wir herausfinden, wie hell diese Leuchtdiode ist, wenn man sie schräg aus einem Winkel von 30° betrachtet. Wir beginnen also recht oben bei der 30° Markierung, folgen der geraden Linie bis zum Ei und folgen dann dem Bogen nach links oder unten bis zur Helligkeits-Skala. Dort kommen wir bei 0,5 heraus. Wir sehen also nur noch halb soviel Licht.

Interessant ist die Tatsache, dass diese Leuchtdiode laut Katalog einen Abstrahlwinkel von 60° hat, aber schon bei 30° sehen wir nur noch die Hälfte der Helligkeit. Die 60° beziehen sich tatsächlich auf den Winkel, ab dem wir fast gar nichts mehr sehen!

Beim Interpretieren der mcd Zahlen sollte man noch eines bedenken: durch eine andere Linsenform kann der Hersteller das Licht des LED-Kristalls stärker bündeln um sie heller wirken zu lassen. Allerdings hat sie dann einen kleineren Abstrahlwinkel. Etwas schräg betrachtet sieht man kaum noch was. Achte beim Kauf also immer darauf, dass der Abstrahlwinkel zum Anwendungsfall passt. Wenn du unterschiedliche Farben in einer Anzeigetafel kombinierst, achte darauf, dass sie alle den gleichen Abstrahlwinkel haben. Für einen guten Gesamteindruck ist das wichtiger, als gleiche Helligkeit.

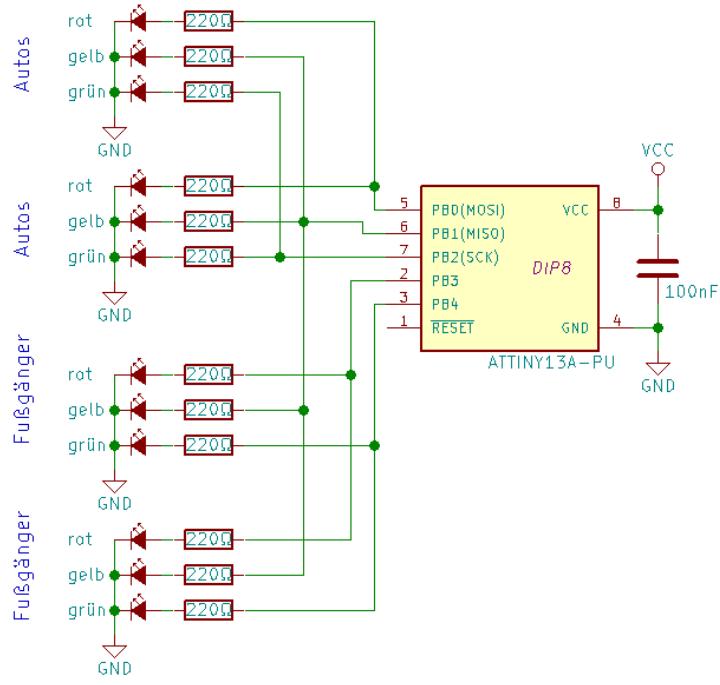
Die Leuchtdioden der 80er Jahre hatten eine Helligkeit von 1 bis 2 mcd. Nach heutigem Maßstab kann man diese Teile nur noch als Glimmfunzeln bezeichnen. Sie leuchten wirklich nicht viel heller als eine glimmende Zigarette. Heute (März 2016) werden diese alten LED's immer noch verkauft, aber für nur wenige Cent mehr bekommt man bei gleicher Stromaufnahme satte 100 bis 400 mcd!

Was sich aber bis heute kaum geändert hat, ist die Stromstärke, die dauerhaft vertragen. Die war damals schon 20 mA und es ist heute immer noch so. Die Helligkeit hängt ungefähr linear vom Strom ab. Bei $\frac{1}{4}$ des Stromes bekommst auch ungefähr $\frac{1}{4}$ der Helligkeit.

Wie viel Strom braucht nun eine moderne 400mcd Leuchtdiode, um ungefähr genau so hell zu wirken, wie eine alte 1mcd Leuchtdiode aus den 80er Jahren? Die Antwort ist ganz einfach, sie braucht nur $1/400$ des Stromes. Also 0,5 mA statt 20 mA! Spätestens jetzt sollte klar sein, warum kaum noch jemand Leuchtdioden mit 20 mA betreiben will.

Es gibt noch einen zweiten praktischen Grund: Die Ausgänge der AVR Mikrocontroller sind bis maximal 40 mA belastbar (und alle Ausgänge zusammen maximal 200 mA). Je geringer der Strom pro LED ist, umso mehr LED's kann man an den Mikrocontroller anschließen. Außerdem halten die Batterien dann länger.

Bei den obigen Experimenten haben wir nur eine Ampel für die Autos und eine zweite für die Fußgänger verwendet. In Echt müsste man natürlich doppelt so viele LED's anschließen, damit die Fußgänger in beide Richtungen etwas sehen können, und ebenso die Autos. Unser Mikrocontroller ist dazu locker stark genug.



Die Stromstärke durch eine Leuchtdiode kannst du messen oder berechnen:

$$(3,6 \text{ V} - 2 \text{ V}) : 220 \Omega = 7,2 \text{ mA}$$

An stärksten ist der Ausgang für Gelb belastet, denn der muss jetzt für 4 LED's ganze 28,8 mA liefern. Zum Anschluss von großen 230 V Glühlampen würde man die LED's durch Opto-Triacs oder Solid-State Relais ersetzen (Siehe Band 2).

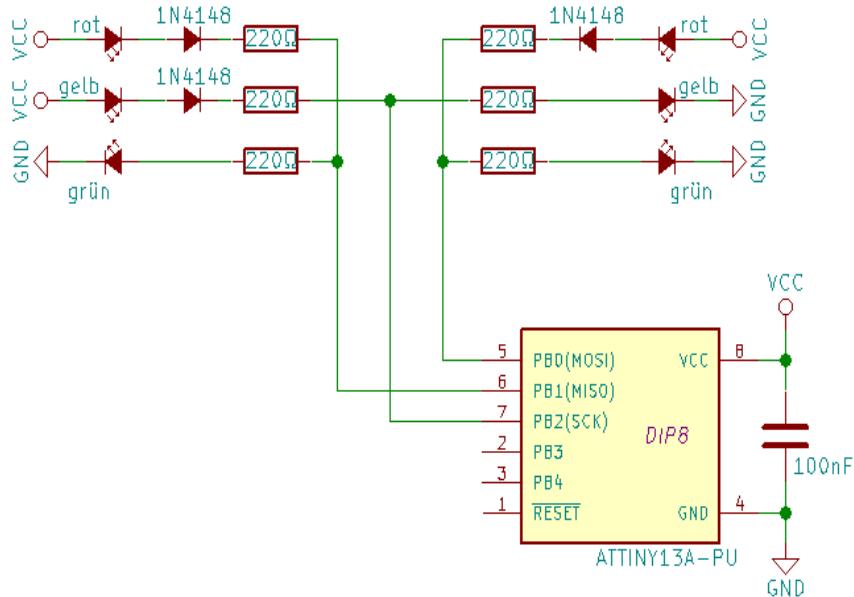
6.4 Kreuzung

Als nächste Ausbaustufe wagen wir uns an eine Straßenkreuzung ohne Fußgänger-Überwege. Der Unterschied zur vorherigen Variante ist, dass wir längere Rot-Phasen brauchen, damit die Kreuzung geräumt wird, bevor die anderen Autos grün bekommen.

Phase	Ampel A	Ampel B
1.	rot	rot+gelb
2.	rot	grün
3.	rot	gelb
4.	rot+gelb	rot
5.	grün	rot
6.	gelb	rot

(und dann wiederholen)

Jetzt haben wir aber ein Problem mit unseren I/O Pins. Denn wir können die gelben Leuchtdioden nicht mehr parallel schalten. Es scheint, als hätten wir nun doch einen Pin zu wenig. Aber es gibt wieder es eine Lösung, die allerdings nur mit drei Akkus (3,6V) funktioniert:



Der Trick ist folgender: Wir belegen jeden Pin mit zwei Leuchtdioden, die niemals gleichzeitig an sein müssen. Bei PB0 leuchtet die rote LED, wenn der Pin auf Low geht. Die grüne LED leuchtet, wenn der Pin auf High geht.

Die 1N4148 wird hier wegen ihrem Spannungsabfall verwendet. Sie verhindert, dass beide LED's gleichzeitig leuchten, wenn der Mikrocontroller kein Signal liefert. Denn:

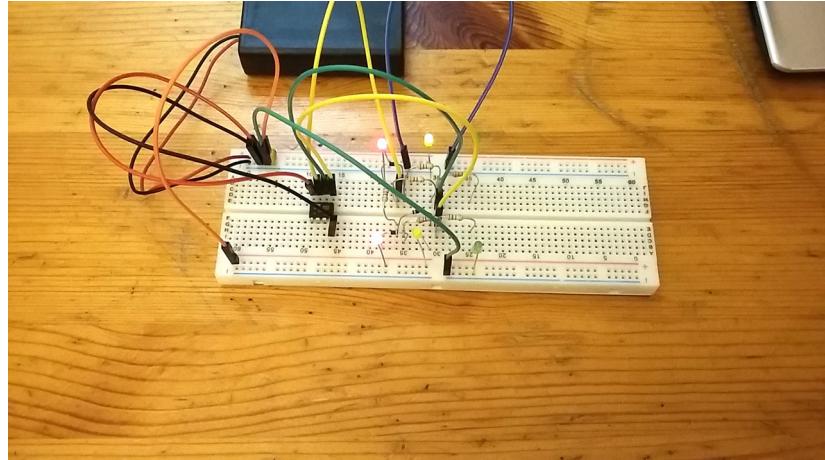
- Die rote LED benötigt mindestens 1,6V.
- Die 1N4148 benötigt mindestens 0,6V.
- Die grüne LED benötigt mindestens 1,8V.

Das macht zusammen 4 Volt. Die Batterie hat aber nur 3,6 Volt. Deswegen kann ohne Signal vom Mikrocontroller kein Strom fließen, und beide LED's sind dann dunkel. Wir nutzen die Ausgänge PB0 und PB1 also für drei Zustände:

- High = grün
- Low = rot
- Kein Signal = kein Licht

Man nennt diesen speziellen Anwendungsfall „Tri-State“. Den selben Trick wenden wir auch bei den gelben LED's mit PB2 an, denn unser Ablaufplan verlangt nicht mehr, dass beide gelben LED's gleichzeitig leuchten.

Wir brauchen also nur drei I/O Pins um alle sechs Leuchtdioden anzusteuern. Jetzt haben wir sogar noch zwei Pins übrig! Noch ein Foto vom Aufbau, wieder ohne Programmier-Adapter:



Die Makros müssen wir jetzt allerdings komplett umschreiben, sonst stiften sie mehr Verwirrung, als sie nützen. Wie man das genau umsetzt, ist Geschmackssache. Also betrachte den folgenden Code nur als Anregung. Wenn er dir nicht gefällt, dann schreibe ihn anders. Du kannst auch gerne Funktionen statt Makros benutzen.

```
#include <avr/io.h>
#include <util/delay.h>

// A = Ampel für die Nord/Süd Strecke
// B = Ampel für die Ost/West Strecke

#define ALLE_AUS { DDRB = 0; }

#define A_ROT      { DDRB |= (1<<PB0); PORTB &= ~(1<<PB0); }
#define A_GRUEN    { DDRB |= (1<<PB0); PORTB |= (1<<PB0); }

#define B_ROT      { DDRB |= (1<<PB1); PORTB &= ~(1<<PB1); }
#define B_GRUEN    { DDRB |= (1<<PB1); PORTB |= (1<<PB1); }

#define A_GELB     { DDRB |= (1<<PB2); PORTB &= ~(1<<PB2); }
#define B_GELB     { DDRB |= (1<<PB2); PORTB |= (1<<PB2); }

int main(void)
{
    // Endlosschleife
    while (1)
    {
        ALLE_AUS;
        A_ROT;
        B_ROT;
        B_GELB;
        _delay_ms(2000);

        ALLE_AUS;
        A_ROT;
        B_GRUEN;
        _delay_ms(4000);

        ALLE_AUS;
        A_ROT;
        B_GELB;
        _delay_ms(2000);

        ALLE_AUS;
    }
}
```

```

A_ROT;
A_GELB;
B_ROT;
_delay_ms(2000);

ALLE_AUS;
A_GRUEN;
B_ROT;
_delay_ms(4000);

ALLE_AUS;
A_GELB;
B_ROT;
_delay_ms(2000);
}

}

```

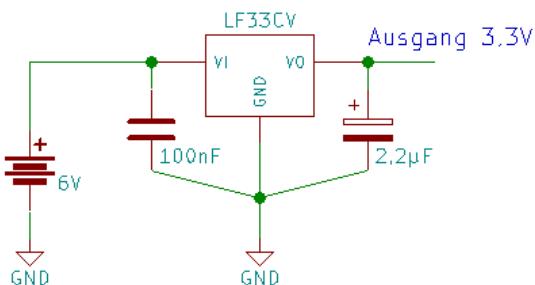
Bei jeder Ampelphase werden zunächst alle LED's mit ALLE_AUS ausgeschaltet, indem alle I/O Pins als Eingang konfiguriert werden. DDRB=0 bedeutet: Alles auf Eingang stellen. Danach werden jeweils die LED's eingeschaltet, die Leuchten sollen. Dazu wird der entsprechende I/O Pin als Ausgang geschaltet und dann auf High oder Low - je nach dem, welche LED denn nun genau leuchten soll.

Versuche, die Schaltung mit zwei Alkaline Batterien (also 3 Volt) zu betreiben. Damit leuchten die LED's nur noch sehr schwach. Versuche stattdessen drei Alkaline Batterien, (also 4,5 Volt). Das ist schon zu viel. Jetzt glimmen Leuchtdioden ständig, anstatt richtig aus zu gehen.

Wir haben jetzt also eine ausgefuchste Schaltung, die mit wenigen I/O Pins auskommt, aber sie ist auch recht empfindlich. Ein bisschen zu viel Spannung, und schon funktioniert sie nicht mehr. Ein bisschen zu wenig Spannung, und sie funktioniert auch nicht mehr.

6.5 Exkurs: Spannungsregler

Die Ampel funktioniert nur mit 3,3 bis 3,6 Volt. Wir brauchen eine stabile Spannungsversorgung die nicht vom Typ und Ladezustand der Batterien abhängt. Das erreicht man am Besten mit einem Spannungsregler IC. Für diese Ampel eignet sich zum Beispiel der LF33CV.



Der Spannungsregler macht aus einer etwas höheren Batteriespannung präzise und stabile 3,3 Volt. Er eignet sich auch prima für Handy Ladegeräte, die ungefähr 5 V liefern.

Beachte, dass Spannungsregler IC's immer Kondensatoren benötigen. Halte dich immer an die Angaben im jeweiligen Datenblatt und schließe die Kondensatoren mit möglichst kurzen Leitungen an. Falsche oder zu weit entfernten Kondensatoren verursachen kuriose Fehlfunktionen, zum Beispiel dass der Chip glühend heiß wird oder gar zu viel Spannung ausgibt.

Apropos heiß werden: Wenn der Spannungsregler die 6 V auf 3,3 V herab setzt, verheizt er die überschüssige Spannung. Die Wärmeleistung (Verlustleistung) berechnen wir, indem wir die überschüssige Spannung mit dem Strom multiplizieren. Die obige Ampel nimmt im Schnitt 20 mA auf. Also gilt die Rechnung: $(6V - 3,3V) \cdot 0,02 A = 0,054 \text{ Watt}$

Der LF33CV wird im TO-220 Gehäuse geliefert. Im Band 2 hatte ich erklärt, dass dieses Gehäuse maximal 1 Watt ohne zusätzlichen Kühlkörper abführen kann. Davon sind wir weit entfernt, also brauchen wir keinen Kühlkörper.

Die obige Rechnung macht aber auch eines deutlich: Der Spannungsregler verheizt Energie. Da Batterien im Vergleich zu Strom aus der Steckdose sehr teuer sind und nur begrenzte Energiemengen speichern, sollte man batteriebetriebene Geräte möglichst sparsam konstruieren. Da wollen wir keine Energie nur zum Spaß verheizen. Also benutzen wir bei Batteriebetrieb nur dann Spannungsregler, wenn es wirklich notwendig ist.

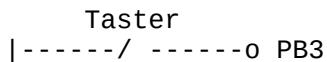
Im Fall unserer Ampel können wir auf den Spannungsregler verzichten, sofern wir sie mit drei NiMh Akkus versorgen.

6.6 Bedarfsampel

Unsere Ampel hat noch zwei Pins frei, die wir sinnvoll einsetzen können. Wir könnten unsere Ampelkreuzung mit einem Bewegungsmelder ausstatten, der den Verkehr einer Neben-Straße überwacht.

Normalerweise hat die Hauptstraße dauernd grüne Welle. Die Ampel schaltet nur dann um, wenn auf der Nebenstraße etwas los ist, also Bedarf für eine Umschaltung besteht. Also fügen wir nun einen Bewegungsmelder hinzu, bzw. wir simulieren ihn durch einen Taster.

Ergänze nun die Kreuzungssampel (von oben) um einen Taster, der mit GND und PB3 verbunden wird.



Und nun ändern wir das Programm so ab, dass es bei einer Grün-Phase auf Tastendruck wartet, anstatt die bisherigen 4 Sekunden.

```

#include <avr/io.h>
#include <util/delay.h>

// A = Ampel für die Nord/Süd Strecke
// B = Ampel für die Ost/West Strecke

#define ALLE_AUS { DDRB = 0; }

#define A_ROT      { DDRB |= (1<<PB0); PORTB &= ~(1<<PB0); }
#define A_GRUEN    { DDRB |= (1<<PB0); PORTB |= (1<<PB0); }

#define B_ROT      { DDRB |= (1<<PB1); PORTB &= ~(1<<PB1); }
#define B_GRUEN    { DDRB |= (1<<PB1); PORTB |= (1<<PB1); }

#define A_GELB     { DDRB |= (1<<PB2); PORTB &= ~(1<<PB2); }
#define B_GELB     { DDRB |= (1<<PB2); PORTB |= (1<<PB2); }

#define INIT_TASTE      { PORTB |= (1<<PB3); }
#define TASTE_GEDRUECKT ((PINB & (1<<PB3))==0)

int main(void)
{
    // Pull-Up am Taster aktivieren
    INIT_TASTE;

    // Endlosschleife
    while (1)
  
```

```

{
    ALLE_AUS;
    A_ROT;
    B_ROT;
    B_GELB;
    _delay_ms(2000);

    ALLE_AUS;
    A_ROT;
    B_GRUEN;

    // Warte auf Taste bzw. Bewegungsmelder
    while (!TASTE_GEDRUECKT) {};

    ALLE_AUS;
    A_ROT;
    B_GELB;
    _delay_ms(2000);

    ALLE_AUS;
    A_GELB;
    B_ROT;
    _delay_ms(2000);

    ALLE_AUS;
    A_GRUEN;
    B_ROT;
    _delay_ms(4000);

    ALLE_AUS;
    A_GELB;
    B_ROT;
    _delay_ms(2000);
}
}

```

Ganz am Anfang vor der Endlos-Schleife initialisieren wir den Pin, wo der Taster angeschlossen ist. Nach einem Reset sind alle Pins als Eingang geschaltet, das ergibt sich schon von selbst. Aber wir müssen noch den internen Pull-Up Widerstand des Mikrocontroller einschalten, damit der Eingang ein definiertes Signal hat während der Taster nicht betätigt wird.

```
PORTE |= (1<<PB3);
```

Erledigt dies. Die zweite Änderung befindet mitten in der Endlos-Schleife. Dort habe ich einen delay Befehl durch eine leere Warteschleife ausgetauscht, die so lange wiederholt wird, wie der Taster nicht gedrückt ist.

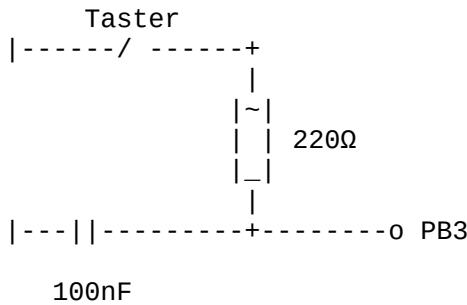
Wenn der Taster gedrückt wird, geht der Eingang PB3 auf Low. Dann liefert das Makro TASTE_GEDRUECKT den Wert „true“, so dass die Warteschleife verlassen wird.

6.7 Exkurs: Entstörung

Wenn du die Mittel dazu hast, dann versuche mal, den Taster über lange Kabel anzuschließen. Zum Beispiel 20 Meter. Du wirst feststellen, dass die Ampel dann ziemlich häufig von selbst auslöst, ohne dass jemand auf den Taster gedrückt hat.

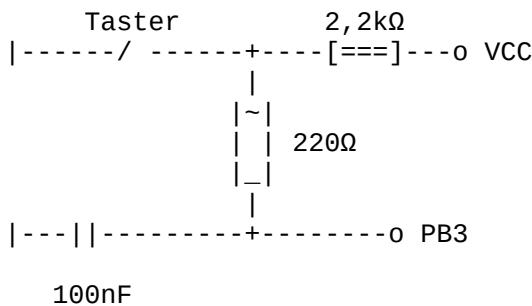
Das kommt daher, dass die lange Leitung eine prima Antenne für Funkwellen darstellt. Sie Empfängt Signale aus der Luft und leitet sie an den Mikrocontroller weiter, der daraufhin falsch reagiert.

Als einfachste Gegenmaßnahme dient ein Kondensator am Eingang des Mikrocontrollers:



Der Kondensator schließt hochfrequente Radiowellen kurz so dass sie eliminiert werden. Der Widerstand begrenzt den Entladestrom des Kondensators, so dass die Kontakte des Tasters beim Schließen nicht beschädigt werden.

Eine weitere Gegenmaßnahme ist, den internen Pull-Up Widerstand durch einen externen kleineren Widerstand zu unterstützen:



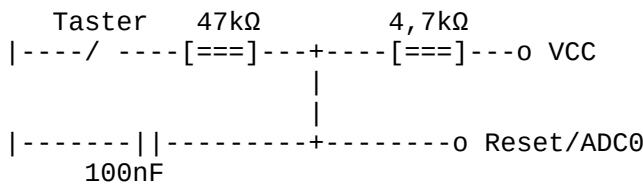
Durch den Zusätzlichen Pull-Up Widerstand wird die Schaltung gegenüber Radiowellen noch unempfindlicher. Außerdem arbeiten viele Taster (und Schalter) länger zuverlässig, wenn sie mit einem gewissen mindest-Strom belastet werden.

6.8 Reset Pin doppelt belegen

Wir sind bisher davon ausgegangen, dass wir den Pin 1 (=Reset, PB5, ADC0) nicht als normalen I/O Pin nutzen können, weil der Programmieradapter die Reset Funktion benötigt, um zu funktionieren.

Das ist soweit richtig. Aber man kann den Pin eingeschränkt als analogen Eingang verwenden, ohne einen Reset auszulösen. Denn laut Datenblatt wird garantiert kein Reset ausgelöst, solange der Eingang mindestens eine Spannung in Höhe von $0,9 \cdot VCC$ hat.

Das ist ganz einfach umsetzbar. Ersetze die obige Schaltung für den Taster durch diese:



Durch den $4,7\text{ k}\Omega$ Widerstand wird der Reset Eingang ständig auf High Pegel gehalten. Wenn der Taster gedrückt wird, fließt ein geringer Strom in Richtung GND ab, so dass die Spannung am Eingang des Mikrocontroller ein kleines bisschen absackt. Der ADC kann den kleinen Unterschied zuverlässig erkennen, und doch löst das Signal sicher keinen Reset aus.

Die beiden $47\text{ k}\Omega$ und $4,7\text{ k}\Omega$ bilden einen Spannungsteiler, wenn der Taster gedrückt wird. Die Stromstärke durch die beiden Widerstände ist:

$$3,6\text{V} : (47\text{ k}\Omega + 4,7\text{ k}\Omega) = 0,07\text{ mA}$$

Die Spannung am $47\text{ k}\Omega$ Widerstand ist:

$$47\text{ k}\Omega \cdot 0,07\text{ mA} = 3,29\text{ V}$$

Das ist gerade hoch genug, um keinen Reset auszulösen, denn:

$$0,9 \cdot 3,6\text{ V} = 3,24\text{ V}$$

Das Makro zum Einlesen der Taste wird durch eine Funktion ersetzt, die eine Messung mit dem ADC durchführt.

```
#include <avr/io.h>
#include <util/delay.h>

// A = Ampel für die Nord/Süd Strecke
// B = Ampel für die Ost/West Strecke

#define ALLE_AUS { DDRB = 0; }

#define A_ROT      { DDRB |= (1<<PB0); PORTB &= ~(1<<PB0); }
#define A_GRUEN    { DDRB |= (1<<PB0); PORTB |= (1<<PB0); }

#define B_ROT      { DDRB |= (1<<PB1); PORTB &= ~(1<<PB1); }
#define B_GRUEN    { DDRB |= (1<<PB1); PORTB |= (1<<PB1); }

#define A_GELB     { DDRB |= (1<<PB2); PORTB &= ~(1<<PB2); }
#define B_GELB     { DDRB |= (1<<PB2); PORTB |= (1<<PB2); }

// Fragt ADC0 ab, ob die Taste gedrückt wurde
uint8_t taste_gedrueckt(void)
{
    // Lese Eingang ADC0 mit VCC als Referenz
    ADMUX=0;
    // Starte Messung mit Prescaler 16
    ADCSRA=(1<<ADEN)+(1<<ADSC)+(1<<ADPS2);
    // Warte, solange die Messung läuft
    while (ADCSRA & (1<<ADSC)) {};
    // Ergebnis abrufen
    return ADC<970;
}

int main(void)
{
    // Endlosschleife
    while (1)
    {
        ALLE_AUS;
        A_ROT;
    }
}
```

```

    B_ROT;
    B_GELB;
    _delay_ms(2000);

    ALLE_AUS;
    A_ROT;
    B_GRUEN;

    // Warte auf Taste bzw. Bewegungsmelder
    while (!taste_gedrueckt()) {};

    ALLE_AUS;
    A_ROT;
    B_GELB;
    _delay_ms(2000);

    ALLE_AUS;
    A_GELB;
    B_ROT;
    _delay_ms(2000);

    ALLE_AUS;
    A_GRUEN;
    B_ROT;
    _delay_ms(4000);

    ALLE_AUS;
    A_GELB;
    B_ROT;
    _delay_ms(2000);
}

}

```

Um die Funktion der Tasten-Abfrage zu verstehen, vergleiche die Befehle mit dem Datenblatt des ATtiny13. Zuerst wird mit dem ADMUX Register eingestellt, dass wir den Eingang ADC0 einlesen wollen und die Versorgungsspannung (VCC) als Referenz verwenden.

```
ADMUX=0;
```

Dann wird der ADC eingeschaltet und eine Messung gestartet. Die Taktfrequenz von 1,2Mhz wird durch 16 geteilt, was 75kHz ergibt. Laut Datenblatt soll die Taktfrequenz des ADC zwischen 50 und 200kHz liegen.

```
ADCSRA=(1<<ADEN)+(1<<ADSC)+(1<<ADPS2);
```

Dann wird gewartet, bis die Messung beendet ist.

```
while (ADCSRA & (1<<ADSC)) {};
```

Und zum Schluss wird geprüft, ob der gemessene Wert kleiner als 960 ist. Wenn ja, dann liefert die Funktion 1 (=true) zurück, sonst liefert sie 0 (=false) zurück.

```
return ADC<970;
```

Wenn die Taste nicht gedrückt ist, liefert der ADC ungefähr den Wert 1023. Das ist der höchste mögliche Wert. Wenn die Taste gedrückt ist, liefert der ADC ungefähr den Wert 930, weil die Spannung geringer ist.

Für den Vergleich habe ich willkürlich einen Wert ungefähr in der Mitte zwischen 930 und 1023 gewählt. Wir können so eindeutig und zuverlässig erkennen, ob die Taste gedrückt ist, oder nicht.

7 Servos

In diesem Kapitel werden wir mit Modellbau-Servos experimentieren.



Wir könnten dazu wieder den kleinen ATtiny13 verwenden, aber um die Sache etwas aufzulockern werden wir dieses mal ein Mikrocontroller Modul mit dem deutlich größeren ATmega328 benutzen.

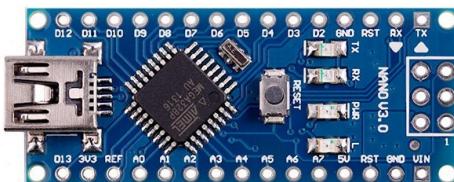
Material:

- 1 Diode 1N4001
- 1 Arduino Nano (Original oder Nachbau aus China¹)
- 1 Mini-USB Kabel für den Arduino Nano
- 1 Widerstand 10 k Ohm ¼ Watt
- 1 Phototransistor für Tageslicht PT331C (oder ähnlicher in Bauform einer LED)
- 1 Modellbau Mini Servo
- 1 Netzteil 5 V, mindestens 500 mA
- 1 Steckbrett und Kabel

7.1 Exkurs: Arduino Nano

Lade dir den Schaltplan des Arduino Nano Moduls von der Seite <https://www.arduino.cc/en/Main/ArduinoBoardNano> herunter und drucke ihn aus. Du wirst ihn ziemlich oft brauchen, um die Zuordnung der Anschlüsse zu den Pins des Mikrocontrollers herauszufinden. Leider sind sie nämlich nicht 1:1 beschriftet.

Die chinesischen Nachbauten verwenden einen anderen USB-UART Chip, haben ansonsten aber die gleichen technischen Daten, wie das Original.



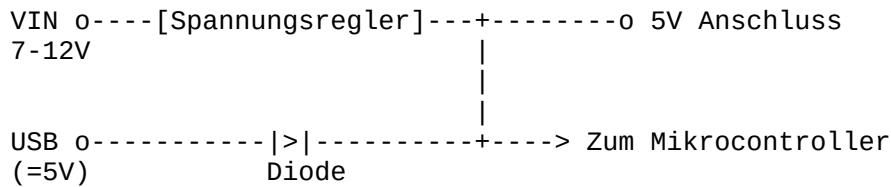
Das Arduino Board kann wahlweise auf folgende Weise mit Strom versorgt werden:

- 7..12 V am Anschluss VIN
- ungefähr 5 V am Anschluss 5V
- Vom PC über das USB Kabel

- 1 Leider kommt es bei chinesischen Nachbauten häufig vor, dass die Fuses falsch gesetzt sind oder der Bootloader nicht installiert wurde. Dann funktioniert Avrdude nicht. Folge ggf. der Anleitung auf <https://www.arduino.cc/en/Hacking/Bootloader?from=Tutorial.Bootloader>

Falls du einen Nachbau mit CH340 Chip hast, musst du eventuell die „Rx“ LED entfernen, damit die USB Kommunikation zuverlässig funktioniert. Es handelt sich hier um einen Fehler im Schaltungsdesign.

Dazu befindet sich auf dem Arduino Board folgende Schaltung:



Unabhängig davon, woher die Stromversorgung kommt, darf man am 5 V Anschluss maximal 200 mA abgreifen. Bei höheren Strömen kann der Spannungsregler, die Diode oder schlimmstenfalls sogar der USB Port des Computers kaputt gehen.

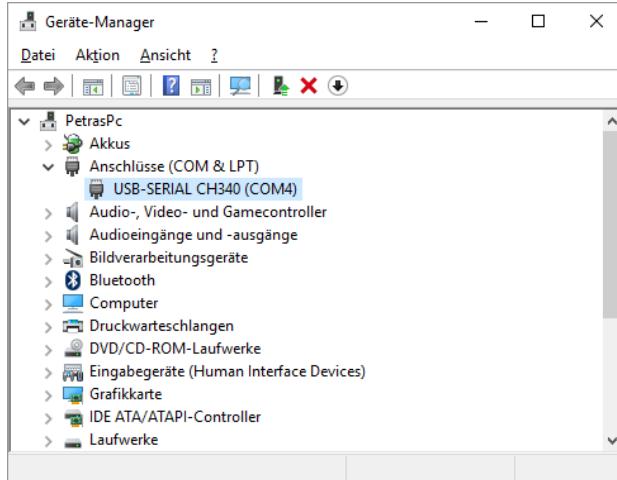
Wenn du deine Experimente mit dem PC verbindest, besteht immer ein gewisses Risiko, den USB Anschluss des PC zu beschädigen. Das lässt sich erheblich reduzieren, indem man einen sogenannten „Self Powered USB Hub“ zwischen PC und Arduino steckt. Immerhin kostet ein defekter Hub viel weniger, als ein neuer PC.

Das Arduino Board kann wahlweise über einen ISP Programmer oder mit Hilfe des vorinstallierten Bootloaders seriell über USB programmiert werden. Wir werden die USB Variante nutzen.

7.1.1 Windows Treiber Installieren

Wenn du das USB Kabel zum ersten mal an den PC steckst, sollte Windows den nötigen Treiber automatisch im Internet finden und installieren. Falls das nicht klappt, musst du den Treiber selbst herunterladen und manuell installieren.

Ob der Treiber funktioniert, verrät ein Blick in den Gerätemanager der Systemsteuerung. Dort erfährst du auch, welche Nummer der virtuelle COM Port hat. Diese Information brauchst du später für Avrdude.



7.1.2 Linux Treiber Installieren

Linux wird bereits mit dem nötigen Treiber ausgeliefert, so dass man nichts installieren muss. Du kannst das mit dem Befehl „sudo dmesg“ überprüfen:

```

stefan@STEFANSPC:~$ sudo dmesg
[9582.086182] usb 2-1.2: new full-speed USB device number 6 using ehci-pci
[9582.179632] usb 2-1.2: New USB device found, idVendor=1a86, idProduct=7523
[9582.179644] usb 2-1.2: New USB device strings: Mfr=0, Product=2, SerialNumber=0
[9582.179650] usb 2-1.2: Product: USB2.0-Serial
[9582.180289] ch341 2-1.2:1.0: ch341-uart converter detected
[9582.182102] usb 2-1.2: ch341-uart converter now attached to ttyUSB0
stefan@STEFANSPC:~$ 

```

Der virtuelle serielle Port für Avrdude ist in diesem Fall /dev/ttyUSB0.

7.1.3 Makefile Vorbereiten

Lade dir meine „Hello World“ Vorlage von meiner Homepage herunter und passe den oberen Teil des Makefile wie folgt an:

```

# Programmer hardware settings for avrdude
AVRDUDE_HW = -c arduino -P /dev/ttyUSB0 -b 57600

# Name of the program
PRG = Servo

# Microcontroller type and clock frequency
MCU = atmega328p
F_CPU = 16000000

```

Ganz oben geben wir für Avrdude an, dass wir den Mikrocontroller mit Hilfe des Arduino Bootloaders beladen wollen. Die Baudrate des Bootloaders ist 57600 oder 115200 und der Port ist zum Beispiel COM3 unter Windows oder /dev/ttyUSB0 unter Linux. Die richtige Nummer des virtuellen seriellen Ports hast du ja vorher kontrolliert, gib hier die Nummer an, die dein Computer dem USB Anschluss zugewiesen hat.

Falls die COM-Port Nummer zweistellig ist (z.B. COM10) musst du es so schreiben: \\.\COM10

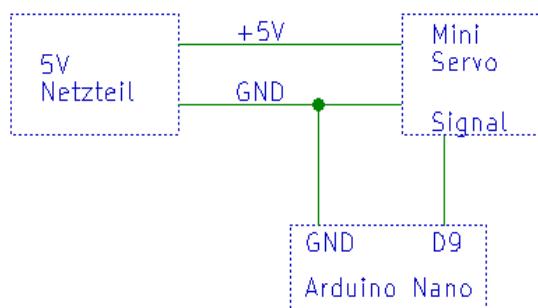
Bei MCU gibst du an, welcher Mikrocontroller auf dein Arduino Board gelötet wurde. Das ist meistens ein atmega328p, kann aber auch ein atmega328, atmega168p oder atmega168 sein.

Bei F_CPU gibt man die Taktfrequenz 16Mhz an, denn das Arduino Board enthält einen 16Mhz Keramik-Resonator.

Die Einstellungen für die Fuses solltest du entfernen. Der Arduino Bootloader kann die Fuses nicht verändern. Das würde nur mit einem ISP Programmieradapter gehen.

7.2 Probe-Aufbau

Stecke den Servo, das 5 V Netzteil und das Arduino Nano Board folgendermaßen zusammen:

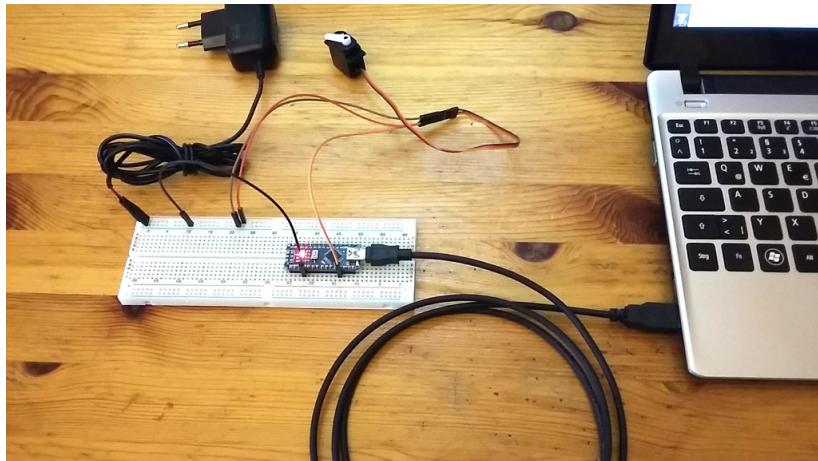


Verbinde das Arduino Nano Modul mit dem USB Anschluss deines Computers.

Beim Servo sind die Farben des Anschlusskabels meistens so:

- GND = schwarz oder braun

- 4,8..6V = rot
- Signal = orange, gelb oder weiß

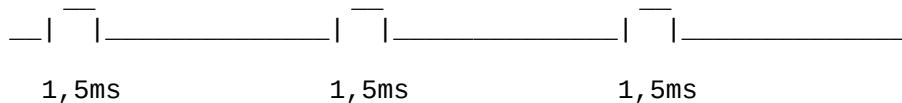


Der 5 V Ausgang des Arduino Nano Moduls ist mit maximal 200 mA belastbar. Da der Servo Motor aber zeitweise erheblich mehr Strom aufnimmt, muss er unbedingt von einem separaten Netzteil mit Strom versorgt werden. Er darf auf keinen Fall an den 5 V Anschluss des Arduino Nano Moduls angeschlossen werden.

7.3 Steuersignal erzeugen

7.3.1 Servo mit 16-bit Timer

Im Band 2 dieser Buchreihe habe ich beschrieben, wie das Steuersignal eines Servos aussehen muss. Wir brauchen Impulse mit ungefähr 1,5ms, die sich alle 20ms wiederholen.

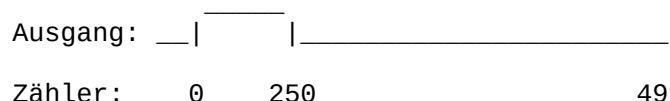


Die Breite der Impulse bestimmt, an welche Position der Servo fährt.

Wir benutzen den Timer 1 im Fast PWM Modus, um das Steuersignal zu erzeugen. Der Fast PWM Modus erzeugt Impulse variabler Breite, die sich regelmäßig wiederholen, also genau das, was wir brauchen.

Der Timer zählt Taktimpulse und vergleicht den Zählerstand mit einem einstellbaren Vergleichswert. Von Null bis zum Vergleichswert geht der Ausgang des Timers auf High. Danach geht er auf Low, bis der Zähler einen einstellbaren Maximal-Wert überschreitet. Dann fängt er wieder bei 0 an zu zählen.

Wenn wir als Vergleichswert die Zahl 250 einstellen und als Maximalwert die Zahl 4999, dann liefert der Timer an seinem Ausgang dieses Signal:



Dieses Signal wird fortlaufend wiederholt. Folgende relevanten Begriffe kommen im Datenblatt des Atmega328 vor:

- PWM: Puls Weiten Modulation
- Clock: Taktung

- Prescaler: Verteiler für die Taktung
- OCR1A: Vergleichswert A
- OC1A: Ausgang vom Vergleicher A
- OCR1B: Vergleichswert B
- OC1B: Ausgang vom Vergleicher B
- TOP / ICR1: Maximalwert für den Zähler

Spätestens jetzt solltest du das Kapitel „16-bit Timer/Counter 1 with PWM“ im Datenblatt durchlesen.

Der Servo ist mit dem Anschluss D9 am Arduino verbunden. Das entspricht (laut Schaltplan des Arduino Boardes) dem Port PB1 des Mikrocontrollers. Dieser Pin hat auch die Spezialfunktion OC1A, also unser PWM Ausgang.

Wir programmieren den Timer mit folgenden Parametern:

- Fast PWM Modus
- Prescaler 64
- TOP Wert 4999
- Wert für „links“ 250
- Wert für „rechts“ 500

Der Verteiler teilt unsere 16Mhz Taktfrequenz durch 64. Die Dauer eines Zählertaktes ist daher:

$$1 : 16000000\text{Hz} : 64 = 0,000004\text{s} = 0,004\text{ms}$$

Der Top Wert 4999 bewirkt, dass der Zähler immer von 0 bis 4999 Zählt, also insgesamt 5000 Takte pro Wiederholung.

Das Wiederholintervall ist daher:

$$5000 \cdot 0,004\text{ms} = 20\text{ms}$$

Der Vergleichswert für „links“ entspricht:

$$250 \cdot 0,004\text{ms} = 1\text{ms}$$

Der Vergleichswert für „rechts“ entspricht:

$$500 \cdot 0,004\text{ms} = 2\text{ms}$$

Editiere die Datei main.c, dass sie so aussieht:

```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    // Starte Timer 1 im Fast PWM Modus
    // mit Prescaler 64 und TOP Wert 4999
    TCCR1A = (1<<COM1A1) + (1<<WGM11);
    TCCR1B = (1<<WGM12) + (1<<WGM13) + (1<<CS10) + (1<<CS11);
    ICR1 = 4999;

    // Aktiviere PWM und LED Ausgänge
    DDRB |= (1<<PB1) + (1<<PB5);

    while (1)
```

```

{
    // Servo nach links fahren
    PORTB |= (1<<PB5);
    OCR1A = 250;
    _delay_ms(10000);

    // Servo nach rechts fahren
    PORTB &= ~(1<<PB5);
    OCR1A = 500;
    _delay_ms(10000);
}

```

Wenn du das Programm mit „make program“ in den Mikrocontroller lädst, wird die LED auf dem Arduino Board alle 10 Sekunden abwechselnd an und aus gehen. Der Servo fährt dabei abwechselnd nach links und rechts.

Versuche, den Wert 250 weiter zu reduzieren, damit der Servo noch weiter nach links fährt. Aber passe auf, dass er dabei nicht an den mechanischen Endanschlag anstößt. Den dann läuft der Motor heiß und brennt nach einigen Sekunden durch. Man kann den Fehler an einem knurrenden Geräusch erkennen. Wenn dir das passiert, dann ziehe sofort das Netzteil aus der Steckdose und erhöhe den Wert wieder.

Auf die gleiche Weise kannst du nun versuchen, den Wert 500 zu erhöhen, damit der Servo noch weiter nach rechts fährt. Bei welchen Werten die Endanschläge erreicht werden, hängt vom Servo-Modell ab. Da verhält sich jedes Servo anders, deswegen kann ich hier keine konkreten Zahlen nennen.

Durch Verwendung des zweiten Vergleichers (B) kannst du noch einen weiteren Servo mit dem selben Timer gleichzeitig ansteuern. Den zweiten Servo schließt du an den Port PB2/OC1B an.

Der elektrische Aufbau war sehr einfach und das Programm ist ziemlich klein. Aber jede einzelne Zeile hat es in sich. Vergleiche das Programm mit den Informationen zum Timer 1 im Datenblatt des Mikrocontrollers.

Wenn du irgend etwas nicht verstehst, dann frage jemanden, der sich auskennt. Im Forum <https://www.mikrocontroller.net/> tummeln sich viele Bastler, die sich mit AVR Mikrocontrollern gut auskennen.

7.3.2 Servo mit 8-Bit Timer

Wenn du den 16-Bit Timer für andere Zwecke benötigst, kannst du auch den 8-Bit Timer verwenden, um den Servo anzusteuern. Das ist allerdings deutlich aufwändiger. Auf der anderen Seite kannst du mit dem aufwändigeren Code viel mehr Servos ansteuern und dazu jeden beliebigen I/O Pin verwenden. Der Aufwand lohnt sich also.

Wir verwenden den 8-Bit Timer 2. Unser Servo ist weiterhin an den Arduino Anschluß D9 gesteckt, das ist beim Mikrocontroller der Port PB1.

Wir starten den Timer wieder im Fast PWM Modus, aber dieses mal mit Vorteiler 256. Ein Timertakt dauert daher:

$$1 : 16000000\text{Hz} : 256 = 0,000016\text{s} = 0,016\text{ms}$$

Um einen Impuls von 1ms (für links) zu erzeugen, stellen wir das Compare Register auf den Wert 62, denn

$$92 \cdot 0,016\text{ms} = 1\text{ms}$$

Um einen Impuls von 2ms (für rechts) zu erzeugen, stellen wir das Compare Register auf den Wert 124, denn

$$124 \cdot 0,016\text{ms} = 2\text{ms}$$

Der 8 Bit Timer läuft schon nach 256 Takten über (eben weil er 8-Bit klein ist). Anders als beim 16 Bit Timer können wir hier keinen anderen TOP Wert festlegen. Daher wiederholen sich die Impulse zwangsläufig in diesem Intervall:

$$256 \cdot 0,016\text{ms} = 4\text{ms}$$

Das ist nicht gut, denn Servos benötigen ein Wiederholungsintervall von 20ms. Die 4ms sind völlig falsch.

Und jetzt kommt der eigentliche Trick: Wir steuern den Servo nicht direkt über einen Timer Ausgang an, sondern rufen stattdessen Software Routinen auf, die dann wiederum den Ausgang ansteuern, aber nur bei jedem fünften mal. So kommen wir auf das korrekte Wiederholungsintervall.

$$5 \cdot 4\text{ms} = 20\text{ms}$$

Schau dir jetzt mal den Quelltext an:

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

uint8_t phase=0;

// Timer ist übergelaufen, neue Phase beginnt
ISR(TIMER2_OVF_vect)
{
    // Servo nur in Phase 0 ansteuern
    if (phase==0)
    {
        PORTB |= (1<<PB1);
    }
    // Nächste Phase 0..4
    if (++phase > 4)
    {
        phase=0;
    }
}

// Timer hat Vergleichswert erreicht
ISR(TIMER2_COMPA_vect)
{
    PORTB &= ~(1<<PB1);
}

int main(void)
{
    // Starte Timer 0 im Fast PWM Modus mit Prescaler 256
    TCCR2A = (1<<WGM21) + (1<<WGM20);
    TCCR2B = (1<<CS22) + (1<<CS21);

    // Erlaube Interrupts
    TIMSK2 = (1<<OCIE2A) + (1<<TOIE2);
    sei();
}
```

```

// Aktiviere PWM und LED Ausgänge
DDRB |= (1<<PB1) + (1<<PB5);

while (1)
{
    // Servo nach links fahren
    PORTB |= (1<<PB5);
    OCR2A = 62;
    _delay_ms(10000);

    // Servo nach rechts fahren
    PORTB &= ~(1<<PB5);
    OCR2A = 124;
    _delay_ms(10000);
}
}

```

Anstatt dass der Ausgang direkt von der Timer-Hardware angesteuert wird, haben wir nun zwei Interrupt-Routinen. Die Interruptroutinen werden vom Timer ausgelöst. Sie unterbrechen die Ausführung des Hauptprogrammes für einen kurzen Moment.

- **TIMER2_OVF_vect**
Wird ausgeführt, wenn der Zähler des Timers überläuft (also von 255 nach 0 springt). Wir schalten dann den Ausgang auf High, aber nur bei jedem fünften mal.
- **TIMER2_COMPA_vect**
Wird ausgeführt, wenn der Zähler des Timers den Vergleichswert aus Register OCR2A erreicht. Wir schalten dann den Ausgang auf Low.

Die Variable „phase“ wird verwendet, um „jedes fünfte mal“ zu bestimmen. Unser Servo erhält nur in Phase 0 einen Impuls. Danach werden die Phasen 1,2,3 und 4 ohne Impuls durchlaufen. Dann kommt wieder Phase 0.

7.3.3 Viele Servos

Mit dem 8-Bit Timer und einem leicht aufgemotztem Programm kannst du 5 Servos gleichzeitig ansteuern. Wir steuern in jeder Phase einen anderen Servo an, immer schön der Reihe nach.

Wir benutzen dieses mal die Ausgänge PB0 bis PB4 für fünf Servos, und PB5 ist immer noch die LED auf dem Arduino Board.

```

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

uint8_t phase=0;

volatile servo[5] = {92,92,92,92,92};

// Timer ist übergelaufen, neue Phase beginnt
ISR(TIMER2_OVF_vect)
{
    // Nur einen Servo auf High setzen
    switch (phase)
    {
        case 0: PORTB |=(1<<PB0); break;
        case 1: PORTB |=(1<<PB1); break;
        case 2: PORTB |=(1<<PB2); break;
        case 3: PORTB |=(1<<PB3); break;
        case 4: PORTB |=(1<<PB4); break;
    }
}

```

```

        }
        // Nächste Phase 0..4
        if (++phase > 4)
        {
            phase=0;
        }
        // Compare-Wert für die nächste Phase
        OCR2A = servo[phase];
    }

    // Timer hat Vergleichswert erreicht
    ISR(TIMER2_COMPA_vect)
    {
        // Alle Servos auf Low setzen
        PORTB &= ~((1<<PB0) + (1<<PB1) + (1<<PB2) + (1<<PB3) + (1<<PB4));
    }

    int main(void)
    {
        // Starte Timer 0 im Fast PWM Modus mit Prescaler 256
        TCCR2A = (1<<WGM21) + (1<<WGM20);
        TCCR2B = (1<<CS22) + (1<<CS21);

        // Erlaube Interrupts
        TIMSK2 = (1<<OCIE2A) + (1<<TOIE2);
        sei();

        // Aktiviere PWM Ausgänge (PB0..PB4)
        // und LED Ausgang PB5
        DDRB |= (1<<PB0) + (1<<PB1) + (1<<PB2) + (1<<PB3) + (1<<PB4) + (1<<PB5);

        while (1)
        {
            // Servo nach links fahren
            PORTB |= (1<<PB5);
            servo[1] = 62;
            _delay_ms(10000);

            // Servo nach rechts fahren
            PORTB &= ~(1<<PB5);
            servo[1] = 124;
            _delay_ms(10000);
        }
    }
}

```

Das Array servo[] speichert die Soll-Positionen aller fünf Servos. Es wird fünf mal dem Wert 92 initialisiert, was 1,5ms für die mittlere Position entspricht.

- servo[0] ist für Port PB0 / Arduino Anschluss D8
- servo[1] ist für Port PB1 / Arduino Anschluss D9, hier ist unser Servo angesteckt.
- servo[2] ist für Port PB2 / Arduino Anschluss D10
- servo[3] ist für Port PB3 / Arduino Anschluss D11
- servo[4] ist für Port PB4 / Arduino Anschluss D12

An Anfang jeder Phase schaltet die Interrupt-Routine TIMER2_OVF_vect schaltet immer abwechselnd einen der fünf Servo-Ausgänge auf High. Dann wird die Nummer der Phase erhöht und das Compare-Register auf den Sollwert der nächsten Phase gesetzt. Der Timer benutzt den neuen Wert erst nach Ablauf der aktuellen Phase.

Die Interrupt-Routine TIMER2_COMPA_vect schaltet das Signal für alle fünf Servos auf Low. Sie wird aufgerufen, sobald der Zähler des Timers den Compare-Wert erreicht hat.

Da wir den Compare-Wert jetzt laufend ändern, kann jeder Servo eine andere Position anfahren.

Im Hauptprogramm wird unser Servo wie gehabt alle 10 Sekunden abwechselnd nach links und rechts gefahren.

Du könntest jetzt noch ein paar mehr Servos anstecken und das Hauptprogramm erweitern, so dass es alle fünf Servos an irgendwelche Positionen fährt. Wenn du das tust, dann achte darauf, dass immer nur ein Servo gleichzeitig in Bewegung ist, sonst überlastest du möglicherweise das Netzteil. Mach also nach jeder Positionsänderung eine ausreichend lange (delay) Pause.

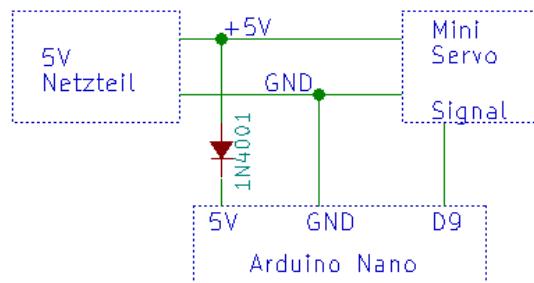
Da der Timer noch ein zweites Compare Register B hat, könnte man diesen Programmcode auf 10 Servos erweitern. Und wenn du den Timer 1 nach dem gleichen Prinzip hinzunimmst, kannst du weitere 10 Servos ansteuern. Aber wir wollen es mal nicht übertreiben.

7.4 Exkurs: Dioden in Stromversorgung

Stecke den Computer vom USB Kabel ab. Das Arduino Board verliert dadurch seine Stromversorgung und arbeitet nicht mehr. Du könntest jetzt dessen 5 V Anschluss mit dem 5 V Netzteil verbinden, damit er läuft.

Aber das wäre keine gute Idee, denn dann hängt auch der Servo Motor an dieser 5 V Leitung. Wenn dann jemand das USB Kabel wieder einsteckt wird er zu viel Strom aus dem PC ziehen. Schlimmstenfalls wird dadurch der USB Port zerstört, wahrscheinlicher ist jedoch, dass nur die kleine Diode auf dem Arduino Board durchbrennt.

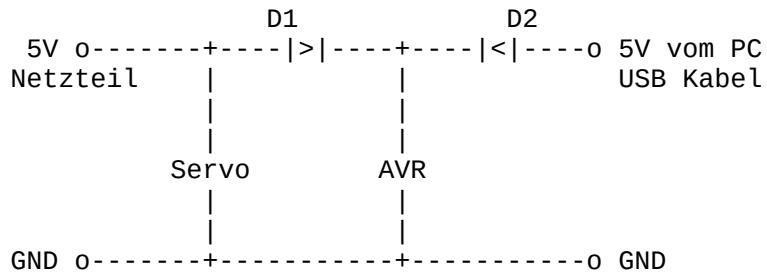
Das wollen wir natürlich beides nicht nicht. Deswegen fügen wir eine weitere Diode ein:



Jetzt bekommt das Arduino Board seine Spannungsversorgung durch die Diode, falls es nicht an den PC angeschlossen ist. In die umgekehrte Richtung (vom Computer über das Arduino Modul zum Servo) kann aber kein Strom fließen. Die Diode verhindert das.

An der Diode gehen etwa 0,7 V von den 5 V verloren. Die Spannungsversorgung des Arduino Moduls beträgt daher in Wirklichkeit nur 4,3 V. Das ist in Ordnung, Du hast ja schon im Band 1 der Buchreihe erfahren, dass diese Mikrocontroller recht flexibel sind, was ihre Stromversorgung angeht.

Insgesamt haben wir nun zwei Dioden in den Stromversorgung-Pfaden. Und zwar D2 auf dem Arduino Board und die externe D1:



Der Servo bekommt ausschließlich Strom vom Netzteil. Er kann den PC gar nicht belasten, weil D1 keinen Strom vom PC in Richtung Servo fließen lässt.

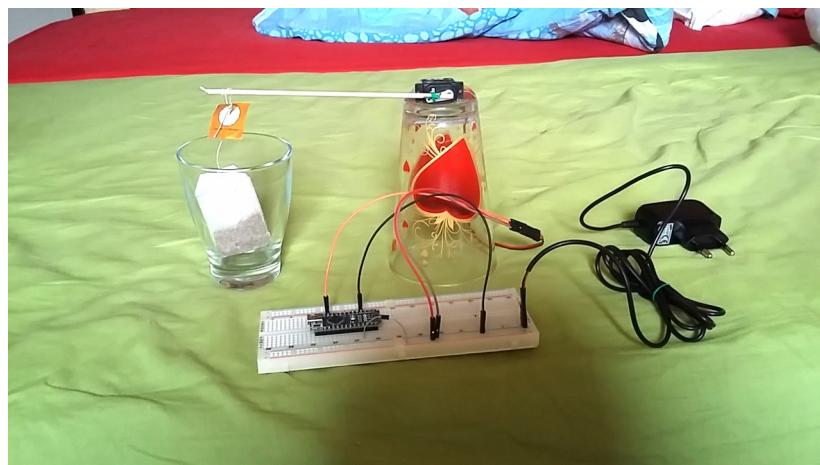
Der AVR Mikrocontroller bekommt von beiden Seiten Strom. Wobei die Seite mit der höheren Spannung gewinnt. Je nach dem, ob links oder rechts mehr Spannung anliegt, wird entweder die Diode D1 oder die Diode D2 leitend. Sie schalten also zwischen den beiden Möglichkeiten um.

Vom Netzteil aus kann kein Strom in den PC fließen, weil die Diode D2 das nicht zulässt. Das ist gut so, denn der PC sollte niemals von außen mit einem Fremden Netzteil versorgt werden.

Diese Methode kannst du bei allen folgenden Schaltungen anwenden, wenn eine wechselweise Stromversorgung über Netzteil oder PC erwünscht ist.

7.5 Tee Roboter

Im diesem Experiment lassen wir uns beim Tee-Kochen von einem Roboter unterstützen.



Zu den elektrischen Teilen, die du seit Anfang des Servo Kapitels verwendest, kommt noch folgendes Material hinzu:

- 1 Hohes Saftglas
- 1 Teeglas
- 1 Teebeutel
- 1 Schaschlik-Stab
- 1 Tesafilm
- 1 Draht zum Anbinden

Binde den Holzstab mit etwas Draht fest an den Hebel des Servos an. Klebe dann den Servo mit Tesafilm oben auf das kopfüber stehende Saftglas. Binde den Teebeutel an das Ende des Holzstabes.

Der Servo wird wie gehabt an den Arduino Anschluss D9 angeschlossen, das entspricht dem Port PB1 vom Mikrocontroller. Das Programm soll folgendes tun:

Nach dem Aufgießen mit heißem Wasser starten wir das Programm durch Druck auf den Reset Taster oder durch Einschalten der Stromversorgung. Zuerst wird der Teebeutel 5 Minuten lang in das heiße Wasser getaucht. Zwischendurch soll er alle 30 Minuten ein paar mal hoch und runter gewackelt werden, damit der Tee besser zieht.

```
#include <stdint.h>
#include <avr/io.h>
#include <util/delay.h>

// Steuerwerte für den Servo
#define HOCH 250
#define MITTE 200
#define RUNTER 150

void wackeln(void)
{
    for (uint8_t i=0; i<3; i++)
    {
        OCR1A = MITTE;
        _delay_ms(500);
        OCR1A = RUNTER;
        _delay_ms(500);
    }
}

int main(void)
{
    initSystemTimer();

    // Starte Timer 1 im Fast PWM Modus
    // mit Prescaler 64 und TOP Wert 4999
    TCCR1A = (1<<COM1A1) + (1<<WGM11);
    TCCR1B = (1<<WGM12) + (1<<WGM13) + (1<<CS10) + (1<<CS11);
    ICR1 = 4999;

    // Aktiviere PWM und LED Ausgänge
    DDRB |= (1<<PB1) + (1<<PB5);

    // LED an
    PORTB |= (1<<PB5);

    // Teebeutel absenken
    OCR1A = RUNTER;

    // 6 mal 30 Sekunden ziehen lassen
    for (uint8_t i=0; i<6; i++)
    {
        _delay_ms(30000);
        wackeln();
    }

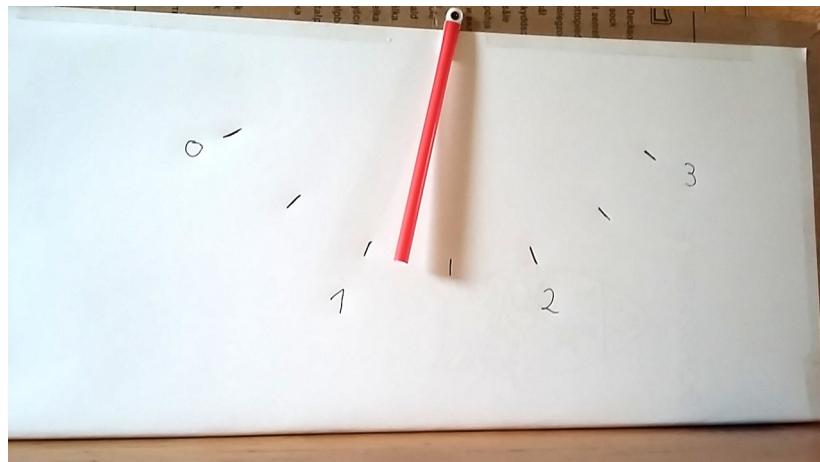
    // Teebeutel anheben
    OCR1A = HOCH;

    // LED aus
    PORTB &= ~(1<<PB5);
}
```

Die drei Werte für den Servo habe ich durch ausprobieren herausgefunden. Da du ein andere Servo hast und deine Gläser auch andere Abmessungen haben, musst du die für dich richtigen Werte selbst herausfinden.

7.6 Zahnpflege-Uhr

Die Zahnpflege-Uhr zeigt an, wie lange wir unsere Zähne putzen sollen.



Zu den elektrischen Teilen, die du seit Anfang des Servo Kapitels verwendest, kommt noch folgendes Material hinzu:

- 1 Pappkarton
- 1 Blatt Papier mit einer Zeitskala, verteilt auf 120°
- 1 Bunter Trinkhalm
- 1 Tesafilm

Bei der Zeitskala ist zu beachten, dass der Servo den gesamten Bereich auch erreichen müssen. Die meisten Servos können etwas weniger als einen Halbkreis (das wäre 180°) abdecken. Wenn wir für die Skala 120° verwenden, sind wir im sicheren Bereich und das ist mit einem Geometrie-Dreieck leicht zu bewerkstelligen.

Oben im Mittelpunkt der Zeitskala machst du ein kleines Loch mit 5mm Durchmesser. Schraube den Hebel vom Servo ab. Stecke dann den Servo von hinten durch das Loch und bringe den Hebel wieder an. Den Servo klebst du nun mit Tesafilm fest, dann steckst du den Trinkhalm auf den Servo-Hebel und schneidest ihn auf passende Länge.

Der Servo wird an den Arduino Anschluss D9 angeschlossen, das ist der Port PB1 vom Mikrocontroller. Das Programm dazu sieht so aus:

```
#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

// Steuerwerte für die beiden End-Positionen
#define LINKS 530
#define RECHTS 215

#define TEILER (9000/(LINKS-RECHTS))

// Wird fortlaufend hoch gezählt,
// erreicht in 3 Minuten den Wert 9000
volatile uint16_t zeit=0;

// Wird alle 20ms aufgerufen
ISR(TIMER1_OVF_vect)
{
    zeit++;
}
```

```

}

int main(void)
{
    // Starte Timer 1 im Fast PWM Modus
    // mit Prescaler 64 und TOP Wert 4999
    TCCR1A = (1<<COM1A1) + (1<<WGM11);
    TCCR1B = (1<<WGM12) + (1<<WGM13) + (1<<CS10) + (1<<CS11);
    ICR1 = 4999;

    // Aktiviere Timer Interrupt
    TIMSK1 = (1<<TOIE1);
    sei();

    // Aktiviere PWM und LED Ausgänge
    DDRB |= (1<<PB1) + (1<<PB5);

    // LED an
    PORTB |= (1<<PB5);

    // Bewege den Zeiger, solange wir unter 3 Minuten sind
    while (1)
    {
        cli();
        uint16_t x=zeit;
        sei();

        OCR1A = LINKS-x/TEILER;

        if (x==9000)
        {
            break;
        }
    }

    // Ende
    // Bewege den Zeiger zurück nach Links
    OCR1A = LINKS;

    // LED aus
    PORTB &= ~(1<<PB5);
}

```

Ganz oben im Programmcode stehe zwei Definitionen die End-Positionen des Zeigers. Da der Zeiger nach unten hängt, bewegt er sich seiten- verkehrt. Der Steuerwert für Links ist nun größer, als der Steuerwert für rechts.

```
#define LINKS 530
#define RECHTS 215
```

Diese Werte habe ich durch Ausprobieren heraus bekommen. Ich habe erst mit 500 und 250 angefangen und dann die Zahlen oft geändert, bis der Servo den Zeiger auf die richtigen Stellen für „0 Minuten“ und „3 Minuten“ bewegt. Da du nicht genau den gleichen Servo hast, wirst du diese Zahlen wohl selbst nochmal anpassen müssen.

Ich habe den Timer 1 sowohl zur Ansteuerung des Servos verwendet, als auch um die Zeit zu messen. Jedesmal, wenn der Servo-Timer auf 0 zurück springt (also alle 20ms), löst er einen Interrupt aus. Und in der Interrupt-Routine wird dann jedesmal die Variable „zeit“ um eins erhöht. In drei Minuten hat die Variable den Wert 9000 erreicht.

Die Definition

```
#define TEILER (9000/(LINKS-RECHTS))
```

berechnet, durch wie viel ich die 9000 dividieren muss, um korrekte Steuerwerte für den Servo zu erhalten. Wir wollen ja, dass der Zeiger beim Wert 9000 nach rechts auf die „3 Minuten“ Marke bewegt wird.

Weiter unten im Hauptprogramm findest du die while Schleife:

```
while (1)
{
    cli();
    uint16_t x=zeit;
    sei();

    OCR1A = LINKS-x/TEILER;

    if (x==9000)
    {
        break;
    }
}
```

Zuerst kopieren wir den Zeit-Zähler in die Variable x. Dann setzen wir den Steuerwert des Servos. Je weiter die Zeit fortgeschritten ist, umso größer ist x und umso weiter wird der Zeiger nach rechts bewegt.

Wenn wir 9000 = 3 Minuten erreicht haben, brechen wir die while Schleife ab. Danach wird die LED aus geschaltet und der Zeiger zurück nach links bewegt. Das Programm endet an dieser Stelle. Für einen Neustart musst du auf den Reset-Taster des Arduinos drücken.

Du kannst auch einen externen Reset Taster an das Arduino Board anschließen, oder du verwendest zur Stromversorgung vier Batterien und einen ein/aus Schalter. Dann startet die Uhr immer neu, wenn der Strom eingeschaltet wird.

7.7 Exkurs: Variablen in Interruptroutinen

Die Variable „zeit“ Bedarf hier besonderer Aufmerksamkeit, weil die sowohl innerhalb als auch außerhalb der Interruptroutine verwendet wird.

Normalerweise geht der C Compiler davon aus, dass der Programmcode so abläuft, wie man es am Quelltext ablesen kann. Er optimiert den Zugriff auf Variablen dementsprechend. Zum Beispiel, wenn wir schreiben:

```
int zeit=3;
int b=zeit+1; // Ergibt 4
int c=zeit+2; // Ergibt 5
int d=zeit+3; // Ergibt 6
```

Dann haben wir anscheinend drei Lesezugriffe auf die Variable „zeit“, nämlich um dreimal etwas zu addieren. Der Compiler geht jedoch davon aus, das zwischen diesen Zeilen niemand an der Variable „zeit“ herum fummelt und optimiert den Maschinencode dementsprechend.

Nach der ersten Addition hat der Prozessor den Wert von „zeit“ in irgendeinem Zwischenregister. Für die zweite und dritte Addition wird er die Variable „zeit“ gar nicht mehr lesen, sondern wieder das selbe Zwischenregister verwenden. Der Maschinencode wird dadurch kompakter und schneller.

In Zusammenhang mit Interruptroutinen kann das aber zu unerwartetem verhalten führen. Stelle dir vor, die erste Addition wurde durchgeführt und dann löst ein Timer einen Interrupt aus, welche den Wert von „zeit“ ändert.

```
int zeit=3;
int b=zeit+1; // Ergibt 4
```

Unterbrechung: zeit wird auf 20 geändert

```
int c=zeit+2; // Ergibt 5, müsste aber 22 sein.
int d=zeit+3; // Ergibt 6, müsste aber 23 sein.
```

Dann müsste die zweite Addition mit dem neuen Wert von „zeit“ weiter rechnen. Tut sie aber nicht, weil die zweite Addition das Zwischenregister von der ersten Addition weiter verwendet. Und auch die dritte Addition wird nun ein falsches Ergebnis liefern.

Um diese Fehlfunktion zu verhindern, deklarieren wir die Variable „zeit“ als „volatile“. Der Compiler wird dadurch angewiesen, die Variable „zeit“ wirklich immer neu zu lesen.

```
volatile int zeit=3;
int b=zeit+1; // Ergibt 4
```

Unterbrechung: zeit wird auf 20 geändert

```
int c=zeit+2; // Ergibt 22
int d=zeit+3; // Ergibt 23
```

Bei der „zeit“ Variable haben wir ein ganz ähnliches Szenario. Auch sie wird in der Interruptroutine verändert – sogar ziemlich oft, 50 mal pro Sekunde. Und im Hauptprogramm wird sie gelesen.

Die zweite Besonderheit betrifft Variablen, die größer als 8 Bit sind, was auf unsere „zeit“ Variable zutrifft. Da der Mikrocontroller einen 8 Bit Prozessorkern hat, muss er größere Variablen zerlegen und in mehreren Teilschritten abarbeiten. Dabei können Interrupts stören. Ein Beispiel:

```
volatile int zeit=0;

while (1)
{
    if (zeit<9000)
    {
        tu irgendwas;
    }
}
```

Hier wird in der Endlosschleife immer wieder geprüft, ob die Variable „zeit“ kleiner als 9000 ist. Der Prozessorkern muss diese 16 Bit Variable in zwei Hälften zu je 8 Bit zerlegen, um sie zu verarbeiten. Unser Vergleichsoperator führt daher zu einem komplexeren Maschinencode, der ungefähr das macht:

1. Setze „zeit“ auf 0.
2. Hole die oberen 8 Bit von Variable „zeit“ in das Zwischenregister R19.
3. Hole die unteren 8 Bit von Variable „zeit“ in das Zwischenregister R18.
4. Wenn R19 größer als 35 ist, gehe zu 7.
5. Wenn R18 größer oder gleich 40 ist, gehe zu 7.
6. tu irgendwas
7. Gehe wieder zu 1

Das funktioniert prima, bis die Variable „zeit“ von 9215 nach 9216 wechselt. In diesem Fall dürfte der Mikrocontroller die Zeile „tu irgendwas“ nicht ausführen, denn beide Zahlen sind größer als 9000. Tatsächlich kommt es jedoch zu folgender Fehlfunktion:

1. „zeit“ ist 9215 (=35 in den oberen 8 Bit und 255 in den unteren 8 Bit)
2. Hole die oberen 8 Bit von Variable „zeit“ in das Zwischenregister R18 (=35).
3. Interrupt: Setzt „zeit“ auf 9216 (=36 in den oberen 8 Bit und 0 in den unteren 8 Bit)
4. Hole die unteren 8 Bit von Variable „zeit“ in das Zwischenregister R18 (=0).
5. Wenn R19 größer als 35 ist, gehe zu 8. (ist nicht der Fall)
6. Wenn R18 größer oder gleich 40 ist, gehe zu 8. (ist nicht der Fall)
7. tu irgendwas
8. Gehe wieder zu 1

Dass heißt, „tu irgendwas“ wird fälschlicherweise ausgeführt, obwohl der Wert von „zeit“ sowohl vor als auch nach dem Interrupt weit über 9000 war.

Du magst einwenden, dass man halt die beiden Bytes in umgekehrter Reihenfolge einlesen soll. Richtig ist, dass dieses konkrete Problem dann nicht mehr auftritt, aber dann gibt es andere Zahlenkombinationen, die Fehlfunktionen auslösen. Eine allgemeingültige Lösung, die der Compiler umsetzen könnte, gibt es nicht.

Deswegen müssen wir Programmierer unser Hirn einsetzen und dem Compiler helfen. Die einfachste Lösung besteht darin, Interrupts an der kritischen Stelle zu verbieten:

```
while (1)
{
    cli();
    if (zeit<9000)
    {
        tu irgendwas;
    }
    sei();
}
```

So funktioniert es richtig, ist allerdings suboptimal, wenn „tu irgendwas“ mehr als nur ein oder zwei Zeilen Code groß ist. Denn wenn du Interrupts über einen längeren Zeitraum sperrst, werden Unterbrechungs-Ereignisse übersprungen. Bei einer Uhr würde das dazu führen, dass die Uhr zu langsam läuft. Bei einem Bedienfeld könnte es dazu führen, dass einzelne Tastendrücke ignoriert werden, was den Benutzer verärgern wird.

Um die Zeit der gesperrten Interrupts auf ein Minimum zu reduzieren, kopiert man die fragliche Variable einfach in eine andere:

```
while (1)
{
    cli();
    int x=„zeit“;
    sei();

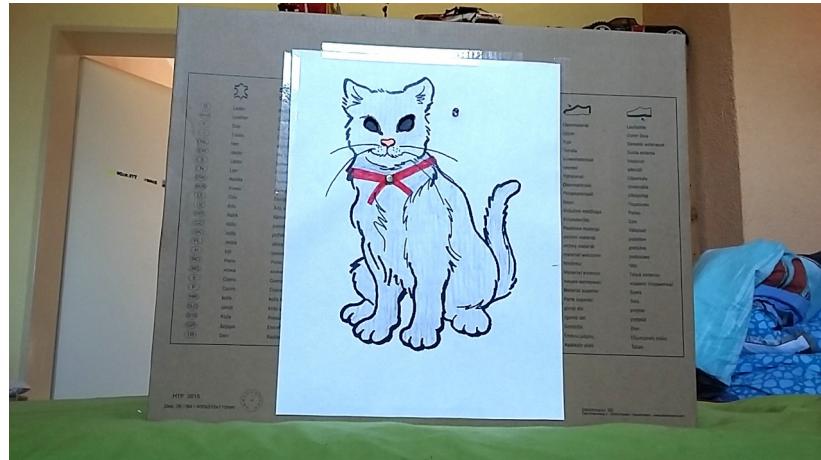
    if (x<9000)
    {
        tu irgendwas;
    }
}
```

Jetzt spielt es keine Rolle mehr, wie lange die Funktion „tu irgendwas“ dauert. Wir sperren Interrupts nur ganz kurz, gerade lang genug, den Wert von „zeit“ nach x zu kopieren. Danach kann uns völlig egal sein, ob ein Interrupt die Variable nochmal verändert, denn wir arbeiten ja ab jetzt mit der Kopie x weiter. Wir vergleichen x mit 9000.

7.8 Vampirkatze

Die gemeine Vampirkatze schläft den ganzen Tag. Nur nachts, wenn es dunkel ist, öffnet sie ihre Augen. Das ist natürlich frei erfunden – aber egal. Wir werden eine solche Katze bauen.

Ein Phototransistor wird die Helligkeit im Raum messen und ein Servo-Antrieb wird die Augen öffnen und schließen.

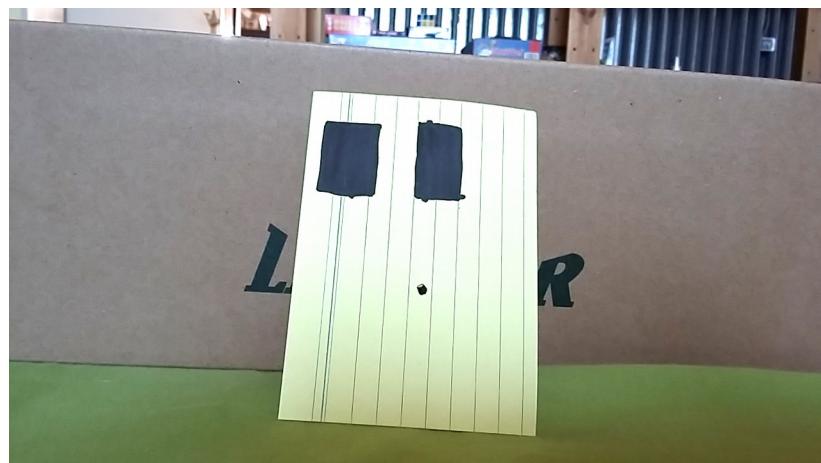


Der ultimative Beweis für meine künstlerische Begabung ☺

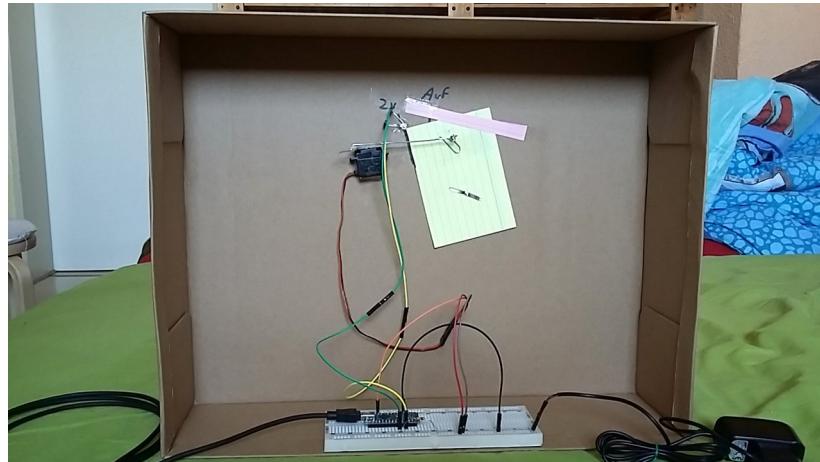
Zu den elektrischen Teilen, die du seit Anfang des Servo Kapitels verwendest, kommt noch folgendes Material hinzu:

- 1 Pappkarton
- 1 Blatt Papier, mit einer Katze bemalt
- 1 Musterbeutelklammer
- 1 Tesafilm
- 1 Stück gelber Karton (Din A6) für die Augen
- 1 Schwarzer Filzstift
- 2 Büroklammern

Das Katzenbild kleben wir von außen auf den Pappkarton. Die Augen werden mit dem Bastelmesser ausgeschnitten. Der gelbe Karton soll die geöffneten Augen darstellen. Wir malen darauf zwei schwarze Flecken, so dass die Augen durch Verschieben des Kartons wechselweise gelb oder schwarz erscheinen.



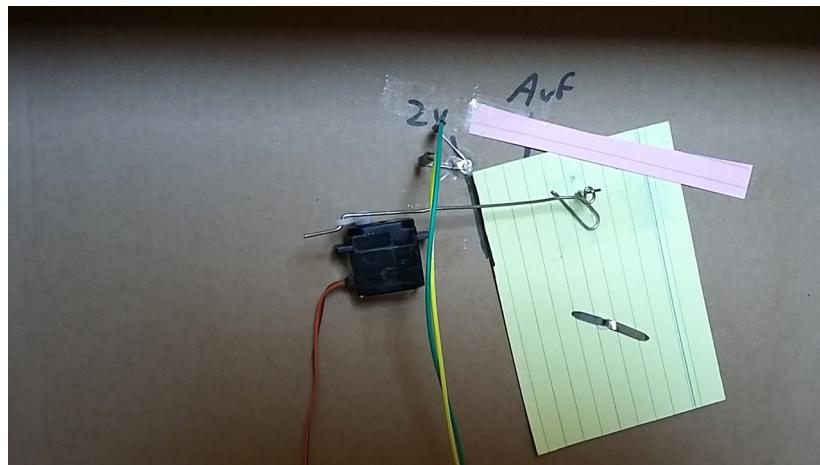
Dann befestigen wir den gelben Karton mit der Musterbeutelklammer an der Rückseite der Katze. Ich habe zusätzlich noch einen rosa Streifen Karton angebracht, um die Stabilität zu verbessern. Wichtig ist, dann man den gelben Karton nun so hin und her schieben kann, dass die Augen mal schwarz und mal gelb erscheinen.



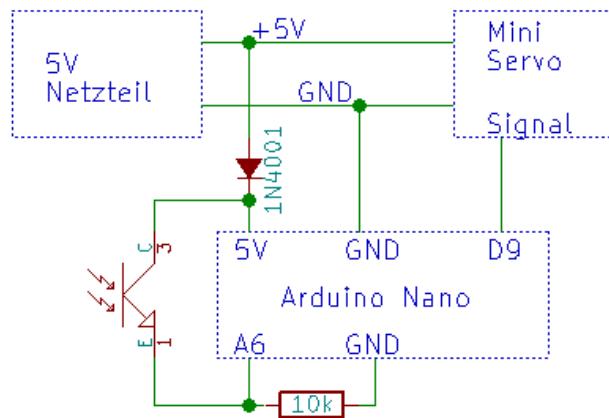
Aus einer Büroklammer formen wir einen Haken, an dem der gelbe Karton später hin und her geschoben wird. Diesen kleben wir mit Tesafilm am oberen drittel des Kartons fest. Von der Seite betrachtet sieht der Haken so aus:



Der Phototransistor soll neben dem Katzenkopf durch den Karton gesteckt werden und mit Tesafilm fixiert werden. Aus der zweiten Büroklammer formen wir eine Schubstange, die den Hebel des Servos mit dem Haken des Augen-Kartons verbindet. Danach befestigen wir den Servo mit Tesafilm.



Der Servo wird wieder an den Arduino Anschluss D9 angeschlossen, das ist beim Mikrocontroller der Port PB1. Im obigen Foto führen die grünen und gelben Kabel zum Phototransistor. Der wird wie folgt an den analogen Eingang 6 angeschlossen.



Der Phototransistor bildet zusammen mit dem $10\text{ k}\Omega$ Widerstand einen Spannungsteiler. Je heller es ist, umso höher ist die Spannung am Pin A6.

Die 5 V Versorgung bezieht der Phototransistor vom Arduino Board, nicht vom 5 V Netzteil. Der Phototransistor braucht unbedingt exakt die gleiche Versorgungsspannung, wie der Mikrocontroller, damit der ADC bei konstanter Helligkeit auch konstante Werte liefert.

Seit hier besonders sorgfältig! Wenn du die 5 V Leitung versehentlich kurz schließt, geht die Diode auf dem Arduino Board kaputt, oder eventuell sogar der USB Port des Computers.

Kontrolliere mit einem Digital-Multimeter die analoge Spannung (an A6 und GND). Sie sollte bei normaler Raumhelligkeit deutlich über 0,1 V liegen und bei Dunkelheit bis auf annähernd 0 V runter gehen. Wenn das bei dir nicht der Fall ist, hast du den Phototransistor wahrscheinlich falsch herum angeschlossen.

```
#include <stdio.h>
#include <stdint.h>
#include <avr/io.h>
#include <util/delay.h>
#include "driver/serialconsole.h"

uint16_t readADC(uint8_t channel)
{
    // ADC verwendet VCC als Referenz und Eingang 6
    ADMUX = (1<<REFS0) + channel;
    // ADC einschalten und Messung starten
    ADCSRA = (1<<ADEN) + (1<<ADSC) + (1<<ADPS0) + (1<<ADPS1) + (1<<ADPS2);
```

```

// Warte
while (ADCSRA & (1<<ADSC)) {};
// Ergebnis zurück geben
return ADC;
}

int main(void)
{
    initSerialConsole();

    // Starte Timer 1 im Fast PWM Modus
    // mit Prescaler 64 und TOP Wert 4999
    TCCR1A = (1<<COM1A1) + (1<<WGM11);
    TCCR1B = (1<<WGM12) + (1<<WGM13) + (1<<CS10) + (1<<CS11);
    ICR1 = 4999;

    // Aktiviere PWM und LED Ausgänge
    DDRB |= (1<<PB1) + (1<<PB5);

    while (1)
    {
        uint16_t messwert=readADC(6);
        printf("Messwert=%ud\n",messwert);

        // Wenn es hell ist
        if (messwert>100)
        {
            OCR1A = 470;
        }

        // Wenn es dunkel ist
        else
        {
            OCR1A = 240;
        }

        // Ein bisschen warten
        _delay_ms(1000);
    }
}

```

In der Datei hardware.h stellst du die Baudrate der seriellen Schnittstelle ein.

```

#ifndef _HARDWARE_H_
#define _HARDWARE_H_

#include <avr/io.h>

// See also serial port settings in driver/serialconsole.h.
#define SERIAL_BITRATE 2400
#define USE_SERIAL0

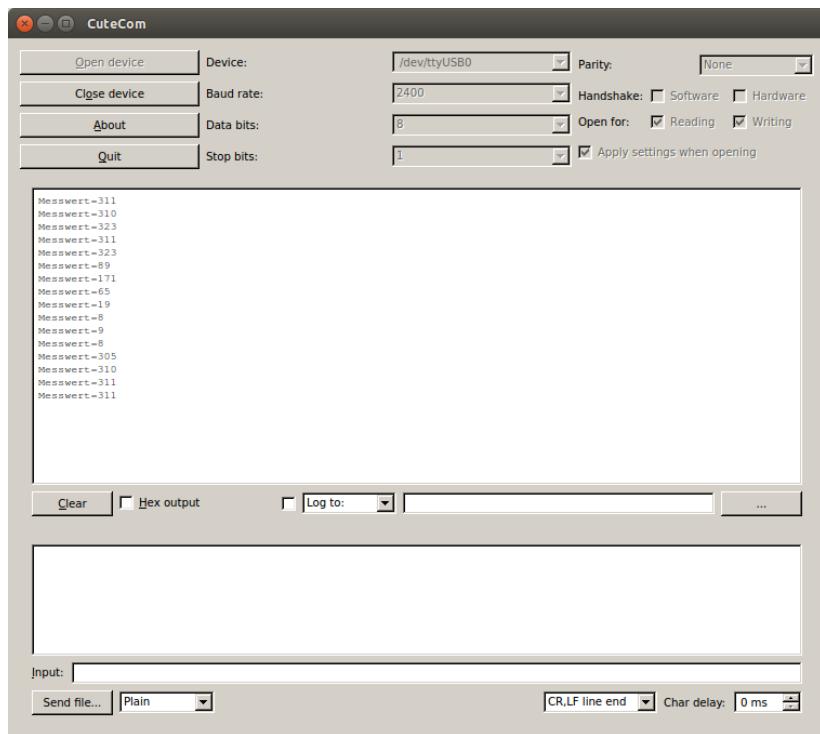
#endif // _HARDWARE_H_

```

Das Programm nutzt den ADC, um die Spannung vom Lichtsensor (Phototransistor) auszulesen. Mit Hilfe der seriellen Konsole aus der „Hello World“ Vorlage, sendet es die gemessenen Werte an den PC.

Starte auf dem PC ein Terminalprogramm, zum Beispiel Cutecom oder das „Hammer Terminal“. Unter Linux gibst bitte den Befehl „sudo hterm“ ein, damit du Zugriff auf den USB Anschluss bekommst. Stelle die richtige Baudrate und den seriellen Port ein.

Auf meinem Linux Computer ist das der virtuelle serielle Port /dev/ttyUSB0 und die Baudrate ist 2400. Unter Windows würde der serielle Port vielleicht COM3 heißen.



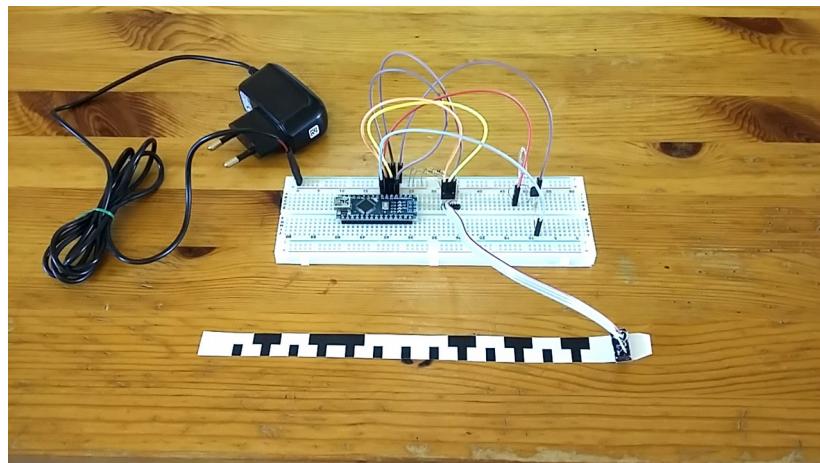
Im Programm steht „if (messwert>100)“ um zu ermitteln, ob es hell ist. Ich habe hier einen Zahlenwert genommen, der eindeutig zwischen den Messwerten für hell und dunkel liegt. Wenn du einen anderen Phototransistor verwendest, musst du wahrscheinlich eine andere Zahl als die 100 einsetzen.

Berücksichtige, dass der USB Port nur von einem Programm gleichzeitig belegt werden kann. Wenn du das nächste mal „make program“ ausführst, musst du das Terminal Programm vorher beenden.

Je nach dem, ob es nun hell oder dunkel ist, wird der Servo nach links oder rechts gefahren. Die exakten Positionsweite 470 und 270 habe ich experimentell herausgefunden. Sie hängen von den Abmessungen der mechanischen Konstruktion ab.

8 Code Scanner

Jetzt bauen wir einen optischen Code-Scanner, der so ähnlich funktioniert, wie die Scanner an der Supermarkt Kasse. Bei unserem Scanner dient ein schmaler Papierstreifen als Schlüssel. Damit kannst du zum Beispiel eine Schatzkiste öffnen.



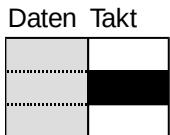
Material:

- 1 Arduino Nano Board
- 2 Optische Sensoren Typ CNY70
- 2 Widerstände 47kOhm ¼ Watt
- 1 Widerstand 220 Ohm ¼ Watt
- 1 Widerstand 2,2k Ohm ¼ Watt
- 1 Transistor BC337-40
- 1 Diode 1N4148
- 1 Glühlämpchen für 5 V maximal 1 Watt
- 1 Steckbrett und Kabel
- 1 Netzteil 5 V mindestens 500 mA
- 1 Tesafilm und Kleber
- 1 Codestreifen aus Laserdrucker (siehe nächste Seite)

Wenn man den richtigen Code-Streifen durch den Sensor zieht, geht die Lampe kurz an. Wenn man den falschen Streifen verwendet oder ihn falsch herum durch zieht, bleibt die Lampe dunkel.

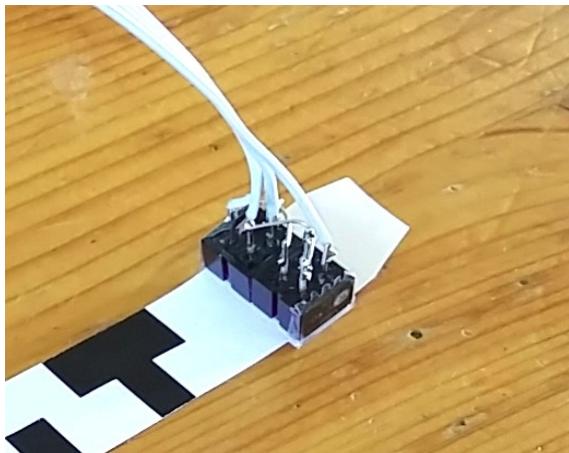
Den Code-Streifen habe ich mit einem Tabellen-Programm (LibreOffice) erstellt. Die Kästchen müssen 7mm breit sein. Die Höhe der Kästchen sollte mindestens 5mm betragen. Am besten funktioniert es, wenn man zum Ausdrucken einen Laserdrucker verwendet.

Die linke Spalte enthält den Code des Schlüssels. Die rechte Spalte liefert ein Taktsignal, das beim Auslesen hilfreich ist. Jedes Bit besteht aus 6 Kästchen:

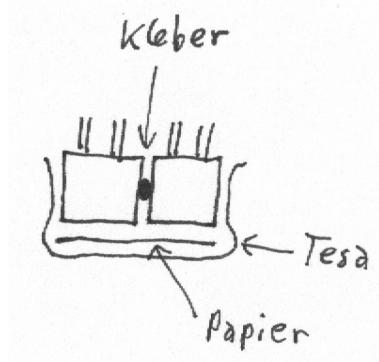


Der Codestreifen enthält von oben nach unten 13 solcher Bits. Wenn wir schwarz als 1 werten und weiß als 0, dann enthält dieser Streifen den Code „1010100011010“. Das entspricht der Dezimalzahl 5402.

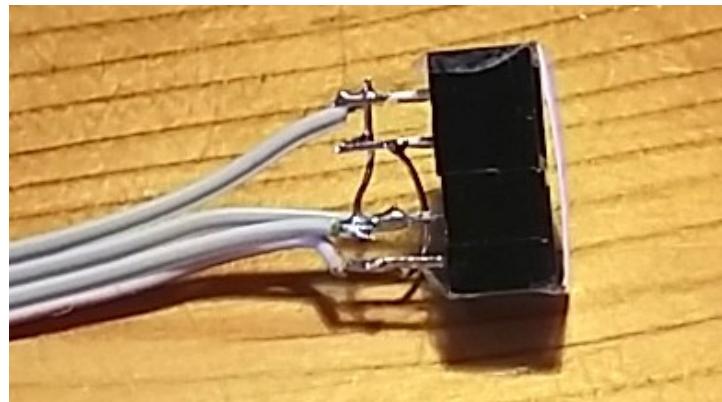
Jetzt müssen wir einen optischen Sensor bauen, der diesen Streifen einlesen kann.



Dazu kleben wir zwei sogenannte Reflex-Lichtschranken aneinander und kleben eine Führung aus Papier und Tesafilm an. Es bleibt ein schmaler Spalt, durch den wir den Codestreifen ziehen können.



Auf diesem Foto kann man den schmalen Spalt zwischen Papier und Sensor erkennen:

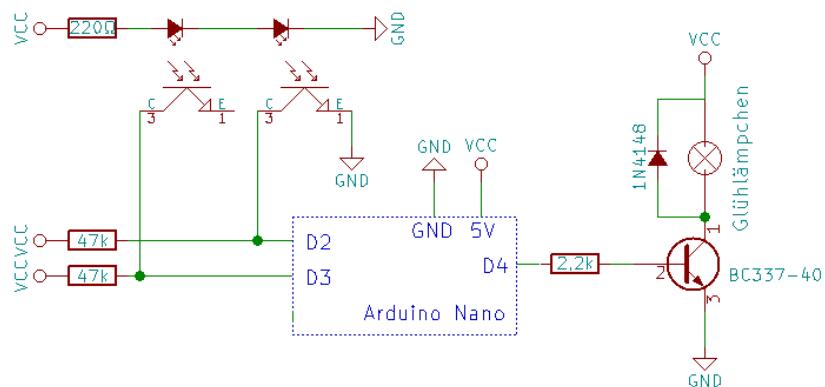


Die beiden CNY70 Sensoren strahlen ständig unsichtbares Infrarot-Licht aus. Das vom Papier reflektierte Licht fällt auf die Phototransistoren, die dann ein Low-Signal an den Mikrocontroller senden.

Wenn hingegen ein schwarzer Fleck vor dem Sensor liegt, dann wird fast kein Licht reflektiert so dass der Phototransistor keinen Strom ableitet. Der Mikrocontroller erhält dann einen High Pegel. Auf meinem Billig-Oszilloskop sieht das so aus:



Die gelbe Linie zeigt das Signal von der Takt-Spur und die blaue Linie zeigt das Signal von der Daten-Spur. Nun zum Schaltplan:



Die 47k Ohm Widerstände wirken parallel zu den internen pull-up Widerständen des Mikrocontrollers. Ich hatte zunächst gehofft, dass die internen Pull-Up Widerstände des Mikrocontrollers ausreichen, aber da waren die High Pegel nicht hoch genug. Deswegen habe ich unterschiedliche zusätzliche Pull-Up Widerstände ausprobiert, bis sowohl High Pegel als auch Low Pegel passten. Low muss unter als 1 V liegen und High muss mehr als 3 V sein.

Da die Glühlampe mehr Strom verbraucht, als der Mikrocontroller liefern kann, haben wir am Ausgang D4 einen Transistor zur Verstärkung. Dort kannst du auch ein Relais anschließen, welches größere Stromverbraucher ansteuert. Die Diode am Ausgang dient als Freilaufdiode (Erklärung siehe Band 2). Sie ist notwendig, wenn du ein Relais oder einen Motor anschließt. Für die Glühlampe ist sie nicht notwendig, schadet aber auch nicht.

Das Programm basiert wieder auf der „Hello World“ Vorlage. Im Makefile gibt's du wieder wie bisher die Parameter für Avrdude und den Mikrocontroller-Typ an:

```
# Programmer hardware settings for avrdude
AVRDUDE_HW = -c arduino -P /dev/ttyUSB0 -b 57600

# Name of the program without extension
PRG = Code

# Microcontroller type and clock frequency
MCU = atmega328p
F_CPU = 16000000
```

In der Datei hardware.h stellen wir die Baudrate der seriellen Schnittstelle ein und sammeln Makro Definitionen für den Zugriff auf die I/O Pins:

```
#ifndef _HARDWARE_H_
#define _HARDWARE_H_

#include <avr/io.h>

// See also serial port settings in driver/serialconsole.h.
#define SERIAL_BITRATE 2400
#define USE_SERIAL0

// Macros for I/O Pins
#define LED_AN { DDRB |= (1<<PB5); PORTB |= (1<<PB5); }
#define LED_AUS { PORTB &= ~(1<<PB5); }

#define LAMPE_AN { DDRD |= (1<<PD4); PORTD |= (1<<PD4); }
#define LAMPE_AUS { PORTD &= ~(1<<PD4); }

#define INIT_SENSOR { PORTD |= (1<<PD2) + (1<<PD3); }
#define SENSOR_TAKT ( PIND & (1<<PD3) )
#define SENSOR_DATEN ( PIND & (1<<PD2) )

#endif // _HARDWARE_H_
```

Das Makro INIT_SENSOR schaltet die internen Pull-Up Widerstände für die beiden optischen Sensoren ein. Dies ist der Quelltext des Programms in main.c:

```
#include <stdio.h>
#include <stdint.h>
#include <util/delay.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
```

```

#include "hardware.h"

uint16_t einlesen(void)
{
    uint16_t code=0;

    // Warte ewig auf das erste Takt-Signal
    while (!SENSOR_TAKT) {};
    LED_AN;

    while (1)
    {
        code=code<<1;
        if (SENSOR_DATEN)
        {
            code=code+1;
        }

        // Warte auf Ende des aktuellen Taktimpulses
        while (SENSOR_TAKT) {};

        // warte maximal 500ms auf nächsten Takt
        timer_t letzterTakt=milliseconds();
        while (!SENSOR_TAKT)
        {
            if (milliseconds()-letzterTakt > 500)
            {
                // Timeout, liefere den Code ab
                LED_AUS;
                return code;
            }
        }
    }
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    INIT_SENSOR;
    while(1)
    {
        LAMPE_AUS;
        uint16_t code=einlesen();
        printf("Code=%u\n",code);
        if (code==5402)
        {
            LAMPE_AN;
            _delay_ms(2000);
        }
    }
}

```

In der Funktion einlesen() geht die LED des Arduino Boardes an, sobald das erste Taktsignal erkannt wird. Dann wird ein Bit nach dem anderen eingelesen, bis 500ms lang kein Takt mehr kommt. Danach geht die LED wieder aus. Das Taktsignal bestimmt also, wann die Daten Bits eingelesen werden.

Beim Einlesen von jedem Bit wird der Inhalt der Variable „code“ immer einen Schritt nach links geschoben und dann wird das Daten-Bit addiert (falls es High ist). Auf diese Weise kannst du Schlüssel beliebiger Länge bis maximal 16 Bit einlesen. Mehr Bits passen in die Variable „code“ nicht rein.

Die Funktion einlesen() benutzt den Systemtimer aus der „Hello World“ Vorlage, um zu erkennen, wann die 500ms abgelaufen sind. Wenn du dir den Quelltext von driver/systemtimer.c anschaugst, wirst du sehen, dass er ganz ähnlich funktioniert, wie die Zeitmessung bei der Zahnpulz-Uhr.

Die serielle Konsole wird von der main() Funktion verwendet, um den eingescannten Code anzuzeigen. Wenn der Code richtig war (=5402), dann wird die Lampe eingeschaltet. Die serielle Konsole hilft bei der Fehlersuche, falls der Code mal falsch eingelesen wird.

8.1 Exkurs: Zeitmessung mit Überlauf

Die Funktion einlesen() benutzt den Systemtimer aus der „Hello World“ Vorlage, um zu erkennen, wann die 500 ms abgelaufen sind. Hier passiert etwas ganz besonderes, wenn der Zeitzähler zwischenzeitlich überläuft.

Der Zeitzähler (den milliseconds() abfragt) wird jede Millisekunde um eins erhöht, also 1,2,3,4...65535. Das ist der höchst mögliche Wert, weil die Zählvariable 16Bit hat. Eine Millisekunde später springt der Zähler wieder auf 0 um erneut hoch zu zählen.

Im Programmcode berechnen wir die Anzahl der verstrichenen Millisekunden, indem wir zwei Zeitwerte voneinander subtrahieren:

```
timer_t letzterTakt=milliseconds();
...
if (milliseconds()-letzterTakt > 500)
{
    ...
}
```

Wenn wir zum Beispiel bei Zählerstand 0 beginnen, dann haben wir 500ms später die Berechnung:

- $500 - 0 = 500$

Und wenn wir bei Zählerstand 14000 beginnen, dann haben wir 500ms später die Berechnung:

- $14500 - 14000 = 500$

Was aber, wenn der Zähler zwischendurch überläuft? Zum Beispiel:

- $300 - 65336 = 500$

Das ergibt laut Taschenrechner -65036, aber der Mikrocontroller kommt stattdessen auf 500. Das ist super praktisch, denn es sind ja auch 500 Millisekunden verstrichen. Aber wie kommt er nur darauf?

Laut Taschenrechner ist -65036 in Binär-Darstellung:

- 1111 1111 1111 1111 0000 0001 1111 0100

Da die Berechnung auf dem Mikrocontroller aber mit 16 Bit Variablen stattfindet, schneidet er die ganzen Einsen links davor weg. Übrig bleibt:

- 0000 0001 1111 0100

Und das entspricht (auch laut Taschenrechner) der Zahl 500.

Der Überlauf des Zählers und die Ungenauigkeit der Berechnung heben sich also gegenseitig auf. Darauf kannst du dich verlassen. Wenn du von einer Zeit eine andere Zeit subtrahierst, erhältst du immer die korrekte Differenz, selbst wenn zwischendurch ein Überlauf stattfindet. Es müssen nur zwei Rahmenbedingungen erfüllt sein:

- Die beiden Variablen, die voneinander subtrahiert werden, müssen vom selben Integer-Typ sein.
- Der Zähler darf zwischenzeitlich nur einmal überlaufen.

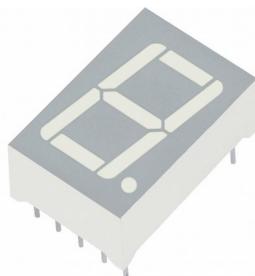
Wenn du längere Zeitspannen als 65 Minuten messen möchtest, musst du in der Datei driver/systemtimer.h die Definition von timer_t ändern. Der nächst größere 32 Bit Integer genügt für 4,2 Millionen Minuten oder 3,2 Jahre.

8.2 Anregungen zum Erweitern

- Anstatt die Lampe einzuschalten, könntest du ein Relais anschließen, welches den Türöffner des Hauses betätigt.
- Du könntest einen Servo ansteuern, welcher die Verriegelung einer Schatzkiste öffnet.
- Anstatt immer nur mit dem Code 5402 zu vergleichen, könntest du mehrere unterschiedliche Codes zulassen.
- Wenn einige male hintereinander ein falscher Code eingescannt wurde, könntest du dafür sorgen, dass das Schloss sicherheitshalber für eine Weile gesperrt wird.
- Füge eine Diode hinzu (siehe Kapitel Exkurs: Dioden in Stromversorgung), damit der Mikrocontroller auch ohne USB Kabel funktioniert.

9 Sieben-Segment Anzeigen

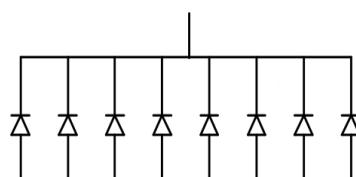
Sieben Segment Anzeigen sind einfach anzusteuern und sehr gut abzulesen, weil sie groß sind und eine hohe Leuchtkraft besitzen.



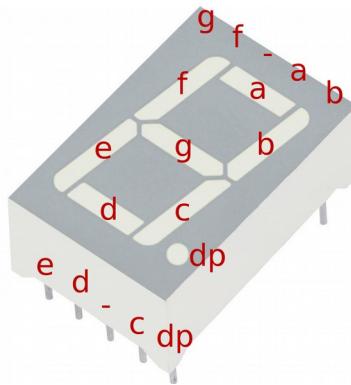
Material:

- 1 Arduino Nano
- 2 Transistoren BC337-40
- 2 Widerstände 2,2 kΩ ¼ Watt
- 8 Widerstände 220 Ω ¼ Watt
- 1 Widerstand 100 kΩ ¼ Watt
- 1 Widerstand 10 kΩ ¼ Watt
- 2 Kondensatoren 100 nF
- 2 7-Segment Anzeigen mit gemeinsamer Kathode, 14-20 mm hoch
- 2 Kurzhubtaster
- 1 Reed-Kontakt
- 1 Kleiner Magnet
- 1 NTC Temperaturfühler 10 kΩ bei 25 °C, zum Beispiel TTC05103JSY
- 1 Steckbrett und Kabel

Sieben-Segment Anzeigen gibt es in unterschiedlichen Größen und Farben zu kaufen. Je nach Bauart sind entweder alle Anoden (+ Pole) oder alle Kathoden (- Pole) miteinander verbunden. Wir werden Anzeigen mit gemeinsamer Kathode verwenden:



Die sieben Striche nennt man „Segmente“. Sie werden mit den Buchstaben „a“ bis „g“ benannt. Der Dezimalpunkt wird „dp“ genannt. Im folgenden Bild habe ich die Segmente und die zugehörigen Anschlüsse beschriftet.



Die Position der Sieben Segmente a-g ist immer gleich. Die Anschlussbelegung ist jedoch nicht immer gleich. Schau daher am Besten in das Datenblatt von deiner Anzeige.

Bei den kleinen 7-Segment Anzeigen befindet sich hinter jedem Segment eine LED. Es gibt auch größere Anzeigen mit mehreren LED's pro Segment. Für den direkten Anschluss an Mikrocontroller eignen sich allerdings nur die Anzeigen mit maximal zwei LED's pro Segment, weil die größeren mehr als 5 V benötigen.

9.1 Ansteuerung

9.1.1 Einfache Anzeige

Als erstes wollen wir versuchen, die sieben Segmente nacheinander aufzuleuchten zu lassen. Den Dezimalpunkt schließen wir nicht an. Baue dazu folgende Schaltung auf dem Steckbrett auf:

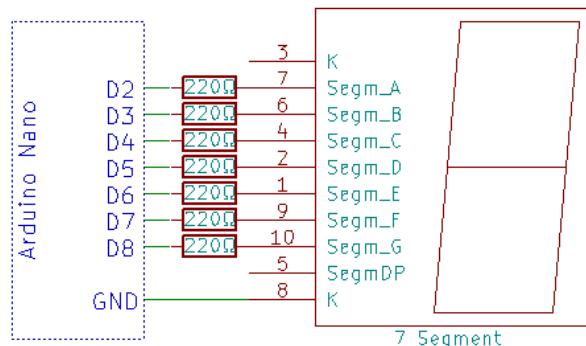
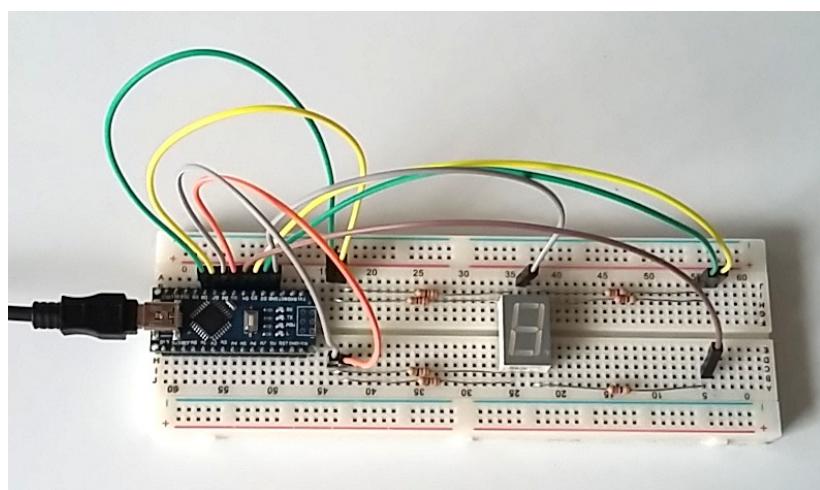


Foto:



Zum Programmieren starten wir wieder mit der „Hello World“ Vorlage. Im Makefile geben wir wie gewohnt die Parameter für Avrdude ein, sowie den Mikrocontroller Typ:

```
# Programmer hardware settings for avrdude
AVRDUDE_HW = -c arduino -P /dev/ttyUSB0 -b 57600

# Name of the program without extension
PRG = Sieben

# Microcontroller type and clock frequency
MCU = atmega328p
F_CPU = 16000000
```

In die Datei hardware.h tragen wir ein paar Makro Definitionen ein, um die Anschlüsse für die 7-Segment Anzeige als Ausgang einzurichten und um sie anzusteuern:

```
// 7-Segment Anzeige

#define SIEBEN_INIT { \
    DDRD |= 0b11111100; \
    DDRB |= (1<<PB0); \
}

#define SIEBEN_AUS { \
    PORTD &= ~0b11111100; \
    PORTB &= ~(1<<PB0); \
}

#define SEGMENT_A { PORTD |= (1<<PD2); }
#define SEGMENT_B { PORTD |= (1<<PD3); }
#define SEGMENT_C { PORTD |= (1<<PD4); }
#define SEGMENT_D { PORTD |= (1<<PD5); }
#define SEGMENT_E { PORTD |= (1<<PD6); }
#define SEGMENT_F { PORTD |= (1<<PD7); }
#define SEGMENT_G { PORTB |= (1<<PB0); }
```

Bei den ersten beiden Makros siehst du, dass sie auch mehrzeilig sein dürfen. Dann muss aber ans Ende jeder Zeile ein Backslash („\“) geschrieben werden.

An Stelle von 0b11111100 darfst du auch 0xFC, 252 oder gar „(1<<PD2)+(1<<PD3)+(1<<PD4)+(1<<PD5)+(1<<PD6)+(1<<PD7)“ schreiben. Alle vier Varianten ergeben letztendlich die gleiche Zahl, so dass es reine Geschmackssache ist.

Und so sieht das Hauptprogramm in main.c aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"

int main(void)
{
    initSerialConsole();
    initSystemTimer();
```

```

SIEBEN_INIT;

while(1)
{
    SIEBEN_AUS;
    _delay_ms(200);
    SEGMENT_A;
    _delay_ms(200);
    SEGMENT_B;
    _delay_ms(200);
    SEGMENT_C;
    _delay_ms(200);
    SEGMENT_D;
    _delay_ms(200);
    SEGMENT_E;
    _delay_ms(200);
    SEGMENT_F;
    _delay_ms(200);
    SEGMENT_G;
    _delay_ms(200);
}

```

Nach der Initialisierung folgt eine Endlosschleife, die zuerst alle Segmente aus schaltet und dann eines nach dem anderen ein schaltet. Mit diesem Programm kannst du sofort sehen, ob alle Segmente korrekt angeschlossen sind und keine Vertauschung vorliegt.

Bei diesem Programm hat es sich sehr gelohnt, die Zugriffe auf I/O Pins in Makros auszulagern. Denn so kann man schön am Quelltext ablesen, was vor sich geht und man kann die Zuordnung von I/O Pin zu Segment beliebig ändern, ohne das eigentliche Programm ändern zu müssen. Je größer das Programm wird, umso nützlicher wird dies.

Nun ändern wir das Hauptprogramm so, um die Ziffern 0 bis 9 anzuzeigen:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"

// Zeigt eine Ziffer auf der 7-Segment Anzeige an
void anzeigen(uint8_t ziffer)
{
    SIEBEN_AUS;
    switch (ziffer)
    {
        case 0:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_E;
            SEGMENT_F;
            break;

        case 1:

```

```
SEGMENT_B;
SEGMENT_C;
break;

case 2:
SEGMENT_A;
SEGMENT_B;
SEGMENT_D;
SEGMENT_E;
SEGMENT_G;
break;

case 3:
SEGMENT_A;
SEGMENT_B;
SEGMENT_C;
SEGMENT_D;
SEGMENT_G;
break;

case 4:
SEGMENT_B;
SEGMENT_C;
SEGMENT_F;
SEGMENT_G;
break;

case 5:
SEGMENT_A;
SEGMENT_C;
SEGMENT_D;
SEGMENT_F;
SEGMENT_G;
break;

case 6:
SEGMENT_A;
SEGMENT_C;
SEGMENT_D;
SEGMENT_E;
SEGMENT_F;
SEGMENT_G;
break;

case 7:
SEGMENT_A;
SEGMENT_B;
SEGMENT_C;
break;

case 8:
SEGMENT_A;
SEGMENT_B;
SEGMENT_C;
SEGMENT_D;
SEGMENT_E;
SEGMENT_F;
SEGMENT_G;
break;

case 9:
SEGMENT_A;
```

```

        SEGMENT_B;
        SEGMENT_C;
        SEGMENT_D;
        SEGMENT_F;
        SEGMENT_G;
        break;
    }
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;

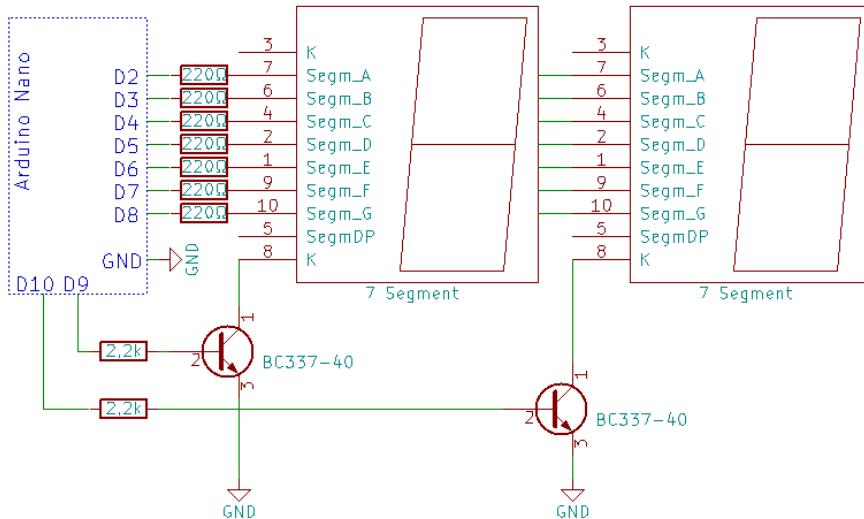
    while(1)
    {
        for (int i=0; i<10; i++)
        {
            anzeigen(i);
            _delay_ms(300);
        }
    }
}

```

Wenn du das Programm ausführst, erscheint die Zahlenfolge 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (und wieder von vorne) auf der Anzeige.

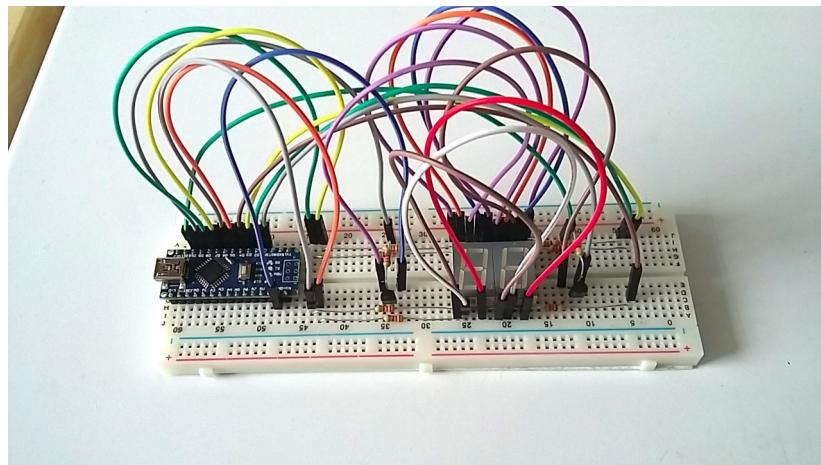
9.1.2 7-Segment Matrix

Als nächsten Schritt bauen wir die zweite Sieben-Segment Anzeige ein. Ihre Eingänge a bis g schließen wir parallel zur ersten Anzeige an. In folgendem Schaltplan habe ich das der Übersicht halber vereinfacht dargestellt:



Die Transistoren werden dazu dienen, die beiden Anzeigen abwechselnd ein und aus zu schalten. Wenn die Basis eines Transistors mit einem High Pegel angesteuert wird, schaltet er durch, so dass die damit verbundene Anzeige leuchten kann. Die 2,2 k Ω Widerstände begrenzen den Strom in die Basis auf etwa 2 mA.

Wegen der 220 Ω Widerstände wird jede LED mit ungefähr 14 mA betrieben. Wenn alle Segmente leuchten, sind das zusammen ca. 98 mA. Das schafft der Transistor Mühe los.



Für die beiden Transistoren muss die Datei hardware.h angepasst werden. Wir steuern jetzt zwei Ausgänge mehr an:

```
// 7-Segment Anzeige
#define SIEBEN_INIT { \
    DDRD |= 0b11111100; \
    DDRB |= 0b00000111; \
}

#define SIEBEN_AUS { \
    PORTD &= ~0b11111100; \
    PORTB &= ~0b00000111; \
}

#define SIEBEN_LINKS { PORTB |= (1<<PB1); }
#define SIEBEN_RECHTS { PORTB |= (1<<PB2); }

#define SEGMENT_A { PORTD |= (1<<PD2); }
#define SEGMENT_B { PORTD |= (1<<PD3); }
#define SEGMENT_C { PORTD |= (1<<PD4); }
#define SEGMENT_D { PORTD |= (1<<PD5); }
#define SEGMENT_E { PORTD |= (1<<PD6); }
#define SEGMENT_F { PORTD |= (1<<PD7); }
#define SEGMENT_G { PORTB |= (1<<PB0); }
```

Um zweistellige Zahlen anzuzeigen, schalten wir zuerst nur die linke Anzeige ein und stellen eine Ziffer dar. Dann schalten wir nur die rechte Anzeige ein und stellen die zweite Ziffer dar. Dann wieder links, usw.

Zwischen dem Umschaltvorgängen warten wir bisschen, damit die LED's genug Zeit bekommen, richtig aufzuleuchten. Wenn wir zu schnell umschalten würden (zum Beispiel eine Millionen mal pro Sekunde) würden die LED's nur schwach leuchten. Außerdem würden wir dann riskieren, den Empfang von drahtlosen Geräten (Radio, WLAN, Fernsehen) in der nahen Umgebung zu stören. Bei so hohen Frequenz verlassen die Elektromagnetischen Wellen nämlich gerne mal die Kabel und breiten sich in der Luft aus.

Das Hauptprogramm sieht nun so aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
```

```

#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"

// Mögliche Werte für "anzeige"
#define LINKS 0
#define RECHTS 1

// Zeigt eine Ziffer auf einer 7-Segment Anzeige an
// ziffer=0..9
// anzeige=LINKS oder RECHTS, bzw. 0 oder 1
void anzeigen(uint8_t ziffer, uint8_t anzeige)
{
    SIEBEN_AUS;
    switch (ziffer)
    {
        case 0:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_E;
            SEGMENT_F;
            break;

        case 1:
            SEGMENT_B;
            SEGMENT_C;
            break;

        case 2:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_D;
            SEGMENT_E;
            SEGMENT_G;
            break;

        case 3:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_G;
            break;

        case 4:
            SEGMENT_B;
            SEGMENT_C;
            SEGMENT_F;
            SEGMENT_G;
            break;

        case 5:
            SEGMENT_A;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_F;
            SEGMENT_G;
            break;
    }
}

```

```

        case 6:
            SEGMENT_A;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_E;
            SEGMENT_F;
            SEGMENT_G;
            break;

        case 7:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_C;
            break;

        case 8:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_E;
            SEGMENT_F;
            SEGMENT_G;
            break;

        case 9:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_F;
            SEGMENT_G;
            break;
    }

    switch (anzeige)
    {
        case 0:
            SIEBEN_LINKS;
            break;
        case 1:
            SIEBEN_RECHTS;
            break;
    }
}

```

```

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;

    while(1)
    {
        for (int i=0; i<10; i++)
        {
            anzeigen(i,LINKS);
            _delay_ms(100);
            anzeigen(9-i,RECHTS);
            _delay_ms(100);
        }
    }
}

```

```

    }
}
```

Starte das Programm, und du wirst sehen, dass die beiden Sieben-Segment abwechselnd folgende Ziffern anzeigen:

Links: 0 – 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 –

Rechts: – 9 – 8 – 7 – 6 – 5 – 4 – 3 – 2 – 1 – 0

Ändere jetzt den Inhalt der while(1) Schleife so ab, um eine Flackerfreie Anzeige zu erhalten:

```

while(1)
{
    for (int i=0; i<10; i++)
    {
        for (int j=1; j<500; j++)
        {
            anzeigen(i, LINKS);
            _delay_ms(1);
            anzeigen(9-i, RECHTS);
            _delay_ms(1);
        }
    }
}
```

Die äußere for-Schleife Zählt immer wieder von 0 bis 9, um die anzuzeigenden Zahlen vorzugeben. Die innere Schleife wird für jede Zahl 500 mal wiederholt. Sie zeigt links die Zahl (i) an und rechts den Wert von (9-i). Da die Delays nun auf 1ms reduziert wurden, können wir das Flackern nicht mehr sehen. Unser Auge ist dazu nicht flink genug.

Dieses wechselweise Ansteuern von LED Anzeigen nennt man „Matrix-Anzeige“. Gewöhnliche Sieben-Segment Anzeigen sind hell genug, um eine Matrix aus bis zu sechs Anzeigen zu bilden.

Einfach nur ein paar Zahlen anzuzeigen, ist natürlich noch nicht wirklich Sinnvoll. Stell dir vor, das Programm zeigt die Uhrzeit an. Oder es Zählt Signale von irgendeinem Sensor. Oder es zeigt die Spannung einer Batterie an.

All diese Anwendungsfälle erfordern Multithreading. Der Mikrocontroller muss sich nicht nur um das Ansteuern der Anzeige kümmern, sondern noch andere Aufgaben gleichzeitig wahrnehmen.

Wir werden das Programm jetzt in zwei Schritten so anpassen, dass Multithreading fähig wird. Der erste Schritt besteht darin, den Code für die Anzeige in eine separate Datei auszulagern. Das hilft uns dabei, den Überblick zu bewahren. Der zweite Schritt besteht darin, das Programm mit den Mitteln des Zustandsautomaten umzuschreiben, da dies Multithreading ermöglicht.

9.1.3 Exkurs: C Module

Größere C Programme teilt man in mehrere Quelltext-Dateien auf. Diese nennt man dann „Module“. Du hast bereits zahlreiche Module verwendet, nämlich die serielle Konsole, den Systemtimer und viele der Module, die der C-Compiler mit sich bringt (zum Beispiel avr/io.h).

Ein C Modul besteht aus zwei Dateien:

- Eine *.c Quelltext-Datei mit dem Quelltext
- Eine *.h Header-Datei mit der Schnittstelle (Interface)

Die Schnittstelle legt fest, welche Funktionen von „außen“ sichtbar sein sollen. So findest du in der Datei systemtimer.h die beiden Funktionen milliseconds() und initSystemTimer(). Aber die Interrupt-Routine gibt es nur in der *.c Datei. Die soll nämlich nach Außen hin nicht sichtbar sein.

In den *.h Dateien schreibt man nur die Namen der Funktionen mit ihren Parameter und Rückgabewerten auf. Der eigentliche Programmcode befindet sich immer in den *.c Dateien.

Wir werden jetzt den Programmcode für die Ansteuerung der Sieben-Segment Anzeige in ein C Modul auslagern. Erstelle die Datei anzeigen.h im src Verzeichnis des Projektes mit folgendem Inhalt:

```
#ifndef ANZEIGE_H
#define ANZEIGE_H

// Mögliche Werte für "anzeige"
#define LINKS 0
#define RECHTS 1

// Zeigt eine Ziffer auf einer 7-Segment Anzeige an
// ziffer=0..9
// anzeige=LINKS oder RECHTS, bzw. 0 oder 1
void anzeigen(uint8_t ziffer, uint8_t anzeige);

#endif
```

Der Rahmen aus ifndef/define/endif gehört in jede Header Datei. Der Compiler würde sonst in manchen Fällen meckern, dass er Sachen doppelt vorfindet.

Unsere Header Datei deklariert nur die eine Funktion anzeigen() ohne ihren Code zu definieren. Das tun wir in einer weiteren Datei mit Namen anzeigen.c:

```
#include <stdint.h>
#include "hardware.h"
#include "anzeige.h"

void anzeigen(uint8_t ziffer, uint8_t anzeige)
{
    SIEBEN_AUS;
    switch (ziffer)
    {
        case 0:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_E;
            SEGMENT_F;
            break;

        case 1:
            SEGMENT_B;
            SEGMENT_C;
            break;

        case 2:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_D;
            SEGMENT_E;
            SEGMENT_G;
            break;

        case 3:
            SEGMENT_A;
            SEGMENT_B;
            SEGMENT_C;
            SEGMENT_D;
            SEGMENT_G;
```

```

        break;

    case 4:
        SEGMENT_B;
        SEGMENT_C;
        SEGMENT_F;
        SEGMENT_G;
        break;

    case 5:
        SEGMENT_A;
        SEGMENT_C;
        SEGMENT_D;
        SEGMENT_F;
        SEGMENT_G;
        break;

    case 6:
        SEGMENT_A;
        SEGMENT_C;
        SEGMENT_D;
        SEGMENT_E;
        SEGMENT_F;
        SEGMENT_G;
        break;

    case 7:
        SEGMENT_A;
        SEGMENT_B;
        SEGMENT_C;
        break;

    case 8:
        SEGMENT_A;
        SEGMENT_B;
        SEGMENT_C;
        SEGMENT_D;
        SEGMENT_E;
        SEGMENT_F;
        SEGMENT_G;
        break;

    case 9:
        SEGMENT_A;
        SEGMENT_B;
        SEGMENT_C;
        SEGMENT_D;
        SEGMENT_F;
        SEGMENT_G;
        break;
    }

switch (anzeige)
{
    case 0:
        SIEBEN_LINKS;
        break;

    case 1:
        SIEBEN_RECHTS;
        break;
}

```

```
}
```

Damit der Compiler diese neue Quelltextdatei nicht einfach ignoriert, müssen wir sie im Makefile angeben:

```
# Objects to compile  
OBJ = driver/serialconsole.o driver/systemtimer.o anzeige.o main.o
```

Im Hauptprogramm (main.c) können wir jetzt die Funktion anzeige() entfernen und durch eine #include Anweisung ersetzen. So sieht die ganze Datei main.c nach der Anpassung aus:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <stdint.h>  
#include <util/delay.h>  
#include <avr/pgmspace.h>  
#include <avr/interrupt.h>  
#include "driver/serialconsole.h"  
#include "driver/systemtimer.h"  
#include "hardware.h"  
#include "anzeige.h"  
  
int main(void)  
{  
    initSerialConsole();  
    initSystemTimer();  
    SIEBEN_INIT;  
  
    while(1)  
    {  
        for (int i=0; i<10; i++)  
        {  
            for (int j=1; j<500; j++)  
            {  
                anzeigen(i, LINKS);  
                _delay_ms(1);  
                anzeigen(9-i, RECHTS);  
                _delay_ms(1);  
            }  
        }  
    }  
}
```

Das war's schon. Du kannst das Programm kompilieren und in den Mikrocontroller hochladen.

9.1.4 Exkurs: Multithreading

Beim Multithreading geht es darum, dass der Mikrocontroller mehrere Vorgänge (Threads) scheinbar parallel abarbeitet. Multithreading mit Zustandsautomaten beruht auf dem Prinzip, dass das Hauptprogramm aus einer dummen Endlosschleife besteht, die einfach alle Threads abwechselnd aufruft.

```
int main(void)  
{  
    ... // Initialisierungen  
  
    while(1)  
    {  
        thread_1();  
    }  
}
```

```

        thread_2();
        thread_3();
        thread_4();
    }
}

```

Bei jedem Aufruf führen die Threads immer nur einen kurzen Arbeitsschritt aus und merken sich, welcher Schritt als nächster dran sein wird. Dadurch sieht es von außen betrachtet so aus, als ob der Mikrocontroller mehrere Vorgänge gleichzeitig abarbeitet.

Der erste Thread wird unsere Anzeige ansteuern. Er besteht aus vier Schritten, die fortlaufend wiederholt werden:

1. Linke Ziffer ausgeben
2. Warte 1ms
3. Rechte Ziffer ausgeben
4. Warte 1ms

Diesen Thread werden wir nun programmieren. Entferne die Funktion `anzeiger` aus der Header Datei `anzeige.h` und füge stattdessen folgendes ein:

```

// Führe den Thread für die Anzeige aus.
void thread_anzeige(void);

// Ziffern für die Anzeige in einem Array speichern
void setze_ziffer(uint8_t ziffer, uint8_t anzeige)

```

Die Quelltext-Datei `anzeige.c` müssen wir ein bisschen erweitern:

```

#include <stdint.h>
#include "driver/systemtimer.h"
#include "hardware.h"
#include "anzeige.h"

// Speichert die Ziffern für die beiden 7-Segment Anzeigen
static uint8_t werte[2];

// Ziffern für die Anzeige in einem Array speichern
void setze_ziffer(uint8_t ziffer, uint8_t anzeige)
{
    werte[anzeige]=ziffer;
}

static void anzeigen(uint8_t ziffer, uint8_t anzeige)
{
    ... Inhalt wie bisher
}

// Thread ausführen
void thread_anzeige(void)
{
    static enum
    {
        ANZEIGEN_LINKS,
        WARTEN_LINKS,
        ANZEIGEN_RECHTS,
        WARTEN_RECHTS
    }
}

```

```

status=ANZEIGEN_LINKS;

static timer_t warteSeit=0;
uint8_t ziffer;
timer_t jetzt;

switch (status)
{
    case ANZEIGEN_LINKS:
        ziffer=werte[LINKS];
        anzeigen(ziffer,LINKS);
        status=WARTEN_LINKS;
        warteSeit=milliseconds();
        break;

    case WARTEN_LINKS:
        jetzt=milliseconds();
        if (jetzt-warteSeit >= 1)
        {
            status=ANZEIGEN_RECHTS;
        }
        break;

    case ANZEIGEN_RECHTS:
        ziffer=werte[RECHTS];
        anzeigen(ziffer,RECHTS);
        status=WARTEN_RECHTS;
        warteSeit=milliseconds();
        break;

    case WARTEN_RECHTS:
        jetzt=milliseconds();
        if (jetzt-warteSeit >= 1)
        {
            status=ANZEIGEN_LINKS;
        }
        break;
}
}

```

Die Funktion anzeigen() trägt nun das Attribut „static“.

```
static void anzeigen(uint8_t ziffer, uint8_t anzeige)
```

An dieser Stelle bewirkt das Wort „static“ dass diese Funktion nur innerhalb der Datei anzeigen.c bekannt ist. In anderen Dateien könntest du eine andere Funktion mit gleichem Namen haben, ohne das der Compiler sich daran stört. Gleiches gilt auch für das Array werte[]:

```
static uint8_t werte[2];
```

Sie ist ebenfalls nur in dieser einen Quelltextdatei bekannt. Für die Aufrufe von außen haben wir eine neue Funktion:

```
void setze_ziffer(uint8_t ziffer, uint8_t anzeige)
{
    werte[anzeige]=ziffer;
}
```

Sie speichert den Wert der Ziffer in das Array ab. Der Thread wird sich darum kümmern, die gespeicherten Ziffern zur Anzeige zu bringen.

Man sieht deutlich die vier Arbeitsschritte in der Aufzählung (enum) und in dem switch/case Konstrukt darunter.

```
void thread_anzeige(void)
{
    static enum
    {
        ANZEIGEN_LINKS,
        WARTEN_LINKS,
        ANZEIGEN_RECHTS,
        WARTEN_RECHTS
    }
    status=ANZEIGEN_LINKS;

    ...
    switch (status)
    {
        case ANZEIGEN_LINKS:
            ...
        case WARTEN_LINKS:
            ...
        case ANZEIGEN_RECHTS:
            ...
        case WARTEN_RECHTS:
            ...
    }
}
```

Hier kommt das Wort „static“ wieder vor, aber innerhalb von Funktionen hat es eine ganz andere Bedeutung als außerhalb. Hier bedeutet es, dass die betroffenen Variablen ihre Werte behalten, wenn die Funktion verlassen wird. Ohne das Wort „static“ würden diese Variablen bei jedem Aufruf der Funktion erneut initialisiert werden.

Die Status Variable ist eine Aufzählung, welche die vier angegebenen Werte enthalten kann. Damit merkt sich der Thread, welcher Schritt als nächstes an der Reihe ist.

Beim ersten Aufruf befindet sich der Thread im Status „ANZEIGEN_LINKS“. Dann macht er folgendes:

```
case ANZEIGEN_LINKS:
    ziffer=werte[LINKS];
    anzeigen(ziffer,LINKS);
    status=WARTEN_LINKS;
    warteSeit=milliseconds();
    break;
```

Zuerst wird der Wert der anzuzeigenden Ziffer aus dem Array geholt und zur Anzeige gebracht. Dann wird der Status auf WARTEN_LINKS umgestellt und in der Variable warteZeit vermerkt, seit wann wir warten.

Wenn der Thread das nächste mal ausgeführt wird, befindet er sich im Status WARTEN_LINKS. Dann macht er folgendes:

```
case WARTEN_LINKS:
    jetzt=milliseconds();
    if (jetzt-warteSeit >= 1)
    {
        status=ANZEIGEN_RECHTS;
    }
    break;
```

Dieser Schritt prüft, ob mindestens eine Millisekunde verstrichen ist. Wenn ja, geht es zum nächsten Schritt weiter. Wenn nicht, wird dieser Schritt beim nächsten Durchlauf wiederholt.

Die nächsten beiden Schritte für die Rechte Ziffer funktionieren ebenso.

Nun zum Hauptprogramm. Wir fangen einfach an, indem wir nur den `thread_anzeige()` aufrufen. Dies ist die ganze Datei `main.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"

#include "anzeige.h"

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;

    while(1)
    {
        thread_anzeige();
    }
}
```

Ich war hier wieder so faul, einige unnötige `#include` Anweisungen stehen zu lassen. Sie schaden nicht und vielleicht brauchen wir sie später noch.

Wenn du das Programm so startest, wird deine Anzeige „00“ lauten, weil wir nämlich noch keinen Code geschrieben haben, der die anzugebenden Werte festlegt. Zu diesem Zweck schreiben wir einen weiteren Thread – und zwar in der Datei `main.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"
#include "anzeige.h"

uint8_t zahl=0;

void thread_zaehlen(void)
{
    static enum
    {
        ZAEHLEN,
```

```

        AUSGEBEN,
        WARTEN
    }
    status=WARTEN;

    static timer_t warteSeit=0;

    switch (status)
    {
        case ZAEHLEN:
            zahl++;
            if (zahl>99)
            {
                zahl=0;
            }
            status=AUSGEBEN;
            break;

        case AUSGEBEN:
            setze_ziffer(zahl/10, LINKS);
            setze_ziffer(zahl%10, RECHTS);
            status=WARTEN;
            warteSeit=milliseconds();
            break;

        case WARTEN:
            if (milliseconds()-warteSeit >= 1000)
            {
                status=ZAEHLEN;
            }
            break;
    }
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;

    while(1)
    {
        thread_anzeige();
        thread_zählten();
    }
}

```

Der Thread zählen hat drei Stati:

- Im Status ZAEHLEN wird die Variable zahl um eins erhöht, bis 99. Dann fängt er wieder bei 0 an. Der nächste Status ist: AUSGEBEN.
- Im Status AUSGEBEN wird die Zahl in das Array "werte" geschrieben. Die Zehner links und die Einer rechts. Außerdem wird der nächste Status mit WARTEN festgelegt und vermerkt, wann das Warten begonnen hat.
- Im Status WARTEN verbleibt der Zustandsautomat so lange, bis mindestens 1000ms verstrichen sind. Erst dann wird wieder in den Status ZAEHLEN gewechselt.

Jetzt hast du einen Timer, der nach dem Einschalten exakte Sekunden hoch zählt. Der Timer führt zwei Threads quasi gleichzeitig aus. Ein Thread steuert die Anzeige an, der andere zählt die Sekunden hoch. Hier jetzt noch einen dritten Thread unterzubringen, ist ganz einfach. Das werden wir gleich machen.

9.2 Stoppuhr

Wir fügen unserem Timer aus dem vorherigen Kapitel zwei Tasten hinzu, um eine Stoppuhr zu erhalten.

```
Start-Taste
GND | -----/ -----o PC0
```

```
Stopp-Taste
GND | -----/ -----o PC1
```

Wenn ein Taster gedrückt wird, geht der entsprechende Eingang des Mikrocontrollers auf Low, ansonsten wird der interne Pull-Up Widerstand ihn auf High ziehen. Die Datei hardware.h wird entsprechend erweitert:

```
// Zwei Taster
#define TASTER_INIT { PORTC |= (1<<PC0)+(1<<PC1); }
#define TASTER_START_GEDRUECKT ((PINC & (1<<PC0))==0)
#define TASTER_STOPP_GEDRUECKT ((PINC & (1<<PC1))==0)
```

Und das Hauptprogramm sieht so aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"
#include "anzeige.h"

uint8_t zahl=0;

enum
{
    STOPP,
    START,
    ZAEHLEN,
    AUSGEBEN,
    WARTEN
}
status=STOPP;

void thread_zahhlen(void)
{
    static timer_t warteSeit=0;

    switch (status)
    {
        case STOPP:
```

```

        // nichts tun
        break;

    case START:
        zahl=0;
        setze_ziffer(zahl/10, LINKS);
        setze_ziffer(zahl%10, RECHTS);
        warteSeit=milliseconds();
        status=WARTEN;
        break;

    case ZAEHLEN:
        zahl++;
        if (zahl>99)
        {
            zahl=0;
        }
        status=AUSGEBEN;
        break;

    case AUSGEBEN:
        setze_ziffer(zahl/10, LINKS);
        setze_ziffer(zahl%10, RECHTS);
        status=WARTEN;
        warteSeit=milliseconds();
        break;

    case WARTEN:
        if (milliseconds()-warteSeit >= 1000)
        {
            status=ZAEHLEN;
        }
        break;
    }

}

void thread_tastenabfrage(void)
{
    if (TASTER_START_GEDRUECKT)
    {
        status=START;
    }
    else if (TASTER_STOPP_GEDRUECKT)
    {
        status=STOPP;
    }
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;
    TASTER_INIT;

    while(1)
    {
        thread_anzeige();
        thread_tastenabfrage();
        thread_zahlen();
    }
}

```

```
}
```

Die Enumeration "status" befindet sich dieses mal außerhalb der Funktion `thread_zahelen()`, damit sie von der Tastenabfrage verändert werden kann. Es gibt jetzt zwei neue Stati:

- STOPP ist der neue Anfangszustand. Der Timer steht still, die Zahl wird nicht hoch gezählt.
- START wird durch einen Druck auf die Start-Taste ausgelöst. Der Timer wird auf 0 zurück gesetzt, die 0 wird auf der Sieben-Segment Anzeige ausgegeben und dann beginnt der Zählvorgang mit der ersten Warte-Sekunde.

Für die Tastenabfrage haben wir einen neuen Thread eingefügt. Da dessen Funktion simpel ist, braucht dieser Thread keine Zustandsvariable.

Wenn die Stoppuhr die maximalen 99 Sekunden überschreitet, fängt sie wieder bei 00 an. Du kannst das verhalten Ändern und sie stattdessen anhalten:

```
if (zahl>99)
{
    status=STOPP; // statt zahl=0
}
```

Du kannst die Geschwindigkeit der Stoppuhr ändern, zum Beispiel auf Zehntel-Sekunden:

```
if (milliseconds()-warteSeit >= 100) // statt 1000
{
    status=ZAEHLEN;
}
```

9.2.1 Genauigkeit verbessern

Die Stoppuhr erfüllt ihren Zweck schon ganz gut, aber sie hat einen Schwachpunkt, der sich offenbart, wenn wir serielle Kommunikation hinzufügen.

```
case ZAEHLEN:
    zahl++;
    if (zahl>99)
    {
        zahl=0;
    }
    printf("Die aktuelle Zahl ist %u\n", zahl);
    status=AUSGEBEN;
    break;
```

Der `printf()` Befehl ist neu. Bei 2400 Baud benötigt er zur Ausgabe der 25 Zeichen ungefähr $25 \cdot 10 : 2400 = 0,1$ Sekunden. In dieser Zeitspanne sind alle Threads blockiert weil die serielle Konsole keine Rücksicht auf andere Threads nimmt. Man sieht das auch am Flackern der Anzeige.

Durch Verwendung einer höheren Baudrate bekommen wir zwar das Flackern weg, aber wir haben noch ein anderes viel ernsteres Problem: Die Uhr läuft jetzt zu langsam!

Wenn wir den `printf()` Befehl beschleunigen könnten, so dass er deutlich weniger als 1ms dauert, hätten wir das Problem gelöst. Das ist tatsächlich auch machbar, indem man einen Sendepuffer einrichtet und das Senden in eine Interruptroutine auslagert.

Ich möchte hier allerdings eine andere rein mathematische Lösung zeigen. Wir haben ja einen Systemtimer, der verlässlich Millisekunden zählt – auch während der seriellen Ausgabe. Wenn wir nicht einfach nur dumm 1000ms ab jetzt warten, sondern auf 1000 ms seit dem vorherigen Zähltakt, dann versaut uns die serielle Ausgabe nicht mehr das Timing. Das geht so:

```

case AUSGEBEN:
    setze_ziffer(zahl/10, LINKS);
    setze_ziffer(zahl%10, RECHTS);
    status=WARTEN;
    warteSeit+=1000; // statt warteSeit=millisekunden();
    break;

```

Dass die vorherige serielle Ausgabe bereits 0,1 Sekunden verbraucht hat, spielt keine Rolle mehr. Dann wird eben nur noch 0,9 Sekunden gewartet und so der Fehler ausgeglichen.

Probiere das Programm aus und vergleiche mit einer Stoppuhr.

Du kannst die serielle Ausgabe (den printf() Befehl) danach wieder heraus nehmen.

9.2.2 Stoppuhr mit Pause

Wir wollen die Stoppuhr noch ein wenig verbessern, indem wir einen PAUSE-Zustand hinzufügen.

- Die Start Taste soll abwechselnd die Uhr laufen lassen und anhalten.
- Die Stopp Taste soll die Uhr auf 00 zurück setzen.

Im Zählen-Thread entfällt daher der STOPP-Zustand, dafür gibt es die neuen Zustände RESET und PAUSE. Der Code für START wird so geändert, dass er die Uhr nur noch fortsetzt und nicht mehr zurück setzt.

```

enum
{
    RESET,
    PAUSE,
    START,
    ZAEHLEN,
    AUSGEBEN,
    WARTEN
}
status=RESET;

void thread_zahhlen(void)
{
    ...
    case RESET:
        zahl=0;
        setze_ziffer(0, LINKS);
        setze_ziffer(0, RECHTS);
        status=PAUSE;
        break;

    case PAUSE:
        // nichts tun
        break;

    case START:
        warteSeit=milliseconds();
        status=WARTEN;
        break;

    case ZAEHLEN:
        ...
}

```

Damit die Stoppuhr nicht sofort nach dem Starten in den PAUSE Status wechselt, muss die Tastenabfrage erweitert werden, so dass sie nach jedem Tastendruck auf das Loslassen wartet, bevor ein erneuter Tastendruck erkannt wird.

Das machen wir wieder mit einem Zustandsautomaten:

```
void thread_tastenabfrage(void)
{
    static enum {
        WARTE_DRUCK,
        WARTE_LOSLASSEN
    }
    tastenStatus=WARTE_DRUCK;

    switch (tastenStatus)
    {
        case WARTE_DRUCK:
            if (TASTER_START_GEDRUECKT)
            {
                if (status==PAUSE)
                {
                    status=START;
                }
                else {
                    status=PAUSE;
                }
                tastenStatus=WARTE_LOSLASSEN;
            }
            if (TASTER_STOPP_GEDRUECKT)
            {
                status=RESET;
                tastenStatus=WARTE_LOSLASSEN;
            }
            break;

        case WARTE_LOSLASSEN:
            if (!TASTER_START_GEDRUECKT &&
                !TASTER_STOPP_GEDRUECKT)
            {
                tastenStatus=WARTE_DRUCK;
            }
            break;
    }
}
```

Probiere das Programm mehrmals aus und achte darauf, wie die Start/Pause Taste reagiert. Du wirst feststellen, dass die Uhr manchmal nicht richtig reagiert. Sie verhält sich dann so, als ob du die Taste zweimal gedrückt hättest.

Dieser Effekt wird durch das Prellen der Kontakte im Taster ausgelöst. Am einfachsten schaltest du einen 100nF Kondensator parallel zum Taster, dann tritt der Fehler nicht mehr auf. Eleganter ist allerdings eine reine Softwarelösung:

```
void thread_tastenabfrage(void)
{
    static enum
    {
        WARTE_DRUCK,
        ENTPRELLEN,
        WARTE_LOSLASSEN
    }
    tastenStatus=WARTE_DRUCK;

    static timer_t gesperrtSeit=0;
```

```

switch (tastenStatus)
{
    case WARTE_DRUCK:
        if (TASTER_START_GEDRUECKT)
        {
            if (status==PAUSE)
            {
                status=START;
            }
            else {
                status=PAUSE;
            }
            tastenStatus=ENTPRELLEN;
        }
        if (TASTER_STOPP_GEDRUECKT)
        {
            status=RESET;
            tastenStatus=ENTPRELLEN;
        }
        break;

    case ENTPRELLEN:
        gesperrtSeit=milliseconds();
        tastenStatus=WARTE_LOSLASSEN;

    case WARTE_LOSLASSEN:
        if (TASTER_START_GEDRUECKT || TASTER_STOPP_GEDRUECKT)
        {
            tastenStatus=ENTPRELLEN;
        }
        else if (milliseconds()-gesperrtSeit >= 20)
        {
            tastenStatus=WARTE_DRUCK;
        }
        break;
}
}

```

Diese Lösung besteht darin, dass nach einem erkannten Tastendruck nicht nur einfach auf das Loslassen gewartet wird, sondern dass die Taste mindestens 20ms lang losgelassen bleiben muss. Wenn die Kontakte eines Tasters innerhalb der 20ms Sperrzeit doch noch einmal schließen (also prellen), beginnt die Sperrzeit erneut.

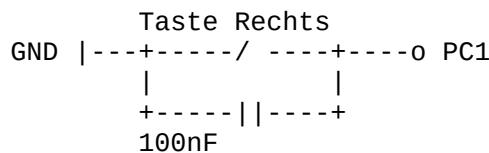
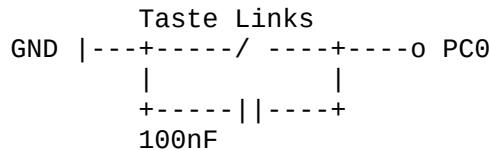
Nun stellt sich die Frage, was besser ist. Entprellung mit Kondensator oder Entprellung mit Software?

Ich persönlich tendiere zum Kondensator, weil es einfacher ist. Auf die paar Cent mehr kommt es bei selbst gebauten Schaltungen nicht an. Einen anderen relevanten Aspekt hatten wir bereits in Kapitel Exkurs: Entstörung thematisiert.

9.3 Torzähler

Mit dem Torzähler zählen wir die erreichten Tore von zwei Fußballmannschaften. Für jede Mannschaft gibt es einen Taster, mit dem wir die Ziffern der Anzeige erhöhen können.

Ausgehend von der Schaltung aus Kapitel 7-Segment Matrix schließen wir zwei Taster mit Kondensatoren zum Entprellen an:



Wir verwenden auch wieder das Modul für die Anzeige (Dateien anzeigen.c und anzeigen.h) aus dem Kapitel 7-Segment Matrix. In der Datei hardware.h bekommen die beiden Taster andere Namen:

```

// Zwei Taster
#define TASTER_INIT { PORTC |= (1<<PC0)+(1<<PC1); }
#define TASTER_LINKS_GEDRUECKT ((PINC & (1<<PC0))==0)
#define TASTER_RECHTS_GEDRUECKT ((PINC & (1<<PC1))==0)
    
```

Und der Quelltext in main.c sieht so aus:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"
#include "anzeige.h"

uint8_t zahl_links=1;
uint8_t zahl_rechts=1;

void thread_tastenabfrage(void)
{
    static uint8_t vorher_links=0;
    static uint8_t vorher_rechts=0;

    uint8_t jetzt_links=TASTER_LINKS_GEDRUECKT;
    uint8_t jetzt_rechts=TASTER_RECHTS_GEDRUECKT;

    if (jetzt_links && !vorher_links)
    {
        zahl_links++;
        setze_ziffer(zahl_links,LINKS);
    }
    if (jetzt_rechts && !vorher_rechts)
    {
        zahl_rechts++;
        setze_ziffer(zahl_rechts,RECHTS);
    }

    vorher_links=jetzt_links;
    vorher_rechts=jetzt_rechts;
}
    
```

```

}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;
    TASTER_INIT;

    while(1)
    {
        thread_anzeige();
        thread_tastenabfrage();
    }
}

```

Der Thread für die Tastenabfrage vergleicht den aktuellen Zustand der Tasten mit dem vorherigen Zustand. Wenn eine Taste vorher nicht gedrückt war und jetzt gedrückt ist, dann wird die links oder rechte Zahl erhöht und zur Anzeige gebracht.

Durch diesen vorher/jetzt Vergleich wird verhindert, dass die Zahl fortlaufend erhöht wird, wenn man die Taste festhält.

Und dann gibt es da noch einen kleinen Trick, und zwar bei der Initialisierung der Variablen:

```

uint8_t zahl_links=1;
uint8_t zahl_rechts=1;

```

Wenn du die Variablen stattdessen mit 0 initialisierst und dann einmal kurz das USB Kabel trennst und wieder ansteckst, wird die Anzeige unerwartet mit „11“ beginnen, statt mit „00“. Das kommt von den beiden 100nF Kondensatoren. Beim Einschalten der Stromversorgung sind sie beide zunächst leer, was so wirkt, als ob die beiden Tasten kurz gedrückt würden.

Durch die Initialisierung mit 1 bringen wir das Programm dazu, anfangs das Loslassen der Tasten abzuwarten. Erst die Tastendrücke danach werden gezählt.

Mit dem Reset Taster auf dem Arduino Board kannst du die Zähler wieder auf „00“ zurück setzen. Für einen realen Aufbau kannst du natürlich auch einen externen Reset Taster an das Arduino Board anschließen.

9.3.1 Änderungsvorschläge

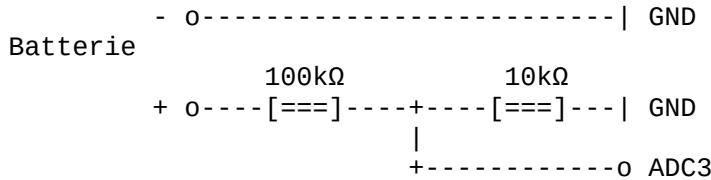
Überlege dir, was du alles ändern müsstest, um die Anzeige auf zwei Ziffern pro Mannschaft zu vergrößern.

Du kannst die selbe Schaltung auch als Rundenzähler für eine Carrera Rennbahn verwenden. Durch Überfahrt der Zielmarke könntest du mit optischen oder magnetischen Sensoren realisieren.

9.4 Batterie Meßgerät

Das Batterie Messgerät kombiniert den ADC Wandler des Mikrocontrollers mit der LED-Anzeige um die Spannung von kleinen Batterien anzuzeigen.

Ausgehend von der Schaltung aus dem Kapitel 7-Segment Matrix brauchst du nur zwei simple Widerstände, um die Spannung von Batterien zu messen:



Die Spannung der Batterie wird mit einem Spannungsteiler herab gesetzt und dann an den analogen Eingang 3 zur Messung geführt. Außerdem dient der Spannungsteiler zugleich als Schutz vor Überspannung.

Da das Messgerät Spannungen von 0,0 bis 9,9V anzeigen soll, legen wir den Dezimalpunkt der linken Sieben-Segment Anzeige über einen Widerstand auf 5 V, so daß er ständig leuchtet.

220Ω
5V o---[==]----o dp der linken Anzeige

Wieder verwenden wir das Modul für die Anzeige (Dateien anzeige.c und anzeige.h) aus dem Kapitel 7-Segment Matrix. Der Quelltext des Hauptprogramms sieht so aus:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"
#include "anzeige.h"

float messen(void)
{
    // Lese Eingang ADC3 mit 1,1V Referenz
    ADMUX=(1<<REFS0)+(1<<REFS1)+3;
    // Starte Messung mit Prescaler 128
    ADCSRA=(1<<ADEN)+(1<<ADSC)+(1<<ADPS0)+(1<<ADPS1)+(1<<ADPS2);
    // Warte, solange die Messung läuft
    while (ADCSRA & (1<<ADSC)) {};
    // Ergebnis abrufen
    return ADC/84.5455;
}

void thread_messen(void)
{
    static enum {MESSEN,WARTEN} status=MESSEN;
    static timer_t warteSeit=0;
    float spannung;
    uint8_t volt;
    uint8_t zehntelVolt;

    switch (status)
    {
        case MESSEN:
            spannung=messen();
            volt=spannung;
            zehntelVolt=(spannung-volt)*10;

```

```

        setze_ziffer(volt, LINKS);
        setze_ziffer(zehntelVolt, RECHTS);
        status=WARTEN;
        warteSeit=milliseconds();
        break;

    case WARTEN:
        if (milliseconds()-warteSeit >= 300)
        {
            status=MESSEN;
        }
        break;
    }

}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;

    while(1)
    {
        thread_anzeige();
        thread_messen();
    }
}

```

Die Funktion messen() führt eine Messung mit dem ADC durch und liefert das Ergebnis als Fließkommazahl in der Einheit Volt zurück. Die Zahl 84.5455 ergibt sich aus folgender Rechnung:
Die Referenzspannung des ADC haben wir auf 1,1 V eingestellt. Das ist aus Sicht des ADC die höchste Spannung, die er messen kann. Dabei fließt durch den 410 kΩ Widerstand ein Strom von:

$$1,1 \text{ V} : 10 \text{ k}\Omega = 0,11 \text{ mA}$$

Der gleiche Strom fließt auch durch den 100 kΩ Widerstand, wo dann so viel Spannung abfällt:

$$10 \text{ k}\Omega \cdot 0,11 \text{ mA} = 11 \text{ V}$$

Diese Summe der beiden Spannungen entspricht Batteriespannung, die nötig wäre, damit der ADC seinen Maximalwert liefert. Also 12,1 V. Der ADC liefert dann die Zahl 1023, das ist sein Maximalwert. Nun berechnen wir das Verhältnis von 1023 zu 12,1 V:

$$1023 : 12,1 \text{ V} = 84,5455$$

Wenn wir die vom ADC gelieferte Zahl durch 84,5455 teilen, erhalten wir daher die Spannung in Volt.

Schauen wir uns den Thread für die Messung an:

```

case MESSEN:
    spannung=messen();
    volt=spannung;
    zehntelVolt=(spannung-volt)*10;
    setze_ziffer(volt, LINKS);
    setze_ziffer(zehntelVolt, RECHTS);
    status=WARTEN;
    warteSeit=milliseconds();
    break;

```

Wir müssen die gemessene Spannung in zwei Ziffern umrechnen. Die linke Ziffer soll die Volt-Zahl anzeigen und die rechte Ziffer soll die Zehntel-Volt (also die erste Stelle nach dem Komma) anzeigen.

Die Umrechnung der Fließkommazahl „spannung“ in die Integer-Zahl „volt“ ist ganz einfach, denn das erledigt der Compiler von ganz alleine durch simple Zuweisung:

```
volt=spannung;
```

Volt enthält praktisch den Spannungswert, aber abgerundet auf 0 Nachkommastellen. Den Rest erhalten wir, indem wir diese gerundete Zahl von der Gesamtspannung subtrahieren. Übrig bleiben die Nachkommastellen. Die multiplizieren wir mit 10 und weisen sie ebenfalls einer Integer-Variable zu, um wieder die Nachkommastellen abzuschneiden.

```
zehntelVolt=(spannung-volt)*10;
```

ZehntelVolt und Volt sind Integer Variablen, während Spannung eine Fließkommazahl ist. Bei spannung=7,5 ist volt=7 und zehntelVolt=5. Diese beiden Ziffern werden zur Anzeige gebracht.

Danach wechselt der Thread in den WARTEN Zustand, wo er abwartet, bis 300ms vergangen sind. Dies verbessert die Lesbarkeit der Anzeige in Grenzfällen, wo die Spannung zum Beispiel ständig zwischen zwei Werten hin und her wechselt. Man könnte sie sonst vor lauter Flackern nicht ablesen.
Lade das Programm in den Mikrocontroller und probiere es mit unterschiedlichen Batterien aus.

9.5 Tachometer

Der Tachometer zeigt die Geschwindigkeit eines Fahrrades an, indem er die Drehzahl des Vorderrades mit einem magnetischen Reed-Kontakt misst.



Wenn du einen Magnet in die Nähe des Reed-Kontaktes bringst, ziehen sich die beiden Metallzungen gegenseitig an und der Strom kann fließen. Häufig reagieren diese Kontakte besser, wenn man den Magneten nicht genau an die Mitte hält, sondern etwas seitlich.

Wir werden den 16-Bit Timer 1 mit seiner Input-Capture Funktion nutzen. Der Timer zählt fortlaufend von 0 bis 65535. Immer wenn der Reed-Kontakt schließt, speichert der Timer den aktuellen Zählerstand in das sogenannte Input-Capture-Register. Unser Programm wird zwei aufeinander folgende Werte miteinander vergleichen und kann daraus die Fahrgeschwindigkeit berechnen.

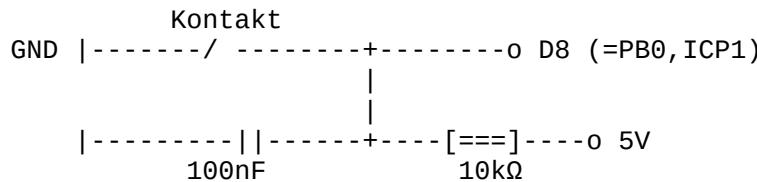
Wir beginnen wieder mit der Schaltung aus dem Kapitel 7-Segment Matrix. Leider müssen wir den Arduino Anschluss D8, wo jetzt noch die LED-Anzeige dran hängt, für den Reed-Kontakt frei machen. Da wir die Zuordnung der I/O Pins jedoch in der Datei hardware.h recht flexibel ändern können, ist das kein Problem.

Stecke also das Kabel vom Anschluss D8 (=PB0) nach D11 (=PB3) um. Und ändere die Datei hardware.h entsprechend:

```
DDRB |= 0b00010110; // statt 0b00000111;  
PORTB &= ~0b00010110; // statt 0b00000111;
```

```
#define SEGMENT_G { PORTB |= (1<<PB4); } // statt PB0
```

An D8 kommt jetzt der Reed-Kontakt mit Entprell-Kondensator und Pull-Up Widerstand:



Der schwache interne Pull-Up Widerstand des Mikrocontroller genügt nicht, weil der Kondensator damit zu langsam aufgeladen würde. Der Tacho würde dann nur mit sehr langsam drehenden Rädern funktionieren.

Der Arduino Anschluss D8 führt zum Mikrocontroller PB0, welcher die Spezialfunktion ICP1 (Input Capture of Timer 1) hat.

Fangen wir erst einmal mit diesem unvollständigen Programm an:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"
#include "anzeige.h"

void thread_berechnen(void)
{
    static enum {RECHNEN,WARTEN} status = RECHNEN;
    static timer_t warteSeit=0;

    switch (status)
    {
        case RECHNEN:
            printf("Messwert = %u\n", ICR1);
            warteSeit=milliseconds();
            status=WARTEN;
            break;

        case WARTEN:
            if (milliseconds()-warteSeit >= 500)
            {
                status=RECHNEN;
            }
            break;
    }
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;
```

```

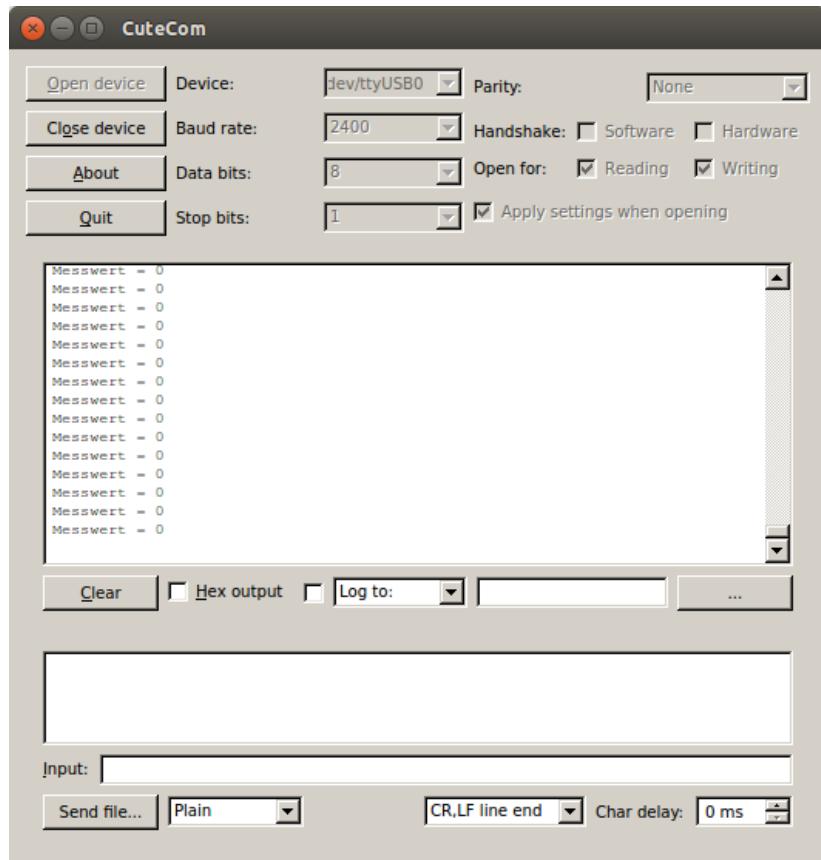
// Timer 1 initialisieren, mit Vorteiler 1024
// und Input-Capture bei fallender Flanke
TCCR1A = 0;
TCCR1B = (1<<ICNC1)+(1<<CS12)+(1<<CS10);
TCCR1C = 0;

while(1)
{
    thread_anzeige();
    thread_berechnen();
}
}

```

Wenn du dieses Programm ausführst, wird die Sieben-Segment Anzeige „00“ anzeigen und ein bisschen flackern. Mehr wird auf der Anzeige noch nicht passieren, weil das Programm unvollständig ist.

Starte auf dem Computer ein Terminal-Programm, öffne den virtuellen seriellen Port mit 2400 Baud (wie in der serialconsole.h immer noch voreingestellt ist) und schau dir die Textnachrichten an, die der Mikrocontroller an den PC sendet:



Der Thread für die Berechnung berechnet noch gar nichts. Er gibt einfach alle 500ms den aktuellen Messwert aus, das ist der Inhalt des Timer-Registers ICR1. Noch erhältst du fortlaufend Nullen, aber das ändert sich, sobald du den Reed-Kontakt betätigst. Bei jedem Impuls schreibt der Timer den entsprechenden Zeitpunkt in das Register ICR1.

Jetzt müssen wir zwei Dinge hinzufügen:

- Die Zeitspanne zwischen zwei Ereignissen (Radumdrehungen) ermitteln.
- Erkennen, wenn sich das Rad nicht mehr dreht.

Dazu fügen wir ein paar Variablen und zwei Interruptroutinen ein:

```

// Zeitpunkt der vorherigen Messung
volatile uint16_t vorher=0;

// Zeitspanne zwischen zwei Radumdrehungen
volatile uint16_t zeitspanne=0;

// Anzahl der Timer-Überläufe
volatile uint8_t overflows=0;

// Interrupt für Radumdrehung
ISR(TIMER1_CAPT_vect)
{
    if (overflows<2)
    {
        zeitspanne=ICR1-vorher;
    }
    vorher=ICR1;
    overflows=0;
}

// Interrupt für Timer-Überlauf
ISR(TIMER1_OVF_vect)
{
    if (overflows<255)
    {
        overflows++;
    }
    if (overflows>1)
    {
        zeitspanne=0;
    }
}

void thread_berechnen(void)
{
    ...
    printf("Zeitspanne = %u, Overflows = %u\n",zeitspanne,overflows);
}

int main(void)
{
    ...

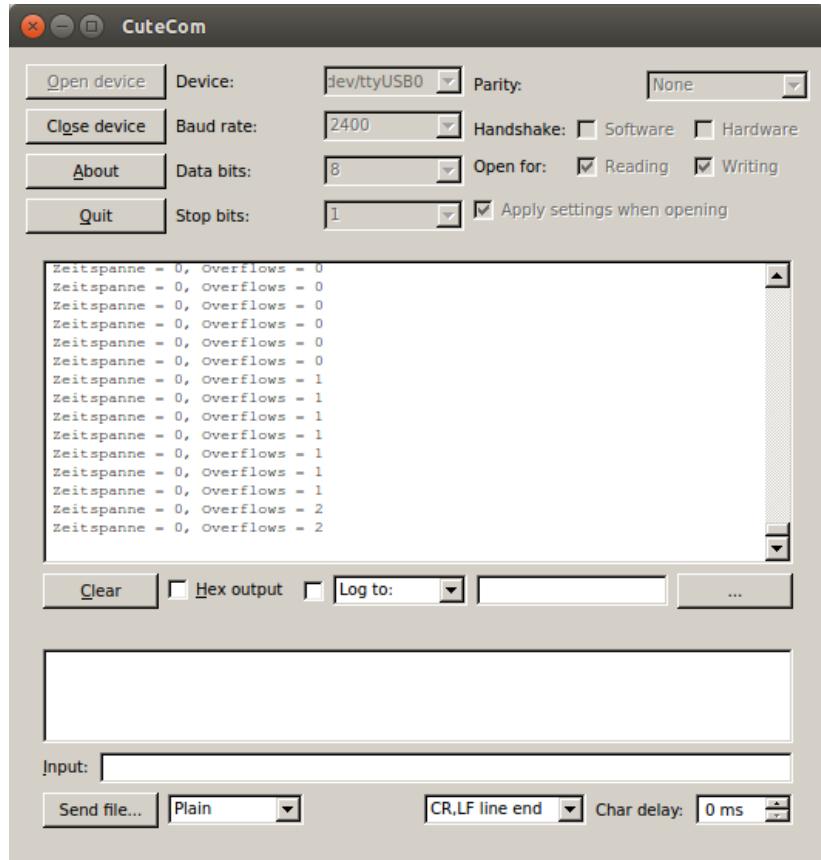
    // Timer 1 initialisieren
    ...

    // Erlaube Interrupts
    TIMSK1 = (1<<ICIE1)+(1<<TOIE1);
    sei();

    while(1)
    {
        thread_anzeige();
        thread_berechnen();
    }
}

```

Vergiss nicht, dass du das Terminal-Programm schließen musst, damit du dein eigenes Programm erneut in den Mikrocontroller hoch laden kannst. Starte das Terminal-Programm danach wieder und schaue, wie sich die Ausgabe verändert hat:



Die gemessene Zeitspanne verbleibt zunächst bei 0 aber die Anzahl der Überläufe erhöht sich ungefähr alle jede vierte Sekunde (bis maximal 255).

Wenn du jetzt den Reedkontakt zweimal betätigst, passiert folgendes:

1. Der Überlauf-Zähler wird auf 0 gesetzt.
2. Die Zeitspanne zwischen dem ersten und zweiten Ereignis wird angezeigt.

Warte nun ab und schau zu, was weiter passiert:

- Wenn der Überlauf-Zähler nach ca. 8 Sekunden den Wert 2 erreicht, wird die Zeitspanne auf 0 zurück gesetzt.
- Danach erhöht sich der Überlauf-Zähler wieder alle ca. alle vier Sekunden, bis maximal 255.

Dass die gemessene Zeitspanne ausgerechnet nach zwei Timer-Überläufen auf 0 zurück gesetzt wird, ist kein Zufall. Erinnere dich an das Kapitel Exkurs: Zeitmessung mit Überlauf. Wir können die Differenz zwischen zwei Zeitpunkten auch nach einem Timer-Überlauf noch korrekt berechnen. Aber nach zwei Überläufen nicht mehr. Deswegen setzen wir die gemessene Zeitspanne nach zwei Überläufen auf Null zurück.

Was jetzt noch fehlt, ist die Berechnung der Geschwindigkeit, damit wir sie anzeigen können. Die Taktfrequenz des Timers beträgt $16000000 : 1024 = 16000$ Hz. Dass heißt, wenn sich unser Vorderrad einmal pro Sekunde dreht, erhalten wir den Wert 16000 als Zeitspanne. Wenn sich das Rad doppelt so schnell drehen würde, erhalten wir den Wert 8000.

Ein 28 Zoll Rad fährt bei einer Radumdrehung ca. 2,2 Meter. Die Strecke hängt ein bisschen vom Reifentyp und Luftdruck ab. Den genauen Wert kannst du ja mal an deinem Fahrrad ausmessen.

Der Messwert 16000, der einer Sekunde entspricht, bedeutet also 2,2 Meter pro Sekunde. Eine Stunde hat 3600 Sekunden, also $2,2 \cdot 3600 = 7,920$ Kilometer pro Stunde. Also müssen wir lediglich die folgende Formel fortlaufend berechnen:

Geschwindigkeit in km/h = $7,920 \cdot 16000 : \text{zeitspanne}$

Das ist im Programmcode ganz einfach umsetzbar. Die Datei main.c sieht nun so aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"
#include "anzeige.h"

// Zeitpunkt der vorherigen Messung
volatile uint16_t vorher=0;

// Zeitspanne zwischen zwei Radumdrehungen
volatile uint16_t zeitspanne=0;

// Anzahl der Timer-Überläufe
volatile uint8_t overflows=0;

// Interrupt für Radumdrehung
ISR(TIMER1_CAPT_vect)
{
    if (overflows<2)
    {
        zeitspanne=ICR1-vorher;
    }
    vorher=ICR1;
    overflows=0;
}

// Interrupt für Timer-Überlauf
ISR(TIMER1_OVF_vect)
{
    if (overflows<255)
    {
        overflows++;
    }
    if (overflows>1)
    {
        zeitspanne=0;
    }
}

void thread_berechnen(void)
{
    static enum {RECHNEN,WARTEN} status = RECHNEN;
    static timer_t warteSeit=0;
    uint8_t kmh;

    switch (status)
    {
        case RECHNEN:
            kmh=7.920*16000/zeitspanne;
    }
}
```

```

        setze_ziffer(kmh/10, LINKS);
        setze_ziffer(kmh%10, RCHTS);

        warteSeit=milliseconds();
        status=WARTEN;
        break;

    case WARTEN:
        if (milliseconds()-warteSeit >= 500)
        {
            status=RECHNEN;
        }
        break;
    }

}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    SIEBEN_INIT;

    // Pull-Up am Timer-Eingang (=Reed Kontakt)
    PORTB |= (1<<PB0);

    // Timer 1 initialisieren, mit Vorteiler 1024
    // und Input-Capture bei fallender Flanke
    TCCR1A = 0;
    TCCR1B = (1<<ICNC1)+(1<<CS12)+(1<<CS10);
    TCCR1C = 0;

    // Erlaube Interrupts
    TIMSK1 = (1<<ICIE1)+(1<<TOIE1);
    sei();

    while(1)
    {
        thread_anzeige();
        thread_berechnen();
    }
}

```

Jetzt hast du einen voll funktionsfähigen Fahrrad Tacho.

9.5.1 Exkurs: Typ von Berechnungen erzwingen

Zur Berechnung der Geschwindigkeit möchte ich noch etwas erklären:

```
kmh=7.920*16000/zeitspanne;
```

Da die Zahl 7,920 eine Fließkommazahl ist, wird der Compiler die ganze Berechnung mit Fließkomma-Arithmetik durchführen. Das Ergebnis wird dann in einen Integer umgewandelt, weil die Variable kmh vom Typ uint8_t ist. Wenn du zur Probe einfach mal

```
kmh=8*16000/zeitspanne;
```

schreibst, wird der Compiler eine Warnmeldung bezüglich Integer Überlauf ausgeben. Denn dieses mal findet er keine Fließkommazahl mehr vor. Er rechnet daher mit Integer Zahlen. Jedoch ist $8 \cdot 16000 = 128000$ und das ist für Integer zu viel.

Mal angenommen, der Radumfang deines Fahrrades ergibt so eine runde Zahl, dann kannst du den Compiler durch einen Trick dazu bringen, trotzdem mit Fließkomma-Arithmetik zu rechnen:

```
kmh=(float)8*16000/zeitspanne;
```

So sagst du ihm, dass er die 8 bitte als Fließkommazahl verarbeiten soll, und das beeinflusst auch den ganzen restlichen Teil der Rechnung.

Alternativ könntest du auch einen größeren Integer-Typ mit mehr Bits erzwingen – aber nur bei ganzen Zahlen ohne Nachkommastellen:

```
kmh=(uint32_t)8*16000/zeitspanne;
```

So bringst du den Compiler dazu, die Rechnung als Long-Integer mit 32 Bits durchzuführen. Und da passen die 128000 locker rein.

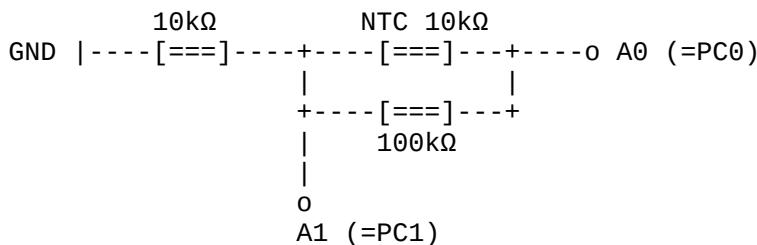
9.6 Thermometer

Es gibt zahlreiche elektronische Bauteile, mit denen man Temperaturen messen kann. Als ein Beispiel möchte ich den NTC Thermistor vorstellen.



NTC Thermistoren sind Widerstände, deren Ohm-Wert mit steigender Temperatur sehr deutlich sinkt. Sie sind preisgünstig und eignen sich für einfache Temperatur Messungen, wo es auf die Genauigkeit nicht allzu sehr ankommt – zum Beispiel um eine Heizung zu regeln.

Wir beginnen wieder mit der Schaltung aus dem Kapitel 7-Segment Matrix. Daran schließen wir den Thermistor folgendermaßen an:



Um die Temperatur zu messen schalten wir den Ausgang PC0 auf High (=5 V) und messen dann die analoge Spannung am Eingang PC1. Je wärmer es ist, umso höher wird die gemessene Spannung sein.

Damit der Temperatursensor sich durch den Stromfluss nicht erwärmt und somit die Messung verfälscht, schalten wir den Ausgang PC0 bei Nicht-Gebraucht auf Low. Dann fließt kein Strom.

Die beiden Widerstände sind nicht zufällig gewählt. NTC Widerstände haben eine nicht-lineare Kennlinie (also keine gerade), welche man durch geschickt gewählte Widerstände jedoch fast gerade biegen kann. Als Faustformel kannst du dir folgendes merken: Ermittle den Widerstandswert bei der mittleren zu messenden Temperatur und schalte einen Festwiderstand mit dem gleichen Wert in Reihe. Schalte außerdem einen 10x so großen Widerstand parallel. Die Messung wird dann bei

der mittleren Temperatur am genauesten sein. Bei hohen und niedrigen Temperaturen wird sie weniger genau sein.

Der verwendete NTC hat bei 25 °C einen Widerstand von 10 kΩ. Wir haben 10 kΩ in Reihe geschaltet, also wird die Schaltung Temperaturen um 25 °C am genauesten messen. Wenn du hingegen eher an Temperaturen um 80 °C interessiert wärst, müsstest du 1,2 kΩ und 12 kΩ Widerstände verwenden. Denn der NTC hat bei 80 °C etwa 1,2 kΩ.

Zurück zur Schaltung mit 10 kΩ/100 kΩ: Ich habe die beiden Widerstände und den NTC zunächst ohne Mikrocontroller an ein 5 V Netzteil angeschlossen und dann durch Vergleich mit einem anderen vertrauenswürdigen Thermometer zwei Probe-Messungen durchgeführt, und zwar ganz bewusst mitten in dem Temperaturbereich, den wir am Ende anzeigen möchten.

- Bei 10 °C → 1,85 V → ADC Wert 378
- Bei 40 °C → 3,43 V → ADC Wert 701

Die Umrechnung von Volt in den ADC Wert geht so:

$$1024 : (5 \text{ V} : \text{Messwert})$$

Die Differenz zwischen 40 °C und 10 °C beträgt 30°. Die Differenz vom ADC Wert ist 701 - 378 = 323. Wenn wir jetzt die 323 durch die 30° teilen, erhalten wir den Umrechnungsfaktor für die Temperaturanzeige:

$$(40^\circ - 10^\circ) : (701 - 378) = 10,77$$

oder:

$$30^\circ : 323 = 10,77$$

Als zweites müssen wir den Offset berechnen. Dazu teilt man einen der beiden Messwerte durch den gerade berechneten Faktor und ermittelt die Differenz zum Sollwert:

$$701 : 10,77 - 40^\circ = 25,09^\circ$$

Damit unser Programm den ADC Wert in eine Temperatur umrechnen kann, muss es demnach immer den ADC Wert durch 10,77 teilen und dann 25,09° subtrahieren. Das Ergebnis ist die Raumtemperatur in Grad Celsius. Das probieren wir gleich mal mit dem Taschenrechner aus:

$$\text{ADC Wert } 378 : 10,77 - 25,09 = 10^\circ\text{C}$$

$$\text{ADC Wert } 701 : 10,77 - 25,09 = 40^\circ\text{C}$$

Der Quelltext des Programms sieht so aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"
```

```

#include "anzeige.h"

float messen(void)
{
    // Lese Eingang ADC1 mit VCC als Referenz
    ADMUX=(1<<REFS0)+1;
    // Starte Messung mit Prescaler 128
    ADCSRA=(1<<ADEN)+(1<<ADSC)+(1<<ADPS0)+(1<<ADPS1)+(1<<ADPS2);
    // Warte, solange die Messung läuft
    while (ADCSRA & (1<<ADSC)) {};
    // Ergebnis in Grad Celsius umrechnen
    return ADC / 10.77 - 25.09;
}

void thread_messen(void)
{
    static enum {SENSOR_EIN,WARTEN,MESSEN,SENSOR_AUS,PAUSE} status=SENSOR_EIN;
    static timer_t warteSeit=0;
    uint8_t grad;

    switch (status)
    {
        case SENSOR_EIN:
            DDRC |= (1<<PC0);
            PORTC |= (1<<PC0);
            status=WARTEN;
            warteSeit=milliseconds();
            break;

        case WARTEN:
            if (milliseconds()-warteSeit >= 10)
            {
                status=MESSEN;
            }
            break;

        case MESSEN:
            grad=messen();
            setze_ziffer(grad/10,LINKS);
            setze_ziffer(grad%10,RECHTS);
            status=SENSOR_AUS;
            warteSeit=milliseconds();
            break;

        case SENSOR_AUS:
            PORTC &= ~(1<<PC0);
            warteSeit=milliseconds();
            status=PAUSE;
            break;

        case PAUSE:
            if (milliseconds()-warteSeit >= 1000)
            {
                status=SENSOR_EIN;
            }
            break;
    }
}

int main(void)
{

```

```

initSerialConsole();
initSystemTimer();
SIEBEN_INIT;

while(1)
{
    thread_anzeige();
    thread_messen();
}
}

```

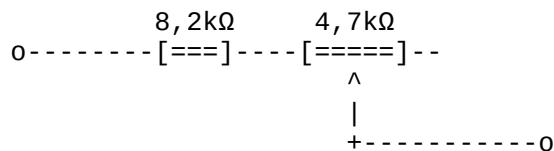
Die Funktion messen() liefert die Temperatur in Grad Celsius als Fließkommazahl. Unsere zweistellige Anzeige kann aber keine Nachkommastellen anzeigen, deswegen ist die Variable „grad“ nur ein einfacher Integer. Der Compiler sorgt automatisch dafür, dass in der Zeile

```
grad=messen();
```

die Fließkommazahl in eine Integer Zahl umgerechnet wird. Diese lässt sich danach wie gewohnt relativ einfach in Zehner und Einer zerlegen.

Wenn du das Programm nun ausführst, wirst du wahrscheinlich nur ungefähr die richtige Temperatur auf der Anzeige sehen. Das liegt daran, dass die NTC Widerstände durchaus einige Prozent von den Soll-Werten abweichen können. Und eine gewisse Ungenauigkeit des ADC kommt auch noch dazu. Bis zu 3 °C Abweichung sind völlig normal.

Du hast nun zwei Möglichkeiten, dein Thermometer zu justieren: Entweder indem du die Zahl 25,09 so änderst, dass die richtige Temperatur erscheint, oder indem du den 10 kΩ Widerstand durch eine einstellbare Kombination aus Festwiderstand und Trimmpti ersetzt:



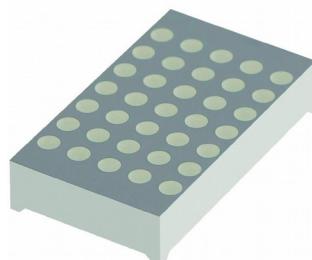
Außerdem empfehle ich, den Temperaturfühler über ein Kabel anzuschließen. Denn wenn er dem Mikrocontroller zu nahe kommt, wir er von dessen Abwärme beeinflusst.

Ein Präzisions-Messgerät ist das dann zwar noch nicht, aber nach korrekter Justierung erreicht es immerhin die gleiche Genauigkeit, wie ein billiges elektronisches 10 Euro Thermometer aus dem Baumarkt. Die funktionieren nämlich genau so.

Für präzisere Messungen gibt es Temperaturfühler mit digitalen Ausgängen. Die sind allerdings viel teurer und nicht so einfach auszulesen.

10 Matrix Anzeigen

LED Matrix Anzeigen sieht man häufig in Bussen und Bahnen, auch an Haltestellen zur Anzeige der nächsten Termine.



Material:

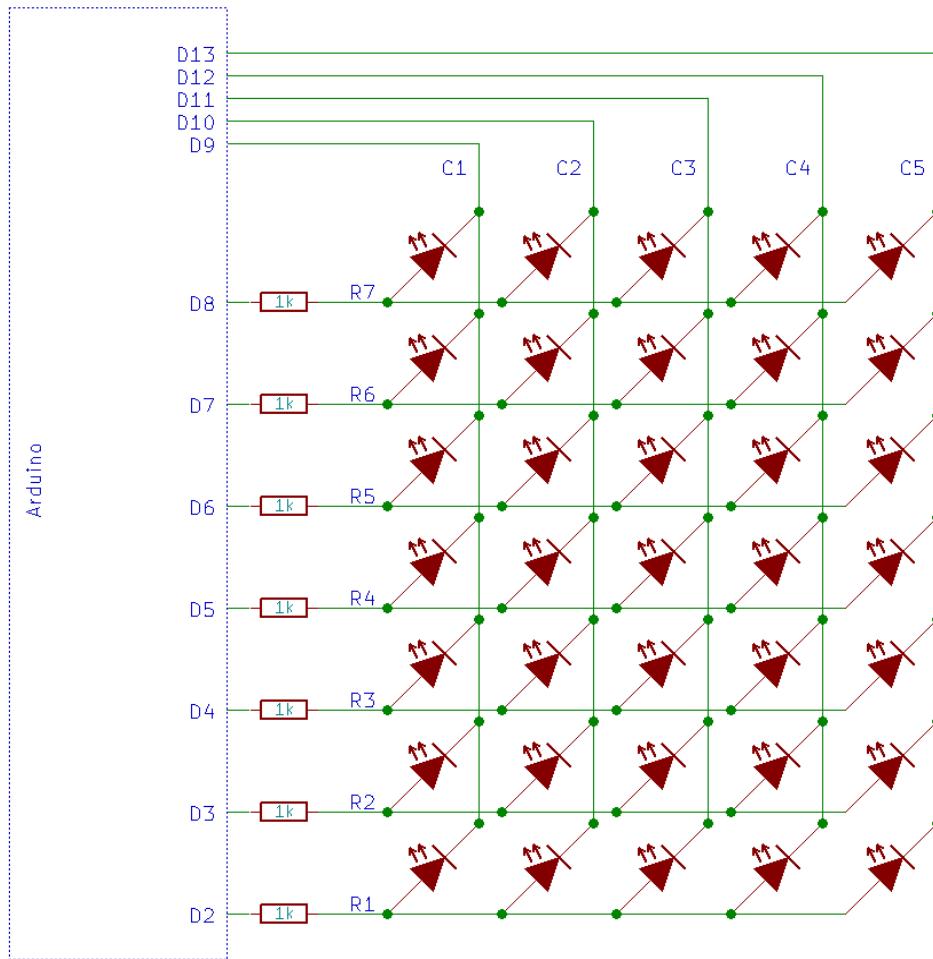
- 1 Arduino Nano
- 7 Widerstände 1 kΩ ¼ Watt
- 1 Punkt-Matrix Anzeige 7x5 LED's mit gemeinsamer Kathode
- 1-2 Steckbretter und Kabel

Diese Anzeigen gibt es in unterschiedlichen Größen, Farben und mit unterschiedlich vielen LED's. Die Variante mit 7x5 LED's findet man am häufigsten vor und ist daher auch die preisgünstigste.

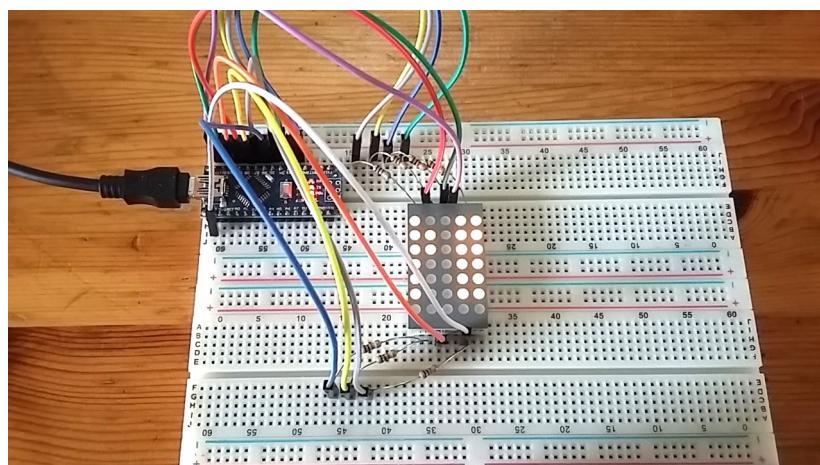
Ein beliebtes Lehrprojekt in Berufsschulen besteht darin, solche Anzeigen aus einzelnen LED's zusammen zu löten. Und dann werden die Anzeigen aller Schüler zu einem riesigen Display miteinander kombiniert.

Bei diesen Projekten passiert es manchmal, dass die gekauften LED's unterschiedlich hell leuchten, was sehr hässlich aussieht. Deswegen rate ich dringend dazu, lieber fertige Matrix-Module zu verwenden. Finanziell macht es auch keinen großen Unterschied.

Zum Ausprobieren schließen wir die Anzeige so an das Arduino Moduls an:



Falls deine Matrix-Anzeige wegen ihrer Größe nicht ins Steckbrett passt, verlängere die Anschlussbeinchen mit Drähten oder verwende zwei Steckbretter – so wie ich das hier gemacht habe:



Wir betreiben die Anzeige mit 1 kΩ Vorwiderständen, so dass die Stromstärke pro LED etwa $(5 \text{ V} - 2 \text{ V}) : 1000 \Omega = 3 \text{ mA}$ beträgt. Alle LED Matrix Anzeigen dieser Bauart dürfen mit viel mehr Strom betrieben werden, aber du wirst sehen, dass die 3 mA innerhalb von Gebäuden schon ausreichen.

Diese Matrix-Anzeige wird ganz ähnlich angesteuert, wie die 7-Segment Matrix aus dem vorherigen Kapitel. Zuerst legt der Mikrocontroller die die gewünschten Reihen (R1..R7) auf High und die erste Spalte (Anschluss C1) auf Low. Dadurch leuchtet die ganz linke Spalte auf. Nach einer kurzen Wartezeit schaltet er wieder alle Lichter aus, und aktiviert die zweite Spalte (C2). Dann wartet er wieder ein bisschen, und fährt mit mittleren Spalten (C3) fort. Und so weiter. Nach fünf Spalten beginnt der Vorgang wieder von vorne.

Die höchste Stromaufnahme kommt vor, wenn alle sieben LED's einer Spalte gleichzeitig leuchten. Das sind zusammen $7 \cdot 3 \text{ mA} = 21 \text{ mA}$. Da der Mikrocontroller bis zu 40 mA pro Pin verträgt, brauchen wir gar keine Transistoren (vergleiche mit dem Kapitel 7-Segment Matrix). Wir können die LED's einfach direkt an den Mikrocontroller anschließen.

Wenn du die Anzeige allerdings draußen verwenden möchtest – zum Beispiel an einem Linienbus, müsstest du mit erheblich mehr Strom arbeiten, dann müssten mindestens die 5 Spalten, oder besser gleich alle 12 Leitungen mit Transistoren verstärkt werden.

Das erste Programm wird alle Leuchtdioden aufleuchten lassen. Es arbeitet absichtlich langsam, damit du nachvollziehen kannst, wie es funktioniert. Wir fangen wieder mit der „Hello World“ Vorlage an, wo wir das Makefile anpassen:

```
# Programmer hardware settings for avrdude
AVRDUE_HW = -c arduino -P /dev/ttyUSB0 -b 57600

# Name of the program without extension
PRG = Matrix

# Microcontroller type and clock frequency
MCU = atmega328p
F_CPU = 16000000
```

In die Datei hardware.h fügen wir Makro Definitionen ein, um die Reihen und Spalten der Matrix-Anzeige anzusteuern:

```
// Die 12 I/O Leitungen sind Ausgänge
// Alle Reihen auf Low
// Alle Spalten auf High
#define MATRIX_INIT { \
    DDRD |= 0b11111100; \
    DDRB |= 0b00111111; \
    PORTD &= ~0b11111100; \
    PORTB &= ~0b00000001; \
    PORTB |= 0b00111110; \
}

// Einzelne Reihen einschalten (=High)
#define MATRIX_REIHE1 { PORTD |= (1<<PD2); }
#define MATRIX_REIHE2 { PORTD |= (1<<PD3); }
#define MATRIX_REIHE3 { PORTD |= (1<<PD4); }
#define MATRIX_REIHE4 { PORTD |= (1<<PD5); }
#define MATRIX_REIHE5 { PORTD |= (1<<PD6); }
#define MATRIX_REIHE6 { PORTD |= (1<<PD7); }
#define MATRIX_REIHE7 { PORTB |= (1<<PB0); }

// Einzelne Spalten einschalten (=Low)
#define MATRIX_SPALTE1 { PORTB &= ~(1<<PB1); }
#define MATRIX_SPALTE2 { PORTB &= ~(1<<PB2); }
#define MATRIX_SPALTE3 { PORTB &= ~(1<<PB3); }
#define MATRIX_SPALTE4 { PORTB &= ~(1<<PB4); }
#define MATRIX_SPALTE5 { PORTB &= ~(1<<PB5); }
```

Das minimale Hauptprogramm sieht so aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"
#include "driver/systemtimer.h"
#include "hardware.h"

void anzeigen(uint8_t bits, uint8_t spalten_nr)
{
    MATRIX_INIT;

    // Jedes Bit entspricht einer LED
    if (bits & 1)
    {
        MATRIX_REIHE1;
    }
    if (bits & 2)
    {
        MATRIX_REIHE2;
    }
    if (bits & 4)
    {
        MATRIX_REIHE3;
    }
    if (bits & 8)
    {
        MATRIX_REIHE4;
    }
    if (bits & 16)
    {
        MATRIX_REIHE5;
    }
    if (bits & 32)
    {
        MATRIX_REIHE6;
    }
    if (bits & 64)
    {
        MATRIX_REIHE7;
    }

    switch (spalten_nr)
    {
        case 0:
            MATRIX_SPALTE1;
            break;
        case 1:
            MATRIX_SPALTE2;
            break;
        case 2:
            MATRIX_SPALTE3;
            break;
        case 3:
            MATRIX_SPALTE4;
            break;
        case 4:
```

```

        MATRIX_SPALTE5;
        break;
    }

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    MATRIX_INIT;

    while(1)
    {
        anzeigen(0b1111111, 0);
        _delay_ms(100);
        anzeigen(0b1111111, 1);
        _delay_ms(100);
        anzeigen(0b1111111, 2);
        _delay_ms(100);
        anzeigen(0b1111111, 3);
        _delay_ms(100);
        anzeigen(0b1111111, 4);
        _delay_ms(100);
    }
}

```

Die serielle Konsole brauchen wir eigentlich nicht, aber ich habe den Code einfach drin gelassen, weil er nicht stört.

Die Hauptschleife steuert immer wieder alle fünf Spalten nacheinander an und lässt dort alle LED's leuchten. Du kannst das deutlich sehen. Verändere jetzt die delays auf 1ms, dann wird es so aussehen, als ob alle LED's gleichzeitig leuchten. Das Flackern ist nicht mehr sichtbar.

Probiere ein anderes interessanteres Leuchtmuster aus. Zum Beispiel ein kleines „f“:

```

while(1)
{
    anzeigen(0b0001000, 0);
    _delay_ms(1);
    anzeigen(0b1111110, 1);
    _delay_ms(1);
    anzeigen(0b0001001, 2);
    _delay_ms(1);
    anzeigen(0b0000001, 3);
    _delay_ms(1);
    anzeigen(0b0000010, 4);
    _delay_ms(1);
}

```

Da das Programm später andere Sachen gleichzeitig machen soll, schreiben wir es jetzt in einem Zustandsautomaten um. Im Prinzip so, wie wir es im vorherigen Kapitel bereits bei der 7-Segment Anzeige gemacht haben.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include "driver/serialconsole.h"

```

```

#include "driver/systemtimer.h"
#include "hardware.h"

uint8_t matrix[5];

void anzeigen(uint8_t bits, uint8_t spalten_nr)
{
    MATRIX_INIT;

    // Jedes Bit entspricht einer LED
    if (bits & 1)
    {
        MATRIX_REIHE1;
    }
    if (bits & 2)
    {
        MATRIX_REIHE2;
    }
    if (bits & 4)
    {
        MATRIX_REIHE3;
    }
    if (bits & 8)
    {
        MATRIX_REIHE4;
    }
    if (bits & 16)
    {
        MATRIX_REIHE5;
    }
    if (bits & 32)
    {
        MATRIX_REIHE6;
    }
    if (bits & 64)
    {
        MATRIX_REIHE7;
    }

    switch (spalten_nr)
    {
        case 0:
            MATRIX_SPALTE1;
            break;
        case 1:
            MATRIX_SPALTE2;
            break;
        case 2:
            MATRIX_SPALTE3;
            break;
        case 3:
            MATRIX_SPALTE4;
            break;
        case 4:
            MATRIX_SPALTE5;
            break;
    }
}

void thread_anzeigen(void)
{

```

```

static enum {ZEIGEN, WARTEN} status=ZEIGEN;
static uint8_t aktuelleSpalte=0;
static timer_t warteSeit=0;
uint8_t bits;

switch (status)
{
    case ZEIGEN:
        bits=matrix[aktuelleSpalte];
        anzeigen(bits, aktuelleSpalte);
        warteSeit=milliseconds();
        status=WARTEN;
        break;

    case WARTEN:
        if (milliseconds()-warteSeit >= 1)
        {
            aktuelleSpalte++;
            if (aktuelleSpalte>4)
            {
                aktuelleSpalte=0;
            }
            status=ZEIGEN;
        }
        break;
}
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    MATRIX_INIT;

    // Buchstabe "f"
    matrix[0]=0b0001000;
    matrix[1]=0b1111110;
    matrix[2]=0b0001001;
    matrix[3]=0b0000001;
    matrix[4]=0b0000010;

    while(1)
    {
        thread_anzeigen();
    }
}

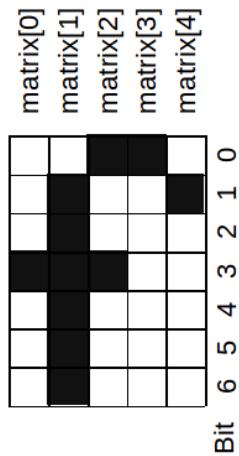
```

10.1 Zeichensätze

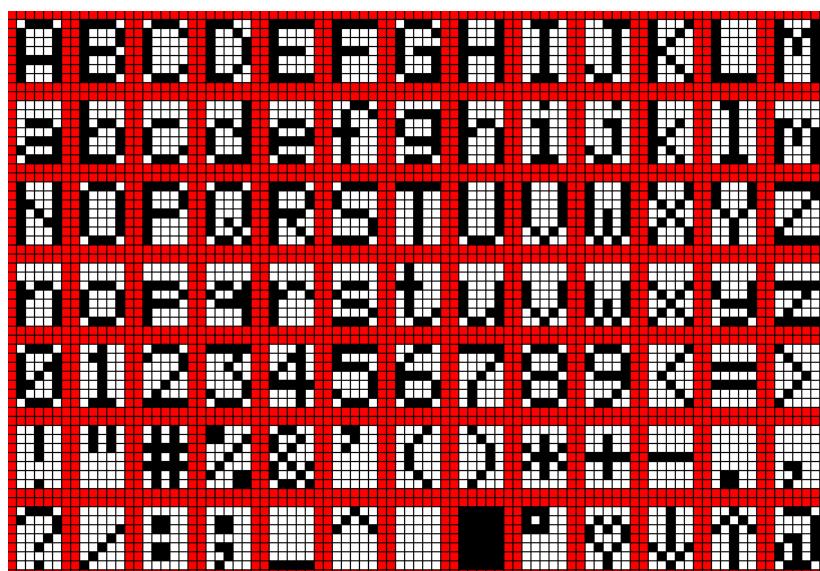
In obigem Programm repräsentiert die Variable

```
uint8_t matrix[5]
```

die Bildpunkte der Matrix-Anzeige. Das erste Byte steht für die ganz linke Spalte und das letzte Byte steht für die ganz rechts Spalte. Die einzelnen Bits entsprechen den sieben LED's. Wobei das Bit 7 ist ungenutzt bleibt.



Wenn du noch andere Schriftzeichen ausprobieren möchtest, dann suche mal im Internet nach Bildern für „7x5 Zeichensatz“. Dort wirst du einige Vorlagen finden. Zum Beispiel diese von Dominic Flins:



Ich zeige dir jetzt, wie man einen solchen Zeichensatz in ein C Programm einbettet:

```
uint8_t zeichensatz[3][5]=
{
    { /* 0=f */
        0b0001000,
        0b1111110,
        0b0001001,
        0b0000001,
        0b00000010
    }, { /* 1=g */
        0b0001100,
        0b1010010,
        0b1010010,
        0b1010010,
        0b0111110
    }, { /* 2=h */
        0b1111111,
        0b0001000,
        0b0000100,
        0b0000100,
        0b1111000
    }
}
```

```

    }

};

void zeige_zeichen(uint8_t nummer)
{
    matrix[0]=zeichensatz[nummer][0];
    matrix[1]=zeichensatz[nummer][1];
    matrix[2]=zeichensatz[nummer][2];
    matrix[3]=zeichensatz[nummer][3];
    matrix[4]=zeichensatz[nummer][4];
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    MATRIX_INIT;

    // Buchstabe "f"
    zeige_zeichen(0);

    while(1)
    {
        thread_anzeigen();
    }
}

```

Die Variable

```
uint8_t zeichensatz[3][5]
```

enthält drei Schriftzeichen (nämlich f, g und h) und für jedes Schriftzeichen werden 5 Bytes belegt. Wenn du die binären Zahlen mit der obigen Grafik von Dominic vergleichst, wirst du die schwarzen Punkte als „1“ und die weißen Punkte als „0“ wieder erkennen.

Damit das Ganze jetzt nicht in eine endlose Tipperei ausartet, habe ich den Zeichensatz auf nur drei Schriftzeichen beschränkt. Du kannst ihn nach diesem Prinzip auf das ganze Alphabet erweitern, wenn du möchtest.

Nun fügen wir einen zweiten Thread hinzu, der die drei Buchstaben f, g und h nacheinander anzeigt:

```

void thread_buchstaben(void)
{
    static enum {BUCHSTABE, WARTEN} status=BUCHSTABE;
    static uint8_t aktuellerBuchstabe=0;
    static timer_t warteSeit=0;

    switch (status)
    {
        case BUCHSTABE:
            zeige_zeichen(aktuellerBuchstabe);
            warteSeit=milliseconds();
            status=WARTEN;
            break;

        case WARTEN:
            if (milliseconds()-warteSeit >= 1000)
            {

```

```

        aktuellerBuchstabe++;
        if (aktuellerBuchstabe>2)
        {
            aktuellerBuchstabe=0;
        }
        status=BUCHSTABE;
    }
    break;
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();
    MATRIX_INIT;

    while(1)
    {
        thread_anzeigen();
        thread_buchstaben();
    }
}

```

Der ganze Zeichensatz ist in einer Array-Variable gespeichert und daher belegt er entsprechend viel RAM. Ab einer gewissen Größe könnte das zu einem Problem werden. Deswegen zeige ich dir jetzt, wie du das Array nur im Programmspeicher anlegst, ohne RAM zu verbrauchen.:

```

uint8_t const zeichensatz[3][5] PROGMEM =
{
    ... wie bisher
};

void zeige_zeichen(uint8_t nummer)
{
    matrix[0]=pgm_read_byte(&(zeichensatz[nummer][0]));
    matrix[1]=pgm_read_byte(&(zeichensatz[nummer][1]));
    matrix[2]=pgm_read_byte(&(zeichensatz[nummer][2]));
    matrix[3]=pgm_read_byte(&(zeichensatz[nummer][3]));
    matrix[4]=pgm_read_byte(&(zeichensatz[nummer][4]));
}

```

Zuerst kennzeichnen wir das Array als „const“, womit es unveränderlich wird. Danach dürfen wir es mit „PROGMEM“ kennzeichnen, was den Compiler dazu anweist, die Daten nur im (Flash-) Programmspeicher abzulegen und sie ausnahmsweise nicht ins RAM zu kopieren.

Um die Daten aus dem Programmspeicher auszulesen, verwenden wir die Funktion pgm_read_byte() und geben als Parameter die Adresse der Speicherzelle an, die wir lesen wollen. Und da wir diese nicht auswendig wissen können, nutzen wir wiederum Ausdrücke wie „&(zeichensatz[nummer][0])“ damit der Compiler sich die Adressen selbst ausrechnet. Das & Zeichen bedeutet „Adresse von“ und in den Klammern dahinter steht, von welchem Objekt wir die Adresse haben wollen.

Diese Vorgehensweise ist in der Dokumentation der AVR C Library detailliert beschrieben:
<http://www.nongnu.org/avr-libc/user-manual/pgmspace.html>

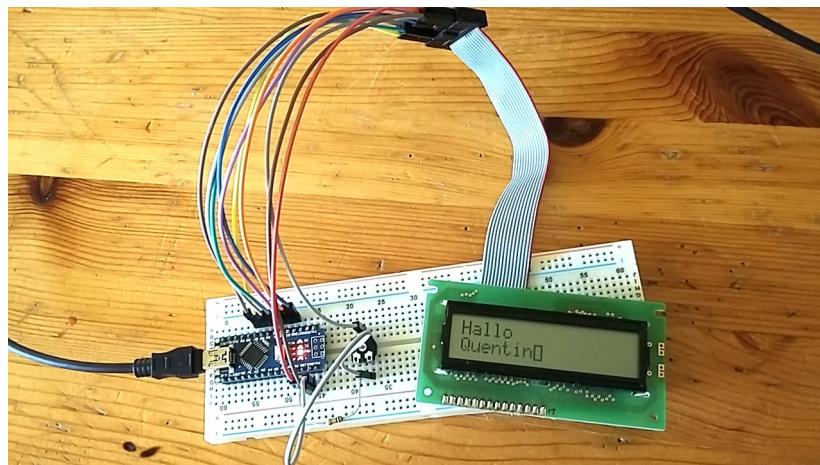
Unsere Matrixanzeige, kann nur einen einzelnen Buchstaben anzeigen. Mit dem bisher gelernten Wissen und mehreren Mikrocontrollern (oder Schieberegistern) könntest du das Ganze auf beliebige Breite vergrößern, um ganze Worte anzuzeigen. Allerdings wäre das für ein Wochenend-Experiment

definitiv viel zu komplex und zu teuer. Außerdem kann man solche Geräte viel billiger als Fertigprodukt kaufen.

Deswegen beende ich das Kapitel „Matrix Anzeigen“ an dieser Stelle. Das bisher Gelernte Wissen wird dir allerdings dabei helfen, kommerzielle Matrixanzeigen sowie LCD-Displays besser zu verstehen und anzuwenden.

11 LCD Anzeigen

Nach den LED Matrix-Anzeigen ist die nächste logische Steigerung eine LCD Matrixanzeige. Diese Anzeigen gibt es in unterschiedlichen Größen, wahlweise mit oder ohne Beleuchtung.



Auf der Webseite <http://sprut.de/electronic/lcd/index.htm> gibt es die umfangreichste deutsche Beschreibung dieser Displays, die ich bisher gefunden habe.

Material:

- 1 Arduino Nano
- 1 Trimmpot 10 kΩ linear
- 1 Widerstand 10 kΩ ¼ Watt
- 2 Dioden 1N4148
- 1 Kondensator 100 nF
- 1 Kondensator 1 µF 16V
- 1 LCD Display 2x16 oder 2x20 Zeichen, HD44780 kompatibel
- 1 Steckbrett und Kabel
- 1 Batteriekasten mit 3 Zellen (ca. 3,6 V)

11.1 Anschlussbelegung

Der Chip HD44780 von Hitachi ist ein ganz alter Klassiker aus den 90er Jahren. Er wurde von zahlreichen anderen Herstellern nachgeahmt. Zur Zeit werden fast alle 1 bis 2-Zeiligen LCD Displays auf die selbe Weise angesteuert.

Diese Displays haben in der Regel eine einfache 14 oder 16 Polige Anschlussleiste die du bitte wie folgt mit dem Mikrocontroller verbindest:

<u>Display</u>	<u>Arduino</u>	<u>Beschreibung</u>
1	GND	GND
2	VCC	Stromversorgung 5 V
3	VO	Kontrastspannung (siehe unten)
4	RS	Register Select (Low=Befehle, High=Daten)
5	R/W	Low=Read, High=Write

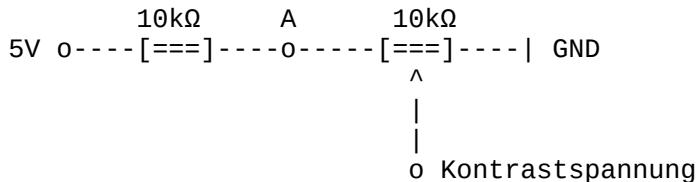
6	E	PD3	Enable (High=Datenfreigabe)
7	DB0		nicht anschließen
8	DB1		nicht anschließen
9	DB2		nicht anschließen
10	DB3		nicht anschließen
11	DB4	PD4	Datenleitung
12	DB5	PD5	Datenleitung
13	DB6	PD6	Datenleitung
14	DB7	PD7	Datenleitung
15	A		Anode der Beleuchtung
16	K		Kathode der Beleuchtung

Vergleiche diese Tabelle mit den technischen Unterlagen zu deinem konkreten Display! Angeblich gibt es Display mit abweichender Anschlussbelegung und du willst es ja nicht kaputt machen.

Um am Mikrocontroller Anschlüsse zu sparen, schließen wir nur die Hälfte der Datenleitungen an. Das Display kann nämlich wahlweise mit 4 oder 8 Datenleitungen angesteuert werden.

Die Beleuchtung kannst du bei den meisten Displays unbenutzt lassen, sie funktionieren auch ohne Licht. Wenn du sie jedoch anschließen möchtest, dann halte dich dabei an die technische Beschreibung des konkreten Displays. Manche schließt man einfach an 5 V an, andere benötigen hingegen einen Vorwiderstand.

Die „Kontrastspannung“ erzeugst du mit einem Trimmervoltmeter wie folgt:



Diese beiden Widerstände bilden einen Spannungsteiler. Da sie beide den gleichen Widerstandswert haben, fällt an beiden die Hälfte von den 5 V ab. Wenn du dein Multimeter an GND und Punkt A anschließt, wirst du etwa 2,5 V messen.

Das Trimmervoltmeter hat einen dritten Anschluss, mit dem man durch eine Drehbewegung jede beliebige Spannung im Bereich 0 bis 2,5 V abgreifen kann. Diese dient dem Display als einstellbare Kontrastspannung. Auch das kannst du mit dem Multimeter überprüfen.

Schließe das USB Kabel an, so dass die Schaltung mit Strom versorgt wird.

Die meisten Displays zeigen nun in der oberen Reihe blass-graue Klötzchen oder Balken an und in der unteren Reihe ist gar nichts zu sehen. Daran erkennt man, dass das Display mit Strom versorgt ist, aber noch nicht initialisiert wurde.

11.2 Timing

Die Kommunikation zum Display muss mit einem bestimmten Timing stattfinden, damit sie zuverlässig funktioniert.

Zu Beginn nach Anlegen der Stromversorgung ist die folgende Initialisierungs-Sequenz nötig:

- 1) Die I/O Pins vom Mikrocontroller als Ausgang konfigurieren.
- 2) Enable Leitung auf Low legen.
- 3) Mindestens 50 ms warten
- 4) RS Leitung auf Low legen.

- 5) Mindestens 40 ns warten.
- 6) Enable Leitung auf High legen.
- 7) DB7 bis DB4 auf 0011 setzen
- 8) Mindestens 250 ns warten.
- 9) Enable Leitung auf Low legen.
- 10) Mindestens 5 ms warten.
- 11) Enable Leitung auf High legen.
- 12) Mindestens 250 ns warten.
- 13) Enable Leitung auf Low legen.
- 14) Mindestens 150 µs warten.
- 15) Enable Leitung auf High legen.
- 16) Mindestens 250 ns warten.
- 17) Enable Leitung auf Low legen.
- 18) Mindestens 250 ns warten.
- 19) Enable Leitung auf High legen.
- 20) DB7 bis DB4 auf 0010 setzen
- 21) Mindestens 250 ns warten.
- 22) Enable Leitung auf Low legen.
- 23) Mindestens 50 µs warten.
- 24) Befehl FUNCTION_SET senden
- 25) Befehl ON_OFF_CONTROL senden
- 26) Befehl CLEAR_DISPLAY senden
- 27) Befehl ENTRY_MODE_SET senden
- 28) Befehl RETURN_HOME senden

Diese lange Initialisierungs-Sequenz ist im Datenblatt beschrieben, allerdings nicht so schön zusammenhängend. Beachte bei den Zeitangaben die wechselnden Einheiten ms, µs und ns.

Das Display wird mit Befehlen gesteuert. Zum Beispiel gibt es einen Befehl, um die Anzeige zu löschen. Und es gibt einen anderen Befehl, um den Cursor zu bewegen. Wenn wir hingegen Daten an das Display senden, erscheinen Schriftzeichen auf der Anzeige.

So sendet man einen Befehl an das Display:

- 1) RS Leitung auf Low legen.
- 2) Mindestens 40 ns warten.
- 3) Enable Leitung auf High legen.
- 4) Die oberen 4 Bits vom Befehl senden.
- 5) Mindestens 250 ns warten.
- 6) Enable Leitung auf Low legen.
- 7) Mindestens 250 ns warten.
- 8) Enable Leitung auf High legen.
- 9) Die unteren 4 Bits vom Befehl senden.
- 10) Mindestens 250 ns warten.
- 11) Enable Leitung auf Low legen.
- 12) Mindestens 37 µs warten (außer beim Befehl RETURN_HOME, der benötigt seltsamerweise 1520 µs).

Daten (also Buchstaben und andere Schriftzeichen) sendet man auf die gleiche Weise, allerdings muss dabei die RS Leitung auf High liegen.

11.3 Befehlssatz

Die Funktionen des Displays werden durch das Senden von Befehlen ausgelöst. Bei manchen Befehlen stehen einzelne Bits für Optionen, also Varianten des Befehls:

FUNCTION_SET = 0010nf00

- Funktion einrichten.
- n steht für die Anzahl der Zeilen. 0=Eine Zeile, 1=Zwei Zeilen.
- f steht für die Wahl des Zeichensatzes. 0=5x8 Pixel, 1=5x10 Pixel. Muss passend zum Display eingestellt werden.

ON_OFF_CONTROL = 00001dcb

- Teile an/aus schalten.
- d steht für „Display“. Wenn dieses Bit den Wert 0 hat, zeigt das ganze Display nichts an.
- c steht für „Cursor“ . Wenn das Bit den Wert 1 hat, wird der Cursor sichtbar.
- b steht für „Blinking of Cursor“. Wenn das Bit den Wert 1 hat, blinkt der Cursor.

CLEAR_DISPLAY = 00000001

- Display löschen.
- Dieser Befehl hat keine Optionen.

ENTRY_MODE_SET = 000001ds

- Art von Einträge einstellen:
- d steht für die Richtung, in die sich der Cursor nach dem Senden eines Zeichens bewegt. 0=links, 1=rechts.
- Wenn s=1 ist, dann wird der Text im Display verschoben.

RETURN_HOME = 00000010

- Bewegt den Cursor in die obere linke Ecke.
- Dieser Befehl hat keine Optionen

Im Datenblatt wird ein „Cursor or Display Shift“ Befehl genannt, den ich hier allerdings als vier einzelne Befehle darstelle, weil er so besser verständlich ist:

SHIFT_CURSOR_LEFT = 00010000

- Cursor nach links bewegen

SHIFT_CURSOR_RIGHT = 00010100

- Cursor nach rechts bewegen

SHIFT_DISPLAY_LEFT = 00011000

- Displayinhalt nach links schieben

SHIFT_DISPLAY_RIGHT = 00011100

- Displayinhalt nach rechts schieben

Das Display hat zwei Speicherbereiche. Einer für den Zeichensatz-Generator, den man benutzen kann, um eigene Schriftzeichen zu definieren. Das funktioniert ganz ähnlich, wie wir das im vorherigen Kapitel mit der LED-Marixanzeige gemacht haben. Ich erkläre später detailliert, wie man das macht.

SET_CGRAM_ADDRESS= 01xxxxxx

- Dieser Befehl bewirkt, dass das nächste Daten-Byte an die angegebene Adresse (xxxxxx) in den Speicher des Zeichensatz-Generator geschrieben wird. Der andere Speicherbereich repräsentiert die sichtbaren Textzeilen.

SET-DDRAM_ADDRESS = 1xxxxxxxx

- Dieser Befehl bewirkt, dass das nächste Daten-Byte an die angegebene Adresse (xxxxxx) in den Text-Speicher des Display geschrieben wird.
- Die erste Zeile beginnt an der Adresse 0000000.
- Die zweite Zeile beginnt bei den meisten Displays an der Adresse 1000000 (= 0x40)

Wir haben also 64 Bytes Speicher für die obere Zeile und nochmal 64 Bytes für die untere Zeile. Das Display kann aber nur 16 oder 20 davon darstellen. Die Befehle SHIFT_DISPLAY_LEFT und SHIFT_DISPLAY_RIGHT können verwendet werden, um den sichtbaren Bereich zu verschieben.

11.4 LCD Library

Wir können das Ganze jetzt in eine Library verpacken. Ich zeige dir hier den gesamten Quelltext. Zur Programmierung beginnen wir wieder mit der „Hello World“ Vorlage und tragen folgendes in die Datei hardware.h ein:

```
// The HD44780 Display is connected to Port D.
#define LCD_PORT_INIT { DDRD |= 0b11111100; }
#define LCD_RS_HIGH { PORTD |= (1<<PD2); }
#define LCD_RS_LOW { PORTD &= ~(1<<PD2); }
#define LCD_E_HIGH { PORTD |= (1<<PD3); }
#define LCD_E_LOW { PORTD &= ~(1<<PD3); }

#define LCD_DATA_LOW_NIBBLE(b) { \
    PORTD = (PORTD & 0x0F) | (b << 4); \
}

#define LCD_DATA_HIGH_NIBBLE(b) { \
    PORTD = (PORTD & 0x0F) | (b & 0xF0); \
}
```

Mit diesen Makro Definitionen steuern wir das Display an. Die beiden letzten Makros dienen dazu, ein Byte in zwei mal vier Bits zu zerlegen. Die unteren vier Bits nennt man Low-Nibble und die oberen vier Bits nennt man High-Nibble. Diese beiden Nibbles werden nacheinander an das Display gesendet, denn wir haben ja nur 4 Datenleitungen angeschlossen.

- **PORTD = (PORTD & 0x0F)**
löscht die oberen vier Bits von Port D, das sind unsere vier Dateileitungen zum Display. Die anderen Pins sind Steuerleitungen, sie sollen unverändert bleiben.
- **... | (b << 4)**
schiebt die rechten vier Bits von dem Byte b nach links und verknüpft sie mit dem vorherigen Ausdruck. Dadurch wird also die rechte Hälfte vom Byte an das Display gesendet.
- **... | (b & 0xF0)**
blendet die rechten vier Bits von dem Byte b aus, so dass nur noch die linken vier Bits übrig bleiben. Diese werden anschließend mit dem vorherigen Ausdruck verknüpft. Dadurch wird also die linke Hälfte vom Byte an das Display gesendet.

Lege eine neue Datei driver/lcd.h mit folgendem Inhalt an:

```
#ifndef __LCD_H__
#define __LCD_H__
```

```

#include <stdint.h>
#include <avr/pgmspace.h>

// This driver provides access to an LCD display
// with HD44780 controller.
// Only the 4bit communication mode is implemented.

// Initialize the display controller.
void LCD_init(uint8_t options);

// Send a command.
void LCD_command(uint8_t cmd);

// Commands and options (add options to the command):

#define LCD_FUNCTION_SET      0b00100000
#define LCD_1_LINE            0b0000
#define LCD_2_LINES           0b1000
#define LCD_5x8_FONT          0b0000
#define LCD_5x10_FONT         0b0100

// Clear display and set cursor to home position.
#define LCD_CLEAR_DISPLAY     0b00000001

// Set cursor to the home position and reset
// the shift position.
// The DDRAM content remains unchanged.
// Note that this command takes much more time
// than the clear display command.
#define LCD_RETURN_HOME        0b00000010

// Set entry mode
// = what happens after writing a character to the DDRAM.
#define LCD_ENTRY_MODE_SET    0b00000100
#define LCD_DECREMENT          0b00
#define LCD_INCREMENT          0b10
#define LCD_SHIFT              0b01

// Control which features are on:
#define LCD_ON_OFF_CONTROL   0b00001000
#define LCD_OFF                0b000
#define LCD_DISPLAY             0b100
#define LCD_CURSOR              0b010
#define LCD_BLINKING            0b001

// Shift the cursor
#define LCD_SHIFT_CURSOR_LEFT  0b00010000
#define LCD_SHIFT_CURSOR_RIGHT  0b00010100

// Shift the display
#define LCD_SHIFT_DISPLAY_LEFT 0b00011000
#define LCD_SHIFT_DISPLAY_RIGHT 0b00011100

// Set address of character generator RAM,
// the following data are written to the CGRAM.
// Add the 6bit address value to the command.
#define LCD_SET_CGRAM_ADDRESS  0b01000000

// Set the address of display data RAM,
// the following data are written to the DDRAM.

```

```

// Add the 7bit address value to the command.
// Note that the second line ususally begins
// at address 0x40.
#define LCD_SET_DDRAM_ADDRESS    0b10000000

// Write one byte to the data register (CGRAM or DDRAM).
void LCD_data(uint8_t data);

// Write a string to the data register (CGRAM or DDRAM).
void LCD_write(char* text);

// Write a string from program memory to the
// data register (CGRAM or DDRAM).
void LCD_write_P(PGM_P text);

#endif //__LCD_H_

```

Lege eine neue Datei driver/lcd.c mit folgendem Inhalt an:

```

#include <util/delay.h>
#include "lcd.h"
#include "../hardware.h"

void LCD_command(uint8_t cmd)
{
    LCD_RS_LOW;
    _delay_us(0.040);
    LCD_E_HIGH;
    LCD_DATA_HIGH_NIBBLE(cmd);
    _delay_us(0.250);
    LCD_E_LOW;
    _delay_us(0.250);
    LCD_E_HIGH;
    LCD_DATA_LOW_NIBBLE(cmd);
    _delay_us(0.250);
    LCD_E_LOW;
    if (cmd==LCD_RETURN_HOME) // return home
    {
        _delay_us(1600);
    }
    else
    {
        _delay_us(60);
    }
}

void LCD_data(uint8_t data)
{
    LCD_RS_HIGH;
    _delay_us(0.040);
    LCD_E_HIGH;
    LCD_DATA_HIGH_NIBBLE(data);
    _delay_us(0.250);
    LCD_E_LOW;
    _delay_us(0.250);
    LCD_E_HIGH;
    LCD_DATA_LOW_NIBBLE(data);
    _delay_us(0.250);
}

```

```

        LCD_E_LOW;
        _delay_us(60);
    }

void LCD_write(char* text)
{
    char c;
    while ((c=*text))
    {
        LCD_data(c);
        text++;
    }
}

void LCD_write_P(PGM_P text)
{
    char c;
    while ((c=pgm_read_byte(text)))
    {
        LCD_data(c);
        text++;
    }
}

void LCD_init(uint8_t options)
{
    LCD_PORT_INIT;
    LCD_E_LOW;
    _delay_ms(50);
    LCD_RS_LOW;
    _delay_us(0.040);
    LCD_E_HIGH;
    LCD_DATA_HIGH_NIBBLE(0b00110000);
    _delay_us(0.250);
    LCD_E_LOW;
    _delay_ms(5);
    LCD_E_HIGH;
    _delay_us(0.250);
    LCD_E_LOW;
    _delay_us(150);
    LCD_E_HIGH;
    _delay_us(0.250);
    LCD_E_LOW;
    _delay_us(0.250);
    LCD_E_HIGH;
    LCD_DATA_HIGH_NIBBLE(0b00100000);
    _delay_us(0.250);
    LCD_E_LOW;
    _delay_us(50);
    LCD_command(LCD_FUNCTION_SET+options);
    LCD_command(LCD_ON_OFF_CONTROL+LCD_DISPLAY);
    LCD_command(LCD_CLEAR_DISPLAY);
    LCD_command(LCD_ENTRY_MODE_SET+LCD_INCREMENT);
    LCD_command(LCD_RETURN_HOME);
}

```

Wenn du diese Library mit den vorherigen Kapiteln oder dem Datenblatt vergleichst, wirst du sehen, dass ich hier die Funktionen des Displays einfach 1:1 in den Programmcode übernommen habe.

11.5 Text Ausgeben

Probiere das folgende Hauptprogramm main.c aus:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include "driver/lcd.h"
#include "hardware.h"

int main(void)
{
    LCD_init(LCD_2_LINES+LCD_5x8_FONT);
    LCD_write_P(PSTR("Hallo"));
    LCD_command(LCD_SET_DDRAM_ADDRESS+0x40);
    LCD_write("Quentin");
}
```

Zuerst wird das Display mit den beiden Optionen LCD_2_LINES und LCD_5x8_FONT initialisiert:

```
LCD_init(LCD_2_LINES+LCD_5x8_FONT);
```

Diese beiden Optionen müssen mit dem technischen Aufbau des Displays überein stimmen, sonst kommt keine sinnvolle Anzeige zustande.

Dann wird mit

```
LCD_write_P(PSTR("Hallo"));
```

ein bisschen Text ausgegeben. Er erscheint links oben in der ersten Zeile des Displays. Als Nächstes wird etwas in die zweite Zeile geschrieben:

```
LCD_command(LCD_SET_DDRAM_ADDRESS+0x40);
LCD_write("Quentin");
```

Der Befehl SET_DDRAM_ADDRESS legt fest, dass der nächste Text an in die angegebene Speicherzelle (und die darauf folgenden) geschrieben werden soll. An Adresse 0x40 beginnt wie bereits geschrieben die zweite Zeile.

Hast du bemerkt, dass ich zwei Unterschiedliche Funktionen zur Textausgabe verwendet habe?

```
LCD_write_P(PSTR("Hallo"));
LCD_write("Quentin");
```

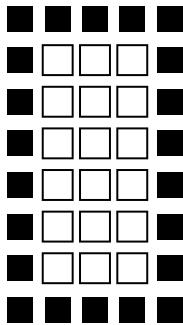
Die obere Variante ist besser, weil die RAM Speicher spart. Bei der unetren Variante wird noch vor dem Ausführen der main() Funktion der Text „Quentin“ ins RAM kopiert und dann erst ausgegeben. Wenn du das mit vielen Textausgaben machst, hast du rasch das ganze verfügbare RAM aufgebraucht. Daher solltest du stets die Variante mit „_P“ bevorzugen, wo immer das möglich ist.

Jetzt, wo das Display initialisiert ist und Text angezeigt wird, kannst du das Trimmoti so einstellen, dass die Anzeige optimal ist. Wenn der Kontrast zu niedrig ist, erscheinen die Schriftzeichen sehr blass. Wenn der Kontrast zu hoch ist, verdunkelt sich der Hintergrund, der eigentlich hell sein soll.

Das Display kann übrigens nicht alle Zeichen darstellen, die du womöglich von deinem PC gewohnt bist. Im Datenblatt des HD44780 Chips (<https://www.sparkfun.com/datasheets/LCD/HD44780.pdf>) sind die Darstellbaren Zeichen auf Seite 17 abgebildet.

11.6 Zeichensatz Generator

Der Zeichensatz-Generator kann neben den vordefinierten Zeichen auch ein paar selbst definierte Zeichen darstellen. Wir werden ein leeres Rechteck erzeugen:



Dazu musst du folgende Befehle ausführen:

```
LCD_command(LCD_SET_CGRAM_ADDRESS+0);
LCD_data(0b11111);
LCD_data(0b10001);
LCD_data(0b10001);
LCD_data(0b10001);
LCD_data(0b10001);
LCD_data(0b10001);
LCD_data(0b10001);
LCD_data(0b11111);
```

Hier wird zuerst festgelegt, dass die folgenden Daten in den Speicher des Zeichensatz-Generators geschrieben werden sollen. Es folgen 8 Bytes entsprechend der 8 Pixel-Reihen des Zeichens. Von jedem Byte werden allerdings nur 5 Bits verwendet. Anschließend kannst du dieses selbst erstellte Zeichen so anzeigen:

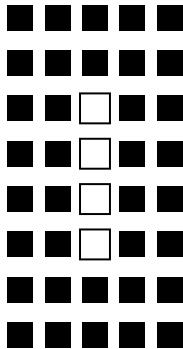
```
LCD_command(LCD_SET_DDRAM_ADDRESS+6);
LCD_data(0);
```

Dadurch wird festgelegt, dass das nächste Datenbyte in den Text-Speicher geschrieben wird, und zwar hinter das Wort „Hallo“. Dann wird der Wert 0 gesendet.

Der Wert 0 bewirkt, dass das erste selbst definierte Zeichen erscheint, welches im Zeichensatz-Generator Speicher ab Adresse 0 angelegt wurde.

Der Wert 1 würde bewirken, dass das zweite selbst definierte Zeichen erscheint, welches im Zeichensatz-Generator Speicher ab Adresse 8 angelegt sein muss.

Das probieren wir direkt mal mit einem anderen Rechteck aus:



Dazu musst du folgende Befehle ausführen:

```
LCD_command(LCD_SET_CGRAM_ADDRESS+8);
LCD_data(0b11111);
LCD_data(0b11111);
LCD_data(0b11011);
LCD_data(0b11011);
LCD_data(0b11011);
LCD_data(0b11011);
LCD_data(0b11111);
LCD_data(0b11111);

LCD_command(LCD_SET_DDRAM_ADDRESS+7);
LCD_data(1);
```

Nun erscheint rechts neben dem leeren Rechteck ein zweites Rechteck mit dickem Rahmen. Du kannst die selbst definierten Zeichen auch mehrfach benutzen. Zum Beispiel so:

```
LCD_command(LCD_SET_DDRAM_ADDRESS+6);
LCD_data(0);
LCD_data(1);
LCD_data(0);
LCD_data(1);
LCD_data(0);
```



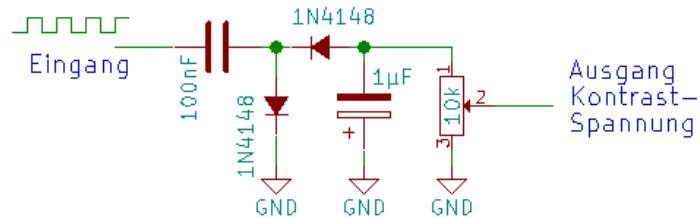
Insgesamt hat der Zeichensatz-Generator Platz für 64 Bytes, also für 8 selbst definierte Zeichen. Jetzt schalten wir noch den blinkenden Cursor an, um zu sehen, wo der sich nun eigentlich befindet und wie er aussieht:

```
LCD_command(LCD_ON_OFF_CONTROL + LCD_DISPLAY + LCD_CURSOR+ LCD_BLINKING);
```

Der blinkende Cursor erscheint rechts neben den selbst definierten Rechtecken, denn dort fand die letzte Ausgabe statt.

11.7 LCD an 3,3V

Die meisten LCD Anzeigen können auch mit 3,3V betrieben werden, allerdings benötigt das Display dann eine negative Kontrastspannung. Diese erzeugt man zum Beispiel mit einer Ladungspumpe:

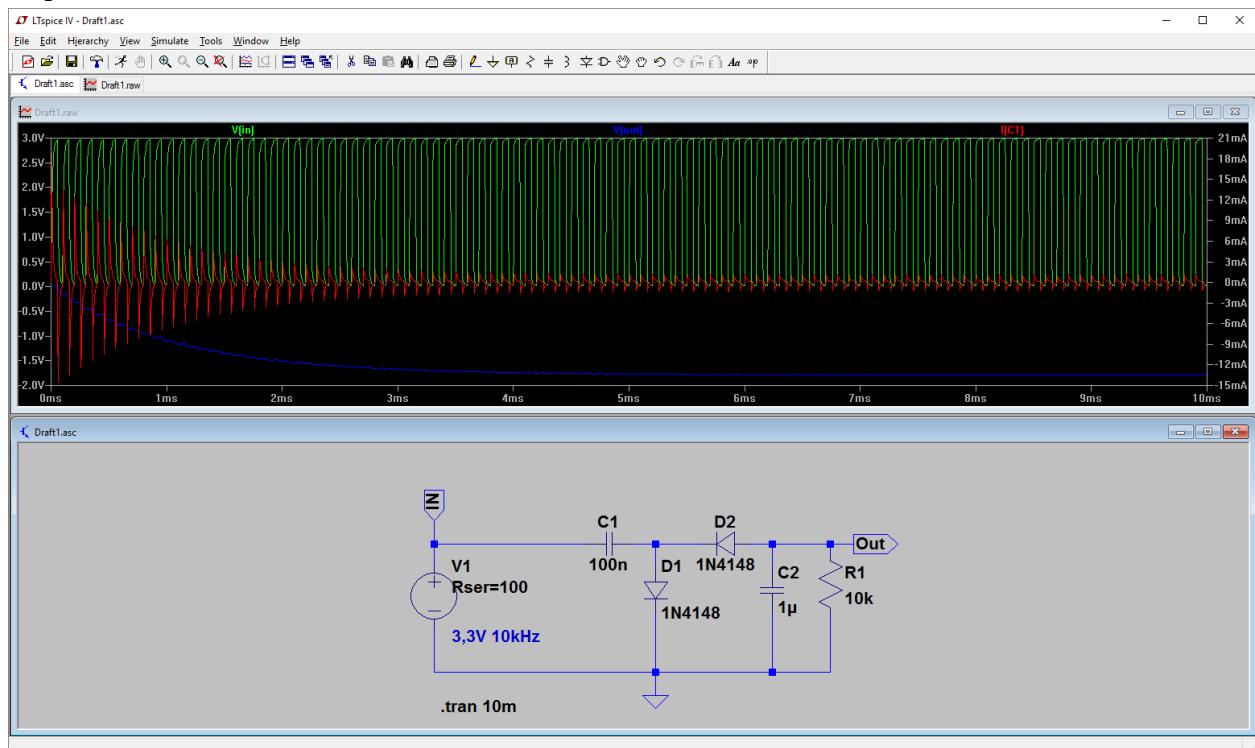


Diese Schaltung erzeugt aus einem positiven Rechteck Signal eine negative Ausgangsspannung von -1,8 Volt. Sie funktioniert so: Wenn das Eingangssignal auf High geht, lädt sich der 100 nF Kondensator über die linke Diode auf 2,5 V auf (das sind die 3,3 V abzüglich Verluste durch die Diode und den Mikrocontroller).

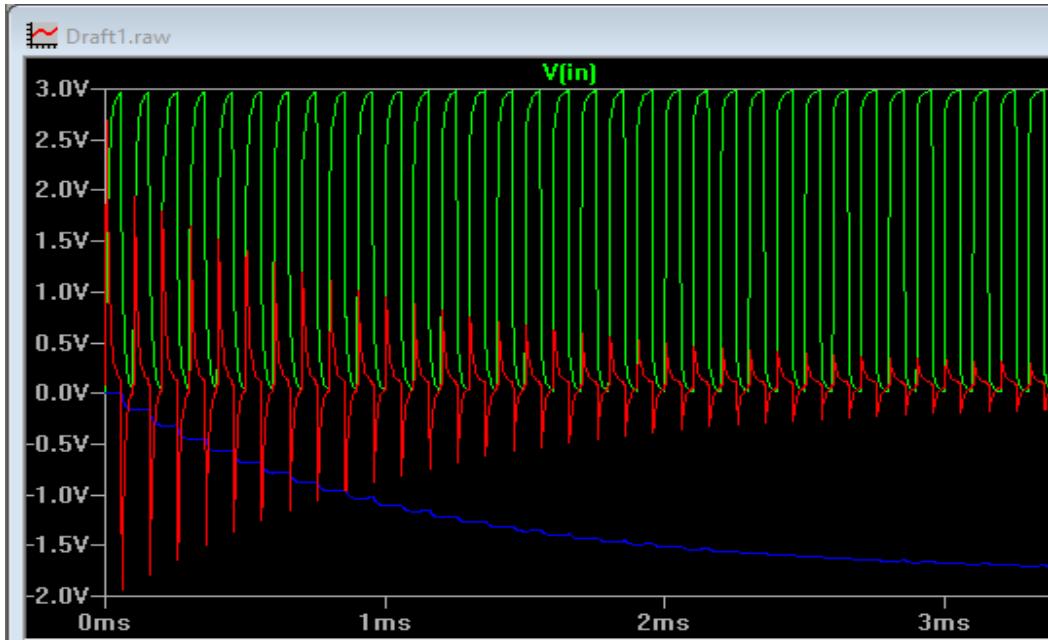
Wenn danach das Eingangssignal auf Low geht, entsteht auf der rechten Seite des 100nF Kondensators eine negative Spannung von von -2,5 V. Diese entlädt sich durch die zweite Diode in den 1 μF Kondensator hinein. Der 1 μF Kondensator stabilisiert die Ausgangsspannung. Er dient als Reservoir um die Lücken zwischen den Umlade-Impulsen zu füllen.

Da auch an der zweiten Diode 0,7 V verloren gehen, kommen hinten -1,8 V heraus. Mit dem Trimmptoti kannst du die Kontrastspannung für die LCD Anzeige im Bereich 0 bis -1,8 V einstellen.

Schaltungen wie diese kann man sehr bequem mit dem kostenlosen Programm LTSpice (<http://www.linear.com/designtools/software/>) entwerfen und simulieren. Ein Bildschirmfoto von LTSpice:



Vergrößerung von dem interessanten Ausschnitt:



Die grüne Linie zeigt das 10 kHz Rechtecksignal. Die rote Linie zeigt den Strom, der durch C1 fließt (positiv=laden, negativ=entladen). Die blaue Linie zeigt die Ausgangsspannung.

Du kannst hier schön sehen, dass die Ausgangsspannung Anfangs zunächst 0 V beträgt und dann treppenförmig immer weiter nach unten geht, bis auf -1,8 V. Nach ungefähr 6 Millisekunden bleibt sie stabil auf diesem Niveau.

Mit LTSpice kann man bequem ausprobieren, wie sich unterschiedliche Spannungen und Frequenzen auf die Schaltung auswirken. Und man kann damit recht schnell die optimalen Größen der Kondensatoren ermitteln. Es lohnt sich also, das Programm benutzen zu lernen. So habe ich zum Beispiel herausgefunden, dass die Eingangsfrequenz mindestens 10 kHz haben muss und dass die Schaltung mit 100 kHz immer noch gut funktioniert.

Nun stellt sich die Frage, wo man so ein 10 kHz Signal her bekommt. Wir haben einen Mikrocontroller vor uns liegen, der das mit seinem Timer erledigen kann.

Füge folgenden Code in den Anfang der main() Funktion ein:

```
// Starte Timer 2 um ein 10kHz Signal
// an OC2A (=Arduino D11) zu erzeugen

TCCR2A=(1<<COM2A0)+(1<<WGM21); // CTC Mode, toggle OC2A
TCCR2B=(1<<CS22)+(1<<CS20); // Prescaler 128
OCR2A=6; // Count from 0..5
TIMSK2=0; // No interrupts
ASSR=0; // No async operation
DDRB|=(1<<PB3); // PB3=OC2A is an output
```

Der Timer 2 wird so konfiguriert, dass er fortlaufend von 0 bis 5 zählt und wieder von vorne. Bei jedem durchlauf wechselt der Ausgang OC2A (das ist Pin PB3, am Arduino als D11 beschriftet) seinen Pegel. Es ergibt sich eine Frequenz von:

16 Mhz : 128 : 6 : 2 = 10,4 kHz

Mit diesem Signal kannst du die Ladungspumpe vorsorgen. Baue die Ladungspumpe auf dem Steckbrett auf und verbinde sie mit dem Anschluss D11 und dem Kontrast-Eingang des Displays. Installiere das Programm auf den Mikrocontroller.

11.7.1 Ladungspumpe Testen

Um die Ladungspumpe testen zu können, müssen wir das Arduino Nano Modul und das Display mit 3,3 bis 4 Volt betreiben. Setze in dazu in den Batteriehalter also drei Akkus ein, oder schwache Einwegbatterien, oder verwende den LF33CV Spannungsregler.

Schließe die Stromversorgung noch nicht an!

Das Datenblatt des Mikrocontrollers verlangt 5 V wenn die Taktfrequenz mehr als 10 Mhz beträgt. Erfahrungsgemäß funktioniert es trotzdem mit 3,3 bis 4 Volt. Für dieses kurze Experiment ist das Ok, bei kommerziellen Anwendungen rate ich allerdings schon dazu, die Vorgaben aus dem Datenblatt ernst zu nehmen.

Beim Testen musst du jetzt bitte sehr vorsichtig vorgehen, damit nichts kaputt geht! Das Arduino Modul darf nicht gleichzeitig an USB und Batterien angeschlossen werden, sonst entsteht ein Kurzschluss.

Entferne nun das USB Kabel und lege es weit weg!

Schließe die Batterien an die Arduino Anschlüsse GND und 5 V an. Das zuvor hochgeladene Programm sollte nun starten und den programmierten Text auf dem Display darstellen.

Am Anschluss D11 kommt ein Rechtecksignal heraus. Du kannst das mit einem Oszilloskop überprüfen. Oder schließe einen Piezo Schallwandler an, dann kannst du die Frequenz hören.

Je nach Batteriespannung liefert die Ladungspumpe eine negative Spannung von ungefähr -2V . Stelle die Spannung mit dem Trimmpoti so ein, dass das Display einen guten Kontrast anzeigt.

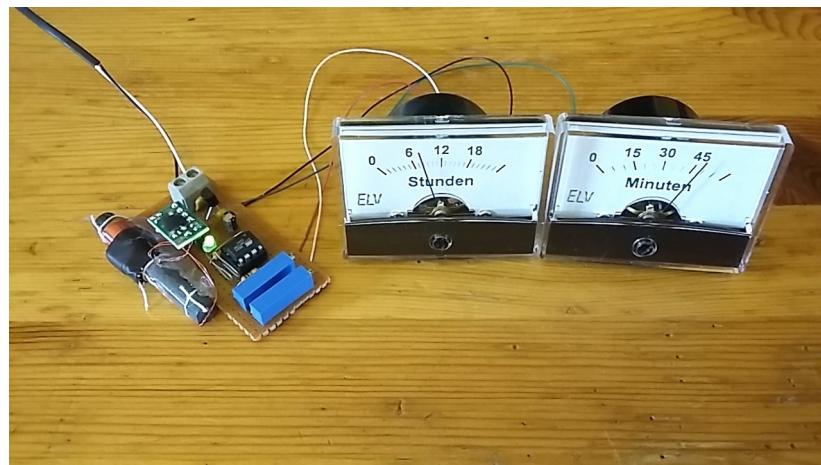
Entferne die Batterie wieder und lege sie weit weg. Schließe erst danach wieder das USB Kabel an.

Das Programm startet wieder, aber auf dem Display wirst du kaum etwas erkennen. Wahrscheinlich ist die ganze Anzeige jetzt total schwarz, weil die Kontrastspannung zu hoch ist. Für den Betrieb an 5 V kannst du die Ladungspumpe also nicht gebrauchen. Entferne sie daher wieder und schließe das Trimmpoti wieder so an, wie im vorherigen Kapitel beschrieben war.

12 Analoge Funkuhr

In dem Ort Mainflingen bei Frankfurt steht die sogenannte „Atomuhr“. Sie gibt für halb Europa die offizielle Uhrzeit vor. Netterweise wird das Zeit-Signal von dort aus kostenlos als Funksignal gesendet, so dass wir es im ganzen Land empfangen können.

Ich stelle dir hier nun eine ganz besondere Uhr vor, die dieses Funksignal verwendet. Sie sieht so aus:



Die Idee ist folgende: Ein AVR Mikrocontroller empfängt das DCF77 Funksignal, dekodiert es und zeigt die Uhrzeit mit Hilfe von PWM Signalen auf den beiden analogen Einbauinstrumenten an.

Material:

- 1 DCF-77 Empfänger Modul „DCF1“ von Pollin.de
- 2 Drehspul Einbau-Meßinstrumente 100 μ A
- 1 Mikrocontroller ATtiny13A-PU (alternativ Attiny25, 45 oder 85)
- 1 LED grün
- 1 Widerstand 220 Ω 1/4 Watt
- 1 Widerstand 1 k Ω 1/4 Watt
- 2 Widerstände 27 k Ω 1/4 Watt
- 1 Widerstand 47 k Ω 1/4 Watt
- 1 Transistor BC337-40
- 2 Trimmtpoti 10 k Ω (vorzugsweise ein Spindeltrimmer)
- 2 Kondensator 100 nF
- 1 Kondensator 2,2 μ F
- 1 Spannungsregler LP2950-3.3 oder LF33CV
- 1 Netzteil 5 V mindestens 20 mA

12.1 Analoge Einbauinstrumente

Für die Anzeige der Uhrzeit werden wir zwei analoge Einbauinstrumente.



Auf der Rückseite findest du die beiden Hauptanschlüsse, sie sind mit „+“ und „-“ gekennzeichnet. Sie führen zum magnetischen Antrieb im Innern des Instrumentes.



Manche andere Einbauinstrumente haben zwei weitere kleinere Anschlüsse, zur Versorgung von kleinen Glühlämpchen, die man allerdings separat dazu kaufen muss.

Öffne die Front-Abdeckung vorsichtig, um dir den Antrieb des Instrumentes anzuschauen. Aber sei dabei sehr vorsichtig, denn die Teile sind so filigran, dass man sie nicht berühren sollte. Du siehst eine kupferfarbene Spule, die drehbar gelagert ist und von einer Spiralfeder gehalten wird. Innerhalb der Spule befindet sich ein fest montierter Magnet.

Wenn die Spule von Strom durchflossen wird, erzeugt sie ein magnetisches Feld, das zu einer Drehbewegung führt. Je mehr Strom fließt, umso stärker ist das Magnetfeld und umso stärker drückt die Spule gegen die Spiralfeder. Der Zeiger bewegt sich dementsprechend mehr oder weniger weit nach rechts.

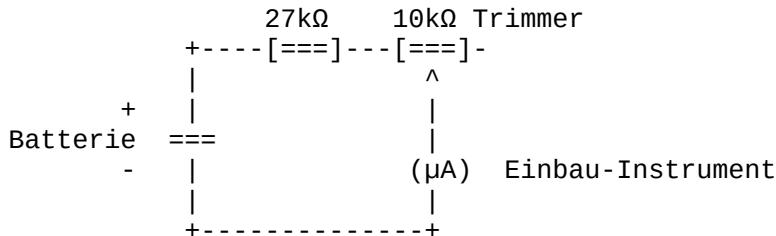
Auf der Front-Blende befindet sich eine schwarze Plastikschraube mit der man die Feder ein bisschen verdrehen kann. Damit stellt man den Nullpunkt ein. Ohne Strom soll der Zeiger genau auf 0 zeigen.

Die Funktionsweise ist bei Wikipedia sehr schön beschrieben:

<https://de.wikipedia.org/wiki/Drehspulmesswerk>

Der Vollausschlag wird bei etwa 100 μA erreicht. Das ist sehr wenig Strom. Und viel mehr verträgt das Gerät auch nicht. Deshalb darfst du das Instrument auf gar keinen Fall direkt an eine Batterie oder an den Mikrocontroller anschließen. Es würde ein viel zu hoher Strom fließen, der die feine Spule binnen Sekunden durchbrennen würde.

Wir verwenden zwei Widerstände, um die Stromstärke auf das richtige Maß zu bringen:



Probeaufbau:



Das $10\text{ k}\Omega$ Trimmpoti hat drei Anschlüsse, von denen wir nur den mittleren und einen seitlichen verwenden. Drehe es mit einem Schraubendreher zunächst in die mittlere Position, dann hat es ungefähr $5\text{ k}\Omega$. Die Spule des Einbauinstrumentes hat typischerweise einen Innenwiderstand von $2\text{ k}\Omega$. Somit haben alle drei Widerstände zusammen $27\text{ k}\Omega + 5\text{ k}\Omega + 2\text{ k}\Omega = 34\text{ k}\Omega$.

Wenn die Batterie $3,6\text{ V}$ hat, geht der Zeiger auf 100% , denn es fließen ungefähr $100\text{ }\mu\text{A}$.

Du kannst das nachrechnen: $3,6\text{ V} : 34\text{ k}\Omega = 0,000105\text{ A} = 105\text{ }\mu\text{A}$

Mit dem Trimmpoti kannst du den Volllausschlag justieren, so dass der Zeiger bei frisch geladener Akkus genau 100% anzeigt. Probiere es mit nur einen Akkus also $1,2\text{ Volt}$. In diesem Fall wird der Zeiger nur ein Drittel so weit ausschlagen.

Die Skalen der Einbauinstrumente habe ich mit selbst gedruckten Etiketten überklebt. Einfaches Kopierpapier ist dazu gut genug. Eine Skala ist von 0 bis 24 Stunden eingeteilt, die andere geht von 0 bis 60 Minuten. Ich habe passende Druckvorlagen im Shop des ELV Verlages bei Artikel Nummer 68-098958 gefunden:

https://files.elv.com/Assets/Produkte/9/989/98958/Downloads/98958_dud1_uhren_skalen.pdf

12.2 DCF Empfänger

Alle größeren Elektronik Versandhäuser haben entsprechende Funkempfänger im Programm. Wir werden das DCF1 Modul vom Pollin Versandhandel verwenden. Es sieht so aus:



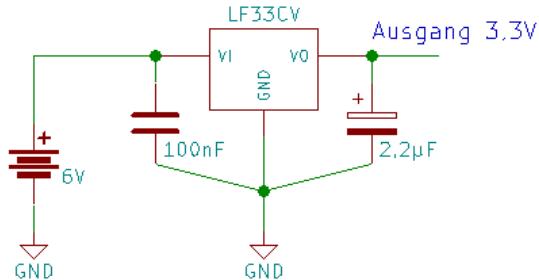
Der schwarze Stab besteht aus magnetisierbarem Metall-Pulver. Er empfängt das magnetische Feld des Funksignals. Die orange Spule wandelt das magnetische Signal in ein elektrisches Signal um, welches von der kleinen Platine ähnlich einem Radio gefiltert und verstärkt wird.

Die Anschlussdrähte des DCF1 Moduls brechen schnell ab. Du solltest sie daher möglichst wenig bewegen und mit etwas Heißleber oder Tesafilm fixieren.

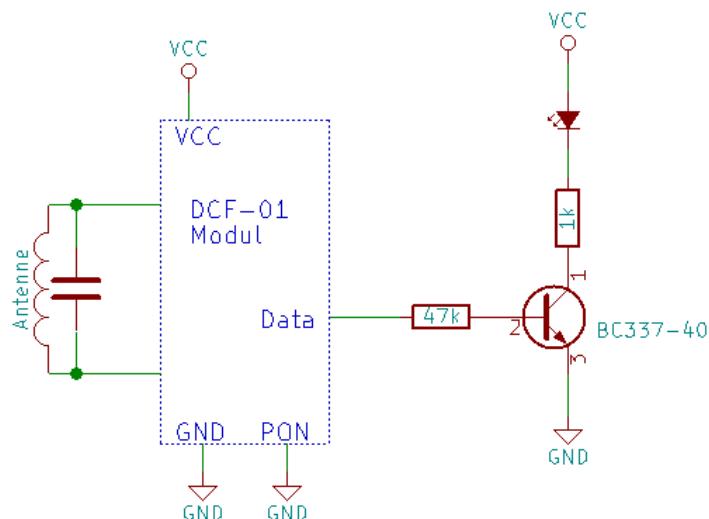
Alle DCF Empfänger (nicht nur der von Pollin) benötigen eine sehr stabile Versorgungsspannung, die frei von Störsignalen ist.

Wir werden unsere Uhr mit einem Netzteil betreiben, denn Batterien würden nur wenige Monaten halten. Manche Schaltnetzteile sind hier ein bisschen problematisch, sie stören den Empfang. Falls das bei Dir der Fall ist, versuche, die GND Leitung zu erden oder probiere ein anderes Netzteil.

Die Ausgangsspannung handelsüblicher Steckernetzteile ist für DCF Empfänger zu hoch, darum schalten wir hinter das Netzteil noch einen Spannungsregler, der stabile 3,3V erzeugt.



Du kannst dazu wahlweise den LF33CV oder den LP2950-3.3 verwenden. An den Ausgang des Funkmoduls schließen wir zunächst einen Transistor und eine Leuchtdiode an:



Der Ausgang des DCF1 Moduls kann nur sehr wenig Strom liefern – zu wenig für die Leuchtdiode. Deswegen nutzen wir den Transistor als Schaltverstärker. Immer wenn der Ausgang auf High geht, fließt ein Strom in die Basis des Transistors. Der schaltet dann durch, so dass die Leuchtdiode an geht.

Stecke diese Teile auf dem Steckbrett zusammen und probiere sie aus. Die Antenne soll auf dem Tisch liegen (nicht hochkant stehen). Drehe die Antenne so, dass die LED regelmäßig jede Sekunde kurz blitzt. Du musst dabei geduldig vorgehen. Nach jeder Bewegung dauert es ein paar Sekunden, bis der Empfänger sich auf das Signal eingestellt hat.

Falls du einen anderen Funkempfänger verwendest, könntest du auf folgende Unterschiede stoßen:

- Das Signal ist eventuell umgekehrt gepolt. Die LED leuchtet dann mit kurzen Aussetzern. Dann musst du die Funktion empfange() im Programm entsprechend anpassen. (siehe unten).
- Manche Module haben zwei Ausgänge. Verwende dann den Ausgang, der positive Impulse liefert.
- Der Ausgang benötigt eventuell einen Pull-Up Widerstand.

12.3 DCF Signal

Schau genau hin, du wirst sehen, dass die LED unterschiedlich lange aufleuchtet. Mal etwas länger, mal etwas kürzer. In diesen Unterschieden ist nicht nur die Uhrzeit, sondern auch das Datum kodiert. Diese Unterschiede wird unser Mikrocontroller erkennen und auswerten.

Im Verlauf von mehreren Minuten, wirst du sehen, dass genau einmal pro Minute ein Impuls fehlt. Dieses Signal kennzeichnet den Beginn einer neuen Minute und damit auch den Anfang eines Daten-Paketes.

Bitte lies nun den Wikipedia Artikel zum Thema DCF77: <https://de.wikipedia.org/wiki/DCF77>

Das DCF77 Signal übermittelt Uhrzeit und Datum seriell mit sehr geringer Bitrate, nämlich einem Bit pro Sekunde. Jedes Datenpaket dauert genau eine Minute.

- Low-Bits werden als kurze Impulse mit 100ms Dauer übertragen.
- High-Bits werden als lange Impulse mit 200ms Dauer übertragen.

Wir verwenden nicht alle Bits, weil unsere Uhr nur Stunden und Minuten anzeigen soll:

Bit	Bedeutung
0	Ist immer Low
1-20	Verwenden wir nicht
21-27	Minuten in BCD Kodierung
28	Parität für die Minuten
29-34	Stunden in BCD Kodierung
35	Parität für die Stunden
36-58	Verwenden wir nicht
59	Ist eine Lücke ohne Impuls

Die BCD Codierung ist so aufgebaut:

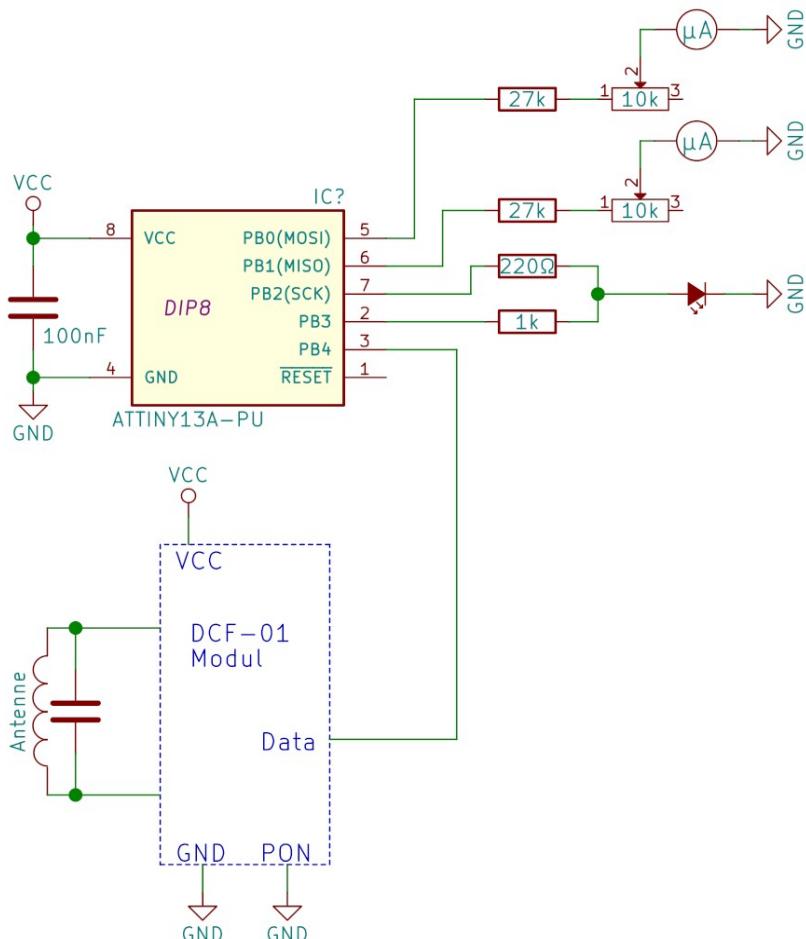
- Die unteren 4 Bits enthalten die Einer als Binärzahl.
- Die oberen Bits enthalten die Zehner als Binärzahl.

Unser Programm wird das in „normale“ Binärzahlen umrechnen.

Die Parität-Bits dienen dazu, einfache Übertragungsfehler zu erkennen. Die Parität gibt an, ob die Anzahl der High-Bits in den Minuten (bzw. Stunden) gerade oder ungerade ist.

12.4 Schaltung

Entferne die LED und den Transistor. Stattdessen schließen wir nun den ATtiny13 Mikrocontroller an. Er soll das Signal dekodieren.



Der Mikrocontroller erzeugt an den Ausgängen PB0 und PB1 eine pulsierende Gleichspannung (PWM-Signal), deren Pulsbreite variiert wird.



Kurze Impulse lassen die Zeiger der Einbauinstrumente nur wenig ausschlagen. Lange Impulse führen zu einem größeren Ausschlag. Mit Hilfe des PWM Timers können wir die Länge der Impulse im Bereich von 0-255 Takte variieren, wir nutzen tatsächlich aber nur den Bereich von 0-240 aus, weil das einfacher zu berechnen ist.

Die mechanische Trägheit der Einbauinstrumente sorgt dafür, dass die Zeiger nicht wild hin und her zucken.

Die Leuchtdiode wird über zwei unterschiedliche Widerstände angesteuert, so dass sie wahlweise hell oder dunkel leuchten kann. Die Software wird das so nutzen:

- Wenn der Empfang in Ordnung ist, leuchtet die LED dunkel (PB3).
- Zusätzlich zeigt ein überlagertes Blinken das empfangenes Signal an (PB2) .

12.5 Programm

Das Programm basiert auf der „Hello-World“ Vorlage, jedoch musst du den Systemtimer und die serielle Konsole entfernen. Der Mikrocontroller wird mit unveränderten Fuses verwendet, so dass er mit 1,2 Mhz Taktfrequenz läuft.

Die Datei hardware.h enthält folgende Zeilen:

```
// Enable the two PWM outputs
#define ENABLE_PWM_OUTPUTS { DDRB |= (1<<PB0) + (1<<PB1); }

// The Status LED output
#define STATUS_LED_ON { DDRB |= (1<<PB2); PORTB |= (1<<PB2); }
#define STATUS_LED_OFF { DDRB &= ~(1<<PB2); PORTB &= ~(1<<PB2); }

// The Signal LED output
#define SIGNAL_LED_ON { DDRB |= (1<<PB3); PORTB |= (1<<PB3); }
#define SIGNAL_LED_OFF { DDRB &= ~(1<<PB3); PORTB &= ~(1<<PB3); }

// The DCF-77 receiver is connected to PB4
#define DCF_SIGNAL (PINB & (1<<PB4))
```

Das ganze Programm besteht nur aus der einen Datei main.c mit folgendem Inhalt:

```
if (++sekunden==60)
{
    sekunden=0;
    if (++minuten==60)
    {
        minuten=0;
        if (++stunden==24)
        {
            stunden=0;
        }
        anzeigen();
    }
}

// Empfange eine Pause und einen Impuls, messe deren Länge.
// Liefert das Ergebnis in den Variablen "puls" und "pause".
void empfange(void)
{
    pause=0;
    while (!DCF_SIGNAL && pause<4000)
    {
        _delay_ms(1);
        pause++;
    }
    SIGNAL_LED_ON;
    puls=0;
```

```

        while (DCF_SIGNAL && puls<4000)
        {
            _delay_ms(1);
            puls++;
        }
        SIGNAL_LED_OFF;
    }

// Prüfe, ob die Sekunde 0 empfangen wurde.
#define IS_START (pause>1500 && pause<2500 && puls>50 && puls<150)

// Prüfe, ob ein High-Bit empfangen wurde
#define IS_HIGH (puls>=150)

// Prüfe, ob eine Empfangsstörung vorliegt
#define IS_ERROR (pause>2500 || pause<700 || puls>250 || puls<50)

int main(void)
{
    init_timer(void);

    // Beide Messinstrumente kurz auf Vollausschlag
    OCR0A=240;
    OCR0B=240;
    _delay_ms(10000);
    OCR0A=0;
    OCR0B=0;

error:
    STATUS_LED_OFF;
    empfangene_minute=255;
    empfangene_stunde=255;

    while(1)
    {
        // warte auf die Sekunde 0
        do
        {
            empfange();
            if (IS_ERROR)
            {
                goto error;
            }
        }
        while (!IS_START);

        // Die zuvor empfangene Zeit anzeigen
        // denn sie wird in der 0. Sekunde gültig
        if (empfangene_stunde<24 && empfangene_minute<60)
        {
            stunden=empfangene_stunde;
            minuten=empfangene_minute;
            sekunden=0;
            anzeigen();
        }

        // überspringe Sekunde 1-20
        for (int i=1; i<=20; i++)
        {

```

```

        empfange();
        if (IS_ERROR)
        {
            goto error;
        }
    }

    // Empfange die Minuten als BCD Zahl (7 Bits)
    uint8_t tmp=0;
    uint8_t parity=0;
    for (int i=0; i<7; i++)
    {
        empfange();
        if (IS_ERROR)
        {
            goto error;
        }
        else if (IS_HIGH)
        {
            tmp=(tmp>>1)|64;
            parity++;
        }
        else
        {
            tmp=(tmp>>1);
        }
    }

    // Empfange und prüfe die Parität
    {
        empfange();
        if (IS_ERROR)
        {
            goto error;
        }
        else if (IS_HIGH)
        {
            parity++;
        }
        if (parity & 1)
        {
            goto error;
        }
    }
    // Die Minuten wurden erfolgreich empfangen

    // Zahl von BCD nach Binär umrechnen
    // Zehner mal 10 + Einer.
    empfangene_minute=(tmp>>4)*10+(tmp & 0x0F);

    // Empfange die Stunden als BCD Zahl (6 Bits)
    tmp=0;
    parity=0;
    for (int i=0; i<6; i++)
    {
        empfange();
        if (IS_ERROR)
        {
            goto error;
        }
        else if (IS_HIGH)
        {

```

```

        tmp=(tmp>>1)|32;
        parity++;
    }
    else
    {
        tmp=(tmp>>1);
    }
}

// Empfange und prüfe die Parität
{
    empfange();
    if (IS_ERROR)
    {
        goto error;
    }
    else if (IS_HIGH)
    {
        parity++;
    }
    if (parity & 1)
    {
        goto error;
    }
}
// Die Stunden wurden erfolgreich empfangen

// Zahl von BCD nach Binär umrechnen
// Zehner mal 10 + Einer.
empfangene_stunde=(tmp>>4)*10+(tmp & 0x0F);

// Die Bits 36 - 59 werden nicht verwendet

STATUS_LED_ON;
}
}

```

Falls du den ATtiny25, 45 oder 85 verwendest, ersetze „TIMSK0“ durch „TIMSK“.

Vergleiche den Programmcode der Funktion init_timer() mit dem Datenblatt des ATtiny, um zu verstehen, was die einzelnen Bits in den Registern bewirken.

Der Timer erzeugt das PWM Signal zur Ansteuerung der Einbauinstrumente. Außerdem berechnet die Interrupt-Routine des Timers die ungefähre Uhrzeit. Wenn einmal der Empfang ausfällt, läuft die Uhr einfach weiter – allerdings recht ungenau, da das Taktignal von einem R/C Oszillator stammt.

Die Funktion empfange() Empfängt einen Impuls vom DCF1 Modul und liefert die Dauer des Impulses und die Dauer der vorherigen Pause zurück. Diese wird vom Hauptprogramm wiederholt aufgerufen, um die einzelnen Bits des DCF77 Signals zu empfangen und zu dekodieren.

In diesem Programm habe ich ausnahmsweise den allgemein unbeliebten Goto Befehl verwendet, da er hier im Rahmen der Fehlerbehandlung sehr praktisch war. Da man mit Goto aber auch sehr schnell total unübersichtlichen Quelltext erzeugen kann, sollte man ihn nur selten und mit Bedacht einsetzen.

12.6 Inbetriebnahme

Nach dem Einsticken des Netzteils passiert folgendes:

1. Beide Messinstrumente schlagen zunächst 10 Sekunden lang voll aus. Du kannst diesen Moment nutzen, um die Trimmpotis richtig einzustellen. Die Nadeln sollen auf das Ende der Skalen zeigen (entsprechend 24 Stunden und 60 Minuten).

2. Die LED blinkt im Sekundentakt, damit zeigt sie das empfangene DCF-Signal an. Diese Anzeige ist hilfreich, um einen Platz mit gutem Empfang zu suchen und die Antenne richtig auszurichten.
3. Es wird auf das Start-Signal gewartet, das ist die Lücke in der 59. Sekunde.
4. Die Minuten und die Stunden werden empfangen, das dauert 35 Sekunden.
5. Ab jetzt pulsiert die LED (hell/dunkel) im Sekundentakt, um zu signalisieren, dass der Empfang erfolgreich war.
6. Nach Ablauf der Minute wird die Anzeige aktualisiert. Ab jetzt gilt die zuvor empfangene Zeit.
7. Sollte eine Empfangsstörung erkannt werden, springt der Programmablauf wieder zu Punkt 2. Die LED blinkt dann wieder anstatt zu pulsieren.

Wenn du möchtest, kannst du auch zwei separate LED's verwenden:

- PB2 = Signal (gelb)
- PB3 = Empfang in Ordnung (grün)

Da der interne R/C Oszillatior des Mikrocontrollers für eine Uhr viel zu ungenau ist, hängt die Schaltung von einem guten DCF77 Empfang ab. Du kannst sie daher nicht an Orten mit schlechtem Empfang verwenden.

12.7 Fehlerbehebung

Bei gutem Empfang sollte die Uhr spätestens 2 Minuten nach den Einschalten die aktuelle Uhrzeit anzeigen. Falls du Schwierigkeiten hast, das DCF77 Signal zu empfangen, versuche folgendes:

- Mache die Leitungen zu den Einbauinstrumenten nicht länger als nötig. Sie strahlen elektromagnetische Felder ab, die eventuell den Empfang beeinträchtigen.
- Stelle die Uhr zwei Meter entfernt von anderen elektrischen Geräten (insbesondere Netzteilen und Smartphones) auf.
- Betreibe die Uhr probeweise mit Batterien, so findest du heraus, ob das Netzteil die Problemursache ist.

13 Experimente mit WLAN

Obwohl ich WLAN Anwendungen super spannend finde, habe ich doch einige Jahre gezögert, entsprechende Experimente zu dokumentieren. Denn im Handel werden alle paar Monate andere Netzwerk-Adapter angeboten, die immer wieder völlig anders zu benutzen waren. Meine Anleitungen wären mangels Verfügbarkeit der Bauteile rasch wertlos geworden.

Nun haben die Chinesen allerdings den unfassbar billigen ESP8266 Chip und damit aufgebaute WLAN Module auf den Markt geworfen. Anfangs funktionierten sie noch sehr schlecht, aber inzwischen ist die Firmware in brauchbarem Zustand.

Damit hat sich die Situation geändert – ich habe mich nun endlich doch dazu entschlossen, eine Reihe spannender WLAN Experimente aufzuschreiben.

Doch zuerst möchte ich mit einer Warnung anfangen, die ich sehr ernst nehme: **Halte gefährliche Sachen vom Internet fern!**

Das soll konkret bedeuten, dass du zum Beispiel nicht die Steuerung deiner Heizungsanlage umbauen sollst, um sie über das Internet erreichbar zu machen. Denn im Internet wimmelt es von Hackern, die Spaß daran haben, anderer Leute Systeme zu manipulieren – und einige von denen nehmen keine Rücksicht auf die Folgen. Das Hacker mit Leichtigkeit einfache Passwort-Abfragen umgehen, brauche ich dir sicher nicht zu erklären.

Um ein WLAN Gerät übers Internet erreichbar zu machen, braucht man nur ein paar wenige Einstellungen im Internet-Router vorzunehmen. Wenn du das machen willst, dann schau dazu in die Bedienungsanleitung deines Routers.

Für das erste Experiment brauchst du:

- 1 Steckbrett und Jumper Kabel
- 1 5 V Netzteil das mindestens 500 mA liefern kann
- 1 Spannungsregler LF33CV
- 1 Kondensator 100 μ F 16 V
- 1 Kondensator 100 nF
- 1 Kondensator 2,2 μ F
- 1 WLAN Modul Typ „ESP-01“ mit AT Firmware
- 1 USB-UART Kabel mit 3,3 V Pegeln
- 2 Widerstände 1 k Ω $\frac{1}{4}$ Watt
- 1 Widerstand 47 Ohm $\frac{1}{4}$ Watt

Für die weiteren WLAN Experimente kommt später noch dazu:

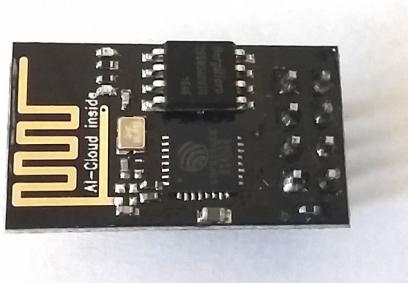
- 1 Arduino Nano
- 1 USB Kabel für den Arduino
- 1 Diode 1N4148
- 1 Zener-Diode 3,6 V 0,5 W
- 1 Widerstand 1 k Ω
- 2 Widerstände 2,2 k Ω $\frac{1}{4}$ Watt

Achte beim Einkaufen darauf, dass das WLAN Modul mit der „AT Firmware“ ausgestattet ist. Das ist zwar der Normalfall, doch es gibt auch Händler, die sie mit anderer Firmware verkaufen (zum Beispiel „NODEMCU“ oder „LUA“).

13.1 Das ESP-01 Modul erforschen

13.1.1 Anschlussbelegung

Lege das ESP-01 Modul so vor dich hin, dass du auf die Bauteile schauen kannst. Ganz links ist die Antenne, ganz rechts die Stifteleiste.



Dies ist die Anschlussbelegung:

	Pin		
RxD	1	2	VCC 3,3 V
	3	4	Reset
	5	6	CH_PD
GND	7	8	TxD

Wenn du den /Reset Eingang kurz mit GND verbindest, startet die Firmware des WLAN Moduls neu. Ansonsten sollte man ihn mit VCC verbinden.

Der CH_PD Anschluss muss normalerweise mit VCC verbunden werden. Wenn er mit GND verbunden wird, geht der Chip in den Power-Down Modus. Dann ist er allerdings weder seriell noch über WLAN ansprechbar. Wenn der Pin danach wieder mit VCC verbunden wird, startet die Firmware des WLAN Moduls neu.

RxD und TxD sind die beiden seriellen Kommunikationsleitungen, mit denen das WLAN Modul gesteuert wird.

Die Anschlüsse 3 und 5 habe ich bewusst nicht beschriftet, weil du sie nicht benutzen sollst. Sie haben nur eine Bedeutung, wenn man eine andere Firmware in das Modul installiert, was wir im Rahmen dieses Buches nicht tun werden.

Alle Anschlüsse vertragen maximal 3,6 V. Dies musst du unbedingt beachten, wenn du das Modul mit einem Computer oder Mikrocontroller verbindest.

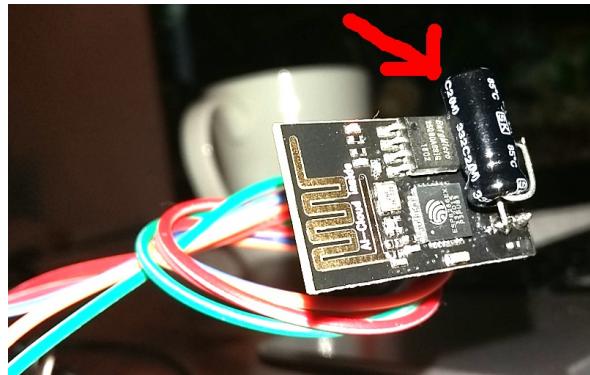
Falls du lieber eines der größeren ESP-Module (z.B. ESP-12) verwenden möchtest, musst du zusätzlich einen 2,2 k Ohm Widerstand zwischen GPIO15 und GND löten!

13.1.2 Stromversorgung

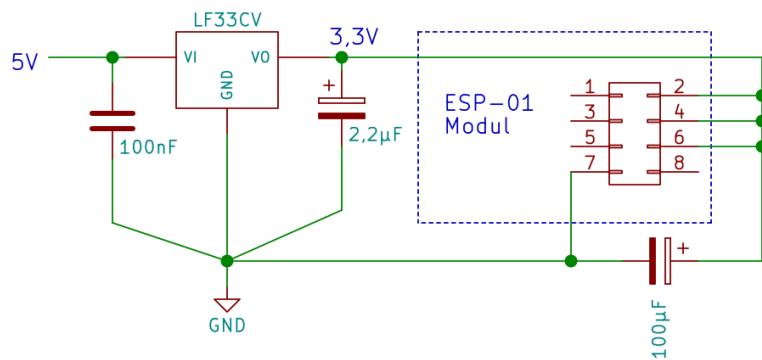
Das ESP-01 Modul funktioniert mit 2,5 bis 3,6 Volt. Seine Stromaufnahme schwankt zwischen 2 und 100 mA, aber für sehr kurze Momente braucht der Chip 400 mA. Es ist daher wichtig, einen Spannungsregler auszuwählen, die diese extremen Strom-Schwankungen schnell ausregeln kann. Der LF33CV ist dazu geeignet.

Zur weiteren Stabilisierung, löte einen 100 μ F Kondensator direkt auf die Stifteleiste des ESP-01 Moduls, und zwar an die Anschlüsse VCC und GND. Dieser Kondensator unterstützt den Spannungsregler und gleicht Verluste in den Zuleitungen zum ESP-01 Modul aus. Ich rate dringend dazu, niemals auf diesen Kondensator zu verzichten. Ohne den Kondensator wird das Modul nicht zuverlässig arbeiten.

In dem folgenden Foto habe ich den Kondensator mit einem Pfeil markiert.



Wir werden ein Steckernetzteil mit 5-6 V verwenden, und einen LF33CV Spannungsregler, der daraus stabile 3,3 V macht. Damit versorgen wir das WLAN Modul.



Die beiden linken Kondensatoren sollen ganz nahe zum Spannungsregler platziert werden. Die Oberfläche des Spannungsreglers genügt, um die Wärme abzuleiten. Es ist kein Kühlkörper nötig.

Schließe das WLAN Modul an den Ausgang des Spannungsreglers an.

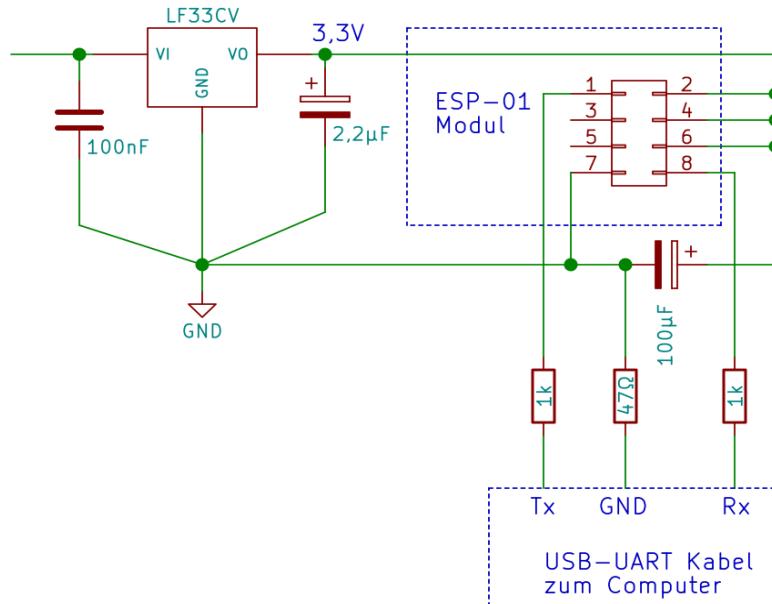
Kontrolliere nochmal, ob alle Leitungen richtig angeschlossen sind und ob das Netzteil richtig herum gepolt ist. Denn bei falscher Stromversorgung geht das ESP Modul sofort kaputt. Es ist wesentlich empfindlicher, als AVR Mikrocontroller.

Wenn du jetzt das Netzteil in die Steckdose steckst, sollte die rote Leuchtdiode auf dem Modul leuchten und die blaue Leuchtdiode sollte kurz flackern. Daran erkennst du, dass das ESP-01 Modul startet. Wenn die blaue LED nicht flackert, ist es defekt oder falsch angeschlossen.

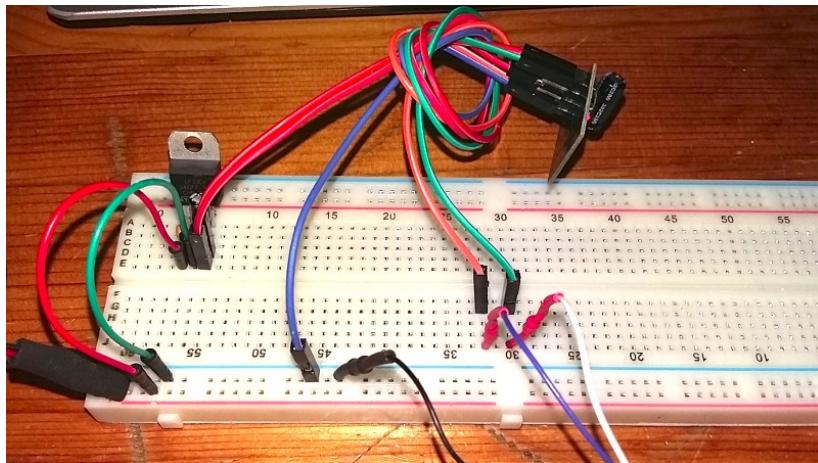
Die blaue LED ist übrigens mit der TxD Leitung des Moduls verbunden. Sie flackert, wenn das Modul etwas zum Computer sendet.

13.1.3 Serielle Verbindung zum PC

Als nächstes sollst du das ESP-01 Modul über ein USB-UART Kabel mit deinem Computer verbinden.



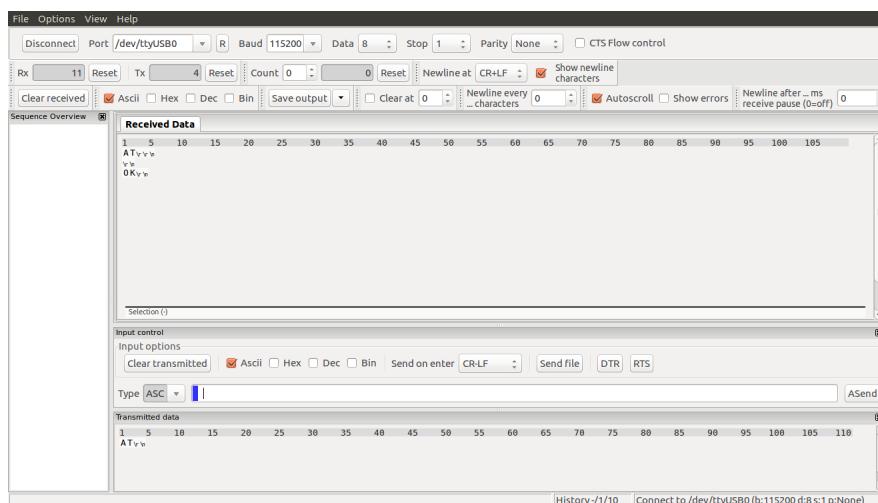
Der 47Ω Widerstand ist optional. Ich habe ihn ans Ende meines USB-UART Kabels gelötet, damit es mir nicht abbrennt, wenn ich die GND Leitung versehentlich an die Spannungsversorgung stecke. Mein Aufbau sieht so aus:



Bei mir sind die Widerstände unter dem Schrumpfschlauch an den Enden des USB-UART Kabels verborgen. Sie beschützen den Computer vor Vertauschung der Leitungen und vor Überspannung.

Wir werden nun die WLAN Parameter einstellen, damit sich das ESP-01 Modul mit deinem Netzwerk verbindet. Auf deinem Computer sollst du dazu ein Terminal-Programm starten, zum Beispiel Cutecom oder „Hammer Terminal“. Stelle das Programm so ein, dass es beim Drücken der Eingabetaste (Enter-Taste, Return-Taste) einen Zeilenumbruch im Format CRLF (\r\n) sendet.

Öffne dann eine Verbindung auf dem virtuellen seriellen Port deines USB-UART Kabels mit 115200 Baud, 8 Datenbits, 1 Stopbit, keine Parität und ohne Flusskontrolle (Handshake). Beim „Hammer Terminal“ unter Linux sieht das so aus:



Sende „AT“, dann antwortet das WLAN Modul mit „OK“. Wenn das klappt, hast du den Beweis, dass die serielle Kommunikation mit dem Computer funktioniert.

Falls es nicht klappt, versuche folgendes:

- Der TxD Ausgang vom WLAN Modul gehört an den RxD Eingang des USB-UART Adapterkabels. Und der RxD Ausgang vom WLAN Modul gehört an den TxD Eingang des USB-UART Adapterkabels. Vielleicht hast du die beiden Leitungen vertauscht.
- Möglicherweise kommuniziert dein WLAN Modul mit 9600 oder 57600 Baud anstatt 115200 Baud. Das musst du dann auch später bei der weiteren Programmierung berücksichtigen.
- Hast du den richtigen seriellen Port verwendet? Unter Windows kannst du das im Gerätetool prüfen. Unter Linux hilft der Befehl „dmesg“ unmittelbar nach dem Einstecken des USB Kabels.

13.1.4 WLAN Netzwerk konfigurieren

Die Befehle, mit denen man das ESP-01 Modul konfiguriert, fangen alle mit „AT“ an. Deswegen heißt es „AT-Firmware“. Wenn ein Befehl erfolgreich verarbeitet wurde, antwortet das Modul mit „OK“.

Als erstes stellen wir den Betriebsmodus 1 ein, damit das Modul sich mit einem bestehendem WLAN Netz verbindet.

```
AT+CWMODE=1
OK
AT+RST
OK
... Informationen zur Firmware Version
ready
```

Warte eine Minute, dann schau nach, welche WLAN Router das Modul gefunden hat:

```
AT+CWLAP
+CWLAP: (4, "EasyBox-4D2D18", -72, "18:83:bf:4d:2d:b2", 2, -46)
+CWLAP: (4, "UPC2827302", -63, "88:f7:c7:52:40:9d", 6, -22)
+CWLAP: (0, "Unitymedia WifiSpot", -64, "8a:f7:c7:52:40:9f", 6, -22)
+CWLAP: (3, "Muschikatze", -45, "5c:49:79:2d:5b:cd", 7, -4)
OK
```

Mein WLAN Router heißt „Muschikatze“, so habe ich es in meiner Fritz-Box eingerichtet. Jetzt konfigurieren wir, mit welchem WLAN Router sich das Modul verbinden soll. Dazu müssen wir das WLAN Passwort kennen.

```
AT+CWJAP="Muschikatze", "supergeheim"  
WIFI CONNECTED  
WIFI GOT IP  
OK
```

Das WLAN ist jetzt fertig konfiguriert. Wenn du das Netzteil aus steckst und wieder einsteckst, wird das ESP-01 Modul sich automatisch mit deinem WLAN Router verbinden. Du kannst das so kontrollieren:

```
AT+CWJAP?  
+CWJAP:"Muschikatze", "5c:49:79:2d:5b:cd", 7, -60  
OK  
AT+CIFSR  
+CIFSR:STAIP, "192.168.2.52"  
+CIFSR:STAMAC, "5c:cf:7f:8b:a9:f1"  
OK
```

Nun weißt du, dass das WLAN Modul mit deinem WLAN Router verbunden ist, und du kennst auch seine IP-Adresse. Im obigen Beispiel ist es die 192.168.2.52. Deines hat sicher eine andere IP-Adresse.

13.1.5 Verbindungstest mit HTTP

Jetzt wollen wir mal sehen, ob wir über WLAN Netz eine IP-Verbindung zu dem ESP-01 Modul aufbauen können. Dazu musst du auf dem Modul den IP-Server starten.

```
AT+CIPMUX=1  
OK  
AT+CIPSERVER=1, 80  
OK
```

Starte nun einen Webbrowser auf einem Smartphone, Tablet oder irgendeinem Computer. Dann gibst du in die Adressleiste die IP-Adresse deines WLAN Moduls ein: <http://192.168.2.52/>

Der Webbrowser wird anzeigen, dass er auf den Empfang einer Webseite wartet. Da wir aber noch keinen Webserver programmiert haben, tut sich hier nicht viel.

Aber im Terminal-Programm gibt es etwas interessantes zu sehen:

```
0,CONNECT  
+IPD,0,317:GET / HTTP/1.1  
Host: 192.168.2.52  
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:49.0) Gecko/20100101  
Firefox/49.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8  
Accept-Language: de,en;q=0.5  
Accept-Encoding: gzip, deflate  
Connection: keep-alive  
Upgrade-Insecure-Requests: 1
```

Die Meldung „0,CONNECT“ bedeutet, dass das WLAN Modul eine Verbindung auf Kanal 0 angenommen hat. Das Modul kann je nach Version bis zu vier oder fünf Verbindungen gleichzeitig annehmen, die dann von 0 bis 4 durchnummeriert werden.

Die Meldung „+IPD,0,317:“ bedeutet, dass das WLAN Modul auf dem Kanal 0 Daten empfangen hat, und zwar genau 317 Bytes.

Die Zeilen darunter geben exakt die Daten wieder, die der Web-Browser gesendet hat. Wenn dir das nicht bekannt vorkommt, dann lies das Kapitel zu Webservern im Band zwei dieser Buchreihe. Dort wird das HTTP Protokoll im Detail erklärt. Wir werden jetzt eine Menge mit diesem Kommunikationsprotokoll arbeiten, weil es in Kombination mit Web-Browsern gar nicht anders geht.

Ich zeige dir nun, wie du eine Antwort an den Webbrowser senden kannst. Falls dein Webbrowser inzwischen die Geduld verloren hat, weil er schon so lange warten musste, wiederhole die Eingabe in der Adressleiste nochmal oder klicke auf den entsprechenden Knopf im Webbrowser, so dass er die Seite erneut lädt.

Beantworte dann im Terminal-Programm die Anfrage des Webrowsers folgendermaßen:

```
AT+CIPSEND=0, 31
HTTP/1.1 200 OK
← Hier eine Leerzeile senden!
Hallihallo
SEND OK

AT+CIPCLOSE=0
0, CLOSED
OK
```

Du musst die Nummer des Kanals verwenden, die vorher beim Verbindungsaufbau gemeldet wurde. Die Leerzeile bei der obigen Eingabe ist beabsichtigt!

Schau auf den Web-Browser: Er hat eine Webseite empfangen, und in der steht „Hallihallo“.

Mit dem „AT+CIPSEND“ befiehlst du dem WLAN Modul, etwas über den Verbindungskanal 0 zu senden – also an den Web-Browser. Und zwar genau 31 Bytes. Die nächsten drei Zeilen sind genau diese 31 Bytes oder Zeichen. Das ist eine HTTP Response, die der Webbrowser versteht und anzeigt.

Alle ESP-Module können 1024 Bytes am Stück senden, manche Firmware Versionen können auch mehr.

Anschließend meldet das WLAN Modul „SEND OK“. Danach bewirkt der Befehl „AT+CIPCLOSE“, dass die angegebene Verbindung geschlossen wird. Dies signalisiert dem Web-Browser, dass die Datenübertragung beendet ist.

Nun versuchen wir die umgekehrte Richtung. Wir wollen eine ganz normale Webseite abrufen:

```
AT+CIPSTART=0, "TCP", "stefanfrings.de", 80
OK

AT+CIPSEND=0, 51
GET /index.html HTTP/1.1
Host: stefanfrings.de
← Hier eine Leerzeile senden!
+IPD, 0, 1460:HTTP/1.1 200 OK
Date: Sat, 08 Apr 2017 14:26:41 GMT
Content-Type: text/html
Content-Length: 1892
Connection: keep-alive
Server: Apache
Last-Modified: Sat, 08 Apr 2017 09:49:52 GMT
ETag: "764-54ca4a9e7fae0"
Accept-Ranges: bytes

<!DOCTYPE HTML PUBLIC ...<title>Willkommen bei Stefan Frings</title>...
```

Geschafft! Du hast soeben die Startseite meiner Homepage abgerufen. Mit dem „AT+CIPSTART“ Befehl hast du eine Verbindung zum Webserver „stefanfrings.de“ auf Port 80 aufgebaut. Anschließend hast du einen HTTP GET Request gesendet, der die Startseite abruft.

Weiterführende Informationen zu ESP-Modulen findest du auf meiner Homepage: <http://stefanfrings.de/esp8266/index.html>

Kannst du dir schon vorstellen, einen eigenen kleinen Webserver mit AVR Mikrocontroller zu programmieren? Ich werde dir dabei in den nächsten Kapiteln helfen.

13.2 Das Internet der Dinge

Wenn Elektronik mit dem Internet verbunden wird, spricht man vom „Internet der Dinge“ oder „Embedded Webserver“. Das sind kleine Server mit eingeschränkter Funktionalität. Sie können viel weniger, als die großen Webserver die man im Internet für gewöhnlich benutzt. Da sie nicht universell sind, sondern einem ganz bestimmten Zweck dienen, genügen dazu schon kleine Mikrocontroller wie der ATmega328P des Arduino Nano Boardes.

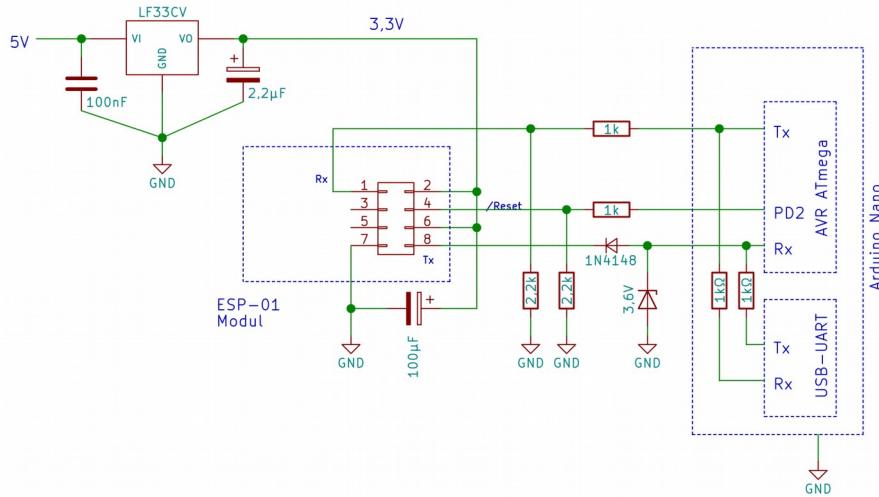
Zwei Hürden muss man dabei allerdings überwinden:

- Das Arduino Nano Modul läuft mit 5V, das ESP Modul verträgt aber nur 3,3 V.
- Das Arduino Nano Modul hat nur eine serielle Schnittstelle, und die ist schon mit der USB-UART Schnittstelle verbunden.

Zur Lösung könnte man natürlich einfach einen größeren AVR Mikrocontroller mit zwei seriellen Ports verwenden und ihn mit 3,3 V betreiben. Aber das ist mir für ein Lehrbuch zu einfach. Ich zeige dir, wie du diese beiden Module trotz der Schwierigkeiten zusammen bekommst.

13.2.1 ESP-01 an Arduino Nano

Entferne das USB-UART Kabel und schließe stattdessen deinen Arduino Nano gemäß folgendem Schaltplan an das WLAN Modul an:



Vergiss beim Aufbauen nicht, die GND Leitung vom WLAN Modul mit dem Arduino Modul zu verbinden!

Eventuell kommst du auf die Idee, das Netzteil weg zu lassen und beide Module über das USB Kabel mit Strom zu versorgen. Du könntest sogar erwägen, den 3,3 V Spannungsregler einzusparen, weil das Arduino Modul ja schon einen 3,3 V Ausgang hat.

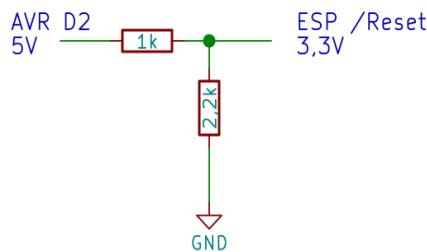
Mach das beides nicht! Denn die Stromaufnahme des WLAN Moduls ist zu hoch. Bedenke, dass es kurzzeitig bis zu 500 mA aufnimmt. Sowohl der 5 V Ausgang als auch der 3,3 V Ausgang des Arduino Moduls darf nicht so hoch belastet werden.

Der serielle Anschluss des AVR Mikrocontrollers ist sowohl mit dem WLAN Modul als auch mit dem USB-UART verbunden. Dadurch kann der Mikrocontroller über USB programmiert werden, während das WLAN Modul deaktiviert ist. Wenn dein Programm startet, schaltet es das WLAN Modul ein und kommuniziert mit ihm.

In den Folgenden Absätzen beschreibe ich im Detail, wie Verbindung zwischen den drei Mikrochips funktioniert.

Reset Leitung

Die Reset Leitung dient dazu, das WLAN Modul neu zu starten. Sie stellt außerdem sicher, dass das WLAN Modul erst startet, nachdem dein Programm auf den AVR hochgeladen ist und läuft.



Da der ESP Chip keine 5 V verträgt, werden hier zwei Widerstände zu einem sogenannten Spannungsteiler kombiniert. Wenn der Ausgang des AVR auf High Pegel liegt, liefert er 5 Volt. Durch die Widerstände wird die Spannung jedoch auf annähernd 3,3 Volt reduziert. Du kannst das so nachrechnen:

$$\frac{(5 \text{ V})}{(1000 \Omega + 2200 \Omega)} = 1,6 \text{ mA} \quad \text{und} \quad 1,6 \text{ mA} * 2200 \Omega = 3,4 \text{ V}$$

Damit lässt sich der Reset Eingang des WLAN Moduls gut ansteuern.

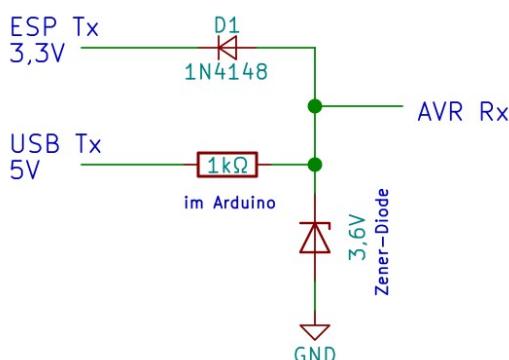
Solange auf dem AVR Mikrocontroller noch kein Programm läuft, liefert er an seinem D2 Ausgang gar kein Signal. In diesem Fall zieht der 2,2 kΩ die Reset Leitung auf Low, dadurch bleibt das WLAN Modul deaktiviert. Er wenn dein Programm läuft, schaltet es das WLAN Modul ein, indem es auf D2 einen High Pegel ausgibt.

Serielle Kommunikation Teil 1

Für die Serielle Kommunikation vom AVR Mikrocontroller (Tx) zum ESP Modul (Rx) wird die gleiche Methode mit dem Spannungsteiler angewendet.

Serielle Kommunikation Teil 2

Für die umgekehrte Richtung wird es etwas komplizierter. Sowohl der USB-UART Chip als auch der ESP Chip des WLAN Moduls können Daten an den AVR Mikrocontroller senden – aber nicht gleichzeitig.



In Ruhelage liefern beide Mikrochips auf der linken Seite einen High Pegel. Beim ESP-Chip sind das 3,3 V und beim USB-UART sind es 5 V. Für den AVR Mikrocontroller ist das In Ordnung, denn alles ab 3 V aufwärts akzeptiert er als gültigen High Pegel.

Wenn der USB-UART kommuniziert, sendet er Low-Impulse aus. Diese erreichen den Eingang des AVR Mikrocontrollers durch den 1 kΩ Widerstand. Dieser Schaltungsteil ist durch das Arduino Board so vorgegeben, der Widerstand ist dort bereits vorhanden.

Wenn der ESP Chip kommuniziert, sendet er ebenfalls Low-Impulse aus. Dabei zieht er den Eingang des AVR über die Diode D1 herunter. Auch hier fallen wieder 0,7 V an der Diode ab, so dass beim AVR tatsächlich 0,7 V ankommen. Das ist Ok, denn der AVR würde sogar 1,2 V noch als gültigen Low Pegel akzeptieren.

Wenn der USB-UART High Pegel (5 V) ausgibt, dann begrenzt die Zener-Diode die Spannung auf 3,6 Volt. Das ist wichtig, denn der ESP verträgt die 5 V nicht. Neben Spannungsteilern sind solche Kombinationen aus Widerstand und Zener-Diode ein beliebtes Mittel, Signalspannungen zu begrenzen.

13.2.2 Inbetriebnahme

Stecke das 5 V Netzteil in die Steckdose. Am WLAN Modul wird die rote LED an gehen, aber die blaue bleibt dunkel. Daran erkennst du, dass die Firmware des WLAN Moduls nicht startet.

Das ist so beabsichtigt. Die Reset-Leitung wird durch den 2,2 kΩ zähneiste auf Low gehalten. Das WLAN Modul bleibt passiv, so dass wir bereit sind, unser eigenes Programm per USB Anschluss auf das Arduino Modul zu übertragen.

Jetzt musst du also ein Programm schreiben. Beginne mit der „Hello World“ Vorlage. Stelle im Makefile den Programmieradapter, den Mikrocontroller und die Taktfrequenz ein:

```
AVRDUDE_HW = -c arduino -P /dev/ttyUSB0 -b 57600
PRG = Arduinowlan
MCU = atmega328p
F_CPU = 16000000
```

Lösche die Datei hardware.h und die entsprechende include-Zeile in main.c. In der Datei driver/serialconsole.h nimmst du bitte folgende Einstellungen vor:

```
#define SERIAL_BITRATE 115200
#define USE_SERIAL0
#define TERMINAL_MODE 1
#define SERIAL_ECHO 0
#define SERIAL_INPUT_BUFFER_SIZE 100
```

Jetzt bist du bereit, das Hauptprogramm zu schreiben. Wir fangen ganz klein an, indem wir den Ausgang PD2 auf High schalten. Dadurch wird das Reset-Signal des WLAN Moduls beendet.

```
int main(void)
{
    initSerialConsole();
    initSystemTimer();

    DDRD |= (1<<PD2);
    PORTD |= (1<<PD2);
}
```

Compiliere das Programm durch Eingabe von: **make clean code**

Du wirst eine Warnung erhalten, dass die Baudrate nicht stimmt. Das liegt daran, dass der 16 Mhz Keramik-Resonator des Arduino Moduls zusammen mit dem Takt-Teiler des seriellen Portes keine exakten 115200 Baud ergibt. Die Arduino Macher hätten besser einen 14,7 Mhz Quarz verwenden sollen. Aber damit müssen wir jetzt leben. Meine Experimente haben ergeben, dass die ganze Schaltung trotzdem funktioniert.

Übertrage das Programm in den Mikrocontroller und beobachte dabei die blaue LED des WLAN Moduls: **sudo make program** (bzw. ohne sudo im Fall von Windows). Etwa zwei Sekunden nach dem Flashen wird die blaue LED flackern. Das heißt, die Firmware des WLAN Moduls hat gestartet.

13.2.3 WLAN Initialisierung

Damit du eine Netzwerk-Verbindung zum WLAN Modul aufbauen kannst, musst du noch zwei Befehle absetzen. Also erweiterst du das Programm entsprechend:

```
int main(void)
{
    initSerialConsole();
    initSystemTimer();

    // Wlan Modul starten
    DDRD |= (1<<PD2);
    PORTD |= (1<<PD2);
    _delay_ms(4000);

    // IP-Server starten
    puts_P(PSTR("AT+CIPMUX=1"));
    _delay_ms(500);
    puts_P(PSTR("AT+CIPSERVER=1, 80"));
    _delay_ms(500);
}
```

Übertrage das Programm wieder in den Mikrocontroller und warte ein paar Sekunden. Danach kannst du mit dem Web-Browser eine Verbindung aufbauen: <http://192.168.2.52/> (verwende hier die IP Adresse von deinem WLAN Modul).

Wieder wird der Webbrower lange auf eine Antwort warten und letztendlich keine empfangen, da unser Programm noch nicht so weit ausgebaut ist. Aber immerhin kannst du sehen, dass der Verbindungsauftbau klappt. Das ist doch mal ein guter Anfang!

Ich erwähnte weiter oben, dass wir die Kommunikation vom AVR zum WLAN Modul über den USB Anschluss mitlesen können. Das probieren wir jetzt mal aus.

Starte dein Terminal-Programm. Öffne dann eine Verbindung auf dem virtuellen seriellen Port deines USB-UART Kabels mit 115200 Baud, 8 Datenbits, 1 Stopbit, keine Parität und ohne Flusskontrolle (Handshake).

Drücke dann den Reset-Knopf auf dem Arduino Modul. Du wirst folgende Ausgabe erhalten:

```
AT+CIPMUX=1
AT+CIPSERVER=1, 80
```

Das sind genau die beiden Befehle, die dein Programm an das WLAN Modul gesendet hat. Die Antworten des WLAN Moduls kannst du allerdings nicht sehen.

Wenn du das brauchst, könntest du dazu dein USB-UART Kabel an die Tx Leitung des WLAN Moduls anschließen und mit einem zweiten Terminal mitlesen, was dort vor sich geht. Aber wir wollen es mal nicht übertreiben, also lassen wir das vorerst sein.

Das Programm soll so erweitert werden, dass es zumindest eine mini-kleine Webseite erzeugt, damit der Webbrower nicht ewig warten muss. Nimm dir Zeit, das folgende Programmbeispiel zu verstehen. Hier sind eine Menge String-Funktionen drin, die du bisher vermutlich noch nicht verwendet hast.

```

int main(void)
{
    initSerialConsole();
    initSystemTimer();

    // Wlan Modul starten
    DDRD |= (1<<PD2);
    PORTD |= (1<<PD2);
    _delay_ms(4000);

    // IP-Server starten
    puts_P(PSTR("AT+CIPMUX=1"));
    _delay_ms(500);
    puts_P(PSTR("AT+CIPSERVER=1, 80"));
    _delay_ms(500);

    while(1)
    {
        // Warte auf eine Zeile
        char line[100];
        fgets(line,sizeof(line),stdin);

        // Wenn Daten vom Netz empfangen wurden...
        if (strncmp(line,"+IPD,",5)==0)
        {
            // Kanalnummer extrahieren
            char channel=line[5];

            // Wenn ein GET Request empfangen wurde
            char* request=strstr(line,"GET");
            if (request>0)
            {
                // Erzeuge minimale HTTP Response
                printf_P(
                    PSTR("AT+CIPSEND=%c,%d\n"),
                    channel,26);
                _delay_ms(20);
                puts_P(PSTR("HTTP/1.1 200 OK\n\nHallo"));
                _delay_ms(20);
                printf_P(PSTR("AT+CIPCLOSE=%c\n"),channel);
            }
        }
    }
}

```

Der Webserver läuft in einer Endlosschleife. Bei jedem Durchgang empfängt er genau eine Zeile Text vom WLAN Modul. Da kann alles Mögliche drin sein, zum Beispiel Statusmeldungen über die Verbindung, Antworten zu vorherigen AT Befehlen und natürlich auch Daten von den WLAN Verbindungen.

Der Webserver interessiert sich nur HTTP Requests von Web-Browsern, also wird für jede empfange Zeile zuerst geprüft, ob das Daten von einer WLAN Verbindung sind. Diese erkennt das Programm daran, dass die Zeile mit „+IPD,“ beginnt. Dahinter kommt die Kanalnummer (0 bis 3) und dahinter der eigentliche HTTP Request.

HTTP Requests enthalten allerdings nicht nur die eine Zeile mit dem GET Befehl, sondern noch weitere Header-Zeilen. Das Programm ist so geschrieben, dass es alle Zeilen ignoriert, die nicht mit „GET“ beginnen.

Wenn denn ein GET Request erkannt wurde, dann antwortet der Server mit einem OK-Header und dem Text „Hallo“. Dann schließt er die Verbindung.

Die Anzahl der Bytes für den AT+CIPSEND Befehl habe ich so gezählt:

- „HTTP/1.1 200 OK\r\n\r\nHallo“ sind 20 Zeichen und zwei Zeilenumbrüche (\r\n).
- puts_P() hängt selbst noch einen weiteren Zeilenumbruch hinten dran.
- Die Zeilenumbrüche zählen doppelt, weil die serielle Konsole alle Zeilenumbrüche in das Format CRLF (\r\n) umandelt.

Das ergibt zusammen 26 Bytes.

Probiere den Webserver aus. Wenn du im Web-Browser die IP-Adresse deines WLAN Moduls eingibst, wird er nun „Hallo“ empfangen und anzeigen.

13.2.4 Basis-Code für Webserver

Nun bauen wir das Programm ein wenig aus, um daraus eine solide Grundlage für vernünftige Anwendungen zu schaffen.

```
// Sende einen String aus dem RAM auf einem WLAN Kanal.
void send(char channel, const char* data)
{
    if (data==0) {
        return;
    }
    // Zähle die Anzahl der Bytes, wobei \n doppelt
    // gezählt wird
    int bytes=0;
    char* ptr=(char*) data;
    while (*ptr!=0)
    {
        if (*ptr=='\n')
        {
            bytes+=2;
        }
        else
        {
            bytes++;
        }
        ptr++;
    }
    // Sende die Daten auf dem Kanal
    printf_P(PSTR("AT+CIPSEND=%c,%d\r\n"),channel,bytes);
    _delay_ms(20);
    puts(data);
    _delay_ms(20);
}

// Sende einen String aus dem Programmspeicher
// auf einem WLAN Kanal.
void send_P(char channel, const char* data)
{
    if (data==0) {
        return;
    }
    // Zähle die Anzahl der Bytes, wobei \n doppelt
```

```

// gezählt wird
int bytes=0;
char* ptr=(char*)data;
char c=pgm_read_byte(ptr);
while (c!=0)
{
    if (c=='\n')
    {
        bytes+=2;
    }
    else
    {
        bytes++;
    }
    ptr++;
    c=pgm_read_byte(ptr);
}
// Sende die Daten auf dem Kanal
printf_P(PSTR("AT+CIPSEND=%c,%d\n"),channel,bytes);
_delay_ms(20);
puts_P(data);
_delay_ms(20);
}

const char okHeader[] PROGMEM = "HTTP/1.1 200 OK\n\n";

// Beantworte einen HTTP Request
void processRequest(char channel, char* request)
{
    // Erzeuge minimale HTTP Response
    send_P(channel,okHeader);
    send_P(channel,PSTR("Hallo\n"));
}

int main(void)
{
    initSerialConsole();
    initSystemTimer();

    // Wlan Modul starten
    DDRD |= (1<<PD2);
    PORTD |= (1<<PD2);
    _delay_ms(4000);

    // IP-Server starten
    puts_P(PSTR("AT+CIPMUX=1"));
    _delay_ms(500);
    puts_P(PSTR("AT+CIPSERVER=1,80"));
    _delay_ms(500);

    while(1)
    {
        // Warte auf eine Zeile
        char line[100];
        fgets(line,sizeof(line),stdin);

        // Wenn Daten vom Netz empfangen wurden...
        if (strcmp_P(line,PSTR("+IPD,"),5)==0)
        {
            // Kanalnummer extrahieren

```

```

char channel=line[5];

// Wenn ein GET Request empfangen wurde
char* request=strstr_P(line,PSTR("GET"));
if (request>0)
{
    // Beantworte den HTTP Request
    processRequest(channel,request);

    // Schließe die Verbindung
    printf_P(PSTR("AT+CIPCLOSE=%c\n"),channel);
}
}

// Wenn der Konfigurationsbefehl
// über USB empfangen wurde
else if (strncmp_P(line,PSTR("CFGWLAN="),8)==0)
{
    printf_P(PSTR("AT+CWJAP=%s\n"),line+8);
}
}

```

Das Programm macht letztendlich immer noch genau das Gleiche wie vorher. Aber es ist besser Strukturiert und kann nun ordentlich erweitert werden.

Die neue send() Funktionen zählt nun selbst die Anzahl der Bytes. Als Programmierer rufst du sie einfach mit der Kanalnummer und dem zu sendenden Text auf. Während bei send() der Text im RAM liegen muss, ist die Funktion send_P() für Texte gedacht, die im Programmspeicher (Flash) liegen.

Jedes mal, wenn der Webserver einen HTTP GET Request empfangen hat, ruft er die Funktion processRequest() auf, deren Aufgabe es ist, den Request zu beantworten. Bei den nächsten Anwendungen wirst du dich auf genau diese Funktion konzentrieren.

Zu guter Letzt kann man über USB die WLAN Zugangsdaten konfigurieren, indem man ins Terminal-Programm eingibt:

```
CFGWLAN="Muschikatze","supergeheim"
```

Der „CFGWLAN“ Befehl wird vom Programm in einen „AT+CWJAP“ Befehl umgesetzt und dann an das WLAN Modul gesendet. Du wirst den Befehl aber nicht brauchen, weil dein WLAN Modul bereits konfiguriert ist.

13.3 Per WLAN Fernsteuern

13.3.1 Befehle in der URL übermitteln.

Wir werden nun ein Licht per WLAN Fernsteuern. Wir wollen zunächst erreichen, dass der Webserver auf folgende Befehle reagiert:

- <http://192.168.2.52/?licht=an>
- <http://192.168.2.52/?licht=aus>

Als „Licht“ verwenden wir der Einfachheit halber die LED auf dem Arduino Board, die mit Port B5 verbunden ist.

Beginne mit dem Basis-Code aus dem Kapitel Basis-Code für Webserver. Ändere die processRequest() Funktion folgendermaßen:

```
const char okHeader[] PROGMEM = "HTTP/1.1 200 OK\r\n\r\n";
```

```

const char badRequest[] PROGMEM =
    "HTTP/1.1 400 Bad Request\n\nFehlerhafte Anfrage\n";

// Beantworte einen HTTP Request
void processRequest(char channel, char* request)
{
    if (strstr(request,"licht=an"))
    {
        // Licht ein schalten
        DDRB |= (1<<PB5);
        PORTB |= (1<<PB5);
        send_P(channel,okHeader);
        send(channel,"An");
    }
    else if (strstr(request,"licht=aus"))
    {
        // Licht aus schalten
        PORTB &= ~(1<<PB5);
        send_P(channel,okHeader);
        send(channel,"Aus");
    }
    else
    {
        send_P(channel,badRequest);
    }
}

```

Probiere das Programm aus. Du kannst nun ganz simpel das Licht fernsteuern, indem du ein Kommando mit irgendeinem Web-Browser auf irgend einem Gerät im Haushalt sendest:

- <http://192.168.2.52/?licht=an>
- <http://192.168.2.52/?licht=aus>

Versuche mal, ein ungültiges Kommando zu senden. Der Webserver wird darauf mit einem HTTP Fehlercode (400) und dem Text „Fehlerhafte Anfrage“ reagieren.

Verglichen mit der ganzen Vorarbeit war das doch erfrischen einfach, nicht wahr?

13.3.2 Fernsteuern mit Formular

Die ist sicher nicht entgangen, dass der Mikrocontroller noch lange nicht voll ist. Wir haben reichlich Platz frei, um dort ein schön gestaltetes Formular zu hinterlegen. Damit bekommt deine Fernsteuerung eine ansprechende Bediener-Oberfläche.

Das Formular testen wir jedoch zuerst ohne Mikrocontroller, bevor wir es zum C-Programm hinzufügen. Lege auf deinem Computer irgendwo eine Datei mit dem Namen test.html an. Sie soll folgenden Inhalt haben:

```

<html>
  <body>
    <h1>Fernsteuerung f&uuml;r das Licht</h1>
    <form>
      <input type="submit" name="licht" value="an">
      <input type="submit" name="licht" value="aus">
    </form>
  </body>
</html>

```

Öffne diese Datei mit einem Web-Browser. So kannst du kontrollieren, ob das Formular gut aussieht und ob es so funktioniert, wie geplant.



Wenn du magst, darfst du das Formular nach allen Regeln der Kunst noch verschönern. Dazu musst du dich dann allerdings mit Web-Design und HTML Entwicklung auskennen.

Das Formular funktioniert und es erzeugt die gewünschten Zeichenketten in der Adressleiste, wenn man auf die Knöpfe klickt. Jetzt übertragen wir es in das C-Programm. Die doppelten Anführungsstriche musst du dabei entweder durch einfache ersetzen, oder jeweils ein „\“ davor schreiben.

```

const char okHeader[] PROGMEM =
"HTTP/1.1 200 OK\nContent-Type=text/html\n\n";

const char lightForm[] PROGMEM = "<html> \
<body> \
<h1>Fernsteuerung f&uuml;r das Licht</h1> \
<form> \
<input type='submit' name='licht' value='an'> \
<input type='submit' name='licht' value='aus'> \
</form> \
</body> \
</html>";

// Beantworte einen HTTP Request
void processRequest(char channel, char* request)
{
    if (strstr(request,"licht=an"))
    {
        // Licht ein schalten
        DDRB |= (1<<PB5);
        PORTB |= (1<<PB5);
    }
    else if (strstr(request,"licht=aus"))
    {
        // Licht aus schalten
        PORTB &= ~(1<<PB5);
    }

    // Zeige das Formular an
    send_P(channel,okHeader);
    send_P(channel,lightForm);
}

```

Das Programm wurde nun so umgeschrieben, dass es zuerst auf die Befehle licht=an oder licht=aus reagiert. Unbekannte Befehle werden einfach ignoriert. Danach liefert der Webserver das Formular aus.

Weil das Formular nicht einfacher Text ist, musste der HTTP Header (okHeader) angepasst werden. Er teilt dem Webbrower nun mit, dass die Antwort eine Dokument im HTML Format enthält.

13.3.3 Status von Etwas Anzeigen

Wäre es nicht schön, wenn das Formular anzeigen würde, ob das Licht gerade an oder aus ist? Kein Problem, das geht zum Beispiel so:

```
const char okHeader[] PROGMEM =
    "HTTP/1.1 200 OK\nContent-Type=text/html\n\n";

const char lightForm1[] PROGMEM = "<html> \
<body> \
    <h1>Fernsteuerung f&uuml;r das Licht</h1> \
    Das Licht ist ";

const char lightForm2[] PROGMEM = "<form> \
    <input type='submit' name='licht' value='an'> \
    <input type='submit' name='licht' value='aus'> \
</form> \
</body> \
</html>";

// Beantworte einen HTTP Request
void processRequest(char channel, char* request)
{
    if (strstr(request,"licht=an"))
    {
        // Licht ein schalten
        DDRB |= (1<<PB5);
        PORTB |= (1<<PB5);
    }
    else if (strstr(request,"licht=aus"))
    {
        // Licht aus schalten
        PORTB &= ~(1<<PB5);
    }

    // Zeige das Formular an
    send_P(channel,okHeader);
    send_P(channel,lightForm1);

    // Wenn das Licht an ist
    if (PORTB & (1<<PB5))
    {
        send(channel,"an");
    }
    else
    {
        send(channel,"aus");
    }

    // Sende den Rest des Formulars
    send_P(channel,lightForm2);
}
```

Ich habe das Formular einfach in zwei Stücke zerlegt und dazwischen fügt das Programm den Text „an“ oder „aus“ ein. Das Resultat sieht so aus:



Du kannst diese Idee beliebig weiter spinnen. Dein Arduino hat noch eine Menge I/O Pins frei, die du alle mit entsprechenden Knöpfen belegen kannst.

Und du kannst auch den Status von ganz anderen Dingen anzeigen. Wie wäre es zum Beispiel, die Raumtemperatur zu messen und anzuzeigen? Lass deiner Phantasie dabei freien lauf!

14 Wie geht es weiter?

Wenn du mit diesem Buch durch bist, schau dir mal meinen Assembler Workshop (http://stefanfrings.de/avr_workshop/index.html) an.

Hier lernst du ein kleines bisschen Assembler Programmierung, was dabei hilfreich ist, die Funktionsweise des Mikrocontroller besser zu verstehen. Ab und zu schauen sich C Programmierer den vom Compiler erzeugten Assembler-Code an, um Probleme zu lösen.

Ich würde dir auch empfehlen, dieses Spielgerät (das in dem Workshop gebaut wird) einfach mal zum Spaß in C zu programmieren. Übersetze das ganze Assembler Programm manuell 1:1 nach C. Dabei wirst du eine Menge lernen.

Kaufe dir einen NiboBee (oder Asuro) Roboter Bausatz, damit kann man spielend lernen.

Auf der Webseite <http://www.elektronik-kompendium.de/> findest du zahlreiche Grundlagen sehr anschaulich erklärt.

Einzelne Bauteile bekommst du gut bei TME.eu und Reichelt. Mikrocontroller-Module findest du günstig bei Amazon. Größere Mengen bekommt man auch gut bei Aliexpress – dauert aber lange.

15 Material-Liste

Mit dem folgenden Material kannst du alle Experimente aus diesem Buch Band 3 nachvollziehen:

- 1 Mikrocontroller ATtiny13A-PU (alternativ ATtiny25, 45 oder 85)
- 1 ISP Programmieradapter
- 1 USB-UART Kabel mit 3,3 V Pegeln
- 1 Arduino Nano Board
- 1 WLAN Modul „ESP-01“ mit AT-Firmware
- 1 Mini-USB Kabel für den Arduino Nano
- 3 Kondensator 100 nF
- 1 Kondensator 10 µF 16V
- 1 Kondensator 2,2 µF 16V
- 1 Kondensator 100 µF 16V
- 1 Kondensator 1 µF 16V
- 2 Optische Sensoren Typ CNY70
- 2 7-Segment Anzeigen mit gemeinsamer Kathode, 14-20 mm hoch
- 1 Punkt-Matrix Anzeige 7x5 LED's mit gemeinsamer Kathode
- 1 LCD Display 2x16 oder 2x20 Zeichen, HD44780 kompatibel
- 2 Drehspul Einbau-Meßinstrumente 100 µA
- 1 DCF-77 Empfänger Modul „DCF1“ von Pollin.de
- 2 Leuchtdioden rot
- 2 Leuchtdioden gelb
- 2 Leuchtdioden grün
- 2 Dioden 1N4148
- 1 Zender-Diode 3,6 V 0,5W
- 1 Diode 1N4001
- 1 Widerstände 100 kΩ ¼ Watt
- 2 Widerstände 47 kΩ ¼ Watt
- 2 Widerstände 27 kΩ ¼ Watt
- 1 Widerstände 10 kΩ ¼ Watt
- 1 Widerstände 4,7 kΩ ¼ Watt
- 2 Widerstände 2,2 kΩ ¼ Watt
- 7 Widerstände 1 kΩ ¼ Watt
- 8 Widerstände 220 Ω ¼ Watt
- 1 Widerstand 47 Ω ¼ Watt
- 2 Trimmoti 10 kΩ linear (vorzugsweise Spindeltrimmer)
- 2 Transistoren BC337-40
- 1 Glühlämpchen für 5 V maximal 1 Watt
- 4 Kurzhubtaster
- 1 Spannungsregler LF33CV
- 1 Phototransistor für Tageslicht PT331C (oder ähnlicher)
- 1 Modellbau Mini Servo

- 1 Reed-Kontakt
 - 1 Kleiner Magnet
 - 1 NTC Temperaturfühler 10 kΩ bei 25° C,
zum Beispiel TTC05103JSY
 - 1 Piezokeramischer Schallwandler mit Gehäuse
 - 1 Knopfzelle CR2032
 - 1 Batterihalter für Knopfzelle CR2032
 - 1 Batteriekasten mit 3 Zellen (ca. 3,6 V)
 - 1 Netzteil 5 V mindestens 500 mA
 - 1 Kleiner Lautsprecher, möglichst mit Gehäuse, 8-100 Ohm
 - 1 Signalgeber ohne Ansteuerung mit mindestens 32 Ohm
(z.B. AL-60P06 oder AL-60P12)
 - 1 IC-Fassung DIL 8 Pin
 - 1 Print-Schraubklemme 2-polig
 - 1 Buchsenleiste 3 Pins
 - 2 Steckbretter mit je mindestens 700 Kontakten
 - 40 Jumper-Kabel für Steckbrett Male-Male
 - 10 Jumper-Kabel für Steckbrett Male-Female
-
- 1 Hohes Saftglas
 - 1 Teeglas
 - 1 Teebeutel
 - 1 Schaschlik-Stab
 - 1 Tesafilm
 - 1 Flüssiger Klebstoff
 - 1 Draht zum Anbinden
 - 2 Pappkartons
 - 2 Blatt Papier
 - 1 Bunter Trinkhalm
 - 1 Musterbeutelklammer
 - 1 Stück gelber Karton (Din A6)
 - 1 Schwarzer Filzstift
 - 2 Büroklammern