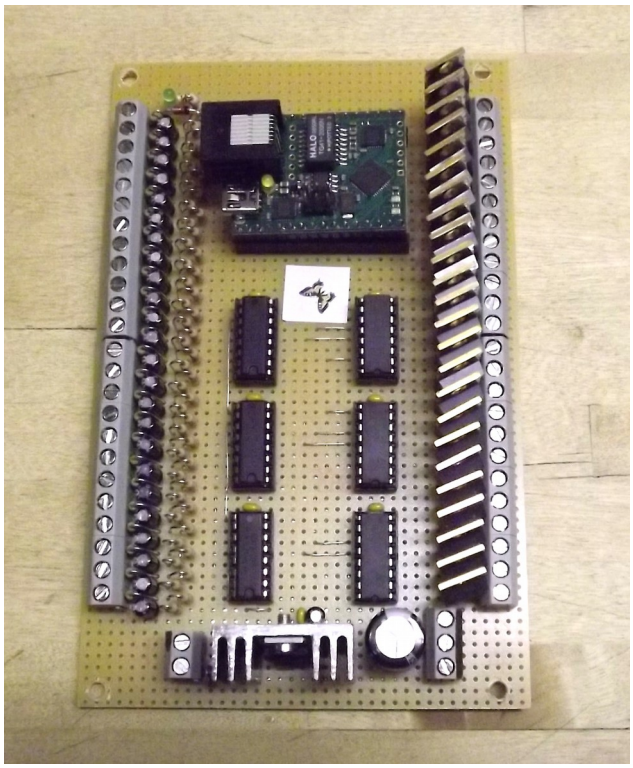
Einstieg in die Elektronik
mit Mikrocontrollern

Band 2

von Stefan Frings



48 Kanal I/O Schnittstelle mit USB und Ethernet

Downgeloaded von http://stefanfrings.de

Inhaltsverzeichnis

1 Einleitung

Im vorliegenden Band 2 des Buches stelle ich dir Bauteile vor, die ich in Verbindung mit AVR und anderen Mikrocontrollern häufig benutze, um Geräte zu bauen.

Ich möchte dir hier einfach nur einen Überblick über die wichtigsten Bauteile verschaffen. Ihre genaue Funktion erlernst du am besten durch ausprobieren und Studieren der Informationen, die im Internet bereit gestellt werden. Vor allem den Datenblättern.

Des weiteren erkläre ich anhand von konkreten Programmbeispielen, wie man in der Programmiersprache C mit Zeichenketten umgeht und wie Web-Browser mit Webservern kommunizieren. Bevor du dich mit dem Band 3 beschäftigst, solltest du wenigstens das Programmier-Kapitel durcharbeiten.

Sicher werden dir auch Beiträge in Wikipedia helfen, Fachbegriffe und Bauteile besser kennen zu lernen. Weitere Anleitungen kannst du in der Online Dokumentation der C Library finden, sowie in der Wissens-Sammlung des Mikrocontroller Forums und des Roboter Netzes. Die „Application Notes“ des Chip-Herstellers ergänzen die Datenblätter um wertvolle Hinweise und konkrete Programmbeispiele.

- http://de.wikipedia.org
- http://www.nongnu.org/avr-libc/user-manual/pages.html
- http://www.roboternetz.de/
- http://www.mikrocontroller.net/
- http://www.efo.ru/ftp/pub/atmel/_AVR_MCUs_8bit/_Technical_Library/appnotes/

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

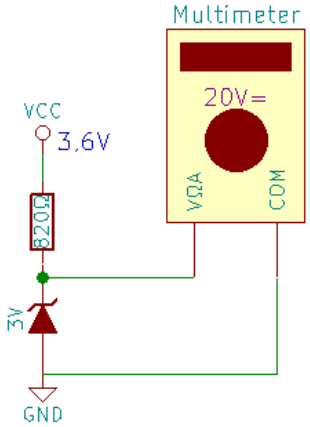
In den Mathematischen Formeln kommen folgende Symbole vor:

- U = Spannung in Volt
- I = Stromstärke in Ampere
- P = Leistung in Watt
- R = Widerstand in Ohm (Ω)
- F = Frequenz in Hertz
- β = Verstärkungsfaktor, auch HFE genannt

2 Bauteilkunde (Fortsetzung)

2.1 Zener-Dioden

Die Zener Diode funktioniert zunächst wie eine normale Diode mit Durchbruch-Spannung um 0,7 Volt. „Falsch herum“ gedreht, leitet sie aber ebenfalls, jedoch mit einer höheren Durchbruchs-Spannung. Zum Beispiel 3 Volt.



Man verwendet Zener Dioden gerne, um Spannungen zu regeln oder zu begrenzen, wenn nur sehr geringe Stromstärken erwartet werden. Allerdings hängt die Spannung ein bisschen von der Stromstärke und der Temperatur ab.

Zener Dioden kann man mit unterschiedlichen Spannungen im Bereich 2 bis 30 Volt kaufen. Die Standard-Ausführungen vertragen bei guter Kühlung bis zu 500 mW oder 1,3 W Verlustwärme.

Die Verlustwärme berechnet man nach der Formel: $P = U_{Diode} \cdot I$

2.2 Transistoren

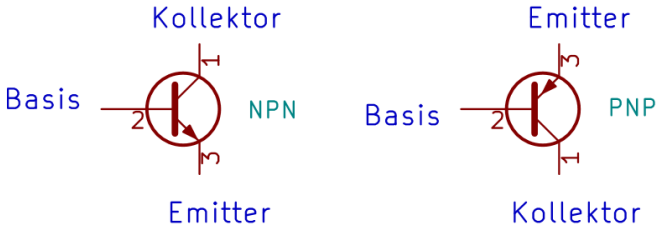
Transistoren benutzt man in digitalen Schaltungen, um große Lasten (Lampen, Motoren, ...) mit schwachen Steuersignalen zu schalten. Im Kapitel „Grundschaltungen“ werde ich zeigen, wie man damit die Ausgänge von Mikrocontrollern verstärkt.

In der Welt von analogen Schaltungen verwendet man Transistoren auch als Verstärker. Dieses Buch beschränkt sich jedoch auf digital schaltende Anwendungen, wo nur die Zustände „an“ und „aus“ vorgesehen sind.

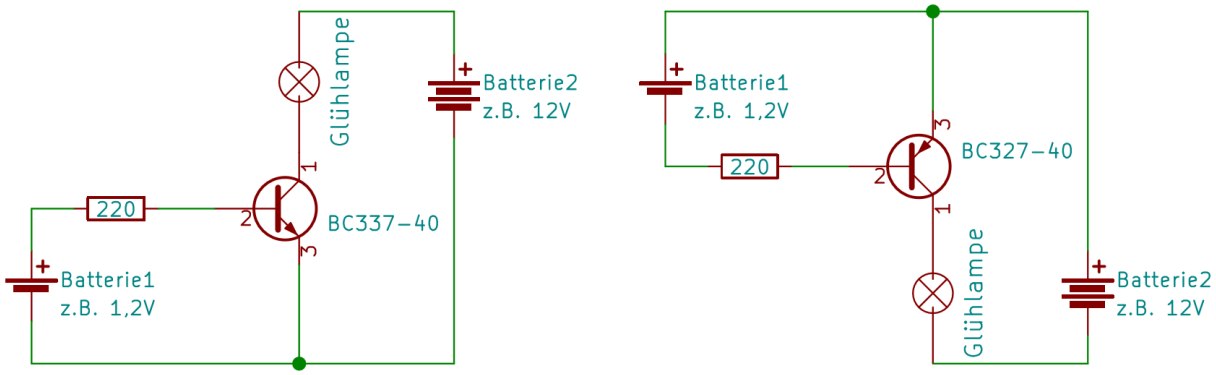
Je nach Anwendungsfall verwendet man kleine oder große Transistoren. Auch hat man die Wahl zwischen Bipolar-Transistoren und den moderneren MOSFET-Transistoren, die ich dir nun beide vorstellen werde.

2.2.1 Bipolare Transistoren

Bipolare Transistoren können mit einem geringen Steuerstrom größere Lasten schalten. Mit der Bauart NPN kann man den Minus-Pol schalten, und mit der Bauart PNP kann man den Plus-Pol schalten.



Die Basis (B) ist der Steuereingang und der Kollektor (C) ist der Ausgang, wo die Last angeschlossen wird. Der Emitter (E) gehört an die Stromversorgung. So benutzen wir diese beiden Transistoren:



Beide Schaltungen funktionieren nach dem gleichen Prinzip. Auf der linken Seite wird ein kleiner Steuerstrom in den Transistor geschickt, welcher auf der rechten Seite einen größeren Laststrom einschaltet.

Der Steuerstromkreis beginnt bei der 1,2 V Batterie und geht über den 220 Ω Widerstand in die Basis des Transistors und kehrt über dessen Emitter wieder zurück zur Batterie. Auf der rechten Seite beginnt der Last-Stromkreis bei der Batterie, geht über die Glühlampe in den Kollektor des Transistors und kehrt über den Emitter wieder zurück zur Batterie.

Für erste Versuche empfehle ich diese Transistoren:

Modell	Typ	Verstärkung mindestens	Maximale Kollektor Spannung	Maximaler Kollektor Strom
BC 337-40	NPN	250	45 V	500 mA
BC 327-40	PNP	250	45 V	500 mA

Je höher der Steuer-Strom durch die Basis ist, umso mehr Last-Strom lässt der Transistor durch den Kollektor fließen. Das Verhältnis dieser Ströme ist der Verstärkungsfaktor. Er wird in den Datenblättern oft als HFE oder β abgekürzt und stets als Bereich (Minimum und Maximum) angegeben. Die Tatsächliche Verstärkung des Transistors liegt in der Praxis immer irgendwo zwischen den beiden Angaben. Ein Transistor mit 2 mA Steuer-Strom und einem Verstärkungsfaktor von 250 würde bis zu 500 mA schalten können.

In den obigen Schaltungen wird die stärke des Steuerstroms durch den Widerstand vor der Basis bestimmt. Dazu muss man noch wissen, dass an der Basis-Emitter Strecke von gewöhnlichen Transistoren immer etwa 0,7 V abfallen. Nun können wir den Steuerstrom ausrechnen:

$$\frac{1{,}2\,V - 0{,}7\,V}{220\,\Omega} = 2{,}3\,mA$$

Beachte aber auch den nächsten Absatz!

2.2.1.1 Verlustleistung

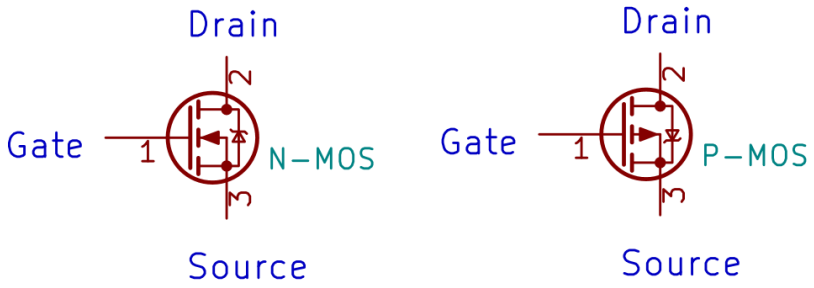
Der Laststrom bewirkt, dass am Transistor (zwischen Kollektor und Emitter) Spannung verloren geht und dass der Transistor deswegen warm wird. Die oben genannten Transistoren vertragen die genannten 500 mA Laststrom nur bei guter Kühlung mit Übersteuerung.

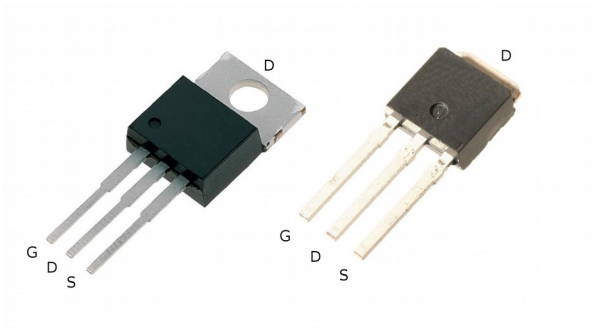
Man reduziert die Verluste deutlich, indem man den Transistor übersteuert. Dass heißt, man steuert ihn mit der dreifachen bis zehnfachen Stromstärke an, als nach der obigen Formel nötig wäre.

Es gibt erheblich größere bipolare Transistoren für höhere Ströme, allerdings haben sie nur sehr geringe Verstärkungsfaktoren und hohe Verlustleistungen. Sie wurden inzwischen weitgehend durch die moderneren MOSFET Transistoren abgelöst.

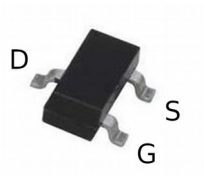
2.2.2 MOSFET Transistoren

Der MOSFET Transistor benötigt keinen Steuerstrom, sondern lediglich eine Spannung. Mit der N-Kanal Variante kann man den Minus-Pol einer Last schalten. Zum Schalten des Plus-Pol verwendet man hingegen die P-Kanal Variante.



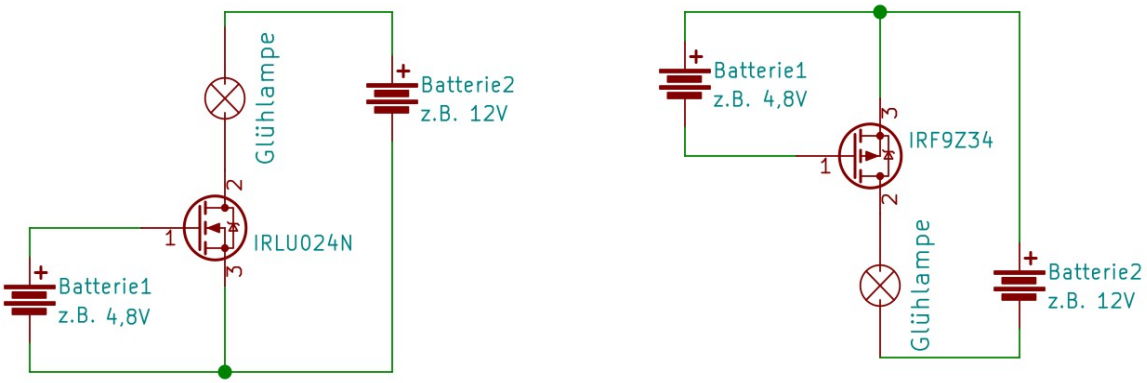


TO220 *iPak*

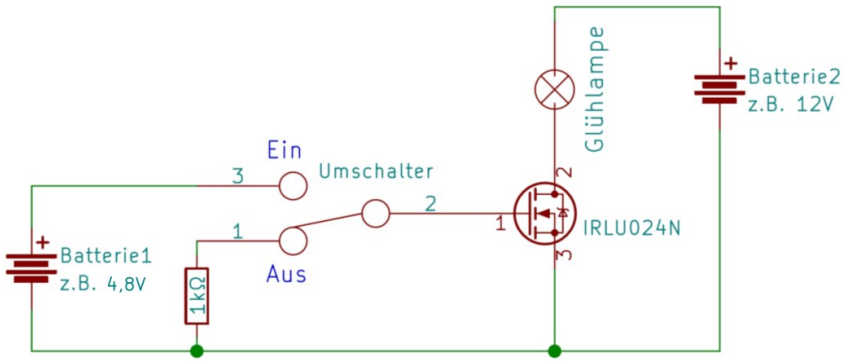


SOT23

Das Gate (G) ist der Steuereingang und Drain (D) ist der Ausgang für den Laststrom. Der Source (S) Anschluss gehört an die Stromversorgung. So verwenden wir MOSFET Transistoren:



Der Eingang des MOSFET verhält sich elektrisch gesehen wie ein kleiner Kondensator. Wenn er erst einmal aufgeladen ist, fließt kein weiterer Strom mehr in den Steuereingang. Alleine das Vorhandensein der Steuerspannung genügt, um den Transistor eingeschaltet zu halten. Wenn du die Batterie weg nimmst, bleibt der Transistor noch lange geladen und eingeschaltet. Du musst ihn aktiv entladen, um ihn auszuschalten, zum Beispiel durch einen Widerstand:



Für erste Experimente empfehle ich dir diese Transistoren:

Modell	**Typ**	**Gehäuse**	**Gate Kapazität**	**Max. Drain Spannung**	**Max. Drain Strom bei Gate Spannung**
IRLU024N	N	iPak	580 pF	55 V	0,3 A bei 3 V 17 A bei 5 V
IRLZ44N IRLIZ44N	N	TO220	1700 pF oder 3300 pF (je nach Hersteller)	55 V	4 A bei 3 V 47 A bei 5 V
IRLML6344	N	SOT23	650 pF	30V	5A bei 2,5 V
IRF9Z34	P	TO220	620 pF	55 V	1 A bei 5 V 17 A bei 10 V
IRML6402	P	SOT23	630 pF	20V	4 A bei 3V
NDP6020P	P	TO220	1600pF	20V	24 A bei 3V

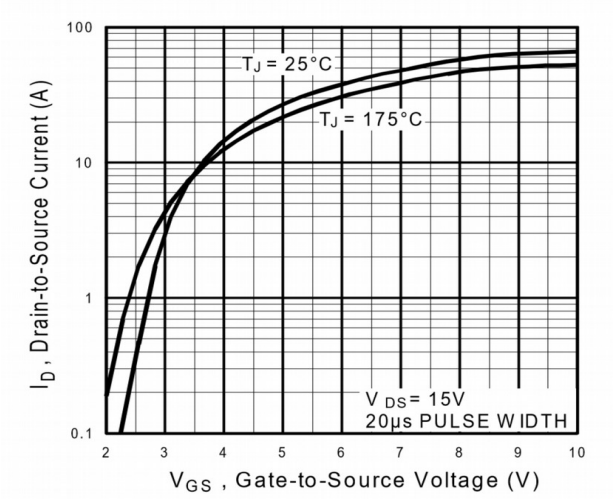
Der etwas teurere IRLIZ44N ist dem IRLZ44N ähnlich, hat als Besonderheit jedoch ein isoliertes Gehäuse, so dass man zur Montage auf einen Kühlkörper keine Isolierscheibe benötigt.

Das I-Pak Gehäuse ist etwas flacher als TO220 und hat an der oberen Kante eine Lötfahne, woran man eine Kühlfläche anlöten kann.

2.2.2.1 Steuerspannung

Beachte, dass P-Kanal MOSFET tendenziell schlechtere Eigenschaften haben, als die N-Kanal Variante, deswegen werden N-Kanal MOSFET allgemein bevorzugt. P-Kanal MOSFET für 3,3V Steuerspannung gibt es offenbar nur in SMD Gehäusen.

Schau dir das folgende Diagramm an, es stammt aus dem Datenblatt des IRLU024N:



Du kannst an diesem Diagramm ablesen, wie viel Strom der Transistor fließen lässt, wenn eine bestimmte Steuerspannung am Gate anliegt. Dieser Transistor lässt bei 3 Volt Steuerspannung und 25°C Temperatur typischerweise bis zu 3 Ampere fließen. Eine Überschreitung dieser Grenze führt dazu, dass am Transistor viel Spannung abfällt und er extrem heiß wird.

Man sollte beim Lesen der Diagramme aber immer bedenken, dass sie nur das typische Verhalten darstellen. Im Datenblatt ist die "Gate Threshold Voltage" mit 1 V bis 2 V angegeben. Du kannst annäherungsweise davon ausgehen, dass der Hersteller mit dem typischen Fall die Mitte dazwischen meint - in diesem Fall also 1,5V.

Im ungünstigsten Fall könnte sich die ganze Kurve somit um 0,5V nach rechts verschieben. Dann kommen wir bei 3 V Steuerspannung und 25°C nur noch auf 0,3 A !

2.2.2.2 Verlustleistung

MOSFET Transistoren haben in voll durchgeschaltetem Zustand typischerweise einen Innenwiderstand von weniger als 0,2 Ω. Die Angaben ganz oben in den Datenblättern gelten allerdings nur für den Idealfall, wenn das Gate mit der maximal zulässigen Spannung angesteuert wird. In den darunter platzierten Tabellen (Electrical Characteristics) findest du normalerweise realistischere Angaben für konkrete niedrigere Gate-Spannungen.

Wenn du den Laststrom mit dem Innenwiderstand multiplizierst, erhältst du die Verlust-Spannung.

$$U_{DS} = R_{DS} \cdot I_{DS}$$

Wenn der Transistor beispielsweise einen Innenwiderstand von 0,05 Ohm hat, und die Last eine Stromstärke von 4 Ampere benötigt, dann fallen am Transistor 0,2 Volt ab. Wenn du die Verlustspannung mit dem Strom multiplizierst, erhält du die Verlustleistung.

$$P = U_{DS} \cdot I_{DS} \quad \text{oder:} \quad P = R_{DS} \cdot I_{DS}^{2}$$

Die Verlustleistung gibt der Transistor in Form von Wärme ab. Je höher sie ist, umso wärmer (oder gar heißer) wird der Transistor. Bei mehr als 1 Watt Verlustleistung muss der Transistor auf jeden Fall an einen Kühlkörper geschraubt oder gelötet werden, sonst brennt er nach wenigen Sekunden durch.

2.2.2.3 Statische Ladungen

Das Gate ist empfindlich gegen Überspannung. Die meisten MOSFET Transistoren vertragen höchstens 20 Volt Zwischen Gate und Source, einige sogar noch weniger. Statische Ladung auf deiner Körperoberfläche kann unter Umständen schon zur Zerstörung des MOSFET Transistors führen. Darum solltest du deinen Körper immer entladen, bevor du eine elektronische Schaltung mit MOSFET Transistoren anfasst.

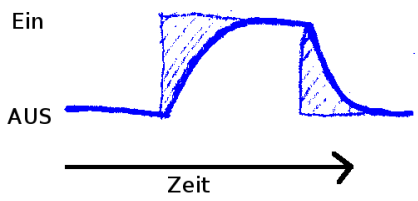
Du kannst deinen Körper entladen, indem du die Metallfläche eines Heizkörpers oder Wasserhahn anfasst. Es ist in diesem Sinne auch vorteilhaft, einen hölzernen Arbeitstisch mit unlackierter Oberfläche zu verwenden und auf das Tragen von Kleidung aus Kunststoff zu verzichten. Das betrifft auch die Schuhe.

2.2.2.4 Gate-Kapazität

Der große Nachteil von MOSFET Transitoren ist ihre hohe Gate-Kapazität. Sie ist typischerweise 100 bis 1000 mal größer, als bei bipolaren NPN und PNP Transistoren!

Die Spannung am Gate-Anschluss kann wegen des Kondensators im Innern des Transistors nicht so schnell von Low nach High (und umgekehrt) wechseln. Demzufolge kann der Transistor auch nicht so schnell zwischen „an“ und „aus“ umschalten.

Die folgende Kurve zeigt, wie so ein Transistor aufgrund der Verzögerung durch die Kapazität umschaltet.



In den schraffiert gezeichneten Zeitabschnitten wird der Transistor warm, weil er dort weder voll durch schaltet noch den Stromkreis komplett unterbricht. In diesen Momenten ist die Verlust-Spannung und die Verlust-Leistung viel höher, als in voll eingeschaltetem Zustand. Wenn der Transistor zwischen den Schaltvorgängen genügend Zeit zum Abkühlen hat, stört das nicht weiter. Je öfter man den Transistor pro Sekunde schalten lässt, umso heißer wird er jedoch.

Bei Frequenzen oberhalb von 1000 Hz muss man diesen Effekt unbedingt berücksichtigen. Man bevorzugt dann Transistoren mit geringer Gate-Kapazität oder verwendet starke MOSFET-Treiber.

2.2.3 Kühlkörper

Wenn ein Transistor mehr als 1 Watt verheizt, muss er an einen Kühlkörper geschraubt werden. Die Oberfläche des Transistors genügt dann nämlich nicht mehr, um die Wärme schnell genug abzuleiten. Der Sinn des Kühlkörpers ist, die Wärme des Transistors aufzunehmen und über eine viel größere Oberfläche an die umgebende Luft abzuleiten. Je mehr Oberfläche ein Kühlkörper hat, umso mehr Wärme kann er daher ableiten. Zusätzlich lässt die Leistung des Kühlkörper durch einen Ventilator erhöhen.

Kühlkörper für Transistoren bestehen in der Regel aus einem schwarz eloxiertem Aluminium-Gerippe, weil die so behandelte Oberfläche ihre Wärme besser abgeben kann, als einfaches blankes Aluminium. Die Leistung eines Kühlkörpers wird in Kelvin pro Watt angegeben. Dieser Wert sagt aus, wie viel Grad der Transistor pro Watt wärmer wird.

Angenommen, der Transistor gibt 2 Watt Verlustwärme ab und der Kühlkörper hat eine Leistung von 17 K/W, dann wird der Kühlkörper 34 Grad wärmer, als seine Umgebungstemperatur. Je größer der Kühlkörper ist, umso kleiner ist sein K/W Wert. In der Praxis hat es sich bei mir bewährt, Kühlkörper so auszulegen, dass sie sich höchstem um 60 Grad erwärmen. Bei einer angenommenen Umgebungstemperatur von 40 Grad (was nicht ungewöhnlich ist) wird der Kühlkörper so maximal

100 Grad heiß. Um den richtigen Kühlkörper zu kaufen, berechnest du die benötigte Leistung nach folgender Formel:

$$K/W = \frac{60}{P}$$

Ein Transistor mit 20 Watt Verlustwärme benötigt demnach einen Kühlkörper mit 3 K/W (oder weniger).

2.2.3.1 Montage

Um den Transistor mit dem Kühlkörper zu verbinden, bohrt man an geeigneter Stelle ein 3mm Loch durch den Kühlkörper.

Ganz wichtig: Die Kanten des Bohrloches müssen entgratet werden! Denn sonst könnten Grate sich zwischen Transistor und Kühlkörper klemmen und so den Kontakt zwischen Transistor und Kühlkörper erheblich verschlechtern. Die Kontaktflächen müssen glatt und sauber sein!

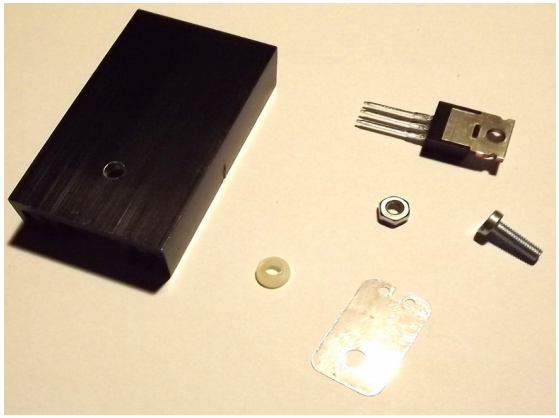
Anschließend gibt man einen winzig kleinen Tropfen Wärmeleitpaste auf den Transistor und verschraubt ihn dann mit einer M3x10 Schraube fest mit dem Kühlkörper.

Die Wärmeleitpaste sieht aus, wie Penaten-Creme - meistens weiß, manchmal auch silbern. Sie enthält Metall-Partikel, welche den Übergang der Wärme zum Kühlkörper verbessern. Die Paste erfüllt ihren Zweck aber nur dann, wenn sie eine hauchdünne Schicht zwischen Transistor und Kühlkörper bildet. Zu viel Wärmeleitpaste ist noch aus einem zweiten Grund schlecht: Sie könnte die Anschlussdrähte des Transistors beschmutzen und zu Kurzschlüssen führen, denn Wärmeleitpaste ist auch elektrisch leitfähig.

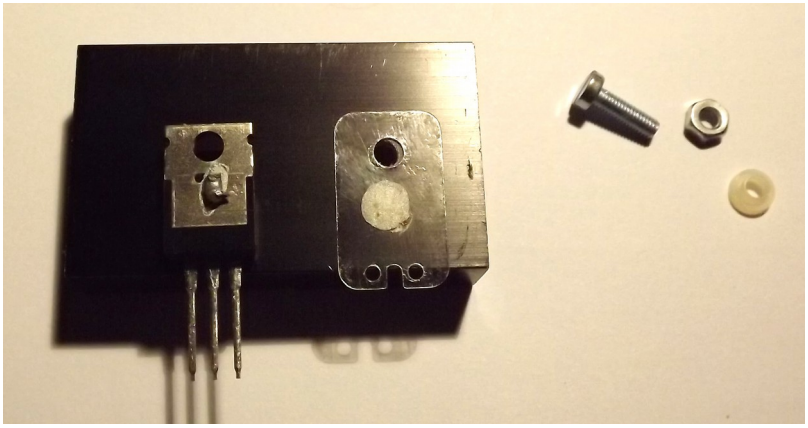
2.2.3.2 Montage mit Glimmer

Bei allen Transistoren im TO220 Gehäuse ist der Kühlkörper-Anschluss mit dem mittleren Pin verbunden (das ist meistens der Drain Anschluss). Der ganze Kühlkörper steht daher unter Strom! Und jeder Transistor benötigt daher seinen eigenen Kühlkörper.

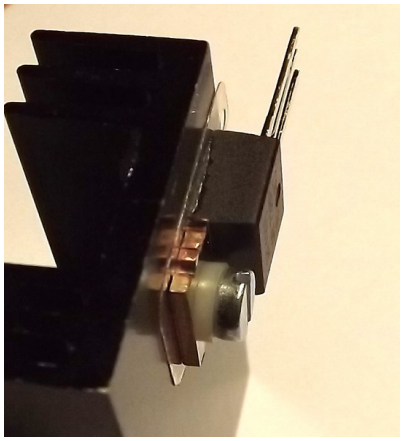
Wenn man den Kühlkörper nicht unter Strom setzen will, benutzt man zwischen Transistor und Kühlkörper eine Glimmerscheibe als Isolator. Derart isolierte Transistoren können sich dann auch einen gemeinsamen Kühlkörper teilen. Im Fachhandel werden entsprechende Montage-Sets (Glimmer, Schraube, Mutter, Isoliernippel) angeboten.

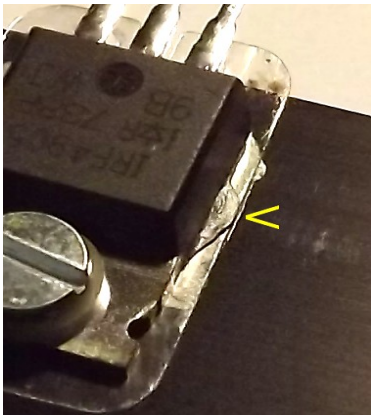


Die Glimmerscheibe sorgt dafür, dass der Transistor den Kühlkörper nicht direkt berührt. Sie ist elektrisch isolierend, leitet Wäme jedoch sehr gut. Zwischen Kühlkörper und Glimmerscheibe platziert man einen winzig kleinen Tropfen Wärmeleitpaste. Auch der Transistor bekommt einen ganz kleinen Tropfen Wärmeleitpaste:



Den Isoliernippel steckt man in die Bohrung des Transistors, dann schraubt man das Ganze so zusammen:



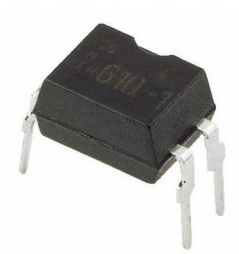


Die Schraube berührt den Transistor nicht, dafür sorgt der Isoliernippel. Nachdem die Schraube fest angezogen ist, verteilt sich die Wärmeleitpaste unter dem Druck. Wenn du zu viel Paste verwendest, quillt sie am Rand über und kann so einen Kurzschluss verursachen.

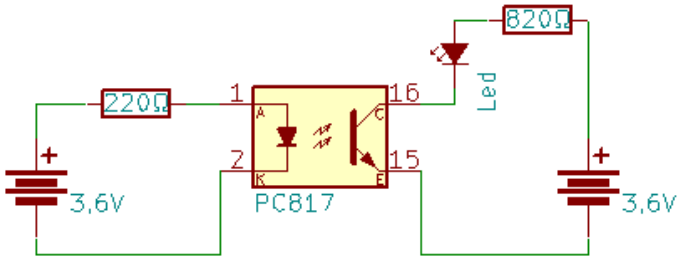
An der markierten Stelle im rechten Bild ist Wärmeleitpaste über die Kante der Glimmerscheibe hinaus gequollen. Da sie leitfähig ist, ist mein Transistor nun doch mit dem Kühlkörper verbunden! Die Isolation durch die Glimmerscheibe ist wegen diesen Fehler wirkungslos. Überschüssige Wärmeleitpaste muss daher sorgfältig entfernt werden. Mit einem Multimeter (im 2000 Ohm Bereich) kannst du überprüfen, ob eine ungewollte Verbindung zwischen Transistor und Kühlkörper vorhanden ist.

2.3 Optokoppler

Optokoppler übertragen Signale durch Licht. Man verwendet sie, um Signale zwischen zwei Geräten zu übertragen, die keine direkte elektrische Verbindung zueinander haben sollen oder keine gemeinsame Masse (GND) haben.



Modell: PC 817



Optokoppler enthalten eingangs-seitig eine Leuchtdiode und Ausgangsseitig einen lichtempfindlichen Transistor. Der Transistor schaltet durch, wenn die Leuchtdiode leuchtet. Die ganze Konstruktion ist in ein Licht-Undurchlässiges Gehäuse verpackt.

Standard-Modelle isolieren typischerweise bis 1000 Volt und haben ein Übertragungsverhältnis von weniger als 100 %. Das bedeutet, dass der Transistor weniger Strom schalten kann, als die Stromstärke, die durch die Diode fließt. Ich verwende gerne die Typen: PC 817, KB 817, LTV 817, CNY-17, 4N 32

2.4 Spulen

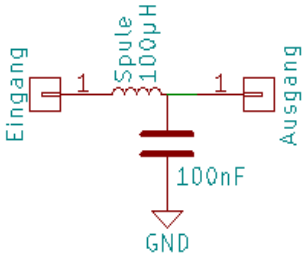
Spulen bestehen aus einem aufgewickeltem Kupferdraht. Die Spule erzeugt ein Magnetfeld, wenn sie von Strom durchflossen wird. Umgekehrt erzeugt sie einen Stromfluss, wenn sich das Magnetfeld in ihrer Umgebung verändert.

Je mehr Windungen eine Spule hat, umso größer ist ihre Wirkung. Die Wirkung der Spule nennt man Induktivität und sie wir in der Einheit „Henry“ gemessen. Ein Kern aus Eisen oder Ferrit verstärkt die Wirkung der Spule.

Die Spule hat für Gleichspannung einen geringen Widerstand. Für hochfrequente Störimpulse hat die Spule jedoch einen hohen Widerstand.

In Computer-Schaltungen setzt man Spulen für ganz unterschiedliche Anwendungen ein:

2.4.1 Versorgungsspannung Entstören



Die Spule ist für hochfrequente Störungen kaum durchlässig, aber Gleichstrom lässt sie beinahe verlustfrei fließen. Der Kondensator wirkt genau umgekehrt. Er hat für Gleichspannung einen hohen Widerstand, wenn er erst mal aufgeladen ist. Für hohe Frequenzen hat der Kondensator jedoch einen geringen Widerstand. Störung, die die Spule noch durch lässt schließt der Kondensator quasi kurz.

Beide Bauteile zusammen ergänzen sich sehr gut, um Störsignale aus einer Leitung heraus zu filtern. Man nennt diese Kombination L/C-Filter. Diese Filter haben eine definierte Grenzfrequenz. Niedrigere Frequenzen lassen sie durch, höhere lassen sie nicht durch.

Mikrocontroller sind nicht sehr Anspruchsvoll, was die Stromversorgung angeht. Sie brauchen in der Regel keine L/C Filter. In der Stromversorgung von analog zu digital Wandlern (ADC) helfen L/C Filter jedoch, präzise rauscharme Messergebnisse zu liefern. Einige Mikrocontroller haben daher für ihren internen ADC einen separaten Pin zur Stromversorgung.

Die Grenzfrequenz berechnet man mit der Formel:

$$F = \frac{1}{2 \cdot \Pi \cdot \sqrt{L \cdot C}}$$

Bei 100 yH und 100 nF beträgt die Grenzfrequenz demnach 16kHz. Störsignale oberhalb dieser Frequenz werden durch den L/C Filter deutlich reduziert.

2.4.2 Energiespeicher in Spannungswandlern

Spannungswandler erhöhen oder verringern die Spannung einer Stromversorgung. Diese Geräte nutzen die magnetische Speicherfähigkeit einer Spule aus.

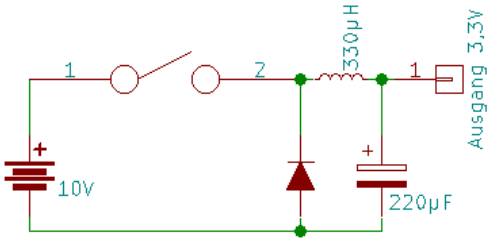
Wenn man einen Strom durch die Spule fließen lässt, baut sie ein Magnetfeld auf. Wenn man anschließend den Strom abschaltet, fällt das Magnetfeld wieder zusammen. Dabei gibt die Spule die in ihr gespeicherte Energie wieder ab, aber umgekehrt gepolt.

Mit Hilfe von Dioden kann man diese Energie auffangen und weiter verwenden. Dabei bevorzugt man schnelle Schottky Dioden (nicht alle Schottky Dioden sind schnell!).

2.4.2.1 Step-Down Wandler

Step-Down Wandler reduzieren die Spannung. Jedes Notebook enthält mehrere Step-Down Wandler, um aus der Batteriespannung (ca. 10 V) die Versorgungsspannungen der Komponenten zu erzeugen: 5 V für die Laufwerke, 3,3 Volt für die Schnittstellen und ungefähr 1 Volt für den Prozessor.

Diese Geräte funktionieren nach folgendem Prinzip:

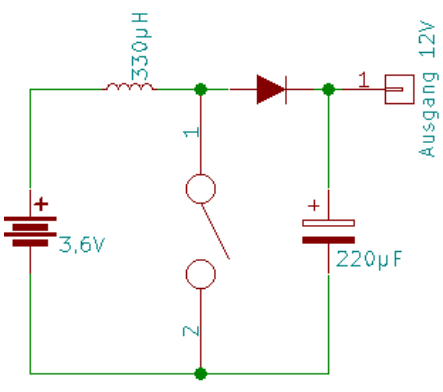


Die Batteriespannung wird durch einen Schalter (Transistor) wiederholt ein und aus geschaltet. Wenn der Schalter eingeschaltet ist, baut die Spule ein Magnetfeld auf. Gleichzeitig wird der Kondensator aufgeladen.

Wenn der Schalter abschaltet, baut die Spule ihr Magnetfeld wieder ab. Dabei erzeugt sie selbst einen umgekehrt gerichteten Strom-Impuls, der durch die Diode auf den Kondensator abgeleitet wird. Eine Steuer-Elektronik misst ständig die Ausgangsspannung und passt die Schaltzeiten so an, dass die Ausgangsspannung konstant und richtig ist.

2.4.2.2 Step-Up Wandler

Step-Up Wandler erhöhen die Spannung einer Batterie. Viele tragbare Musik-Abspielgeräte erhöhen die Spannung aus 1-2 Batterien auf 3,3 Volt. Diese Geräte funktionieren nach folgendem Prinzip:



Der Schalter (Transistor) wird wiederholt ein und aus geschaltet. Wenn er eingeschaltet ist, fließt ein ansteigender Strom durch die Spule, wodurch sich ein Magnetfeld aufbaut. Wenn der Schalter dann abschaltet, baut sich das Magnetfeld wieder ab, wobei die Spule eine umgekehrt gepolte Spannung abgibt. Diese Spannung addiert sich zur Batteriespannung und wird über die Diode auf den Kondensator abgeleitet.

Eine Steuer-Elektronik misst ständig die Ausgangsspannung und passt die Schaltzeiten so an, dass die Ausgangsspannung konstant und richtig ist.

2.5 Polyfuse Sicherungen

Polyfuse Sicherungen sind sehr praktisch, denn sie stellen sich automatisch wieder zurück, wenn man sie abkühlen lässt. Leider reagieren Polyfuse Sicherungen sehr träge (typischerweise in 3-5 Sekunden) und nach der ersten Auslösung ist ihr Innenwiderstand dauerhaft etwas höher, als in neuem Zustand.

Polyfuse Sicherungen setzt man gerne ein, um Bauteile zu schützen, die kurzzeitig aber nicht auf Dauer überhöhte Ströme vertragen. Zum Beispiel Kabel, Motoren, Transformatoren, Leistungs-Transistoren.

Ich benutze 400 mA Polyfuse Sicherungen gerne bei USB betriebenen Schaltungen, um mein Notebook abzusichern.

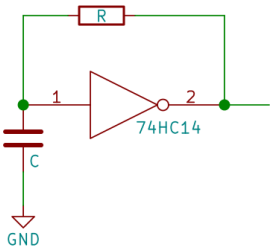
3 Grundschaltungen

3.1 Taktgeber

Taktgeber bestimmen die Geschwindigkeit, mit der digitale Schaltungen arbeiten.

3.1.1 R/C Oszillator

Fast alle Mikrocontroller enthalten mindestens einen R/C Oszillator.



Dieser Oszillator lädt einen Kondensator durch einen Widerstand immer abwechselnd auf und ab. Jeder Umlade-Vorgang braucht eine gewisse Zeit, welche von der Kapazität des Kondensator und dem Wert des Widerstandes abhängt.

Der rechts abgebildete Oszillator funktioniert so:

Wenn die Spannung am Eingang des IC über 60% ansteigt, schaltet der Ausgang auf Low um. Und wenn die Eingangsspannung unter 30% sinkt, schaltet der Ausgangs auf High um.

Zunächst ist der Kondensator leer, so dass der Eingang des IC auf Null Volt liegt und der Ausgang auf High. Dann lädt sich der Kondensator durch den Widerstand langsam auf. Wenn die Spannung vom Kondensator die 60% Schwelle erreicht hat, schaltet der Ausgang des IC auf Low um.

Nun entlädt sich der Kondensator durch den Widerstand. Sobald die Spannung vom Kondensator die 30 % Schwelle erreicht hat, schaltet der Ausgang des IC wieder auf High um.

Dann lädt sich den Kondensator wieder auf, und so weiter. Es entsteht eine endlose Wiederholung auf Aufladen und Entladen – eine Schwingung.

Die Frequenz ist ungefähr: $t = \frac{1}{(C \cdot R)}$

Bei 1 µF und 1 MΩ erhält man ungefähr 1Hz, also einen Takt pro Sekunde.

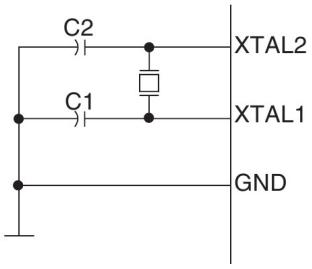
Diese Schaltung ist einfach und billig, aber auch ziemlich ungenau. Denn die Abweichungen der Werte von Kondensator, Widerstand und IC multiplizieren sich. R/C Oszillatoren aus hochwertigen Bauteilen weichen typischerweise bis zu 10% von der Soll-Frequenz ab.

Die Oszillatoren in den AVR Mikrocontrollern werden allerdings vom Hersteller vor dem Verkauf justiert, so dass sie typischerweise nur 2% abweichen.

3.1.2 Quarz-Oszillator

Wenn genaues Timing erforderlich ist, verwendet man Quarze als Zeit-bestimmendes Element. In der silbernen Dose befindet sich ein kleines Kristall-Plättchen, dass ziemlich präzise mit einer ganz bestimmten Frequenz mechanisch schwingt. Die Abweichung von der aufgedruckten Frequenz beträgt typischerweise weniger als 0,002 Prozent.

Quarze mit Frequenzen im Bereich 0,4...20 MHz kann man direkt an Mikrocontroller anschließen. Zusätzlich brauchst du zwei kleine keramische Kondensatoren, meistens passen 22pF.

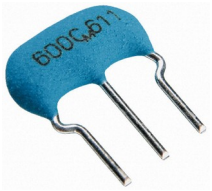


Die Leitungen zwischen Mikrocontroller und den drei Bauteilen müssen unbedingt so kurz wie möglich gestaltet werden. Damit der Mikrocontroller den Quarz anstelle des internen R/C Oszillators verwendet, änderst du die Fuse-Bits CKOPT und CKSEL entsprechend der gewählten Frequenz.

Um die Quarze zu berechnen, muss man die „Lastkapazität“ des Quarzes kennen, die üblicherweise im Angebot steht. Da die beiden externen Quarze in Reihenschaltung wirken, muss man sie doppelt so groß auslegen. Davon aber subtrahiert man wiederum mutig geschätzte 10pF für die Leitungen und die Eingänge des angeschlossenen IC. Für einen Quarz mit 18pF Lastkapazität rechnet man zum Beispiel: $18pF \cdot 2 – 10pF = 26pF$

Wenn die Taktfrequenz noch genauer sein soll, dann tausche C2 durch einen Trimmer-Kondensator mit etwa 5-50 pF aus. Du kannst dann die Frequenz mit einem kleinen Schraubendreher justieren und so die eventuell vorhandene Ungenauigkeit des Quarzes ausgleichen. Eine konstante Temperatur ist auch hilfreich, denn die Schwingfrequenz von Quarzen hängt ein kleines bisschen von der Temperatur ab.

3.1.3 Keramik-Resonator



Als Alternative zum Quarz werden häufig Keramik-Resonatoren verwendet. Sie sind kleiner und billiger, dafür sind sie aber auch weniger genau. Ihre Abweichung von der Soll-Frequenz ist typischerweise etwas weniger als 0,5 Prozent.

Die nötigen Kondensatoren befinden bereits sich im Keramik-Resonator, deswegen hat er drei Anschlüsse. Der mittlere gehört an GND.

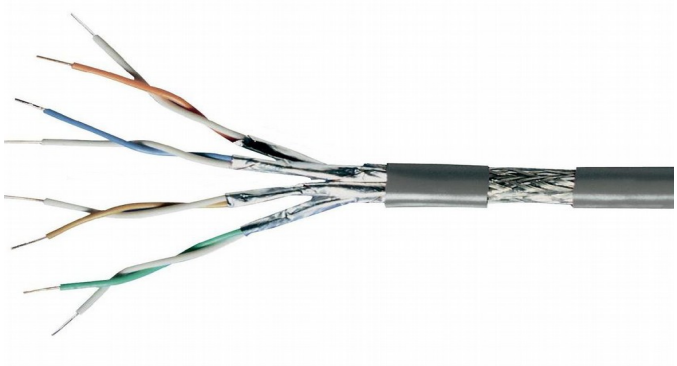
3.2 Lange Kabel

Überall in der Luft, ganz besonders in Gebäuden, sind elektrische und magnetische Felder. Diese Felder verfälschen die eigentlichen Signale, die man übertragen will. In digitalen Schaltungen muss man diesen Effekt bei Kabellängen ab 30 cm berücksichtigen, damit sie zuverlässig funktionieren.

In diesem Kapitel geht es um Elektromagnetische Verträglichkeit, kurz EMV.

3.2.1 Abschirmung

Man verwendet abgeschirmte Datenkabel. Die Signal-Litzen dieser Kabel sind mit einer metallenen Folie und einem Geflecht aus Kupfer umwickelt, welche elektromagnetische Felder abschwächt. Je aufwändiger die Abschirmung gemacht ist, umso wirksamer. Die Abschirmung kann darüber entscheiden, ob ein Kabel 1 Euro oder 20 Euro kostet.

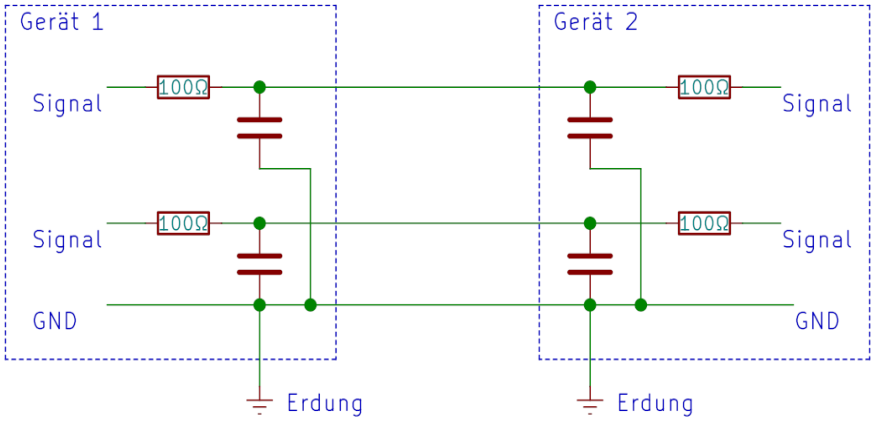


Doppelt abgeschirmtes Kabel mit paarweise verdrillten Leitungen von Conrad Elektronik

Die Abschirmung der Kabel wird idealerweise geerdet, so dass die abgefangenen Ströme zur Erde abgeleitet werden. Computer ohne Erdung leiten die Signale stattdessen zur Masse (GND) ab. Das wirkt weniger gut, aber immer noch besser, als gar keine Abschirmung.

3.2.2 EMV Schutz mit Tiefpässen

Leider sind Datenkabel trotz Abschirmung noch empfänglich für starke elektrische Felder, zum Beispiel die Funkwellen von Smartphones und Radiosendern. Dagegen helfen kleine Kondensatoren von 220 pF an den Enden der Signal-Leitungen:



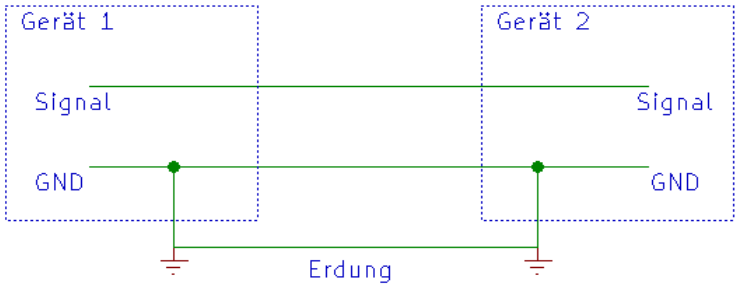
Die Kondensatoren begrenzen zusammen mit den Widerständen die übertragbare Signalfrequenz auf ein paar hundert Kilohertz. Radiowellen werden nun durch die Kondensatoren abgefangen und nach GND abgeleitet. Eine Zusätzliche Erdung der Geräte hilft, die Wirkung zu verbessern.

Die Widerstände beschützen außerdem die angeschlossenen Mikrochips vor zu hohen Strömen bei geringer Überspannung. Das kommt sehr häufig vor, zum Beispiel wenn das eine Gerät Spannung liefert, während das andere noch ausgeschaltet ist. Oder wenn man an Steckverbindern hantiert, während Spannung anliegt.

Diese Schaltung kann elektromagnetische Störungen nicht ganz beheben, aber immerhin deutlich reduzieren. Sie war viele Jahre gut genug, um Drucker an Personal Computer anzubinden.

3.2.3 Masseschleifen

Masseschleifen entstehen, wenn die GND-Leitungen irgendwie einen Ring bilden. Das passiert zum Beispiel, wenn du deine Stereoanlage mit dem Computer verbindest und beide Geräte geerdet sind.



Im unteren Bereich der Zeichnung befindet sich die Ringförmige Verbindung, welche aus der GND Verbindung und der Erdung der beiden Geräte besteht.

Magnetische Felder, die von elektrischen Geräten und Gewittern hervorgerufen werden, induzieren hier einen Strom, der bewirkt, dass das GND Potential auf der linken Seite nicht exakt dem GND Potential auf der rechten Seite entspricht.

Bei Audio-Verbindungen bewirkt dies ein unangenehmes Brummen im Ton. Bei digitalen Signalen kommt es zu Übertragungsstörungen, wenn der Potential-Unterschied (also die magnetisch induzierte Wechselspannung) mehr als 0,5 Volt beträgt.

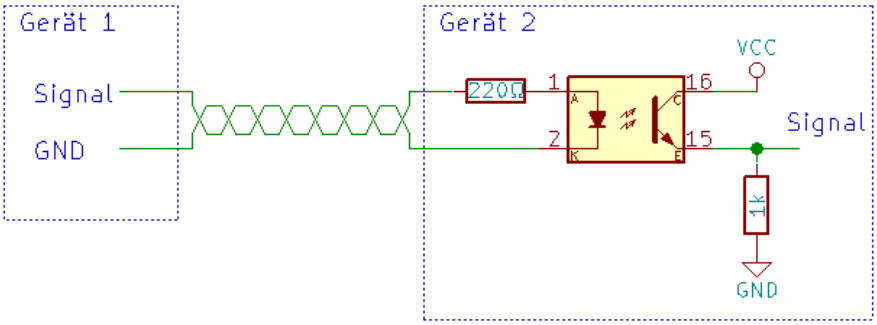
Die Mikrochips der RS485 Schnittstelle tolerieren Potential-Unterschiede bis zu 7 Volt. Damit sind sie nicht völlig immun gegen Masseschleifen, aber für die allermeisten Fälle genügt das.

Wenn selbst die 7 V nicht ausreichen, ist eine vollständige Trennung der Geräte notwendig. Man nennt das dann „Galvanische Trennung“. Bei der galvanischen Trennung fließt kein Strom von einem Gerät zum anderen. Masseschleifen können sich daher nicht störend auswirken. In der Industrie legt man häufig Wert auf Galvanische Trennung.

Die Signale können dann allerdings nicht mehr elektrisch übertragen werden. Man muss sich etwas anderes einfallen lassen. Den folgenden Lösungsansatz habe ich von der Midi Schnittstelle in elektronischen Musikinstrumenten abgeguckt.

3.2.4 Galvanische Trennung mit Optokoppler

Bei digitalen Signalen mit Frequenzen bis ca. 100kHz kann man Masseschleifen recht einfach mit einem Optokoppler verhindern:

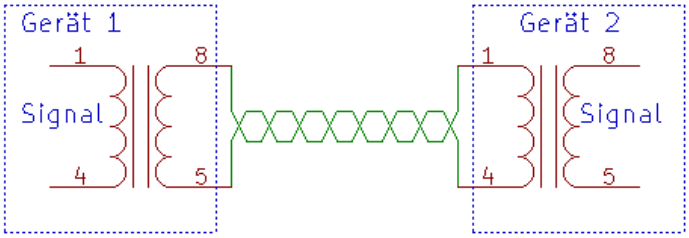


Die beiden GND Anschlüsse der Geräte müssen nun nicht mehr miteinander verbunden sein. Und selbst wenn sie verbunden wären, würde die Masseschleife das zu übertragende Signal nicht mehr stören, denn die LED im Optokoppler arbeitet unabhängig vom GND Potential.

Damit die elektromagnetischen Störfelder möglichst wenig Einfluss auf das Signal haben, bevorzugt man verdrillte Kabel mit Abschirmung.

3.2.5 Galvanische Trennung mit Übertrager

Bei Ethernet wird die Galvanische Trennung auf magnetischem Weg mit Übertragern (= kleine Transformatoren) erreicht. Sie wandeln das elektrische Signal in ein magnetisches Feld und und dann wieder zurück in ein elektrisches Signal.



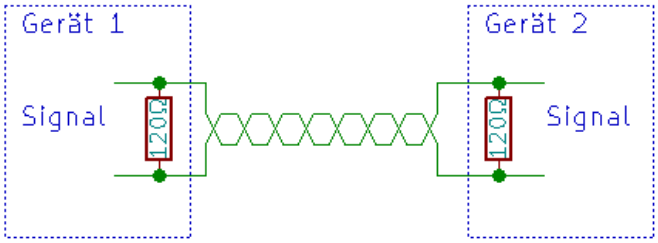
Allerdings stellen die Übertrager umfangreiche Anforderungen an die Signalform und auf der Empfängerseite muss das verzerrte Signal wieder aufbereitet werden. Für Anfänger ist das zu kompliziert.

3.2.6 Symmetrische Übertragung

Durch Symmetrische Signalübertragung reduziert man die Auswirkungen von elektromagnetischen Störungen ganz erheblich. Symmetrische Übertragung ist der Schlüssel zu hohen Übertragungs-Geschwindigkeiten und langen Leitungen.

Symmetrische Übertragung ist bei professioneller Audio-Technik (im Studio und auf der Bühne) alltäglich. Im digitalen Umfeld kann ich Ethernet und RS485 als weit verbreitete Beispiele dieser Technologie nennen. Wobei Ethernet sowohl symmetrisch als auch galvanisch getrennt ist, während RS485 nur symmetrisch aber ohne galvanische Trennung arbeitet.

Im Kapitel über serielle Schnittstellen stelle ich dir einen Mikrochip für RS485 vor.



Der Trick besteht darin, das Nutzsignal auf zwei verdrillten Leitungen zu übertragen, die immer genau entgegen gesetzte Polarität haben. Und dass der Empfänger sich nur für die Spannungs-Differenz zwischen den beiden Leitungen interessiert.

Wenn die Spannung auf der einen Leitung um 1 Volt steigt, dann sinkt die Spannung der anderen Leitung um 1 Volt. Durch diese entgegengesetzten Pegel wirkt das Kabel nach außen neutral. Es strahlt weder elektrische noch magnetische Felder in seine Umgebung ab.

Außerdem ist das Kabel umgekehrt gegen Störung unempfindlich. Denn Störungen, die von außen auf das Kabel einwirken, betreffen beide Leitungen gleichermaßen. Wenn z.B. die Spannungen beider Leitungen wegen einer heftigen Störung um 6 V ansteigen, ändert das nichts an der Differenz der Leitungen zueinander und somit bleibt das Nutzsignal unverfälscht.

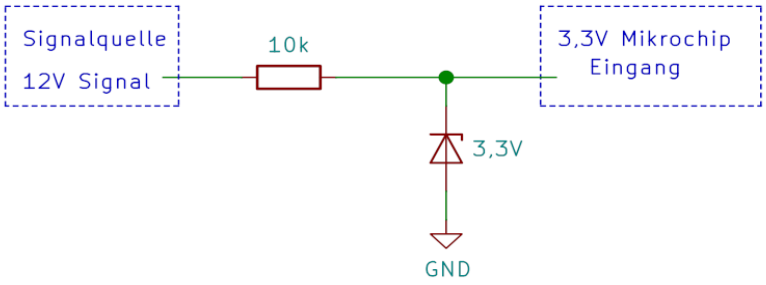
Die beiden Widerstände an den Enden der Leitung nennt man „Abschlusswiderstände“. Sie unterdrücken Reflexionen innerhalb des Kabels, wenn sie den richtigen Wert haben. Die 120 Ω sind ein guter Anhaltswert für handelsübliche Telefon- und Netzwerkkabel.

3.3 Eingangs-Schaltungen

In diesem Kapitel zeige ich dir, wie du Eingänge beschalten kannst, um höhere Spannungen zu akzeptieren.

3.3.1 Zener Diode

Wenn du ein Steuersignal mit hoher Spannung an einen Mikrochip bringen musst, verwende einen Widerstand und eine Zener-Diode.

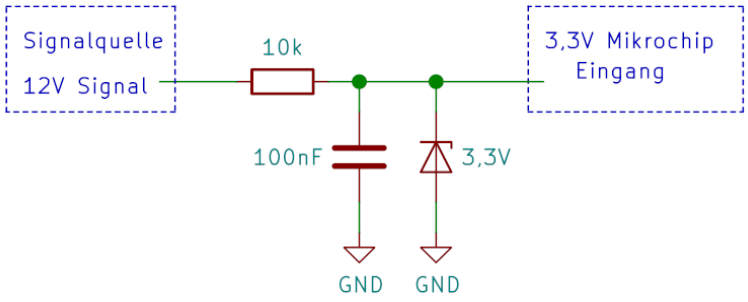


Die Zenerdiode stellt sicher, dass am Mikrocontroller nicht zu viel Spannung ankommt, denn ungefähr ab 3 Volt wird sie leitend. Der 10 kΩ Widerstand verheizt die übrigen 9 Volt und begrenzt dabei die Stromstärke in diesem Fall auf ca. 1 mA. Passe den Widerstand an die höchste erwartete Eingangsspannung an.

$$R = \frac{12\,V - 3{,}3\,V}{1\,mA}$$

Die meisten Zenerdioden sind ab 1mA nutzbar, etwas mehr Strom wäre auch OK. Zener Dioden habe eine relativ hohe Kapazität, so dass sie zusammen mit dem Vorwiderstand einen Tiefpass bilden, der hohe Signal-Frequenzen abschwächt. Auch der Eingang des Mikrochips hat ein paar pF Kapazität, die hier eine Rolle spielen. Solche Schaltungen eignen sich daher nur für Frequenzen unter 1 MHz.

Ein zusätzlicher Kondensator kann Störungen herausfiltern, die durch elektromagnetische Felder in das Kabel eingekoppelt werden (z.B. wenn ein Motor oder eine Leuchtstoffröhre geschaltet wird). Dadurch senkt man die nutzbare Signalfrequenz allerdings noch weiter ab:

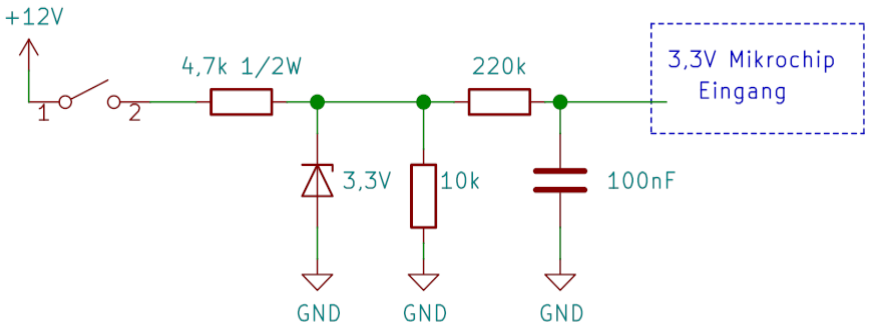


Aufgrund der Größenverhältnisse kann man bei der Berechnung der Grenzfrequenz die wenigen pF von der Zenerdiode und dem Mikrochip vernachlässigen:

$$f = \frac{1}{2 \cdot \pi \cdot R \cdot C} = \frac{1}{2 \cdot \pi \cdot 10\,k\Omega \cdot 100\,nF} = 160\,Hz$$

Die Grenzfrequenz der obigen Schaltung beträgt also 160 Hz. Signale mit dieser Frequenz werden bereits auf die Hälfte abgeschwächt. Erheblich höhere Frequenzen werden damit wirkungsvoll unterdrückt und erheblich niedrigere Frequenzen können genutzt werden.

Für den Anschluss von Schaltkontakten mit höherer Spannung empfehle ich folgende Variante:



Der zusätzliche Pull-Down Widerstand zieht das Signal bei geöffnetem Schaltkontakt auf Low (GND). Zusammen mit dem Eingangswiderstand bildet er einen Spannungsteiler, der den gültigen Bereich für Low Pegel auf etwa 0 - 2 Volt verdoppelt, was für die Störfestigkeit vorteilhaft ist. Für einen High Pegel muss man mindestens 5 Volt anlegen.

Die maximale Eingangsspannung ergibt sich aus der thermischen Belastbarkeit des Eingangswiderstandes, was in diesem Fall ungefähr 70 Volt sind. Wenn hier jemand aus Versehen 230V anschließt, wird mit hoher Wahrscheinlichkeit nur dieser Widerstand abbrennen. Der Rest bleibt heile.

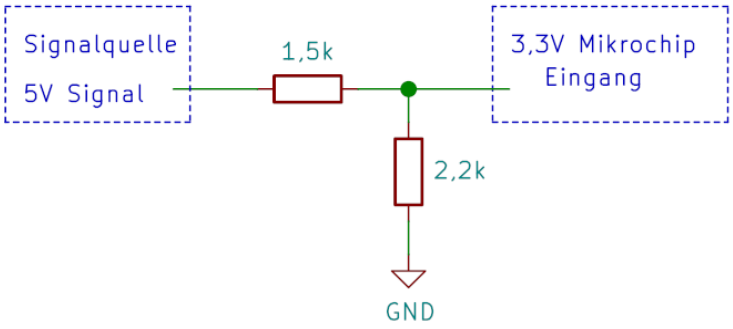
Auf der rechten Seite bildet der 220kΩ Widerstand zusammen mit dem Kondensator einen Tiefpass, der selbst starke elektromagnetische Störungen und auch das Prellen von Kontakten wirksam heraus filtert. Außerdem beschützt er den Mikrocontroller vor allzu großen Stromstärken, falls die

Zenerdiode doch kaputt geht.

Tipp: Anstelle der Zenerdiode kann man auch eine weiße oder blaue LED verwenden.

3.3.2 Spannungsteiler

Alternativ zur Zenerdiode geht auch ein Spannungsteiler. Wenn man kleine Widerstandswerte verwendet, eignen sie sich für höhere Frequenzen bis in den einstelligen MHz Bereich. Allerdings bietet diese Schaltung nur geringen Schutz vor Überspannung und Verpolung.

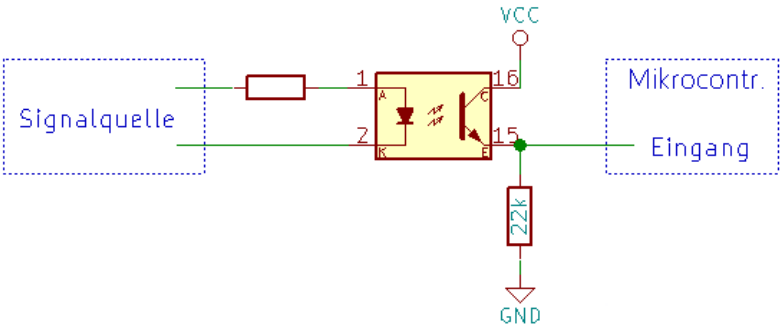


Das Verhältnis der Widerstände muss an die Eingangsspannung angepasst werden. Die Berechnungsformel für das Verhältnis der Widerstände lautet:

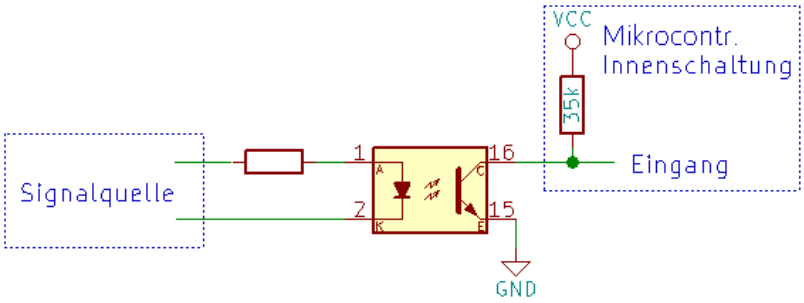
$$\frac{U_{signal}-3{,}3V}{3{,}3V}=\frac{R1}{R2}$$

3.3.3 Optokoppler

Optokoppler enthalten Eingangsseitig eine Leuchtdiode und Ausgangsseitig einen Transistor. Der Transistor schaltet durch, wenn die Leuchtdiode leuchtet. Sie können digitale Signale bis etwa 100 kHz übertragen.



Oder so:



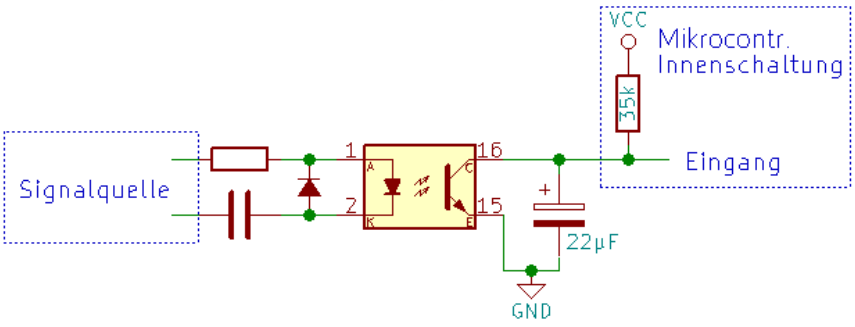
Bei der oberen Schaltung geht der Eingang des Mikrocontrollers auf High, wenn die LED im Optokoppler leuchtet. Bei der unteren Schaltung geht der Eingang stattdessen auf Low, wenn die LED leuchtet.

1mA Stromstärke durch die LED genügt. Die LED‘s in Optokopplern haben 1,3 V Betriebsspannung, so dass du den Vorwiderstand nach folgender Formel berechnen kannst:

$$R \approx \frac{U_{signal} - 1{,}3\,V}{0{,}001\,A}$$

3.3.4 Optokoppler an Wechselspannung

Optokoppler kann man auch an Wechselspannung betreiben. Am einfachsten geht es so:



Der Kondensator vor dem Optokoppler regelt die Stromstärke. Der Widerstand begrenzt die Stromstärke im Einschaltmoment. Bei Netzspannung (230V) nehme ich immer 100 nF und 220 Ohm. Der Kondensator lässt bei 50 Hz etwa 7 mA fließen.

Der Wechselstrom-Widerstand des Kondensators ist:

Rc = 1 : (2 · 3,14 · 50 Hz · C)

In diesem Fall: Rc = 1 : (2 · 3,14 · 50 Hz · 0,0000001 F) = 32 kΩ

Und die Stromstärke ist: I = U : (Rc + Rv)

In diesem Fall: I = 230 V : (32000 Ω + 220 Ω) = 0,007 A

Da die LED des Optokoppler nur in eine Richtung leitet, muss eine weitere Diode für die andere Stromrichtung davor gesetzt werden. Dazu kann man wahlweise eine LED oder eine nicht leuchtende Diode verwenden, wie die 1N4148.

Der Transistor des Optokopplers schaltet nur bei jeder positiven Halbwelle durch. Die Pausenzeiten dazwischen überbrückt der 22 µF Kondensator..

Bei Netzspannung muss der linke Kondensator für mindesten 630V ausgelegt sein, damit er die üblichen Spitzen-Spannungen des Stromnetzes aushält. Die Industrie setzt hier oft 400 V Kondensatoren ein, was zu knapp bemessen ist. Sie gehen oft kaputt. Mit 630 V habe ich gute

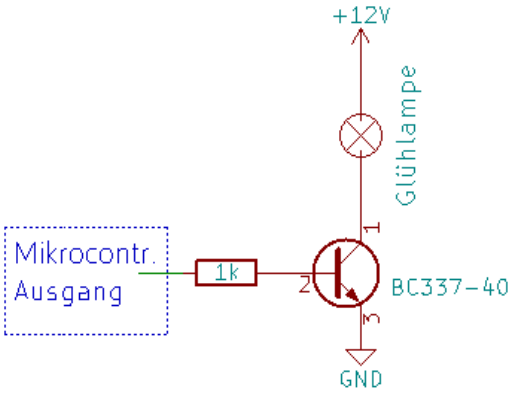
Erfahrungen gemacht, viele Elektroniker empfehlen sogar 1000 V.

3.4 Ausgänge verstärken

Die Ausgänge von Mikrocontrollern sind stark genug, um einzelne Leuchtdioden anszusteuern. Aber für Glühlampen, Motoren, Relais, usw. benötigen Sie Verstärkung durch Transistoren. Hierzu eignen sich sowohl bipolare, als auch MOSFET Transistoren.

3.4.1 NPN Transistor

Am häufigsten findet man einen einfachen NPN Transistor vor. Der BC337-40 eignet sich für Lasten bis zu 45 Volt und 0,5 Ampere.



Wenn der Ausgang des Mikrocontrollers einen High-Pegel liefert, schaltet der Transistor durch, so dass die Lampe leuchtet.

Den Widerstand vor der basis musst du an den maximal erwarteten Strom anpassen, damit der Transistor „gut genug“ leitet. Wenn du R zu hoch ansetzt, leitet der Transistor nur halb und wird heiß.

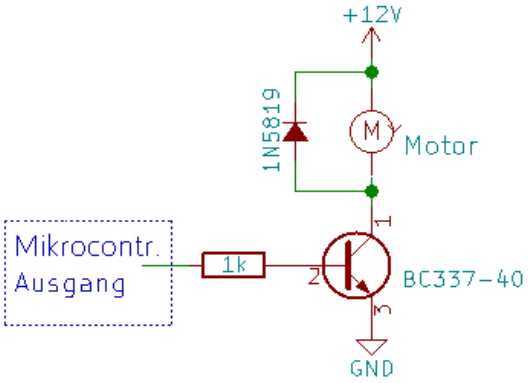
Der oben gezeigte Transistor kann den Steuerstrom mindestens um Faktor 250 verstärken. Setze in der folgenden Formel zur Berechnung des Vorwiderstandes also für β die Zahl 250 ein und für I_{last} die Stromstärke der Last (in diesem Fall die Glühlampe).

$$R = \beta \cdot \frac{3{,}3\,V - 0{,}7\,V}{I_{Last}}$$

Für den BC337-40 und 500 mA Laststrom soll der Vorwiderstand nach dieser Formel 1300 Ohm haben. Besser ist wesentlich weniger (ein Drittel), um den Transistor zu übersteuern und somit die Verlustwärme zu reduzieren.

3.4.1.1 Freilaufdiode

Wenn die Last eine Spule enthält (wie z.B. bei einem Relais oder einen Motor), brauchst du zusätzlich eine sogenannte Freilauf-Diode.



Die Diode wird in dem Moment wirksam, wo der Transistor abschaltet. Die Spule im Innern des Motors erzeugt dabei nämlich einen sehr kurzen aber hohen Spannungsimpuls mit umgekehrter Polarität, welche ohne Schutzvorkehrung den Transistor zerstören würde. Durch die Freilauf-Diode wird der Impuls kurzgeschlossen. Ohne Freilaufdiode würde der Transistor kaputt gehen.

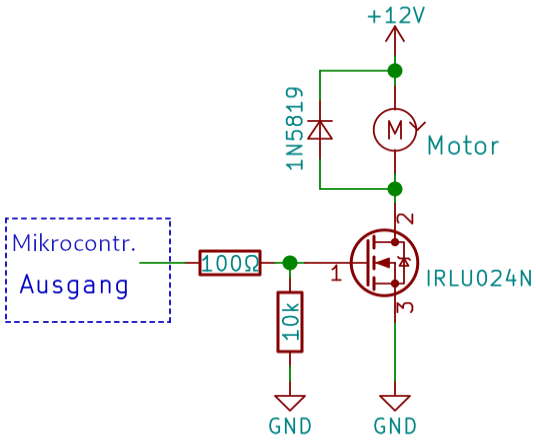
Die Diode soll man so auslegen, dass sie den normalen Betriebsstrom der Last aushalten kann. Außerdem muss die Diode für die Schaltfrequenz geeignet sein. Zu jeder Diode gibt es ein Datenblatt, wo die Schaltgeschwindigkeit oder zumindest der vorgesehene Anwendungsbereich angegeben ist.

Für kleine Ströme bis 200 mA verwende ich die 1N4148. Diese Diode ist sehr schnell und billig. Für bis zu 1A empfehle ich die 1N5819.

Die preisgünstige 1N400x Reihe eignet sich wegen ihrer relativ großen Trägheit nur für niedrige Schaltfrequenzen bis 100 Hz.

3.4.2 N-Kanal MOSFET

MOSFET Transistoren sind beliebt geworden, weil sie hohe Ausgangs-Lasten schalten können und (zumindest bei niedrigen Frequenzen) praktisch gar keinen Steuerstrom brauchen.



Ich hatte bereits weiter oben darauf hingewiesen, dass MOSFET Transistoren eine relativ große Gate-Kapazität haben. Der 100 Ω Widerstand begrenzt dabei den Ladestrom, so dass der Mikrocontroller und die Spannungsversorgung nicht übermäßig belastet werden.

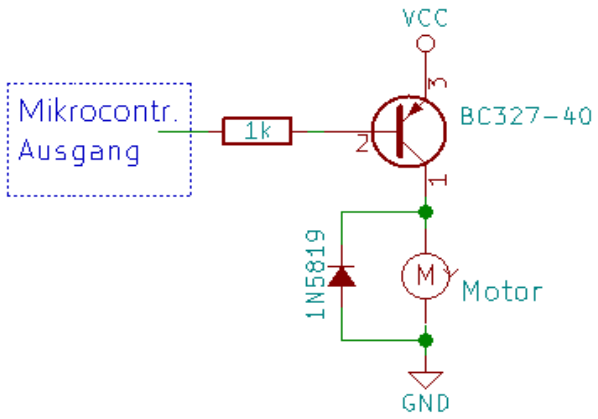
Der 10 kΩ Widerstand verhindert einen undefinierten Zustand in der Reset-Phase des Mikrocontrollers. Solange das Programm den I/O Pin noch nicht als Ausgang konfiguriert hat, würde die Schaltung ohne diesen Widerstand für elektrische Felder aus der Luft empfänglich sein. Es könnte passieren, dass der MOSFET unbeabsichtigt ein schaltet. Schlimmer wäre jedoch, wenn er nur halb einschalten würde, denn dann wird er heiß – unter Umständen zu heiß.

Der 10 kΩ Widerstand zieht die Gate-Leitung auf Low, solange der Anschluss des AVR Mikrocontrollers noch nicht als Ausgang konfiguriert ist. Somit bleibt der Transistor zuverlässig ausgeschaltet.

Bei Frequenzen über 1kHz würde ich zu einem „MOSFET Driver IC“ raten, die können das Gate schneller umladen und dadurch Verluste (Wärme) reduzieren. Wenn die Last eine Spule enthält (wie z.B. ein Relais oder ein Motor) benötigt man in der Regel zusätzlich eine Freilaufdiode.

3.4.3 PNP Transistor

PNP Transistoren benutzt du, wenn unbedingt der Plus-Pol geschaltet werden muss. Solange die zu schaltende Spannung mit der Versorgungsspannung des Mikrocontrollers identisch ist, geht das ganz einfach:



Wenn der Ausgang des Mikrocontrollers einen Low-Pegel liefert, schaltet der Transistor durch, so dass sich der Motor dreht.

Den Widerstand R musst du an den maximal erwarteten Strom anpassen, damit der Transistor „gut genug“ leitet. Wenn du R zu hoch ansetzt, leitet der Transistor nur halb und wird heiß.

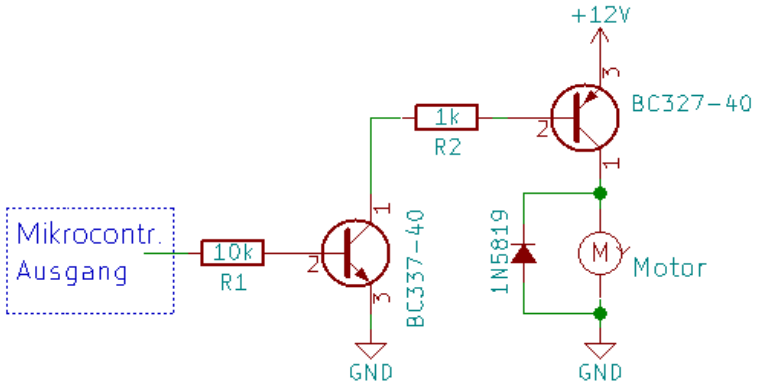
Der oben gezeigte Transistor kann den Steuerstrom mindestens um Faktor 250 verstärken. Setze in der folgenden Formel zur Berechnung des Vorwiderstandes also für β die Zahl 250 ein und für I_{last} die Stromstärke der Last (in diesem Fall der Motor).

$$R = \beta \cdot \frac{3{,}3V - 0{,}7V}{I_{Last}}$$

Für den BC327-40 und 500 mA Laststrom soll der Vorwiderstand nach dieser Formel 1300 Ohm haben. Besser ist wesentlich weniger (ein Drittel), um den Transistor zu übersteuern und somit die Verlustwärme zu reduzieren. Wenn die Last eine Spule enthält (wie z.B. ein Relais oder ein Motor) benötigt man zusätzlich eine Freilaufdiode.

3.4.3.1 PNP bei höherer Spannung

Meistens ist die zu schaltende Spannung allerdings viel höher, zum Beispiel 12 Volt. In diesem Fall kommst du nicht umhin, zwei Transistoren zu kombinieren.

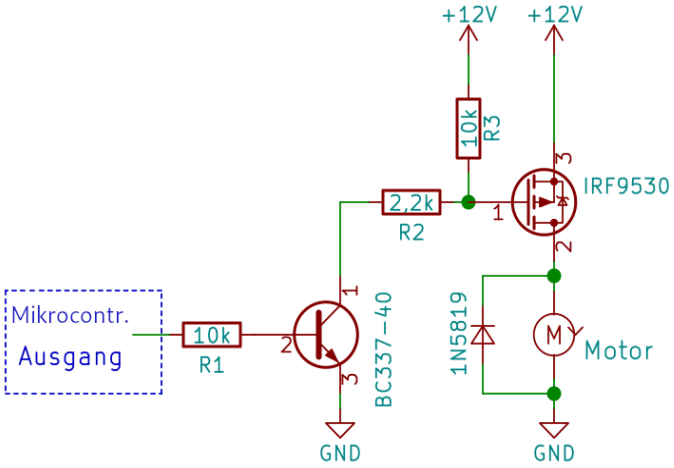


Der erste Transistor ermöglicht uns, eine höhere Spannung zu schalten. Sein Kollektor verträgt bis zu 45 V. Es gibt natürlich andere Transistoren die noch mehr Spannung vertragen. Der Zweite Transistor schaltet den Laststrom.

Den Widerstand R2 musst du an den maximal erwarteten Last-Strom anpassen, wie bereit im vorherigen Kapitel erklärt. Für den Widerstand R1 kannst du einfach immer 4,7 oder 10 kΩ verwenden. Wenn die Last eine Spule enthält (wie z.B. ein Relais oder ein Motor) benötigt man zusätzlich eine Freilaufdiode.

3.4.4 P-Kanal MOSFET

P-Kanal MOSFET Transistoren verwendest du, wenn unbedingt der Plus-Pol geschaltet werden muss und die Stromstärke hoch ist. Die folgende Schaltung ähnelt der vorherigen Variante mit PNP Transistor. Wir benutzen wieder einen Transistor um höhere Spannungen schalten zu können und dahinter einen Zweiten, der den Plus-Pol der Last durch schaltet.



Wenn der Mikrocontroller einen High-Pegel ausgibt, schaltet T1 durch. Er zieht die Spannung am Gate des MOSFET Transistors nach unten, so dass dieser einschaltet.

Der Spannungsteiler (R2/R3) soll so ausgelegt werden, dass der MOSFET sicher einschaltet, aber auch nicht überlastet wird. Für die meisten MOSFET‘s sind 10 V ideal. Aber schau dazu sicherheitshalber ins Datenblatt des MOSFET Transistors.

$$R3 = R2 \cdot \frac{U_{Last} - 10\,V}{10\,V}$$

Wegen der Gate-Kapazität sollte dieser Spannungsteiler nicht allzu hochohmig sein, denn sonst schaltet der MOSFET Transistor zu träge ein und wird heiß. Die oben gezeigten Widerstände eignen sich für wenige Schaltvorgänge pro Sekunde. Wenn du jedoch häufiger zwischen ein/aus wechselst, solltest du kleinere Widerstände wählen. Für eine Drehzahlregelung mittels PWM würde ich 220 Ω und 1 kΩ versuchen. Bei Frequenzen über 1kHz würde ich zu einem „MOSFET Driver IC“ raten, die können das Gate schneller umladen und dadurch Verluste (Wärme) reduzieren.

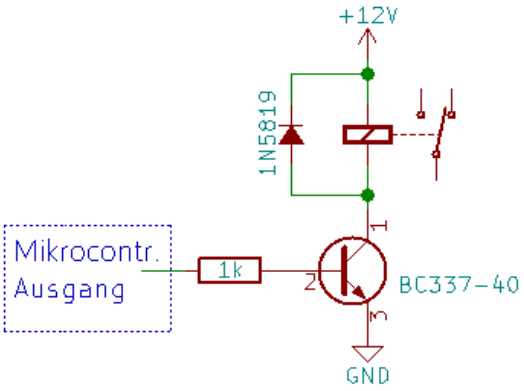
Wenn die Last eine Spule enthält (wie z.B. ein Relais oder ein Motor) benötigt man in der Regel zusätzlich eine Freilaufdiode.

3.5 Relais

Relais sind eine einfache und narrensichere Lösung, um Lasten mit beliebiger Spannung und Strom zu schalten. Relais enthalten eine Magnet-Spule, die einen oder mehrere mechanische Schalter betätigt. Sie können allerdings nur langsam umschalten, in der Regel nicht viel schneller als 5 mal pro Sekunde.

Bei Wikipedia ist die Funktionsweise von Relais sehr schön beschrieben:
https://de.wikipedia.org/wiki/Relais

Relais kannst du nicht direkt mit dem Mikrocontroller ansteuern, weil ihre Spulen zu viel Strom aufnehmen. Du benötigst zur Verstärkung immer einen Transistor und eine Freilaufdiode.



Unabhängig vom Relais-Modell kannst du stets den Transistor BC337-40 mit einem Vorwiderstand von 1 bis 2,7 kΩ verwenden. Als Freilauf Diode eignet sich beinahe jede beliebige Diode, z.B. die 1N4148 oder die 1N5819.

Achte beim Kauf von Relais auf die richtige Spulen-Spannung und natürlich auch darauf, dass die Kontakte des Relais für die Last geeignet sind.

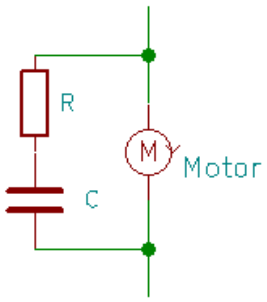
3.5.1 Funkenlöschkreis

Wenn die Last eine Spule enthält, wie es bei Motoren, Transformatoren und vielen Leuchtstofflampen der Fall ist, entstehen beim Abschalten kräftige Funken zwischen den Kontakten des Relais. Das passiert bei geringen Spannungen aus Batterien ebenso, wie bei Netzspannung.

Die Funken entstehen, weil die Spule im Abschaltmoment ihr magnetisches Feld abbaut, was einen kurzen Hochspannungs-Impuls auslöst.

Wie empfindlich Relais auf Funken reagieren, hängt sehr von ihrer Bauart ab. Manche Relais sind so robust, dass ihnen die Funken gar nichts ausmachen. Für durchschnittliche Relais sind Funken allerdings langfristig schädlich, da ihre Hitze die metallische Oberfläche der Kontakte beschädigt.

Bei Lasten mit Gleichspannung unterdrückt man die Funken meist mit einer Freilaufdiode, wie in den vorherigen Kapiteln gezeigt. Bei Lasten mit Wechselspannung muss man jedoch die deutlich teurere „Snubber“ Schaltung anwenden:



Der Kondensator nimmt die Energie auf, welche die Motor-Spule beim Abschalten des Stromes erzeugt. Der Widerstand begrenzt die Stromstärke im Moment des Einschaltens.

Große Spulen brauchen große Kondensatoren. Solange du die komplexe Mathematik noch nicht kennst, die dahinter steckt, benutze folgende Bauteile für 230 Volt:

R = 220 Ω, ¼ Watt

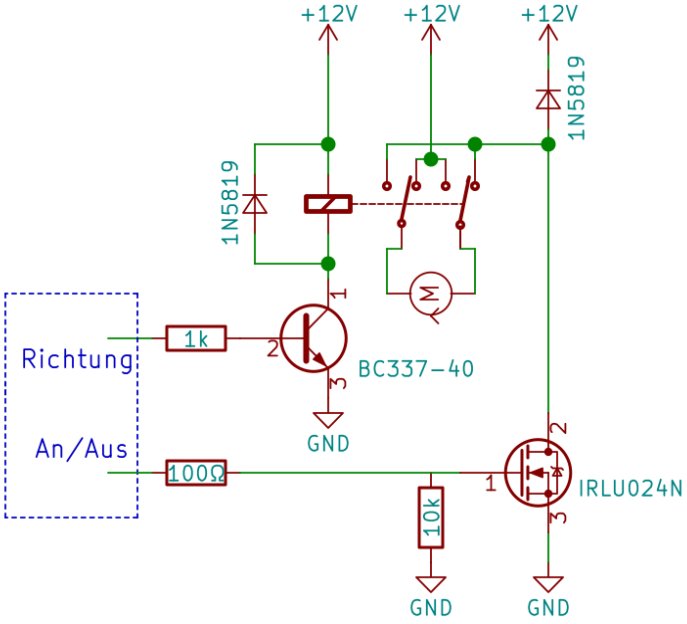
C = 100 nF 630 V

Für 230 V~ bräuchte man rein theoretisch einen Kondensator mit wenigstens 350 V Belastbarkeit, denn so hoch sind die regelmäßigen Spitzen der Wechselspannung. Aber im Stromnetz hat man immer wieder Störimpulse, die erheblich höher sind, deswegen rate ich zu Kondensatoren, die wenigstens 630 V vertragen. Besser wären sogar 1000 V, aber die sind teuer und schwer zu bekommen.

Diese Kombination nennt man auch R-C Glied. Man kann sie als kombiniertes Bauteil kaufen, dann ist die Höhe der Wechselspannung aufgedruckt, für die sie ausgelegt sind, z.B. 250 V~.

3.5.2 Relais als Richtungs-Umschalter

Ich benutze Relais gerne, um die Laufrichtung von Motoren umzuschalten. Zwar sind Relais zugegebenermaßen nicht sehr modern, aber sie sind unkompliziert und robust. Um die Stromrichtung für einen Motor umzuschalten brauchst du ein Relais mit zwei Umschaltkontakten. In Katalogen werden sie mit „2xUM“ gekennzeichnet.



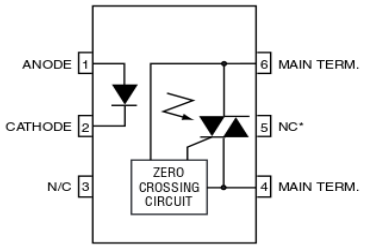
Wenn die beiden Schalter im Relais so stehen, wie ich es hier gezeichnet habe, dann liegt am Motor links der Plus-Pol an und rechts der Minus-Pol.

Der Mikrocontroller kann das Relais über den oberen Transistor umschalten, dann fließt der Strom anders herum durch den Motor.

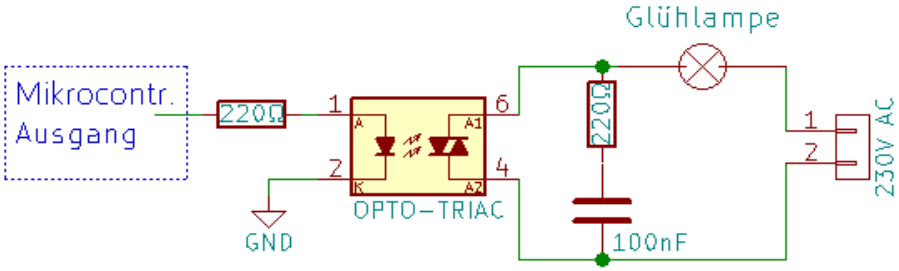
Durch den unteren Transistor schaltet der Mikrocontroller den Motor an und aus. Mit einem PWM Signal kann man auch die Drehzahl steuern.

3.5.3 Opto-Triac

Wechselspannungen kann man mit Relais schalten, aber auch mit rein elektronischen Bauteilen schalten. Zum Beispiel mit Optio-Triacs.



Ich verwendet gerne den MOC 3063 oder MC P3063. Er sieht aus wie ein Optokoppler in 6-poligem Gehäuse und so ähnlich funktioniert er auch.



Die Leuchtdiode im Eingang des Opto-Triacs benötigt einen Strom von etwas mehr als 5 mA. Ausgangsseitig kann er bis zu 100 mA schalten. Bei 230 Volt ist er somit für Lasten bis zu 23 Watt geeignet.

Zum Schutz vor Spannungsspitzen solltest du Opto-Triacs immer mit einem Funken-Löschkreis versehen. Der Kondensator im Funkenlöschkreis sollte für mindestens 630 V (besser 1000 V) ausgelegt sein.

3.5.4 Halbleiter Relais

Für größere Lasten verwendest du ein Halbleiter Relais (SSR), wie zum Beispiel das S202SE2 von Sharp, welches bis zu 8A belastbar ist, oder das S216SE2 welches 16 A verträgt. Beide Relais haben eine Leuchtdiode im Eingang, die mit etwas mehr als 8mA angesteuert werden muss.

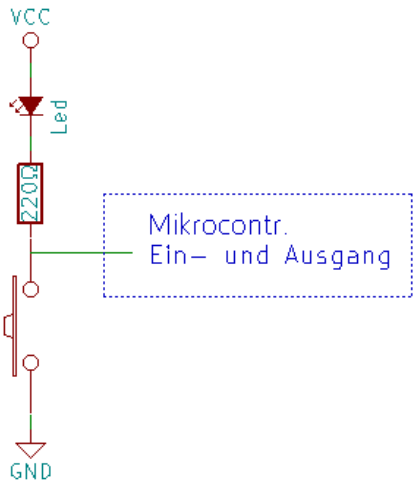
Zum Schutz vor Spannungsspitzen solltest du sie genau wie Opto-Triacs immer mit einem Funken-Löschkreis versehen.

3.6 Anschlüsse mehrfach belegen

Wenn die verfügbaren Anschlüsse des Mikrocontrollers knapp werden, kann man unter bestimmten Voraussetzungen Anschlüsse mehrfach belegen. Zur Anregung zeige ich dir hier ein paar Beispiele.

3.6.1 Taster und Leuchtdiode

Man kann ganz einfach einen Taster und eine Leuchtdiode zusammen legen, und zwar so:



Den Pin des (AVR) Mikrocontrollers konfigurierst du als Ausgang und setzt ihn auf Low, wenn die LED leuchten soll.

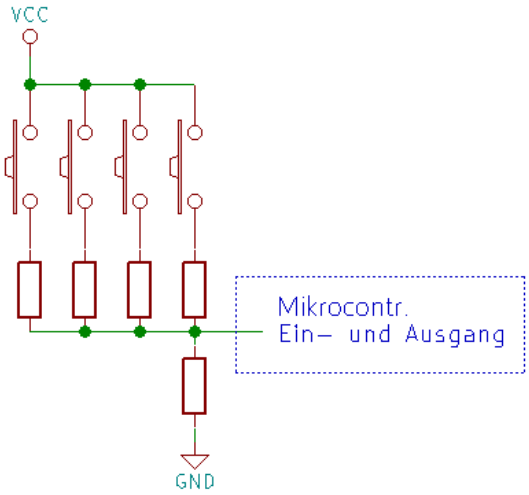
Um den Taster abzufragen, konfigurierst du den Anschluss als Eingang mit Pull-Up Widerstand. Du erhältst ein Low-Signal, wenn der Taster gedrückt wurde, und ein High-Signal, wenn er nicht gedrückt wurde. Direkt nach der Abfrage stellst du den Anschluss wieder als Ausgang ein, mit dem vorherigen Signalpegel, damit die LED ggf. wieder leuchtet.

Wenn du den Taster in regelmäßigen kurzen Intervallen mit genügend Zeitabstand abfragst, merkt man gar nicht, dass die LED dabei flackert. 20 Millisekunden Zeitabstand zwischen den Abfragen wäre zum Beispiel geeignet.

Der Nachteil dieser Schaltung ist, dass die LED immer an geht, wenn man den Taster drückt.

3.6.2 Taster am analogen Eingang

Analoge Eingänge kann man ganz einfach mit mehreren Tastern gleichzeitig belegen. Ich habe diese Idee von meiner Hifi-Anlage ab-geguckt:



Jeder Taster bekommt einen anderen Widerstand. Zum Beispiel:

Oben: 1 kΩ 2,2 kΩ 3,3 kΩ 4,7 kΩ …

Unten: 10 kΩ

Wenn kein Taster gedrückt ist, liegt der Eingang des Mikrocontroller aufgrund des Pull-Down Widerstandes auf null Volt. Je nachdem, welcher Taster gedrückt ist, erhält man eine andere Spannung und dementsprechend einen anderen Messwert vom ADC.

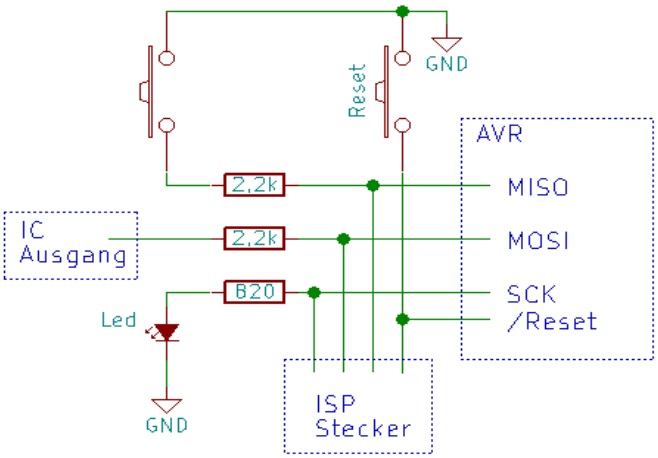
$$ADC = 1024 \cdot \frac{R_{unten}}{R_{oben} + R_{unten}}$$

Ein bisschen Abweichung zum berechneten Wert sollte dabei allerdings von der Software zugelassen werden, weil der tatsächliche Wert immer ein klein Wenig vom Idealen Wert abweicht.

Der Nachteil dieser Schaltung ist, dass die Abfrage der Taster in der Software aufwändiger ist und mehr Zeit in Anspruch nimmt, als Taster am digitalen Eingang.

3.6.3 ISP Schnittstelle doppelt belegen

Bei AVR Mikrocontrollern kann man die Programmierschnittstelle doppelt belegen. Durch Widerstände gibt man dem ISP Programmieradapter Vorrang, wenn er aktiv ist. Ansonsten können die Pins wie gewöhnliche I/O Pins verwendet werden. Das folgende Beispiel zeigt dies für drei unterschiedliche Fälle:



Alle vier Leitungen der SPI Schnittstelle (Reset, MISO, MOSI, SCK) dürfen nur gering belastet werden. Eine LED mit reduziertem Strom von 2 mA ist gerade noch in Ordnung, aber eine LED mit vollen 20mA würde die Leitungen zu stark belasten. Dann geht zwar nichts kaputt, aber es würde den Programmiervorgang stören.

Anschlüsse, die du als Eingang benutzt, musst du mit Widerständen entkoppeln. So hat der ISP Programmieradapter Vorrang, falls er angeschlossen ist. Ich habe mit 2,2 kΩ gute Erfahrungen gemacht.

Taster am Reset-Pin dürfen ausnahmsweise ohne Entkoppelungs-Widerstand angeschlossen werden, weil der ISP-Programmieradapter den Reset-PIN niemals auf High zieht. Ein Kurzschluss kann nicht entstehen, selbst wenn du den Taster während des Programmiervorganges drückst.

Du solltest auch berücksichtigen, dass die Leitungen der ISP Schnittstelle während der Programmierung beinahe zufällige Signale führen. Was auch immer du an diese Leitungen anschließt, sollte dies tolerieren. Wenn eine LED wild flackert ist das sicher in Ordnung. Aber wenn an diesen Leitungen beispielsweise eine Speicherkarte angeschlossen ist, könnte dessen Inhalt verändert werden, was vielleicht nicht gewünscht ist.

4 Stromversorgung

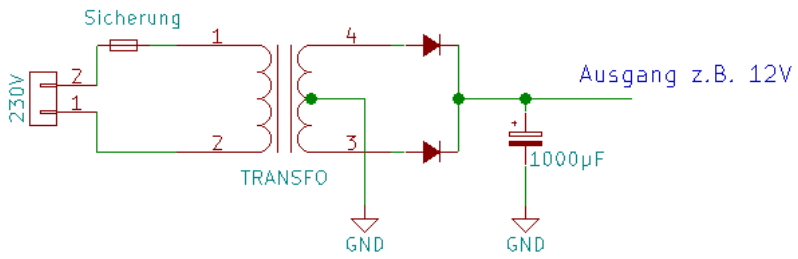
Auch wenn du in der Regel fertige Netzteile aus Massenproduktion verwendest, kann es nützlich sein, ganz grob zu wissen, wie sie funktionieren.

4.1 Transformator-Netzteile

Transformator-Netzteile wandeln die Spannung aus der Steckdose mit zwei Spulen und einem dicken Eisenkern um, also auf magnetischem Wege. Die Vorteile von Transformator-Netzteilen sind:

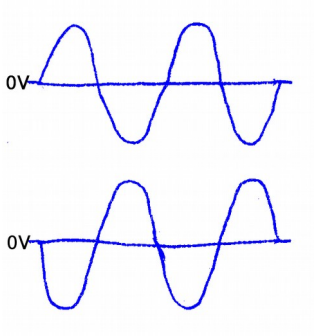
- Sehr zuverlässige Funktion. Liefert im Fehlerfall keine überhöhte Ausgangsspannung.
- Unempfindlich gegen kurzzeitige Überlastung.
- Filtert hochfrequente Störsignale und Impulse sehr Wirkungsvoll heraus, und zwar in beide Richtungen.

Dafür sind Transformatoren leider teuer und groß. Ein einfaches ungeregeltes Transformator-Netzteil kann so aufgebaut sein:

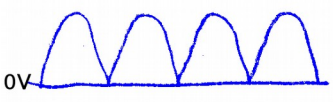


Der Transformator wandelt die Wechselspannung aus der Steckdose in ein Magnetfeld um, welches auf seiner Ausgangs-Seite wiederum in zwei geringere Ausgangsspannungen umgewandelt wird. Das hast du wahrscheinlich schon in der Schule gelernt.

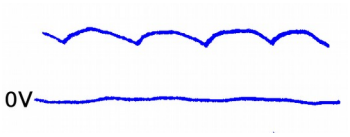
Die beiden Ausgangsspannungen des Transformators haben entgegen gesetzte Polarität. Wenn der eine Ausgang eine positive Spannung abgibt, gibt der andere gerade eine negative Spannung ab.



Es folgen zwei Dioden, die jeweils nur die positiven Halbwellen durchlassen. Am Ausgang der Dioden erhalten wir daher eine pulsierende Gleichspannung.



Der Kondensator lädt sich damit auf und füllt die Lücken, so dass am Ausgang des Netzteils letztendlich eine Spannung heraus kommt, die etwa so aussieht:



Je stärker das Netzteil belastet wird, umso ausgeprägter werden die Wellen. Für Mikrocontroller ist ein solches Netzteil nicht gut genug. Aber für Beleuchtungen und Motoren reicht es. Für Mikrocontroller-Schaltungen benutzt man zusätzlich einen Spannungsregler oder man verwendet ein geregeltes Schaltnetzteil.

4.2 Schaltnetzteile

Schaltnetzteile erhöhen die Netzfrequenz künstlich mit einem Schalt-Transistor und benötigen daher nur einen viel kleineren Transformator. Angesichts der steigenden Kosten für Metalle erfreuen sich Schaltnetzteile daher zunehmender Beliebtheit. Ihre Vorteile sind:

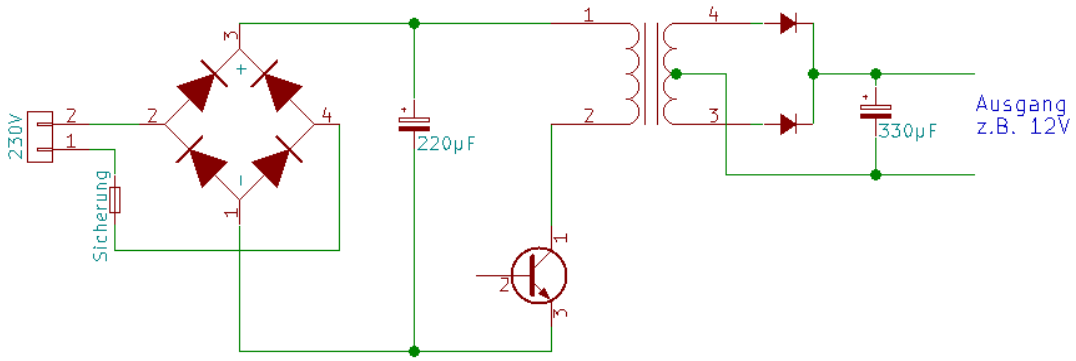
- klein
- leicht
- konstante geregelte Ausgangsspannung
- Billiger, weil weniger Metall benötigt wird

Aber Schaltnetzteile haben auch Nachteile:

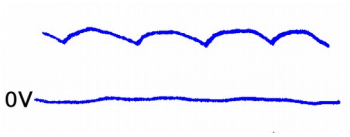
- Störanfällig, da viel mehr Bauteile drin sind.
- Empfindlich gegen Spannungsspitzen im Stromnetz.
- Gefährlicher, da wesentliche Teile der Schaltung unter Netzspannung stehen.
- Wenn die Regelung versagt, kann das Netzteil eine erheblich überhöhte Ausgangsspannung abgeben, die das angeschlossene Gerät zerstört.
- Geringe Fähigkeit, hochfrequente Störsignale zu filtern.
- Neigt selbst dazu, hochfrequente Störsignale zu erzeugen.

Aus dem zuletzt genannten Grund rate ich dringend davon ab, Schaltnetzteile selbst zu bauen. Die korrekte Funktion ist nur mit teurer Messtechnik überprüfbar, die kaum ein Hobby-Elektroniker finanzieren kann. Ein mangelhaftes Schaltnetzteil kann die Funktion anderer elektrischer Geräte in der Nachbarschaft massiv stören – sogar Handies.

Darum zeige ich hier auch keinen vollständigen Schaltplan, sondern nur ein Prinzip-Schaltbild. An diesem Schaltbild kannst du ablesen, wie so ein Netzteil prinzipiell funktioniert.



Im linken Bereich wird die Netzspannung durch vier Dioden gleichgerichtet und damit ein dicker Kondensator aufgeladen. Der Spannungsverlauf an dem linken Kondensator entspricht einem herkömmlichen Transformator-Netzteil:



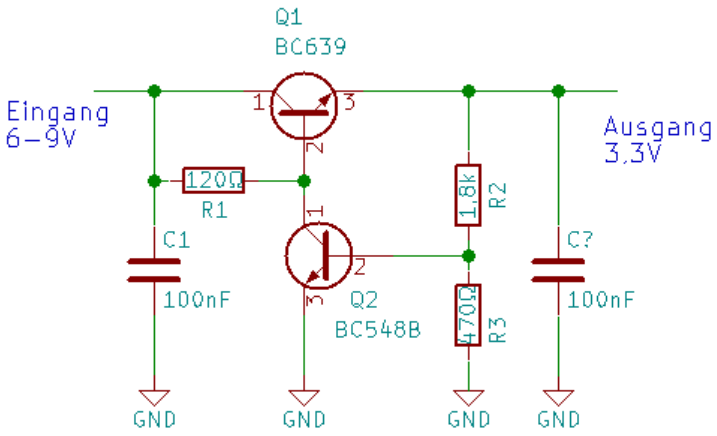
Allerdings haben wir es hier mit Werten von knapp über 300 Volt zu tun! Der Schalt-Transistor wird so angesteuert, dass er mit einer hohen Frequenz (meist im Bereich 10 kHz ... 100 kHz) ein und aus schaltet geht. Die so „zerhackte“ Spannung wird einem kleinen Transformator zugeführt, der sie auf eine geringere Spannung übersetzt. Die Spannung am Ausgang des Transformators wird wie bei einem herkömmlichen Transformator-Netzteil nochmal gleichgerichtet und mit einem weiteren Kondensator stabilisiert.

Ein hier nicht eingezeichneter Mikrochip überprüft ständig die Ausgangsspannung und beeinflusst die Schaltzeiten des Transistors so, dass eine konstante Ausgangsspannung entsteht.

Die Welligkeit der Ausgangsspannung ist bei guten Schaltnetzteilen viel geringer als bei herkömmlichen Transformator-Netzteilen. Gute Schaltnetzteile enthalten außerdem einen Schutz vor Überlast.

4.3 Lineare Spannungsregler

Lineare Spannungsregler benutzt man hinter einem Netzteil oder einer Batterie, um eine konstante Versorgungsspannung ohne Wellen zu erhalten. Die folgende Schaltung zeigt einen funktionsfähigen Linear-Regler, den du genau so aufbauen könntest:



Q1 lässt zunächst den maximalen Strom fließen. Dadurch lädt sich der Kondensator am Ausgang auf und die Ausgangsspannung steigt an. Wenn die Ausgangsspannung 3,3 Volt überschreitet, liegt an der Basis vom Q2 eine Spannung von 0,7 Volt an, so dass dieser leitfähig wird.

Der Q2 entzieht dann dem Q1 einen Teil seines Steuerstromes, so dass der nun weniger Last-Strom fließen lässt. Dadurch sinkt die Ausgangsspannung des Reglers.

Fällt die Ausgangsspannung unter 3,3 Volt, dann liegt an der Basis des Q2 weniger Spannung an, so dass seine Leitfähigkeit sinkt. Er lässt dem Q1 wieder mehr Steuerstrom übrig. Der Q1 lässt daher wieder etwas mehr Last-Strom fließen.

Auf diese Weise regelt diese Schaltung eine konstante Ausgangsspannung. Beide Transistoren arbeiten analog mit kontinuierlich veränderlichen Spannungen und Strömen.

Linear-Regler sind unkompliziert zu handhaben. Nachteilig ist lediglich, dass sie die Überschüssige Spannung „verheizen“. Es entsteht relativ viel Abwärme.

Es gilt die Formel: $P = \frac{Uin - Uout}{I}$

Wenn beispielsweise die Eingangsspannung 9 V beträgt und der Regler mit 100mA belastet wird, verheizt der Transistor bereits 0,57 Watt, was nahe an seiner Belastbarkeit-Grenze liegt. Für größere Ströme braucht man zwangsläufig größere Transistoren mit Kühlkörper.

Im Handel gibt es praktische Linear-Regler als Mikrochip. Sie arbeiten noch erheblich präziser, als die obige Schaltung und sie sind meistens sowohl gegen Überhitzung als auch gegen Kurzschluss geschützt. Folgende Linear-Regler benutze ich gerne:

Typ	Ausgang-Spannung	Strom max.	Eingangs-Spannung
LM317	1,2...37 V (einstellbar)	1,5 A	3,2…40 V min. 2 V mehr als die Ausgangsspannung.
LM7833	3,3V	1 A	5,3 V .. 25 V
LF33CV	3,3 V	500 mA	3,8..16 V
LP2950-3.3	3,3 V	100 mA	3,7...30 V
LM7805	5 V	1 A	7...25 V
LF50CV	5 V	500 mA	5,5..16 V
LP2950-5.0	5,0 V	100 mA	5,4..30 V

Die LP und LF Regler haben einen besonders geringen Stromverbrauch von typischerweise 75 µA und sie brauchen nur etwa 0,5 V mehr Eingangsspannung, was ihn für Batteriebetrieb attraktiv macht.

Alle diese Regler benötigen 1-2 externe Kondensatoren , um zuverlässig zu funktionieren. Schau in das jeweilige Datenblatt, um die richtigen Werte für die Kondensatoren zu erfahren.

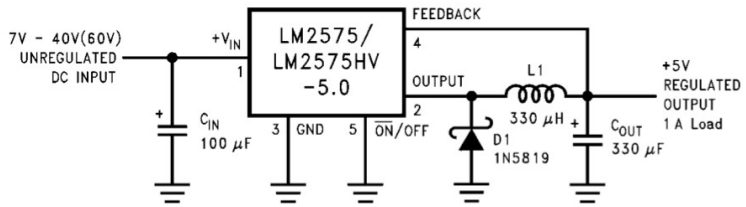
4.4 Step-Down Wandler

Step-Down Wandler benutzt man, um eine wesentlich höhere Gleichspannung herab zu setzen. Wenn die Eingangsspannung mehr als 4 Volt höher ist, als die gewünschte Ausgangsspannung, lohnt sich die Verwendung eines Step-Down Wandler.

Im Gegensatz zu Linear-Reglern verheizen Step-Down Wandler die überschüssige Spannung nicht, sondern transformieren sie mit Hilfe einer Spule. Ihr Wirkungsgrad liegt typischerweise bei ca. 80%.

Das Funktionsprinzip habe ich bereits im Kapitel über Spulen beschrieben: Die Spule wird immer abwechselnd mit der Eingangsspannung aufgeladen und dann in den Ausgangskondensator entladen.

Das folgende Bild kommt aus dem Datenblatt des Schaltreglers LM2575-5.0:



Bei der Beschaltung solcher Mikrochips solltest du dich sehr genau an die Vorgaben aus dem Datenblatt halten, damit die Schaltung zuverlässig arbeitet und keine Funk-Störungen verursacht.

Aber Step-Down Wandler haben auch Nachteile:

- Wellige Ausgangsspannung

- Bei Fehlfunktion eventuell stark überhöhte Ausgangsspannung, welche die dahinter liegende Schaltung zerstören kann.

Ich verwende gerne die folgenden Mikrochips:

Typ	Ausgang-Spannung	Strom max.	Eingangs-Spannung
LM2574-3.3	3,3 V	0,5 A	4,3...40 V
LM2574-5.0	5 V	0,5 A	6...40 V
LM2576-3.3	3,3 V	3 A	4,8...40 V
LM2576-5.0	5.0 V	3 A	6,5...40 V
LT1076	2,5-50 V	2 A	8...60 V
LT1074	2,5-50 V	5 A	8...60 V

Für Schaltregler benötigst du spezielle Spulen, spezielle (Low-ESR) Kondensatoren und schnelle Schottky-Dioden. Zum Beispiel:

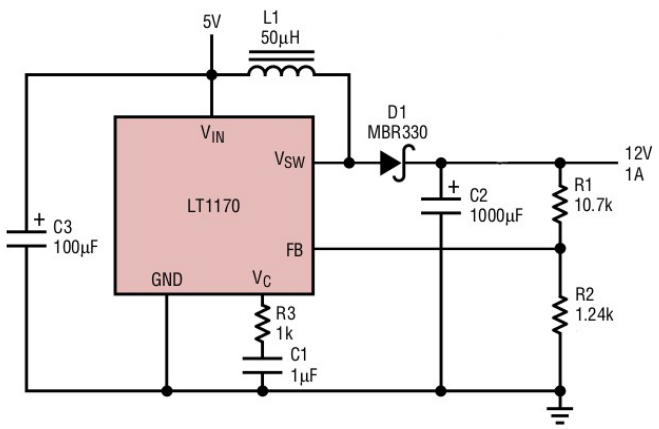
- BY500 für bis zu 5 A
- 1N5822 für bis zu 3 A
- 1N5819 für bis zu 1 A

Im Fachhandel werden übrigens komplett fertige Schaltregler-Module sehr preisgünstig angeboten.

4.5 Step-Up Wandler

Step-Up Wandler erhöhen eine Gleichspannung mit Hilfe einer Spule. Das Funktionsprinzip habe ich bereits im Kapitel über Spulen beschrieben: Die Spule wird immer abwechselnd mit der Eingangsspannung aufgeladen und dann in Reihe zur Eingangsspannung in den Ausgangskondensator entladen. Auch hier liegt der Wirkungsgrad bei ca. 80 %, sofern die Ausgangsspannung nicht mehr als doppelt so hoch ist, wie die Eingangsspannung. Mit Abstrichen am Wirkungsgrad lässt sich ungefähr die zehnfache Ausgangsspannung erreichen.

Das folgende Bild kommt aus dem Datenblatt des LT1170:



Ich verwende gerne die folgenden Mikrochips:

Typ	Ausgang-Spannung	Strom max.	Eingangs-Spannung
LT1172	3...50 V	1,2 A	3...40 V
LT1170	3...50 V	5 A	3...40 V

In den Datenblättern dieser Bauteile findest du konkrete Schaltungsvorschläge an die du dich halten solltest, damit sie nicht unbeabsichtigt zu Störsendern werden.

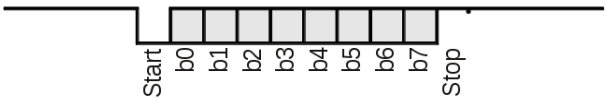
5 Serielle Schnittstellen

Wenn Daten zwischen zwei Maschinen oder Mikrochips übertragen werden, dann passiert das heutzutage fast immer seriell. Seriell bedeutet, dass die Bits der Daten nacheinander durch eine Leitung auf ihre Reise geschickt werden.

Die gewöhnliche serielle Schnittstelle besteht aus zwei Signal-Leitungen:

- Tx oder TxD ist der Ausgang zum Senden
- Rx oder RxD ist der Eingang zum Empfangen

Die serielle Datenübertragung funktioniert folgendermaßen:



- Im Ruhezustand ist die Leitung auf High Pegel
- Jedes Bit hat eine fest definierte Zeitdauer.
- Die Übertragung von jedem Byte beginnt mit einem sogenannten Start-Bit, das immer den Wert 0 (Low) hat.
- Dann folgen die acht Datenbits.
- Zum Abschluss wird ein Stopp-Bit übertragen, das immer den Wert 1 (High) hat.

Die Übertragungsrate bestimmt, wie viel Zeit die Schnittstelle den einzelnen Bits zuweist. Bei der Übertragungsrate 9600 Baud ergeben 9660 Bits zusammen genau eine Sekunde.

Gängige Übertragungsraten sind: 9.600, 19.200, 115.200, 921.600

Damit die Übertragungsrate ausreichend präzise eingehalten wird, muss der Mikrocontroller mit einem Quarz getaktet werden. Der interne R/C Oszillator ist nicht genau genug. Außerdem sind nicht alle Quartz-Frequenzen geeignet. Im Datenblatt des Mikrocontrollers findest du Tabellen, aus denen du je nach gewünschter gewünschter Übertragungsrate geeignete Quarz-Frequenzen ablesen kannst.

Manche serielle Schnittstellen unterstützen neben der Bitrate noch folgende Einstellmöglichkeiten:

- Anzahl der Datenbits 5,6,7 oder 8. Normal ist 8.
- Ein zusätzliches Parity Bit vor dem Stop-Bit in der Variante „Odd“ oder „Even“.
  Das Parity Bit ist gesetzt, wenn die Anzahl der vorherigen High-Bits gerade oder ungerade ist. Normal ist kein Parity Bit.
- Länge des Stop-Bits 1, 1,5 und 2 Takte. Normal ist 1.

5.1 Terminal Programme

Terminal Programme sind prima geeignet, um die seriellen Schnittstellen des PC zu benutzen. Terminal Programme senden jeden Buchstaben, den du eintippst an die serielle Schnittstelle und sie zeigen alle empfangenen Buchstaben auf dem Bildschirm an.

Mit Hilfe einer seriellen Schnittstelle und einem Terminal-Programm bekommt dein Mikrocontroller die Möglichkeit, Texte auf dem Bildschirm deines Computers anzuzeigen. Das kann vor allem während der Softwareentwicklung sehr nützlich sein. Du könntest zum Beispiel die Werte von Variablen anzeigen, oder anzeigen, welche Funktionen das Programm gerade ausführt.

Umgekehrt kannst du so auch Befehle in die Tastatur eintippen, die dein Mikrocontroller empfängt und verarbeitet.

Ich empfehle das Programm „Hammer Terminal“.

5.2 Serielle Schnittstellen am AVR Mikrocontroller

Der ATtiny13 hat keine serielle Schnittstelle, die anderen aber schon. Manche haben eine USI Schnittstelle und manche haben eine USART Schnittstelle. Sie sind prinzipiell beide geeignet, aber die USI Schnittstelle ist erheblich umständlicher zu programmieren und sie kann nur abwechselnd senden und empfangen.

Für die Programmierung hat Atmel konkrete Programmierbeispiele unter dem Namen „Application Notes“ veröffentlicht:

- AVR306: Using the AVR UART
- AVR307: Half duplex UART using the Universal Serial Interface

Weiterhin hat ATMEL auch Programmierbeispiele für serielle Kommunikation ohne Hardwareunterstützung, unter Nutzung ganz gewöhnlicher Ein-/Ausgabe-Pins:

- AVR304: Half Duplex Interrupt Driven Software UART
- AVR305: Half Duplex Compact Software UART

Die Appnotes kannst du von der folgenden Seite herunterladen:
http://www.efo.ru/ftp/pub/atmel/_AVR_MCUs_8bit/_Technical_Library/appnotes/

Probiere die serielle Schnittstelle, indem du den AVR mit Hilfe eines USB-UART Adapters an deinen Personal Computer anschließt. Im Handel werden die Kabel oft falsch „USB TTL Cable“ genannt. Auf dem Computer startest du ein Terminal Programm, welches alle empfangenen Texte auf dem Bildschirm anzeigt.

5.3 RS232

Rund 30 Jahre lang war jeder Personal Computer mit mehreren seriellen RS232 Schnittstellen ausgestattet. Man hatte dort z.B. das Modem für den Internet Zugang angeschlossen. Die RS232 Schnittstelle wurde inzwischen durch USB abgelöst.

Für moderne Computer ohne RS232 Schnittstellen bietet der Handel entsprechende USB Adapter an. Man kann diese Schnittstellen also recht einfach nachrüsten.

Die seriellen Schnittstellen der Personal Computer haben 9-polige Sub-D Stecker, mit folgender Belegung:

1 = DCD, Data Carrier Detect, Eingang
2 = RxD, Receive Data, Eingang
3 = TxD, Transmit Data, Ausgang
4 = DTR, Data Terminal Ready, Ausgang
5 = GND
6 = DSR, Data Set Ready, Eingang
7 = RTS, Request To Send, Ausgang
8 = CTS, Clear To Send, Eingang
9 = RI, Ring Indicator, Eingang

Die RS232 Schnittstelle arbeitet nominal mit inversen +/- 12 Volt Pegeln.

- +3 … 15 Volt ist Low
- -3 … 15 Volt ist High

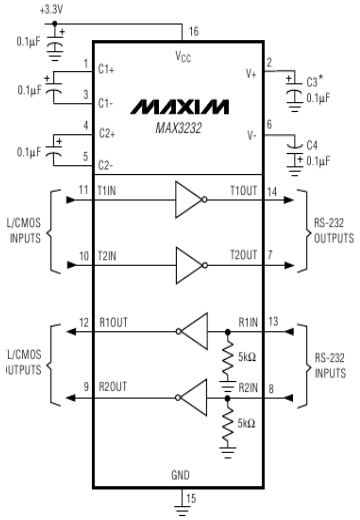
Die Funktion der Signal-Leitungen Leitungen orientiert sich an der Arbeitsweise von Modems:

- RxD und TxD sind die eigentlichen Daten-Leitungen.

- Wenn das Modem eingeschaltet ist, signalisiert es mit der DSR Leitung, dass es Betriebsbereit ist. Der Computer zeigt dem Modem mit der DTR Leitung an, dass er seinerseits empfangsbereit ist.
- Mit der RTS Leitung kündigt der Computer an, dass er Daten senden möchte. Das Modem bestätigt seine Bereitschaft mit der CTS Leitung.
- Wenn das Modem eine Verbindung über die Telefonleitung aufgebaut hat, legt es die DCD Leitung auf High. Wenn das Telefon klingelt, legt es die RI Leitung auf High.

Zur seriellen Kommunikation benötigt man im Minimal-Fall nur drei Leitungen, nämlich RxD, TxD und GND. Alle anderen Leitungen sind optional. Manche Programme und Geräte benutzen sie, andere wiederum nicht.

5.3.1 Transceiver MAX3232



Wenn du einen Mikrocontroller mit einer klassischen RS232 Schnittstelle verbinden möchtest, benötigst du einen Transceiver, der die Spannungspegel übersetzt. Dazu empfehle ich den MAX3232 von Maxim.

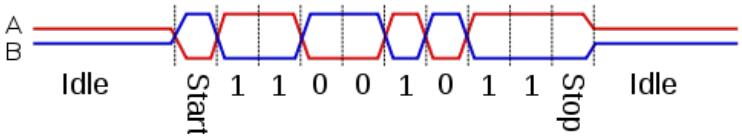
Dieser Mikrochip setzt die 3,3 Volt vom Mikrocontroller auf etwa +/- 6 Volt um, ohne dazu ein extra Netzteil zu benötigen. Die höhere Versorgungsspannung erzeugt sich der Chip selbst.

In der Regel wirst du nur einen der beiden Sender benutzen (für die TxD Leitung) und nur einen Empfänger (für die RxD Leitung).

Aber diesen Chip braucht man nur noch selten, denn die seriellen USB Adapter gibt es auch mit normalen Logikpegeln statt +/-12V. Sie werden oft mit dem fachlich nicht korrektem Namen „USB zu TTL Konverter“ verkauft.

5.4 RS485

Die RS485 Schnittstelle kann als besserer Nachfolger von RS232 betrachtet werden. Der fundamentale Unterschied zu RS232 besteht darin, dass die Signale symmetrisch übertragen werden:



Die RS485 Schnittstelle verwendet immer zwei Leitungen paarweise.

- High = Spannung an A ist mindestens 0,3V größer als B
- Low = Spannung an A ist mindestens 0,3V kleiner als B

Man verwendet Kabel mit paarweise verdrillten Leitungen (z.B. CAT5 Kabel, die auch bei Ethernet zum Einsatz kommen). Die Enden des Kabel werden mit 120 Ohm Widerständen abgeschlossen, um Reflexionen zu unterdrücken.

Dadurch dass die beiden Leitungen immer exakt entgegengesetzt angesteuert werden, strahlen sie nach außen hin kein magnetisches Feld ab und sind auch gegen äußere Einflüsse unempfindlich. So ist selbst bei mehreren hundert Metern Kabellänge immer noch 1 Megabit Übertragungsrate machbar.

RS485 nutzt meistens nur ein Leitungspaar abwechselnd zum Senden und zum Empfangen. Diese Übertragunsart nennt man Half-Duplex.

Es gibt auch eine Variante mit zwei Leitungspaaren. So kann das Senden und Empfangen gleichzeitig stattfinden, also Full-Duplex. Der Half-Duplex Betrieb ist allerdings weiter verbreitet, als der teurere Full-Duplex Betrieb.

Für RS485 gibt es keine genormte Stecker-Belegung. Sehr häufig werden einfache Schraubklemmen verwendet.

Ein sehr gängiger Mikrochip für diese Schnittstelle ist der SN75ALS176. Er setzt die einfachen RxD und TxD Signale des Mikrocontrollers in die symmetrischen Signale A und B um. Über einen Steuer-Eingang muss der Mikrocontroller festlegen, ob gerade gesendet oder empfangen werden soll.

5.5 USB

Der aktuelle Klassiker unter den PC-Schnittstellen ist der USB Bus.

In Version 1.0 bis 2.0 besteht die USB Schnittstelle aus vier Leitungen:

- +5V Stromversorgung, maximal 500 mA
- positive Datenleitung, mit 3,3 V Pegel
- negative Datenleitung, mit 3,3 V Pegel
- GND

Die Daten werden seriell über zwei Leitungen übertragen. Wenn die eine Leitung High ist, dann ist die andere Low und umgekehrt. Die Elektromagnetische Abstrahlung des Kabels ist minimal, weil sich die elektromagnetischen Felder aufgrund der komplementären Polarität der Datenleitungen gegenseitig aufheben.

Es gibt keine separaten Leitungen für die Sende- und Empfangs- Richtung. Stattdessen werden die Daten über die gleichen Leitungen abwechselnd gesendet und Empfangen.

Jeder USB Anschluss kann durch USB Hubs vervielfacht werden, auf theoretisch bis zu 127 Anschlüsse. Jedes Gerät bekommt durch die Hubs eine eindeutige Nummer zugewiesen, über die der Computer das gewünschte Gerät ansprechen kann.

Für Hobby-Elektroniker ist der USB Bus aus mehreren Gründen attraktiv:

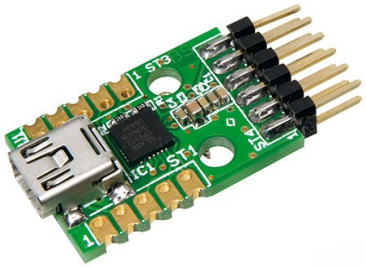
- Der Computer kann als Stromversorgung für die angeschlossenen Geräte dienen, sofern man nicht mehr als 2,5 Watt braucht.
- Man hat mit Hilfe von billigen Hubs beinahe beliebig viele USB Anschlüsse verfügbar.
- Jeder Computer hat USB Anschlüsse.

Das Übertragungsprotokoll der USB Schnittstelle ist ziemlich kompliziert. Die Spezifikation umfasst mehrere hundert Seiten Text. Obwohl ich die USB Schnittstelle seit Jahren benutze, habe ich immer noch nur eine sehr grobe Vorstellung davon, wie das Übertragungsprotokoll funktioniert.

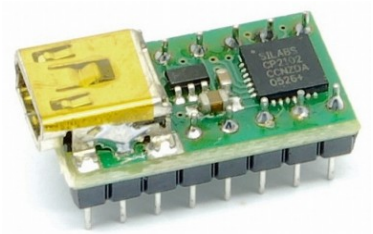
Die USB 3.0 Schnittstelle hat mehr Leitungen, ist schneller und kann mehr Strom liefern. Für Hobbyelektroniker ist sie allerdings (noch) nicht verwendbar. Aber du kannst ältere USB Geräte in die neuen USB 3 Buchsen stecken, da sie dazu abwärtskompatibel sind.

5.5.1 USB-UART

Um selbst gebaute Geräte via USB anschließen zu können, empfehle ich den Einsatz sogenannter USB-UART Module. Diese Module enthalten einen vorprogrammierten Mikrocontroller, einen USB Stecker sowie ein paar Kleinteile drumherum.



UM2102 von elv.de



ioMate-USB1 von chip45.com

USB-A UART Bridge von ic-board.de

UART Kabel gefunden bei Ebay

Die bekanntesten vorprogrammierten UART Chips (Stand 2016) sind:

- Silabs CP2102
- Future Technology Devices International (FTDI) FT232
- Prolific PL2302
- CH340 und CH341 aus China

Alle vier Mikrochips sind zu Linux, Mac OS/X und Windows kompatibel. Ab Windows 8 muss man bei Prolific allerdings aufpassen, denn die aktuellen Treiber unterstützen nicht alle Versionen des Chips. Ggf muss man auf ältere Treiber zurückgreifen, siehe http://stefanfrings.de/avr_tools/index.html#pl2303 .

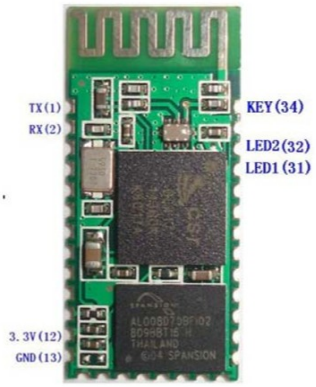
Windows 10 installiert den nötigen Treiber in der Regel vollautomatisch. Linux und Mac OS/X haben die nötigen Treiber bereits vorinstalliert.

Wenn du einen USB-UART an den Personal Computer anschließt, richtet das Betriebssystem eine virtuelle serielle Schnittstelle ein, z.B. COM5, die du wie eine echte serielle Schnittstelle durch entsprechende Programme nutzen kannst.

Bei chip45.com kannst du sehr kompakte Platinen mit unterschiedlichen AVR Mikrocontrollern und USB-UART bekommen. Mit einem solchen Modul habe ich das Gerät auf der Titelseite dieses Buches gebaut. Ebenso kann ich die chinesischen Arduino Nano Nachbauten empfehlen.

5.6 Bluetooth

Drahtlose Bluetooth Schnittstellen sind sehr einfach mit dem chinesischen HC-05 Modul realisierbar. Dieses Modul ist für 3,3 V vorgesehen, man kann es aber auch mit Adapterplatine für 5 V kaufen.



Von den vielen Anschlüssen brauchst du nur ganz wenige:

- 1 = TxD (Ausgang)
- 2 = RxD (Eingang)
- 12 = Versorgung 3,3 V 50 mA (3 bis 4,2 V)
- 13 = GND
- 31 = LED 1, blinkt ständig
- 32 = LED 2, Low bei Bereitschaft und High bei Verbindung.
- 34 = Eingang Low=Normalbetrieb, High=Befehlsmodus (unbeschaltet = Low)

Um eine Verbindung mit dem PC aufzubauen klickst du auf das Bluetooth Symbol in der Task-Leiste und durchsuchst dann die Umgebung nach erreichbaren Geräten. Der Computer wird das Modul finden und nach einem Passwort fragen. Es lautet: 1234. Nach Eingabe des Passwortes richtet der Computer eine virtuelle serielle Schnittstelle ein, z.B. COM5, die du wie eine echte serielle Schnittstelle durch entsprechende Programme nutzen kannst.

Die Übertragungsrate ist standardmäßig 9600 Baud. Über die „echte“ Serielle Schnittstelle (also vom Mikrocontroller aus) kann man sowohl das Passwort als auch die Übertragungsrate mit Befehlen um konfigurieren. Die Befehle sind im Datenblatt angegeben. Ich rate allerdings davon ab, die Übertragungsrate zu erhöhen, weil die Reichweite und Zuverlässigkeit darunter leidet.

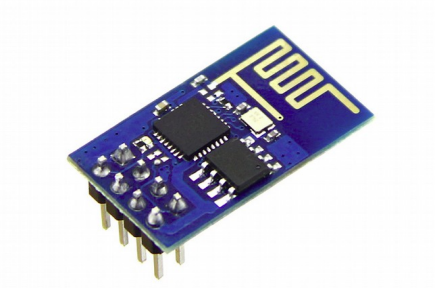
Das Modul funktioniert auch unter Linux problemlos.

Lass dich nicht durch fragwürdige Versprechungen bezüglich der Reichweite in die Irre führen. Viele Händler versprechen bis zu 100 Meter Reichweite. In der Praxis erreicht man jedoch oft nur 3 bis 5 Meter.

5.7 Ethernet und WLAN

Super spannend ist natürlich, den eigenen Mikrocomputer mit dem Internet oder anderen Computer Netzwerken zu verbinden.

Am preisgünstigsten geht das zur Zeit mit einem **ESP-01** Modul:



Es enthält einen eingebauten IP-Stack, der gleichzeitig bis zu 4 TCP oder UDP Verbindungen gleichzeitig aufbauen kann. Das Modul kann sich wahlweise in ein bestehendes WLAN Netz einbuchen oder selbst Access-Point spielen.

Die Verbindung zum Mikrocontroller erfolgt seriell (UART). Über Befehle in Text-Form konfiguriert und steuert man das Modul. Das Senden von Daten geschieht Paketweise, ebenfalls per AT-Befehl. Empfangene Datenpakete werden mit einem kleinen Header (der die Kanalnummer und die Anzahl der Zeichen angibt) an den Mikrocontroller übergeben.

Das Modul ist noch sehr jung, die Firmware ist daher schlecht dokumentiert und ändert sich häufig in Details. Dennoch kann man damit erfolgreiche Basteleien umsetzen.

Eine umfangreiche Dokumentation zu diesem Produkt findest du hier:
http://stefanfrings.de/esp8266/index.html .

Wenn du etwas ähnliches für Kabel suchst, dann werden dir Module mit Chips der Firma Wiznet gefallen. Die habe ich selbst noch nicht verwendet, aber ihre technische Beschreibung sieht vielversprechend aus.

Ansonsten empfehle ich gerne auch meine AVR Webserver Software, die auf den kompakten **CrumbX1-NET** Modulen von Chip45 läuft:

Der sehr zuverlässige Webserver basiert auf der Arbeit des Wissenschaftlers Adam Dunkels, einem winzig kleinen TCP/IP Protokollstack, den er speziell für Mikrocontroller entwickelt hat.

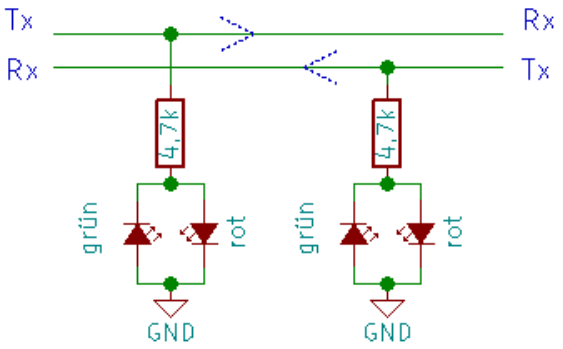
Das Programm ist modular aufgebaut, so dass es relativ einfach ist, eigene Webseiten oder Funktionen hinzuzufügen. Du kannst die Firmware mitsamt Dokumentation und Schaltplänen von meiner Homepage downloaden.

Andere Alternativen findet man unter den Namen **Lantronics Xport**, sowie **Digi Connect**. Diese Module reichen die Daten der Netzwerkverbindung transparent an den Mikrocontroller weiter. Das bedeutet: Wenn ein Computer „Hallo“ an dem Mikrocontroller sendet, kommt dort auch einfach nur Hallo am seriellen Port an. Nicht mehr und nicht weniger. Umgekehrt kann der Mikrocontroller auch einfach Texte (und Daten) senden, sofern eine Verbindung aufgebaut ist.

Diese Module sind am einfachsten zu programmieren, sie unterstützen jedoch nur eine Verbindung gleichzeitig. Und sie sind ziemlich teuer.

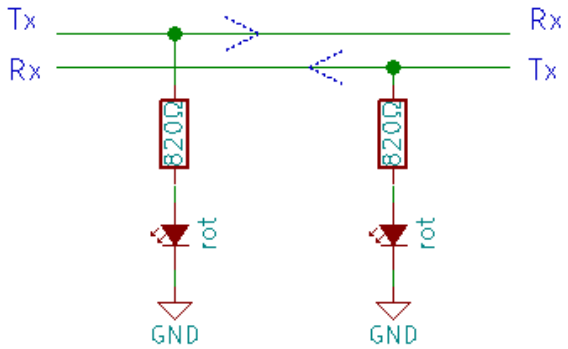
5.8 Serielle Schnittstellen entstören

Falls die Kommunikation zwischen Mikrocontroller und Computer 'mal nicht klappt, schließe Leuchtdioden mit Vorwiderstand an die beiden Datenleitungen an. So siehst du ob etwas gesendet oder empfangen wird. Variante für +/- 12 V Pegel:



Bei High Pegel (+12 V) leuchtet die rote LED. Bei Low Pegel (-12 V) leuchtet die grüne LED. Wenn weder die rote noch die grüne LED leuchtet, dann führt die Leitung ein ungültiges Signal, wahrscheinlich, weil sie irgendwo unterbrochen ist oder RxD/TxD vertauscht verbunden sind.

Variante für 3,3 - 5 V Pegel:



Wenn der Computer oder der Mikrocontroller trotz flackernder LED's gar nichts empfängt, liegt es meistens an:

- RxD und TxD vertauscht
- Falsch eingestellte Parität (muss beim Sender und Empfänger gleich sein)
- Falsch eingestellte Länge des Stopp-Bits (muss beim Sender und Empfänger gleich sein)

Wenn der Empfänger nur wirre Zeichen oder anderen Quatsch empfängt, liegt es meisten an:

- Falsch eingestellte Übertragungsrate (muss beim Sender und Empfänger gleich sein)
- Falscher Quarz beim Mikrocontroller
- Fehlende Kondensatoren am Quarz
- Falsch eingestellte System-Taktfrequenz in der Entwicklungsumgebung

5.9 I²C Bus

Der I²C Bus wurde von der Firma Philips erfunden, um Baugruppen innerhalb von Geräten (z.B. Fernseher) miteinander zu verbinden. Beim I²C Bus geht also um relativ kurze Leitungslängen.

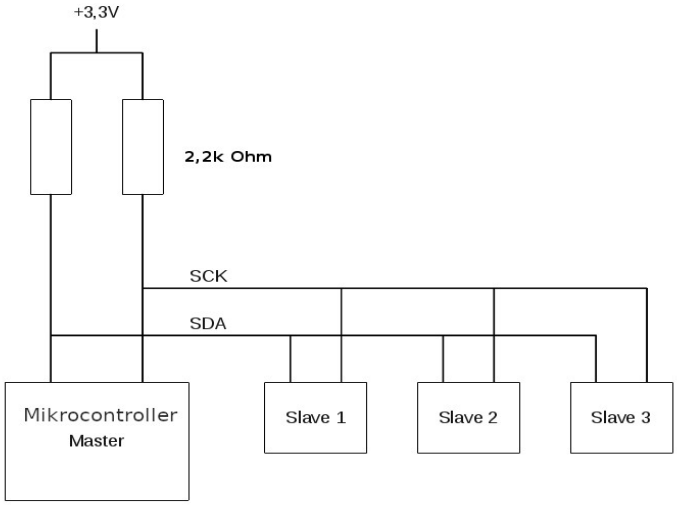
Die Halbleiter Sparte von Philips wurde von NXP gekauft und NXP wurde von Qualcom gekauft. Wenn du nach Datenblättern und Anwendungsbeispielen suchst, kann dieses Wissen nützlich sein.

Es gibt zahlreiche Mikrochips mit I²C Bus, zum Beispiel Tastaturen, Displays, Sensoren, Speicher, Verstärker, Klangregler, Radio und TV Empfänger, Batterie-Laderegler, usw.

5.9.1 Hardware

Der I²C Bus besteht aus zwei Leitungen:

- Eine Takt-Leitung, und
- Eine bidirektionale Datenleitung (zum Senden und Empfangen)



Der I²C Bus wird von einem Mikrocontroller gesteuert, den man „Master“ nennt. Der Master kann über einen einzigen Bus bis zu 112 Geräte steuern, die man „Slaves“ nennt. Jeder Slave hat eine Adresse (7-bit Wert), die ihn identifiziert.

Jeder Mikrochip am I²C Bus hat sogenannte Open-Kollektor Ausgänge. Das bedeutet, dass sie die Leitungen nur entweder auf Low ziehen können oder loslassen können. Der High-Pegel kommt durch die beiden Pull-Up Widerstände zustande, wenn alle Mikrochips die Leitungen loslassen.

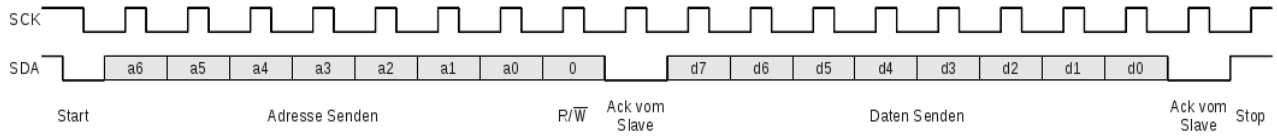
5.9.2 Übertragungsprotokoll

Die Übertragungsrate darf bei den meisten Mikrochips maximal 100.000 oder 400.000 Bits pro Sekunde betragen, sie darf aber auch beliebig langsamer sein. Der langsamste Mikrochip am Bus bestimmt die höchst-mögliche Taktfrequenz. Weil der I²C Bus keine präzise Taktfrequenz benötigt, man kann ohne Quarz auskommen.

Es gibt folgende besondere Signale:

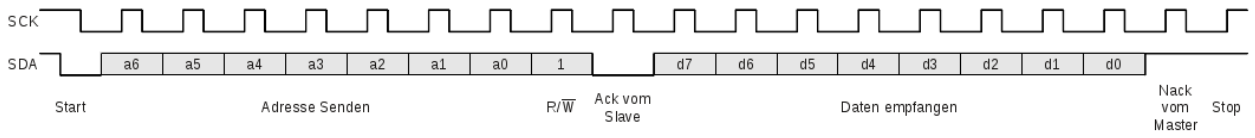
- **Start**
  Der Master zieht zuerst die SDA Leitung und dann die SCL Leitung auf Low.
- **Stop**
  Der Master lässt zuerst die SDA Leitung und dann die SCL Leitung los (auf High).
- **Ack**
  Der Empfänger (Master oder Slave) zieht die SDA Leitung nach Empfang des Bytes für die Dauer eines Taktes auf Low.
- **Nack**
  Ist das Gegenteil von Ack, also wenn die SDA Leitung nicht auf Low gezogen wird.

So sendet man ein Byte an den Slave:



Zuerst sendet der Master das Start-Signal, gefolgt von der 7-Bit Adresse des Slave und dem R/$\overline{W}$ Bit mit dem Wert **Low**. So signalisiert er, dass er anschließend Daten an den Slave senden möchte. Der Master darf mehrere Daten-Bytes an den Slave senden, solange dieser mit Ack antwortet.

Und so empfängt man ein Byte vom Slave:



Zuerst sendet der Master das Start-Signal, gefolgt von der 7-Bit Adresse des Slave und dem R/$\overline{W}$ Bit mit dem Wert **High**. So signalisiert er, dass er anschließend Daten vom Slave lesen möchte. Der Slave antwortet in den folgenden Taktzyklen mit so mehreren Bytes, bis er entweder keine Daten mehr hat oder der Master die Übertragung durch das Nack Signal beendet.

Abgesehen vom Start-Signal darf der Pegel der Datenleitung immer nur dann verändert werden, während die Taktleitung auf Low liegt.

Der Slave kann die Übertragungsgeschwindigkeit herabsetzen, indem er die Taktleitung in den vom Master gesendeten Low-Phase seinerseits auch auf Low zieht. Der Master erkennt diesen Zustand und wartet ggf. ab, bis der Slave die Taktleitung wieder los lässt und sie somit auf High geht.

Wenn der Slave die Adresse oder gesendete Daten nicht mit Ack beantwortet, muss der Master die Datenübertragung abbrechen. Umgekehrt muss auch der Slave das Senden von Daten abbrechen, wenn der Master ein Nack Signal sendet.

5.9.3 Übertragung von Registern

Viele Mikrochips mit I^{2}C Bus arbeiten mit sogenannten Registern. Einige Register dienen der Konfiguration des Mikrochips, andere Register wiederum dienen der Übertragung von Nutzdaten. Ob ein Mikrochips Register hat, und wie viele Register es sind, hängt ganz vom jeweiligen Mikrochip ab und kann in dessen Datenblatt nachgelesen werden.

Man kann Register beschreiben und auslesen. Bei ihnen läuft die Kommunikation etwas komplexer ab. So sendet der Master Daten an ein Register:

- Start-Signal senden
- Slave-Adresse senden
- R/$\overline{W}$ Bit mit dem Wert Low senden (=Write)
- Slave antwortet mit ACK
- Registernummer senden (ein Byte)
- Slave antwortet mit ACK
- Daten-Byte senden
- Slave antwortet mit ACK
- Optional: Daten-Byte für das nächste Register senden und jeweils auf ACK warten
- Stop-Signal senden

Der Master kann in der Regel beliebig viele aufeinander folgende Register beschreiben, indem er nur die Nummer des ersten Registers sendet, gefolgt von mehreren Daten-Bytes.

So liest der Master ein Register aus:

- Start-Signal senden
- Slave-Adresse senden
- R/$\overline{W}$ Bit mit dem Wert Low senden (=Write)
- Slave antwortet mit ACK
- Registernummer senden (ein Byte)
- Slave antwortet mit ACK
- erneut Start-Signal senden
- nochmal Slave-Adresse senden
- R/$\overline{W}$ Bit mit dem Wert High senden (=Read)
- Slave antwortet mit ACK
- Daten-Byte(s) empfangen
- Jeweils mit Ack oder Nack antworten
- Stop-Signal senden

In der Regel kann man mehrere aufeinander folgende Register am Stück auslesen. Der Slave sendet so lange Daten, bis entweder alle verfügbaren Daten übertragen wurden oder der Master die Übertragung durch Nack beendet.

5.9.4 Adressierung

Ein bisschen Verwirrung stiftet oft die Slave-Adresse und das R/W Bit. Aus Sicht des Mikrocontrollers sendest du immer ein Byte, das aus Slave-Adrese UND R/W Bit besteht:

Bit	7	6	5	4	3	2	1	0
Belegung	a6	a5	a4	a3	a2	a1	a0	R/$\overline{W}$

Wenn im Datenblatt eines I^2C Slaves steht, dass seine Adresse 0x17 (=0010111) sei, dann musst du auf dem Mikrocontroller diese Adress-Bits alle um eine Position nach links schieben, um rechts Platz für das R/$\overline{W}$ Bit zu schaffen. Der Mikrocontroller sendet also den Wert 0x2E für eine Write-Kommunikation oder 0x2F für eine Read-Kommunikation.

5.9.5 I^2C Bus am AVR Mikrocontroller

Manche AVR's haben eine USI Schnittstelle und manche haben eine TWI Schnittstelle. Sie sind prinzipiell beide für den I^2C Bus geeignet, aber die USI Schnittstelle ist erheblich umständlicher zu programmieren.

Für beide Schnittstellen enthalten die Application Notes konkrete Programmierbeispiele:

- AVR155: Accessing I2C LCD Display Using the AVR 2-Wire Serial Interface
- AVR310: Using the USI module as a I2C master
- AVR311: Using the TWI module as I2C slave
- AVR312: Using the USI module as a I2C slave
- AVR315: Using the TWI module as I2C master

Siehe http://www.efo.ru/ftp/pub/atmel/_AVR_MCUs_8bit/_Technical_Library/appnotes/

Wenn du die Programmierung üben möchtest, versuch es zuerst mit dem Chip PCF8574. Der Chip hat 8 I/O Leitungen, die du über den I2C Bus setzen und abfragen kannst.

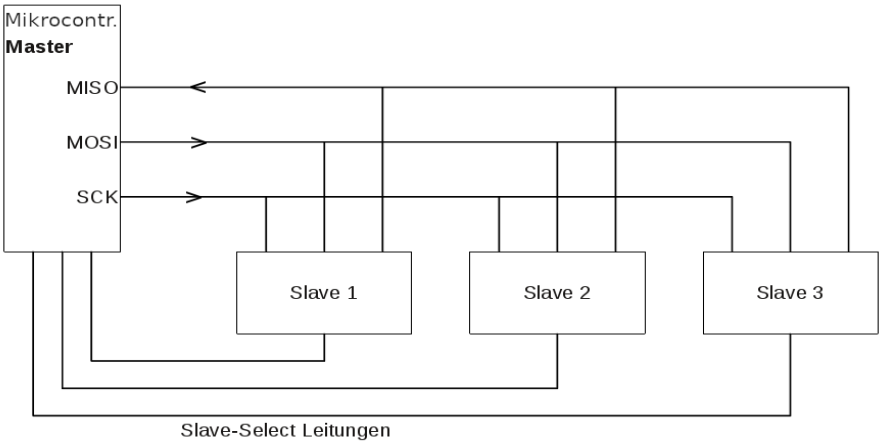
Im Anhang findest du eine C-Funktion, die du nutzen kannst, um über das TWI Modul mit einem Slave zu kommunizieren.

5.10 SPI Bus

Der SPI Bus ist als serielle Schnittstelle zwischen Mikrochips auf einer Platine gedacht. Er ermöglicht sehr hohe Übertragungsraten, typischerweise um 1 Megabit. Die Leitungen dürfen allerdings nur wenige Zentimeter lang sein.

SPI Schnittstellen findet man oft an Speicherchips, zum Beispiel an SD Karten, aber auch an Sensoren, ADC Wandlern und vielen anderen Mikrochips. Sie besteht aus vier Leitungen:

- MOSI, Master Out to Slave In
- MISO, Master In to Slave Out
- SCK, Serial Clock
- $\overline{SS}$, Slave Select, jeder Slave hat seine eigene Select Leitung



Der I^2C Bus wird von einem Mikrocontroller gesteuert, den man „Master“ nennt. Der Master kann über einen einzigen Bus mehrere Geräte steuern, die man „Slaves“ nennt. Je niedriger die Übertragungsrate ist, umso mehr Slaves kann man anschließen.

Im Gegensatz zum I^2C Bus, kann man beim ISP Bus mehrere Mikrochips mit unterschiedlichen Taktfrequenzen wechselweise ansprechen. Es ist hier nicht so, dass sich der gesamte Bus an den langsamsten Chip anpassen muss.

Das Übertragungsprotokoll ist nicht genau festgelegt, kann also bei jedem Slave anders funktionieren. Es gibt nur diese wenigen Grundregeln:

- Der Master beginnt eine Kommunikation, indem er die Slave-Select Leitung des gewünschten Slaves auf Low setzt.
- Dann sendet oder empfängt er Daten, wobei er durch die Clock-Leitung den Takt vorgibt.
- Der Master beendet die Kommunikation, indem er die Slave-Select Leitung auf High setzt.

5.10.1 SPI Bus am AVR Mikrocontroller

Bei AVR Mikrocontrollern gibt es drei unterschiedliche Hardware Schnittstellen, die für SPI Bus geeignet sind: Am einfachsten geht es mit einer „echten“ SPI Schnittstelle, aber auch USI und UART sind geeignet, allerdings umständlicher zu programmieren.

Für alle drei Schnittstellen enthalten die Application Notes konkrete Programmierbeispiele:

- AVR107: Interfacing AVR serial memories
- AVR151: Setup and Use of the SPI
- AVR317: Using the USART on the ATmega48/88/168 as a SPI master
- AVR319: Using the USI module for SPI communication

- AVR320: Software SPI Master

http://www.efo.ru/ftp/pub/atmel/_AVR_MCUs_8bit/_Technical_Library/appnotes/

Probiere die SPI Schnittstelle mit einem MCP3208 aus. Das ist ein 8-Fach ADC mit relativ ausführlichem Datenblatt, welches die SPI Kommunikation detailliert beschreibt.

6 Port Erweiterungen

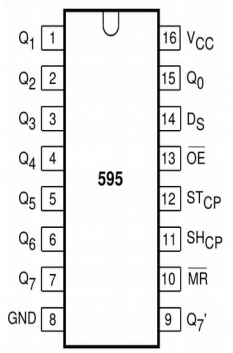
In diesem Kapitel stelle ich gängige Methoden vor, mit denen die Anzahl der Ein-/Ausgabe Pins von Mikrocontrollern erhöht werden kann.

Am Einfachsten ist es natürlich, einen größeren Mikrocontroller zu verwenden. Aber Mikrochips mit mehr als 40 Pins gibt es nur in SMD Form. Dort sind die Abstände zwischen den Anschlüssen so klein, dass man sie in Handarbeit nicht verarbeiten kann.

Für das Hobby eignet sich daher eher die Strategie, mehr Ausgänge durch mehr Mikrochips bereit zu stellen und diese miteinander zu verbinden. Eine durchaus praktikable Möglichkeit besteht darin, mehrere Mikrocontroller zu benutzen und miteinander zu vernetzen. Dann muss man sich aber erst einmal Gedanken darüber machen, wie die Mikrocontroller untereinander kommunizieren sollen und entsprechende Programme schreiben.

Für den Anfang ist es sicher wesentlich einfacher, die Anschlüsse mit nicht programmierbaren Mikrochips zu erweitern.

6.1 Ausgabe-Schieberegister

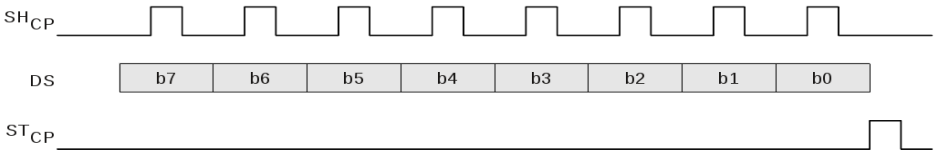


Der Mikrochip 74HC595 (oder 74HCT595) liest Daten seriell ein und gibt sie an acht Ausgängen Q0 bis Q7 aus. Du musst nur drei Leitungen mit dem Mikrocontroller verbinden:

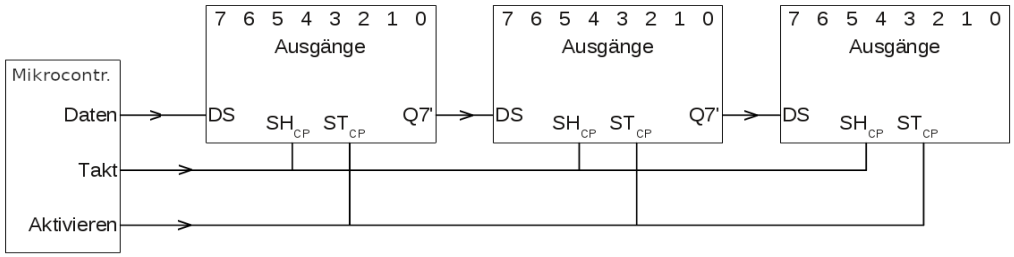
- DS ist der serielle Daten-Eingang
- SH_{CP} ist der Takt-Eingang
- Ein Impuls an ST_{CP} überträgt die gesammelten Daten auf die Ausgänge.

Ein Low-Impuls an $\overline{MR}$ löscht den Speicher, so dass alle Ausgänge auf Low gehen. In vielen Anwendungen wird diese Leitung nicht benötigt und daher fest mit VCC verbunden.

Der Anschluss $\overline{OE}$ kann dazu benutzt werden, die Ausgänge zu deaktivieren. Wenn du ihn nicht benötigst, verbinde ihn fest mit GND. Den Chip steuert man so an:

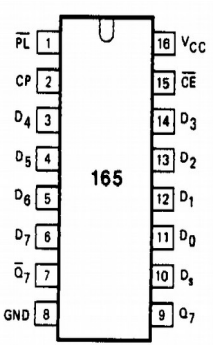


Sende 8 Bits nacheinander an DS, wobei du bei jedem Bit einen Taktimpuls sendest. Sende danach einen Impuls an ST_{CP}, dann erscheinen die gesammelten Bits an den Ausgängen Q0-Q7. Den Ausgang Q7' kann man verwenden, um mehrere dieser Mikrochips zu verketten:



Das kannst du beliebig oft machen. So kann man mit nur drei Steuerleitungen auf beliebig viele Ausgänge kommen. Die Taktfrequenz darf locker 10 Mhz betragen, bei 5 V sogar noch etwas mehr.

6.2 Eingabe-Schieberegister

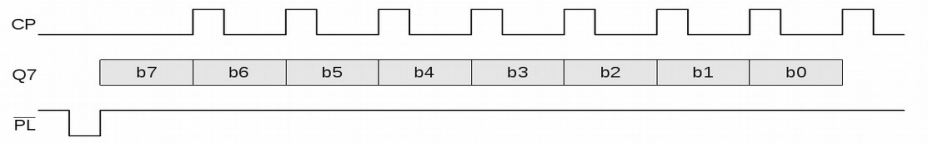


Der Mikrochip 74HC165 (oder 74HCT165) liest Daten von den acht Eingängen D0 bis D7 ein und gibt sie seriell an Q7. Du musst nur drei Leitungen mit dem Mikrocontroller verbinden:

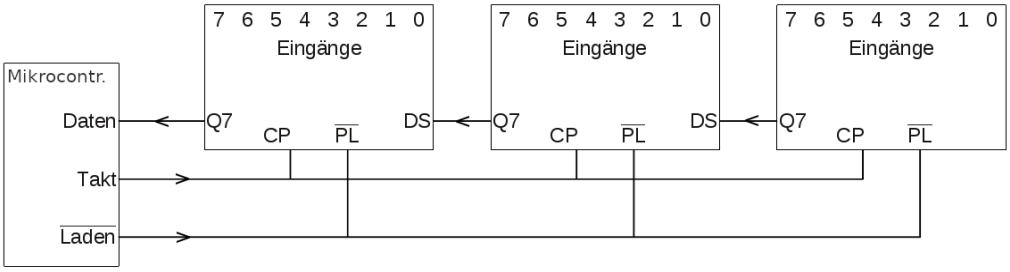
- Q7 ist der serielle Daten-Ausgang
- CP ist der Takt-Eingang
- $\overline{PL}$ überträgt die Eingänge ins Schieberegister

Den Anschluss $\overline{CE}$ verbindest du normalerweise fest mit GND.

Den Chip steuert man so an:

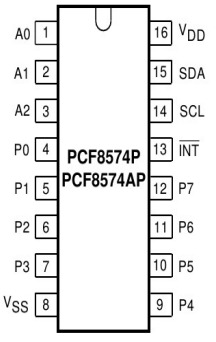


Sende zuerst einen Low-Impuls an $\overline{PL}$, dadurch werden die acht Eingänge in das Schieberegister eingelesen. Am Ausgang Q7 kommt der Wert vom Eingang D7 heraus. Lies acht Bits von Q7 nacheinander ein, wobei du nach jedem Bit einen Impuls an CP sendest. Den Eingang DS kann man verwenden, um mehrere dieser Mikrochips zu verketten:



Das kannst du beliebig oft machen. So kann man mit nur drei Steuerleitungen auf beliebig viele Eingänge kommen. Die Taktfrequenz darf locker 10 Mhz betragen, bei 5V sogar noch etwas mehr.

6.3 Digital über I²C Bus



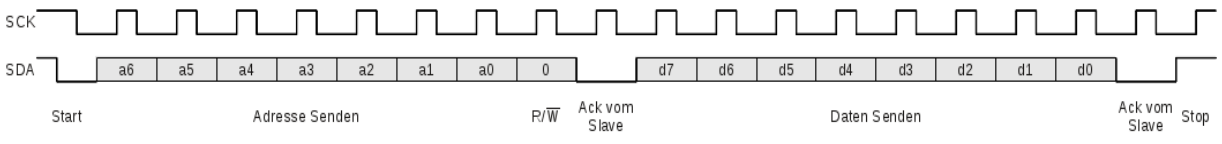
Ein absoluter Klassiker für den I²C Bus ist der PCF8574 von Philips. Er hat acht Leitungen (P0-P7), die man sowohl als Eingang als auch als Ausgang verwenden kann.

Auf Low geschaltet können die Ausgänge Lasten bis 10mA ansteuern, zum Beispiel LED's ansteuern.

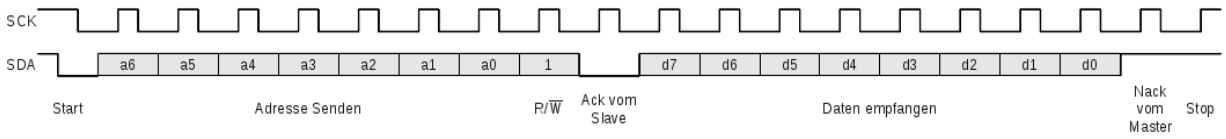
Auf High geschaltet liefern die Anschlüsse einen geringen Strom von nur 100 µA. In diesem Zustand kann man die Anschlüsse auch als Eingang verwenden.

Die Taktfrequenz darf maximal 100 khz betragen.

Die Kommunikation findet exakt nach dem I²C Standard statt. So sendest du ein Byte, um die Ausgänge P0-P7 anzusteuern:



Und so liest du den Zustand der acht Port Pins P0-P7 ein:

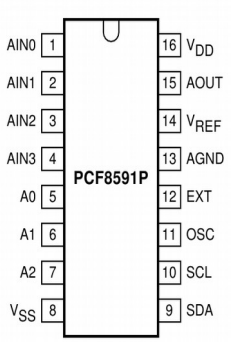


Die 7-Bit Slave-Adresse des PCF8574 ist 0100xxx und die des PCF8574A ist 0111xxx. Wobei xxx durch die drei Eingänge A0, A1 und A2 einstellbar ist. Du kannst also an einen einzigen I^2C Bus von den beiden Mikrochips jeweils bis zu acht Stück anschließen.

In dem Datenblatt des PCF8574 ist der I^2C Bus recht detailliert beschrieben. Ich finde, dass man diesen Chip daher sehr gut als Musterbeispiel für den Bus verwenden kann.

Ein kleiner Hinweis: Was Philips im Datenblatt V_{SS} nennt, entspricht GND, der gemeinsamen Masse. Die 3,3 Volt Stromversorgung schließt man an den V_{DD} Pin an.

6.4 Analog über I^2C Bus



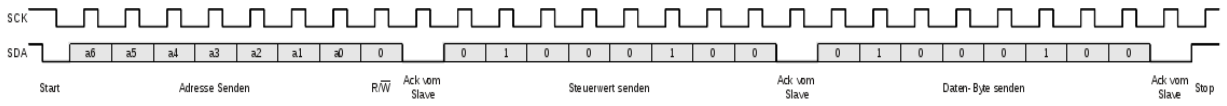
Der PCF8591 enthält einen analog zu digital Wandler mit vier Eingängen (AIN0-3), sowie einen digital zu analog Wandler mit einem Ausgang (AOUT).

Die Spannungen der analogen Eingänge werden mit der Spannung vom Eingang V_{REF} verglichen, so dass du durch äußere Beschaltung den Messbereich selbst festlegen kannst. Ich verbinde den Anschluss V_{REF} meistens mit der 3,3 V Spannungsversorgung, dann kann ich analoge Signale im Bereich 0-3,3 Volt messen.

Der Wandler hat eine Auflösung von 8 Bit, also deutlich weniger, als der interne ADC des Mikrocontrollers. Er liefert demnach Werte im Bereich 0 bis 255.

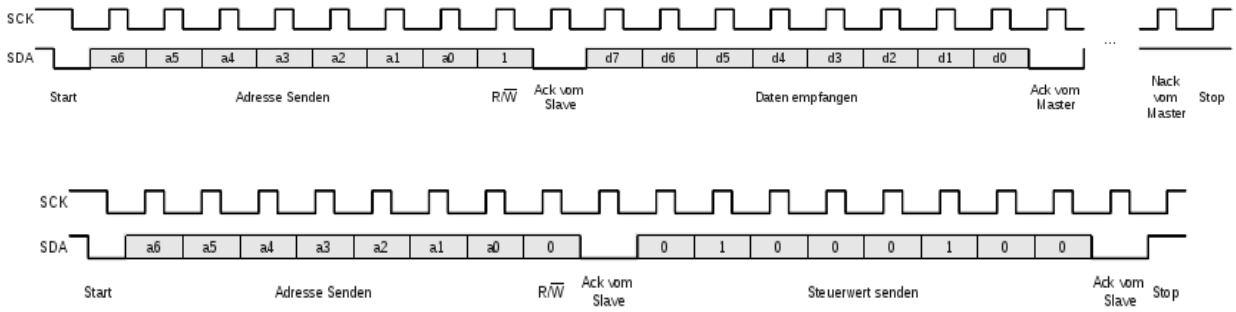
Über den I^2C Bus kannst du wahlweise einen einzelnen analogen Eingang abfragen, oder alle vier Eingänge nacheinander in einem Rutsch. Die Taktfrequenz darf maximal 100khz betragen.

Um den analogen Ausgang ansteuern, musst du drei Bytes senden:



1. Adresse mit R/$\overline{W}$ Bit=0 senden.
2. Steuerwert 01000100 senden.
3. Daten-Byte senden.

Um die analogen Eingänge abzufragen, gehst du so vor:



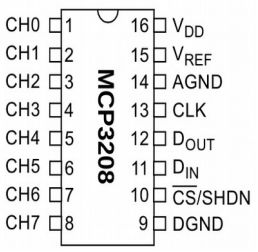
1. Adresse mit R/$\overline{W}$ Bit=0 senden.
2. Steuerwert 01000100 senden.
3. Adresse mit R/$\overline{W}$ Bit=1 senden.
4. Fünf Bytes empfangen.

Das erste empfangene Daten-Byte ist nutzlos. Die darauf folgenden vier Bytes entsprechen den Messwerten der vier analogen Eingänge.

Die 7-Bit Slave-Adresse des PCF8591 ist 1001xxx , wobei xxx durch die drei Eingänge A0, A1 und A2 einstellbar ist. Du kannst also an einen einzigen I^2C Bus also bis zu acht Mikrochips dieses Typs anschließen.

Was Philips im Datenblatt V_{SS} nennt, entspricht GND, der gemeinsamen Masse. Die 3,3 Volt Stromversorgung schließt man an den V_{DD} Pin an. Den EXT Pin musst du mit GND verbinden und an V_{REF} legst du die Referenzspannung an, im einfachsten Fall ebenfalls 3,3 Volt.

6.5 ADC mit SPI Schnittstelle



Der MCP3208 ist mein Favorit, was ADC Wandler angeht, denn er hat eine Auflösung von 12 Bit.

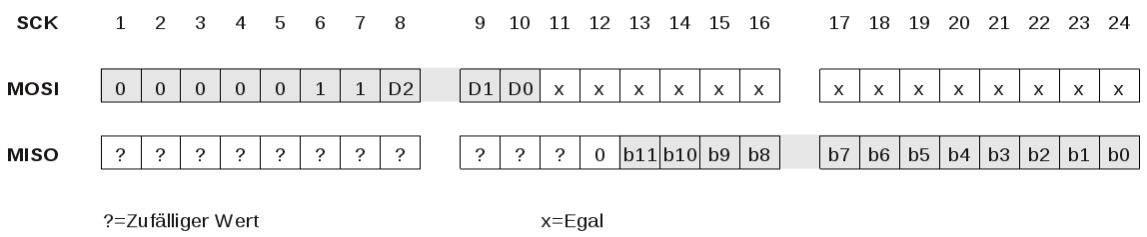
Die acht Eingänge heißen CH0 bis CH7. Der Wandler vergleicht die Eingangsspannung mit der Spannung an V_{REF} und liefert einen Messwert im Bereich 0 bis 4095.

Die Taktfrequenz muss zwischen 10.000 Hz und 1.000.000 Hz liegen.

Verbinde den Mikrochip so mit dem SPI Bus des Mikrocontrollers:

- D_{OUT} an MISO
- D_{IN} an MOSI
- CLK an SCK
- $\overline{CS}$ an irgendeinen Port Pin, entspricht der Slave-Select Leitung.

Bei der seriellen Kommunikation sendet und empfängt der Mikrocontroller gleichzeitig jeweils drei Bytes, von denen allerdings nicht alle Bits benutzt werden:



Die Datenübertragung besteht stets aus insgesamt 24 Taktzyklen, entsprechend 3 mal 8 Bit. In den ersten 10 Taktzyklen sendet der Mikrocontroller einen Steuerwert an den ADC, durch den der gewünschte analoge Eingang ausgewählt sind. In den letzten 12 Taktzyklen liefert der ADC das Messergebnis ab.

Konfiguriere die SPI Schnittstelle des AVR so:

- MSTR=1
  der Mikrocontroller ist der Master
- DORD=0
  es wird immer zuerst das höherwertigste Bit (MSB) übertragen
- CPOL=0
  die Taktleitung sendet positive Impulse
- CPHA=0
  bei der steigenden Flanke des Taktimpulses liest der AVR ein Bit von der Leitung MISO ein und bei der fallenden Flanke gibt er das nächste Bit an MOSI aus.

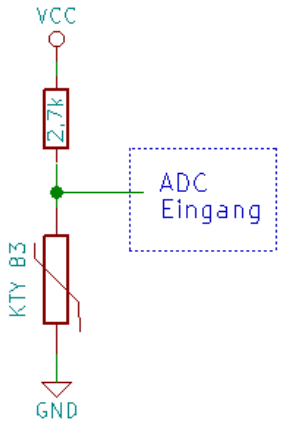
7 Sensoren

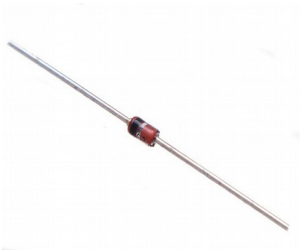
In diesem Kapitel stelle ich dir einige Sensoren vor, die ich bisher verwendet habe. Sie sind vor für den Bau von Maschinen und Robotern gut geeignet.

7.1 Temperatur

7.1.1 KTY-Sensoren

Temperaturen von Gasen (Luft) kann man ziemlich einfach mit den Sensoren der Serie KTY83 messen. Diese Sensoren wirken (richtig herum angeschlossen) wie Widerstände mit etwa 1000 Ohm bei 25 °C. Je höher die Temperatur ist, umso höher ist auch der Widerstands-Wert.



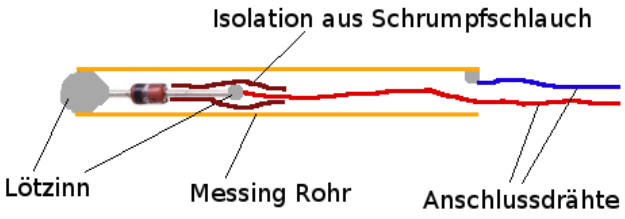


Schließe den Sensor zusammen mit einem 2,7 kΩ Widerstand an einen analogen Eingang des Mikrocontrollers an, und konfiguriere den ADC so, dass er die Spannungsversorgung von 3,3 Volt als Referenzspannung verwendet. Bei Zimmertemperatur erhältst du von einem AVR Mikrocontroller ungefähr den Messwert 280. Je höher die Temperatur ist, umso höher der Messwert.

Das Datenblatt des Temperatursensors enthält Tabellen, an denen du den Widerstandswert für viele Temperaturen ablesen kannst. Den erwarteten ADC Messwert pro Widerstandwert berechnest du so:

$$ADC = \frac{1024 * R_{KTY}}{R_{KTY} + 2700}$$

Wenn du die Temperatur von Flüssigkeiten messen willst, musst du den Sensor in ein dünnes Rohr aus Messing einlöten, damit er nicht nass wird:



7.1.2 NTC Thermistoren

Ein anderes häufig verwendetes Bauteil zum Messen von Temperatur ist der NTC Thermistor, auch bekannt als Heißleiter.

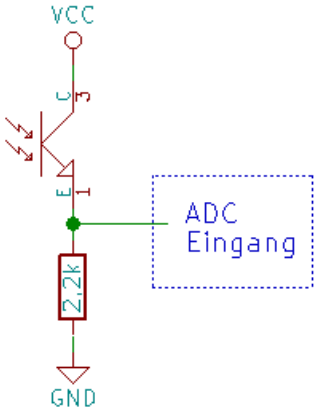
Bei diesen Bauteilen ist der Widerstand kleiner, je höher die Temperatur ist. Ihre Kennlinie ist nicht linear, aber mit zwei Widerständen kann man dennoch eine Genauigkeit erreichen, die für den Hausgebrauch ausreichend ist. Auf der Webseite

http://www.electronicdeveloper.de/MesstechnikNTCLinearR.aspx

findest du ein Tool, das bei der Berechnung der Widerstände hilft.

7.2 Helligkeit

Die Helligkeit von Licht kann man mit einem Photo-Transistor messen. Je mehr Licht auf den Phototransistor fällt, umso besser leitet er den Strom. Wenn du ihn wie im folgenden Schaltbild an einen analogen Eingang des Mikrocontrollers anschließt, liefert der ADC einen kleinen Wert bei Dunkelheit und einen großen Wert bei hellem Licht.



PT331C

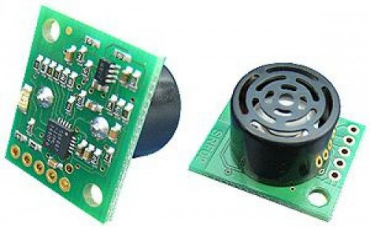
Osram BP 103

Die meisten Phototransistoren sehen aus, wie Leuchtdioden. Es gibt Modelle mit Infrarot-Filter, die auf das unsichtbare Infrarot Licht viel stärker reagieren, als auf sichtbares Licht. Ihr Gehäuse sieht für das menschliche Auge schwarz und undurchsichtig aus.

7.3 Distanz

7.3.1 Mit Ultraschall

Distanzen kann man mit Ultraschall messen. Dazu gibt es fertige Module, zum Beispiel das Modul SRF02 von Devantech. Es wird über einen I^2C Bus an den Mikrocontroller angeschlossen.



Das Modul funktioniert folgendermaßen: Es sendet einen kurzen Piepston mit Ultraschall-Frequenz aus. Dann benutzt es den Schallwandler als Mikrofon und wartet ab, bis das Echo zurück kommt.

Aus der Zeitspanne zwischen Sendung und Empfang ergibt sich der Abstand zur nächsten Wand oder zum nächsten Hindernis. So kann der Sensor Distanzen im Bereich 15-600 cm erfassen. Das Ergebnis wird als Integer-Zahl in Zentimetern geliefert. Devantech produziert weitere Varianten dieser Sensoren mit anderem Messbereich.

Die Stromversorgung des SRF02 muss ungefähr 5 V haben.

7.3.1.1 Programmierung

Die Steuerung über den I^2C Bus funktioniert über Register.. Die 7-Bit Adresse des Sensors ist 1110000. Um eine Messung zu starten, schreibst du den Wert 0x51 in das Register 0. Dann wartest du 65 Millisekunden. Anschließen kannst du das Messergebnis auf den Registern 2 und 3 auslesen. Die Distanz in Zentimetern beträgt

$$\text{Distanz} = (\text{Register2} \cdot 256) + \text{Register3}$$

7.3.2 Mit Licht

Eine ganze andere aber ebenfalls durchaus praktikable Methode der Distanzmessung findet bei den Optischen Distanz Sensoren von Sharp statt.

Diese Sensoren senden auf der einen Seite einen unsichtbaren Infrarot-Licht-Strahl aus und empfangen das von Hindernissen reflektierte Licht auf der anderen Seite. Je nach Distanz kommt das Licht mehr oder weniger seitlich versetzt beim Empfänger an. Genau dieser Versatz wird gemessen und als analoge Spannung ausgegeben. Du kannst diese Sensoren daher direkt an die analogen Eingänge des Mikrocontrollers anschließen.

- GP2D120 für 4 – 30 cm
- GP2Y0A21YK0F für 10 – 80 cm
- GP2Y0A02YK für 20 - 150 cm

Auch diese Sensoren benötigen 5 V Spannungsversorgung, und die sollte sehr sauber sein, am Besten aus einem separaten linearen Spannungsregler kommen.

In den Datenblättern der optischen Sharp Distanz Sensoren findest du Diagramme, an denen du die Spannung für unterschiedliche Distanzen ablesen kannst.

7.4 Beschleunigung und Neigung

Beschleunigung misst man mit einem „Accelleration Sensor“. In Fahrzeugen misst der Sensor, wie stark man beschleunigt oder bremst und wie stark die Fliehkraft bei Kurvenfahrt ist. Bei extremen Kräften lösen Beschleunigungs-Sensoren die Airbags aus.

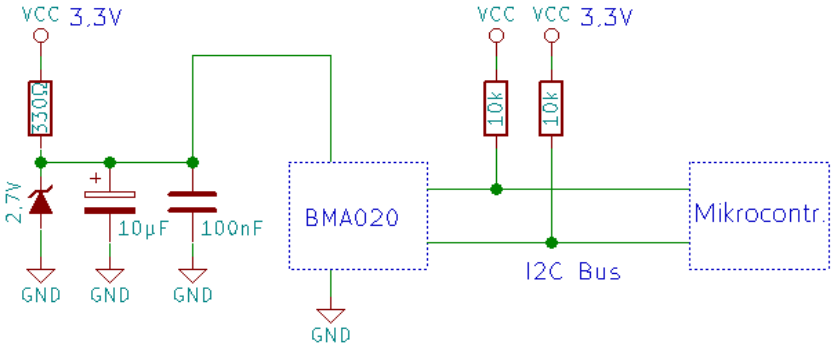
Manche Computer schalten ihre stoß-empfindlichen Festplatten schnell aus, wenn sie vom Tisch fallen – was sie mit Hilfe eines Beschleunigungs-Sensors erkennen.

Solange die Erde uns nach unten zieht, kann man mit solchen Sensoren auch den Neigungswinkel messen, da der Sensor die Richtung der Erdanziehung erfasst.

Ich benutze gerne den BMA020 von Bosch. Er enthält drei mechanische Sensoren, einen Messverstärker, sowie einen Mikrocontroller. Das alles ist in ein winziges Gehäuse mit nur 3mm Kantenlänge verpackt.

Dieser Sensor kann Beschleunigung in alle Richtungen messen (nach oben, unten, links, rechts, vorne und hinten). Er liefert das Messergebnis in Form von drei Integer Zahlen wahlweise über I^2C oder SPI Bus.

Die Stromversorgung muss im Bereich 2,0 bis 3,6 Volt liegen und sollte möglichst frei von Störungen sein. Da die Stromaufnahme des Sensor unter 1 mA liegt, kann man die Spannungsversorgung bequem mit einer Zenerdiode und ein R/C Filter realisieren:



Die Zenerdiode reguliert die Versorgungsspannung für den Sensor auf 2,7 Volt. Die beiden Kondensatoren stabilisieren sie, so dass die Spannung frei von Störimpulsen ist. Das die Spannungsversorgung des Sensors geringer ist, als die des I²C Bus stört nicht weiter. Der Sensor ist dafür ausgelegt.

Der Sensor hat 12 Anschlüsse, die du folgendermaßen beschaltest:

- 2+5+9 = 2,7 Volt
- 3+7 = GND
- 6 = SCK vom I²C Bus
- 8 = SDA vom I²C Bus

Alle anderen Pins lässt du unbenutzt.

Beim ELV Versand kannst du unter der Artikel-Nummer 68-091521 eine kleine Platine bestellen, auf der sich der BMA020 befindet, sowie ein Spannungsregler und ein paar Kleinteile drumherum.

7.4.1.1 Programmierung

Der Sensor stellt über den I²C Bus Zugriff auf seine Register bereit, wo man den aktuellen Messwert auslesen kann und diverse Funktionen konfiguriert. Die 7-Bit Adresse des Sensors ist 1110000.

Der Sensor ist standardmäßig so konfiguriert, dass er 25 mal pro Sekunde eine Messung im 2G Messbereich durchführt. Lies das Messergebnis aus den Registern 2 bis 7 ein und rechne die Zahlen wie folgt zusammen:

```
int8_t daten[6];
… Register 2-7 über I2C in dieses Array einlesen
int16_t x = (daten[0] >> 6) + (daten[1] << 2);
int16_t y = (daten[2] >> 6) + (daten[3] << 2);
int16_t z = (daten[4] >> 6) + (daten[5] << 2);
```

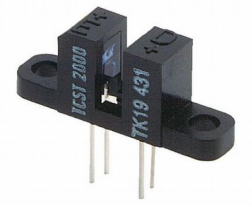
Die Variablen x, y und z enthalten dann die Messwerte als 10-bit Integer Zahl mit Vorzeichen. Der Wert 1023 oder -1023 bedeutet Beschleunigung in Höhe von 2G und das Vorzeichen gibt die Richtung an.

7.5 Lichtschranken

Lichtschranken benutzt man im Maschinenbau oft, um helle Markierungen an bewegten Objekten zu erfassen oder um die Drehzahl von Motoren oder Rädern zu messen.

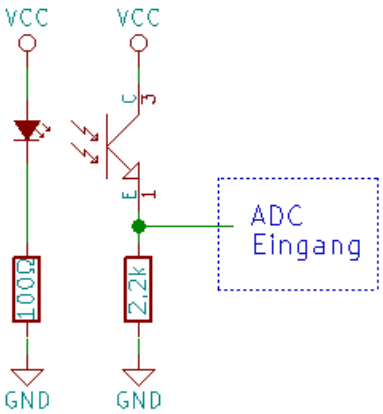
7.5.1 Gabel-Lichtschranken

Gabel-Lichtschranken wie die CNY-37 bestehen aus einer Infrarot-Leuchtdiode und einem Fototransistor. Die Leuchtdiode leuchtet ständig. Der Fototransistor kann so erkennen, ob der Lichtstrahl durch ein undurchsichtiges Objekt unterbrochen wird.



Oft benutzt man diese Gabel-Lichtschranken zusammen mit gelochten Rändern, um die Drehzahl vorn irgendwelchen rotierenden Achsen zu messen.

In CD-Laufwerken benutzt man solche Lichtschranken, um die Startposition der beweglichen Linse festzulegen.



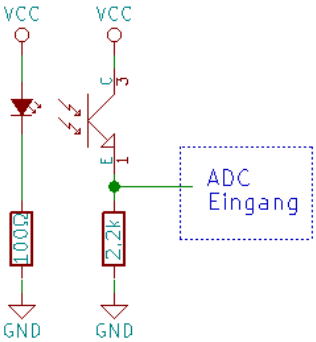
7.5.2 Reflex-Koppler

Bei Reflex-Kopplern wie dem CNY 70 befindet sich eine Infrarot-Leuchtdiode direkt neben einem Fototransistor. Wenn sie ein reflektierendes Objekt kurz vor dem Sensor befindet, dann schaltet der Fototransistor durch.

Auch diese Sensoren kann zur Erfassung von Positionen und Drehzahlen verwenden, indem man auf das dunkle bewegliche Objekt einen hellen Fleck malt (oder umgekehrt).

Roboter benutzen solche Sensoren, um auf den Boden gemalte Linien zu verfolgen.

Diese Sensoren schließt man genau so an den Mikrocontroller an, wie eine Gabellichtschranke. Wenn du einen analogen Eingang verwendest, kannst du sogar ganz grob die Helligkeit oder den Abstand des Objektes messen.



Je kleiner der Vorwiderstand der Leuchtdiode ist, umso heller leuchtet sie und umso größer wird die Reichweite des Sensors. Die Reichweite hängt sehr vom Sensortyp ab und liegt typischerweise im Bereich weniger Millimeter bis zwei Zentimeter.

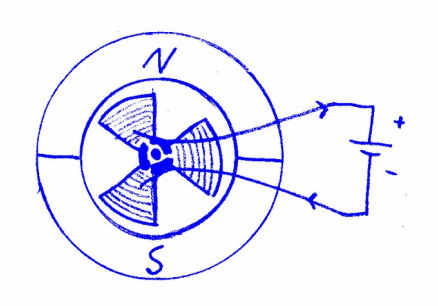
8 Motoren

8.1 DC-Motoren

DC Motoren werden mit Gleichspannung betrieben. Sie enthalten meistens 3 oder 5 rotierende Spulen, die von einem fest stehenden magnetischen Ring umschlossen sind.

8.1.1 Funktionsweise

Die folgende Zeichnung zeigt einen DC Motor im Querschnitt.



Der Punkt in der Mitte ist die rotierende Achse. Die drei Spulen sind auf einem drei-flügeligen Eisenkern gewickelt, den man Anker nennt. Der ganze Anker mitsamt Spulen kann sich drehen, während der äußere magnetische Ring fest steht.

Die drei Spulen werden durch Schleifkontakte in der Nähe der Achse mit Strom versorgt. Diese Schleifkontakte sind nicht ewig haltbar. Bei vielen Motoren kann man sie als Ersatzteil nachkaufen und austauschen.

Durch die rechts liegende Spule fließt der Strom gegen den Uhrzeigersinn, daher wird sie vom Südpol angezogen. Durch die beiden links liegenden Spulen fließt der Strom im Uhrzeigersinn, daher werden sie vom Nordpol angezogen. Dementsprechend dreht sich der Anker fortlaufen rechts herum. Wenn man die Batterie umdreht, dreht sich der Motor links herum.

8.1.2 Eigenschaften

Die Drehzahl von DC Motoren hängt hauptsächlich von Höhe der angelegten Spannung ab. Eine Verdoppelung der Spannung ungefähr zu doppelter Drehzahl.

Je höher der Motor belastet wird, umso mehr Strom verbraucht er. Wenn der Motor unter Spannung blockiert wird, steigt die Stromaufnahme locker auf den zehnfachen Wert.

Wenn man den Motor überlastet, wird er zu heiß. Wenige Sekunden Überlast verträgt jeder Motor, aber danach benötigt er genug Zeit, um sich wieder abkühlen zu können. Daher muss man Motoren vor länger anhaltenden Blockierungen schützen.

Wenn man DC Motoren von außen mechanisch antreibt, werden sie zum Generator. Je schneller man sie dreht, umso mehr Spannung geben sie ab.

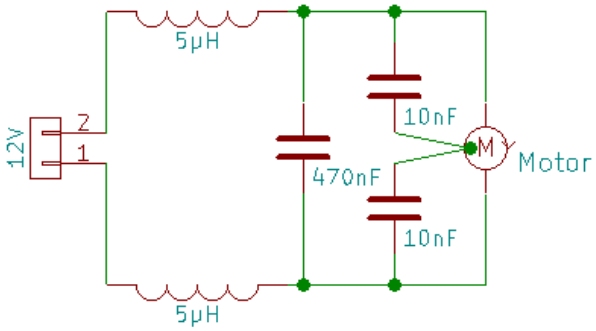
Elektronische Schaltungen, die Motoren ansteuern müssen diesen Effekt ggf. berücksichtigen. Zum Beispiel werden elektrische Spielzeug-Autos von den Kindern gerne über den Boden geschoben. Dabei können sie durchaus mehr Spannung abgeben, als die normale Betriebsspannung des Autos beträgt.

Wenn man die Stromzufuhr zum Motor abschaltet, drehen sie sich aufgrund des Schwungs noch eine ganze Weile weiter und erzeugen dabei Strom. Du kannst den Motor zu schnellerem Stillstand bringen, indem du die beiden Anschlüsse des Motors miteinander kurzschließt. Durch den Kurzschluss wird der „Generator“ stark belastet, so dass die Energie des Schwungs schnell verbraucht ist.

Aus dem Stand heraus benötigen DC Motoren recht viel Spannung und Strom, um anzulaufen. Wenn du die Spannung von Null auf langsam erhöhst, beginnen die meisten Motoren erst bei 30-50 % der Nennspannung, sich zu drehen. Aber sobald sie in Bewegung sind, kannst du die Spannung wieder herabsetzen, um die Drehzahl zu reduzieren.

8.1.3 Entstörung

An den Schleifkontakten entstehen ständig kleine Funken, wenn der Motor läuft. Funken erzeugen starke elektrische und magnetische Wellen, welche die angeschlossene Elektronik und andere Drahtlose Geräte in der Nähe massiv stören können. Darum muss man DC Motoren unbedingt entstören – auch die ganz kleinen. Das macht man mit der folgenden Schaltung:

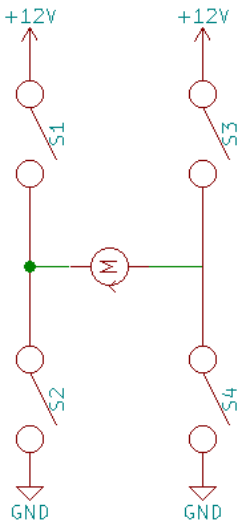


Die beiden kleinen 10nF Kondensatoren sollen direkt an das Gehäuse des Motors und die beiden Stromanschlüsse angelötet werden. Das Problem dabei ist, dass du ohne teure Messgeräte nicht herausfinden kannst, ob dein Motor korrekt entstört ist. Deswegen solltest du beim Kauf Motoren bevorzugen, die mitsamt Entstör-Schaltung geliefert werden.

8.1.4 H-Brücke

DC Motoren steuerst du mit MOSFET Transistoren an. Die Laufrichtung kannst du mit einem Relais umschalten, wie ich im vorherigen Kapitel über Relais beschrieben habe.

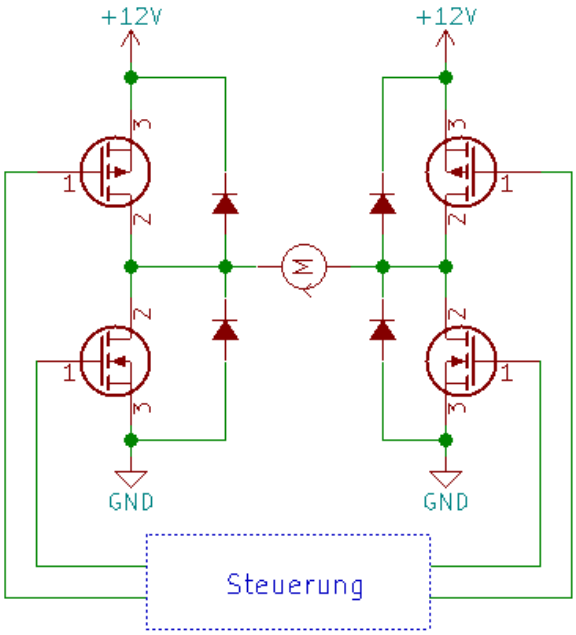
Eine weitere übliche Schaltung ist die sogenannte H-Brücke. Sie kommt ohne Relais aus.



Bei der H-Brücke gibt es folgende Zustände:

- S1 und S4 schalten durch, dann dreht sich der Motor rechts herum.
- S2 und S3 schalten durch, dann dreht sich der Motor links herum.
- S1 und S3 (oder S2 und S4) schalten durch, der Motor wird abgebremst.
- alle Schalter sind offen, der Motor ist Stromlos.

Wenn der Motor nun durch einen Mikrocontroller angesteuert wird, verwendet man anstelle der Schalter starke MOSFET Transistoren mit Freilaufdioden. Manche MOSFETS beinhalten diese Dioden übrigens bereits..



Eine solche H-Brücke kann man auch mit einem PWM Signal ansteuern, um nicht nur die Laufrichtung, sondern auch die Drehzahl des Motors zu beeinflussen. Von einer guten Steuerung erwartet man außerdem einen Gewissen Schutz vor Überlast.

Praktisch alle Motoren benötigen mehr als 3,3 Volt Spannungsversorgung, so dass die Steuerung eine Umsetzung der geringen Signalspannung vom Mikrocontroller auf die höhere Versorgungsspannung des Motors leisten muss.

Für kleine Motoren gibt es dazu fertige Mikrochips, die sowohl die Steuerung als auch die MOSFET Transistoren enthalten:

Chip	**Spannung**	**Strom**	**Beschreibung**
LB1649	6-24 V	1 A	Zwei H-Brücken, ohne Überlast-Schutz
L6202	12-48 V	1 A	Eine H-Brücke, thermischer Schutz
L293D	4,5-36 V	600 mA	Zwei H-Brücken, thermischer Schutz
L298N	6-46 V	2 A	Zwei H-Brücken, thermischer Schutz

Für größere Leistungen musst du die Schaltung aus einzelnen Transistoren aufbauen. Auch dafür gibt es hilfreiche Mikrochips, die das vereinfachen.

8.2 Schrittmotoren

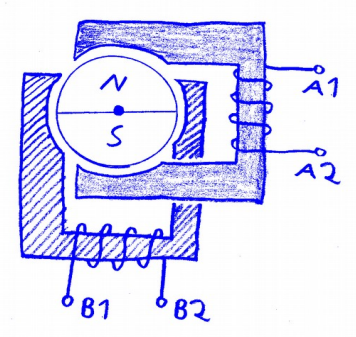
Schrittmotoren haben diesen Namen erhalten, weil sie sich sich schrittweise drehen. Bei jedem Schritt dreht sich die Achse nur ein kleines Stück auf einen ganz bestimmten Winkel.

Schrittmotoren eignen sich daher gut, um bewegliche Maschinenteile zu positionieren, zum Beispiel den Druckkopf eines Tintenstrahl-Druckers.

8.2.1 Funktionsweise

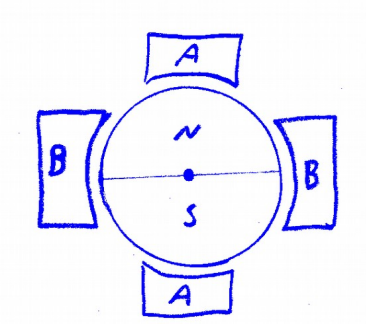
Schrittmotoren enthalten einen drehbaren Magneten auf der Achse und zwei feststehende Elektromagneten (Spulen mit Eisenkern) außen herum. Eine elektronische Steuerung legt Spannungen wechselweise an die Spulen A und B, so dass der magnetische Kern in Rotation versetzt wird.

Die folgende Zeichnung zeigt einen einfachen Schrittmotor im Querschnitt. Dieser Motor hat vier Schritte pro Umdrehung, also 90 Grad pro Schritt:

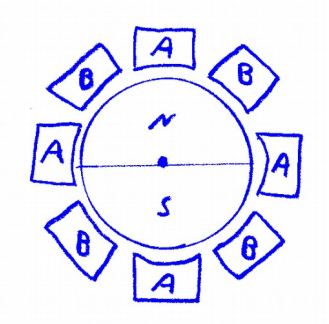


	A1 = plus A2 = minus B = stromlos
	A = stromlos B1 = plus B2 = minus
	A1 = minus A2 = plus B = stromlos
	A = stromlos B1 = minus B2 = plus

Handelsübliche Schrittmotoren machen allerdings viel kleinere Schritte, zum Beispiel 200 pro Umdrehung, entsprechend 1,8 Grad pro Schritt. Dies erreichen die Hersteller, indem sie die Elektromagneten mit mehr „Fingern“ ausstatten, die immer abwechselnd den Spulen A und B zugeordnet sind:



4 Schritte je 90 Grad



8 Schritte je 45 Grad

8.2.2 Eigenschaften

Verglichen mit DC Motoren sind Schrittmotoren sehr schwach. Um die gleiche mechanische Kraft zu erreichen, muss ein Schrittmotor viel größer sein und er verbraucht auch viel mehr Strom. Dafür können Schrittmotoren ganz exakt an der gewünschten Position angehalten werden, was bei DC Motoren nicht möglich ist. Aufgrund der stark induktiven Wirkung hängt sinkt die Stromstärke und damit die Kraft des Motors bei höheren Schrittfrequenzen.

Die Nennspannung des Motors ist die maximale Spannung bei Stillstand und niedrigen Drehzahlen. Je schneller sich der Motor dreht, umso mehr Spannung kann man anlegen. Letztendlich kommt es darauf an, den maximal zulässigen Strom nicht zu überschreiten, sonst wird der Motor zu heiß.

Eine gute Steuerelektronik ist daher imstande, mit erheblich höheren Spannungen, als der Nennspannung, zu arbeiten und die Stromstärke zu regulieren. Nur so kann man die die volle Leistung aus dem Motor heraus holen.

8.2.3 Ansteuerung

Im Prinzip kannst du Schrittmotoren mit zwei H-Brücken ansteuern, wie sie im vorherigen Kapitel beschrieben sind. Um die volle Leistung ausnutzen zu können, brauchst du jedoch eine Steuerung, welche die Stromstärke entsprechend der Schrittfrequenz reguliert. Eine solche Steuerung ist recht komplex.

Auf der Webseite http://www.roboternetz.de/schrittmotoren.html findest du einen entsprechenden Bauplan mit Erklärung. Solche Steuerungen kannst du auch bei Händlern für Robotik und Industrie kaufen. Zum Beispiel gibt es den RN-Stepper297 bei http://shop.robotikhardware.de:

Dieses Modul unterstützt sogar den Halbschritt-Modus, der die Anzahl der Schritte pro Umdrehung verdoppelt, indem er zeitweise beide Spulen unter Strom setzt:

	A1 = plus A2 = minus B = stromlos
	A1 = plus A2 = minus B1 = plus B2 = minus
	A = stromlos B1 = plus B2 = minus
	A1 = minus A2 = plus B1 = plus B2 = minus
	A1 = minus A2 = plus B = stromlos
	A1 = minus A2 = plus B1 = minus B2 = plus
	A = stromlos B1 = minus B2 = plus
	A1 = plus A2 = minus B1 = minus B2 = plus

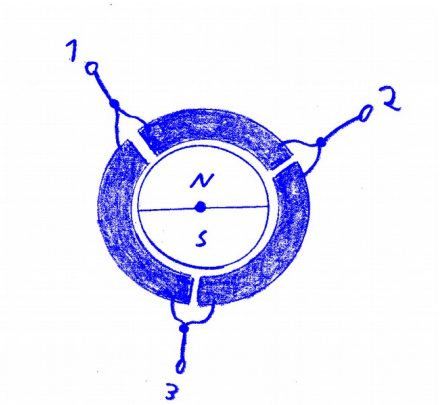
8.3 Brushless/Drehstrom Motoren

Im Modellbau benutzt man die Begriffe „Brushless Motor“ und „Brushless Controller“. Die technischen Fachbegriffe sind jedoch „Drehstrom-Motor“ und „Wechselrichter“.

Drehstrom-Motoren sind bei gleicher Größe und Gewicht erheblich Stärker als herkömmliche DC Motoren und außerdem Wartungsfrei,weil sie keine Schleifkontakte enthalten.

8.3.1 Funktionsweise

Drehstrom-Motoren enthalten einen rotierenden Magneten, der von drei fest stehenden Spulen umschlossen ist.



Die Steuerelektronik legt Spannung in einem rotierenden Muster an diese drei Spulen an. Der Magnet folgt dem rotierenden Magnetfeld der Spulen und dreht sich mit. Drehstrom-Motoren mit drei Spulen werden in sechs Phasen angesteuert:

	1 = minus 2 = minus 3 = plus
	1 = plus 2 = minus 3 = plus
	1 = plus 2 = minus 3 = minus
	1 = plus 2 = plus 3 = minus
	1 = minus 2 = plus 3 = minus
	1 = minus 2 = plus 3 = plus

8.3.2 Eigenschaften

Drehstrom-Motoren sind wartungsfrei, weil sie keine Verschleißteile, wie Schleifkontakte (engl. Brushes) enthalten. Sie sind auch erheblich stärker, als gleich große DC Motoren, aber sie sind auch aufwändiger anzusteuern. Sofern im Modellbau-Katalog nicht anders angegeben, sind Brushless-Motoren für eine Versorgungsspannung von 7,2 Volt ausgelegt. Es kommt allerdings weniger auf die korrekte Spannung an, sondern vielmehr auf die Temperatur, die durch mechanische Belastung entsteht.

In Modellbau-Katalogen ist die Drehzahl normalerweise in der Einheit kV angegeben. Die Angabe „3200 kV“ bedeutet, dass der Motor ohne Last auf 3.200 Umdrehungen pro Minute und pro Volt kommt. Bei 7 Volt Spannung erreicht er also ohne Last 22.400 Umdrehungen pro Minute.

Unter Last wird die Drehzahl etwas weniger sein.

Die Stromaufnahme hängt wie bei DC Motoren von der Belastung ab. Die Motoren können bei Überlast und beim Anlaufen kurzzeitig einen sehr hohen Strom aufnehmen. Sie werden dabei heiß, vertragen Überlast daher nur für wenige Sekunden und benötigen danach genügend Zeit zum Abkühlen.

8.3.3 Ansteuerung

Drehstrom-Motoren steuert man prinzipiell ähnlich der vorherigen H-Brücke mit 6 Transistoren an. Eine passende Steuerung sorgt für ein rotierendes Magnetfeld, dessen Drehzahl sich der aktuellen Drehzahl des Motors automatisch anpasst.

Das elektrisch erzeugte Magnetfeld darf sich nicht schneller drehen, als der Motor ihm folgen kann. Wenn das elektrische Magnetfeld mehr als 120 Grad im Voraus läuft, blockiert der Motor schlagartig oder wechselt sogar seine Drehrichtung.

Manche Steuerungen nutzen einen separaten Winkel-Sensor, um das Drehfeld korrekt anzupassen. Sie funktionieren vor allem beim Anlaufen unter Last zuverlässiger, als Steuerungen ohne Sensor.

Der Eigenbau entsprechender Schaltungen lohnt sich kaum, weil man im Modellbau-Fachgeschäft fertige Module zu fairen Preisen bekommen kann. Diese Regler sind in der Regel für Batteriespannungen im Bereich 6 bis 14,4 Volt ausgelegt und sie vertragen Ströme um 100 Ampere! Im Modellbau werden diese Controller kurz ESC genannt.

Bei vielen Brushless-Controller kann man das Beschleunigungsverhalten und die Bremsleistung konfigurieren. Besonders preisgünstig sind die Sets aus Motor und Controller, wie das folgende Set von Conrad Elektronik:

Die Schnittstelle zum Mikrocontroller ist bei allem Brushless-Controllern ein drei-poliges Kabel mit folgender Belegung:

- GND = schwarz oder braun
- VCC = rot (5 oder 6 Volt)
- PWM Signal = orange, gelb oder weiß

Die CC Leitung ist aus Sicht des Reglers ein Ausgang. Mit dieser Leitung wird bei Modell-Autos der Funk-Empfänger und die Lenkung versorgt. Die Leitung darfst du normalerweise mit bis zu 1,5 Ampere belasten. Wenn du einen Mikrocontroller anschließt, der schon eine eigene Spannungsversorgung hat, brauchst du diese Leitung nicht.

Um den Brushless-Controller anzusteuern, sendet der fortlaufend alle 20 Millisekunden einen positiven Impuls mit 0,5 bis 2,5 Millisekunden Länge.

- Bei der mittleren Puls-Länge von 1,5ms steht der Motor still.
- Je länger die Impulse sind, umso schneller dreht der Motor vorwärts.
- Je kürzer die Impulse sind, umso schneller dreht der Motor rückwärts.
- Die meisten ESC führen ein hartes Brems-Manöver aus, wenn man die Richtung plötzlich von Vorwärts nach Rückwärts ändert. Mann muss danach eine Pause von etwa einer halben Sekunde mit 1,5mV Pulsen einlegen, damit sich die Bremse wieder löst.

8.4 Servos

Modellbau-Servos sind sehr kompakte Getriebemotoren mit einem Winkelsensor und eingebauten Steuer-Elektronik. In Modell-Autos bewegen Servos die Lenkung, in Flugzeugen bewegen sie die Klappen und Ruder.

Servos drehen ein Kreuz oder einen Hebel in einem Bereich von Null bis 180 Grad, also eine halbe Umdrehung. Servos gibt es je nach Anwendungsfeld in unterschiedlichen Größen und Leistungen.

Sie werden aber alle auf die gleiche Weise über ihr drei-poliges Anschlusskabel angesteuert:

- GND = schwarz oder braun
- Stromversorgung 4,8 bis 6 V = rot
- PWM Signal = orange, gelb oder weiß

Zur Ansteuerung muss der Mikrocontroller fortlaufend alle 20 Millisekunden einen positiven Impuls mit ungefähr 1,5 Millisekunden Länge an den Servo senden. Je nach Länge des Impulses fährt der Servo einen anderen Dreh-Winkel an.

Die Zuordnung von Pulslänge zu Winkel sind nicht genau festgelegt. Jedes Servo verhält sich hier etwas anders. Für Servos mit einem Drehbereich von 180° sind 1 bis 2 ms üblich.

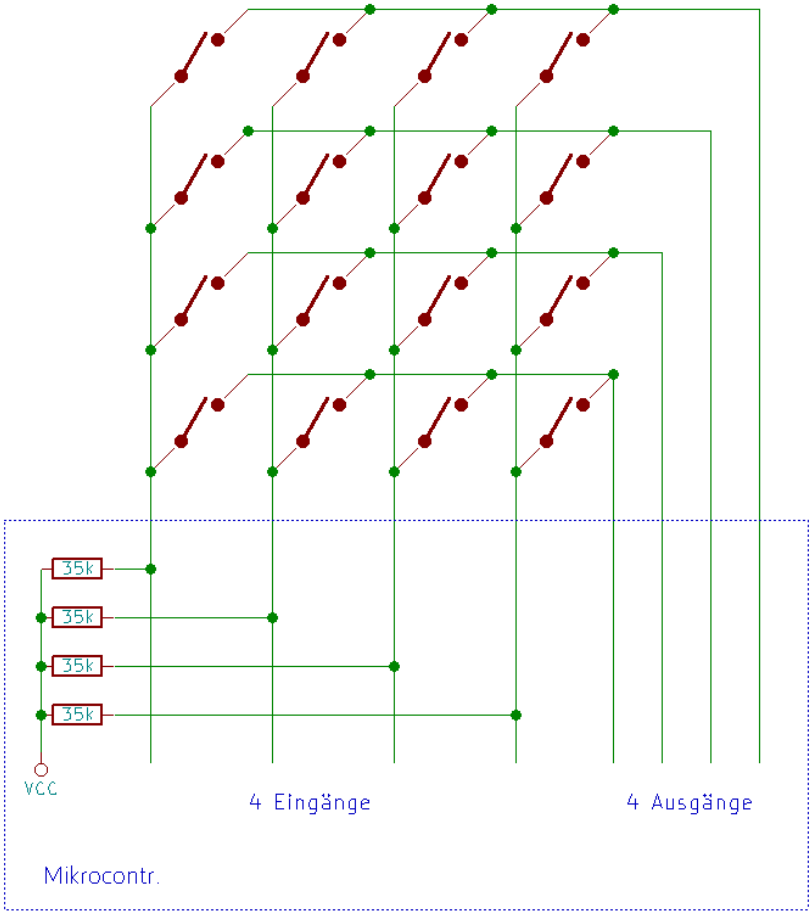
Solange der Servo regelmäßige Impulse empfängt, korrigiert er seine Position automatisch nach, falls der Hebel (oder das Kreuz) durch äußere Kräfte verstellt wird. Wenn der Servo kein Signal mehr empfängt, schaltet er die Stromzufuhr zum Motor aus.

Schau dir dieses Video an: https://youtu.be/yQMcr3PNxV8

9 Bedienfelder

9.1 Tasten-Matrix

Der Sinn einer Tasten-Matrix besteht darin, viele Taster an wenige Leitungen anzuschließen. Die Tastaturen von Computern sind nach diesem Prinzip konstruiert.Das folgende Schaltbild zeigt eine Matrix mit 4x4 Tasten. Werden nur acht Anschlüsse benötigt, um alle 16 Tasten auszulesen.



Die Abfrage der Tasten funktioniert folgendermaßen:

- Bei den vier Eingängen müssen die Pull-Up Widerstände aktiviert sein.
- Danach aktiviert das Programm eine Reihe, indem der entsprechende Pin als Ausgang mit Low Pegel konfiguriert wird. Die anderen Reihen bleiben inaktiv (also als Eingang konfiguriert).
- Nachdem eine Reihe aktiviert wurde, kann das Programm an den Eingängen erkennen, welche Tasten in dieser einen Reihe gedrückt sind.
- Danach wird die Reihe deaktiviert (indem der Pin als Eingang konfiguriert wird) und die nächste Reihe ausgelesen. Und so weiter.

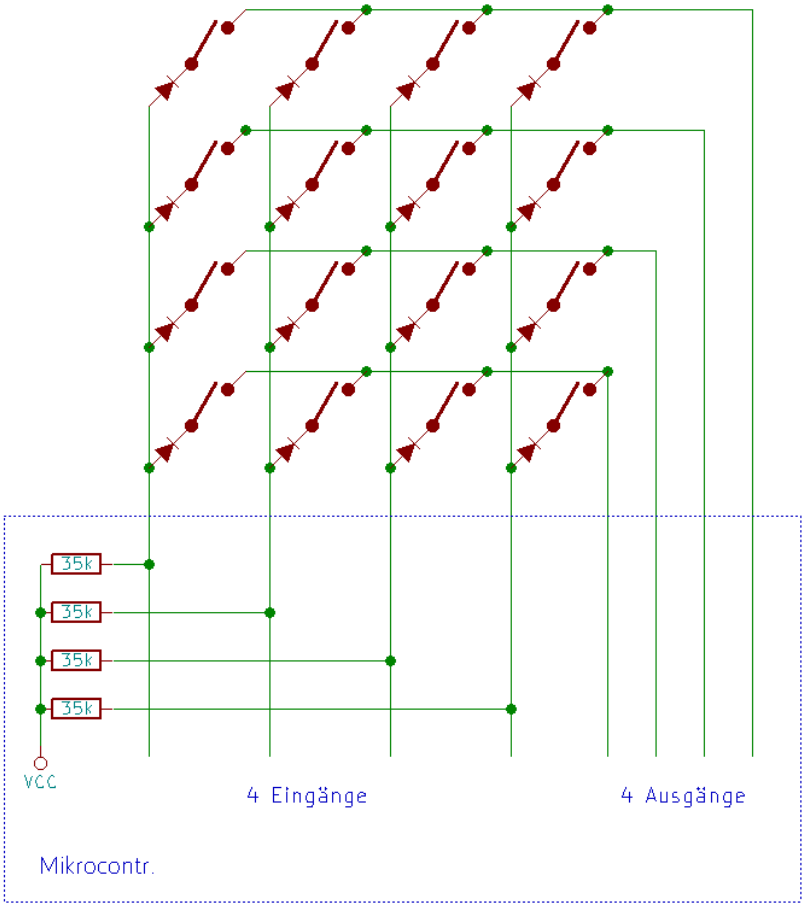
So wird von der ganzen Tastatur eine Reihe nach der anderen ausgelesen.

Diese Schaltung geht davon aus, dass niemals Tasten aus mehreren Reihen gleichzeitig gedrückt werden. Sonst kommt es zu falschen Auslese-Ergebnissen.

9.2 Schalter-Matrix

Wenn man die Taster einer Tasten-Matrix durch Schalter ersetzt, kann das Szenario mit mehreren gleichzeitig geschlossenen Schaltern aber schon viel eher auftreten.

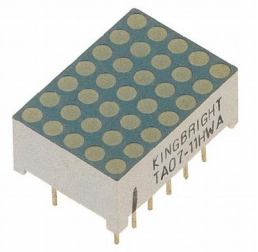
Damit der Mikrocontroller dennoch jeden Schalter einzeln korrekt abfragen kann, benötigt man zusätzlich für jeden Schalter eine Diode:



In der so erweiterten Matrix dürfen beliebig viele Schaltkontakte gleichzeitig geschlossen sein, und dennoch kann der Mikrocontroller jeden einzelnen korrekt abfragen.

Die Dioden verhindern, dass der Strom die Matrix in die falsche Richtung (rückwärts) durchläuft.

9.3 LED-Matrix

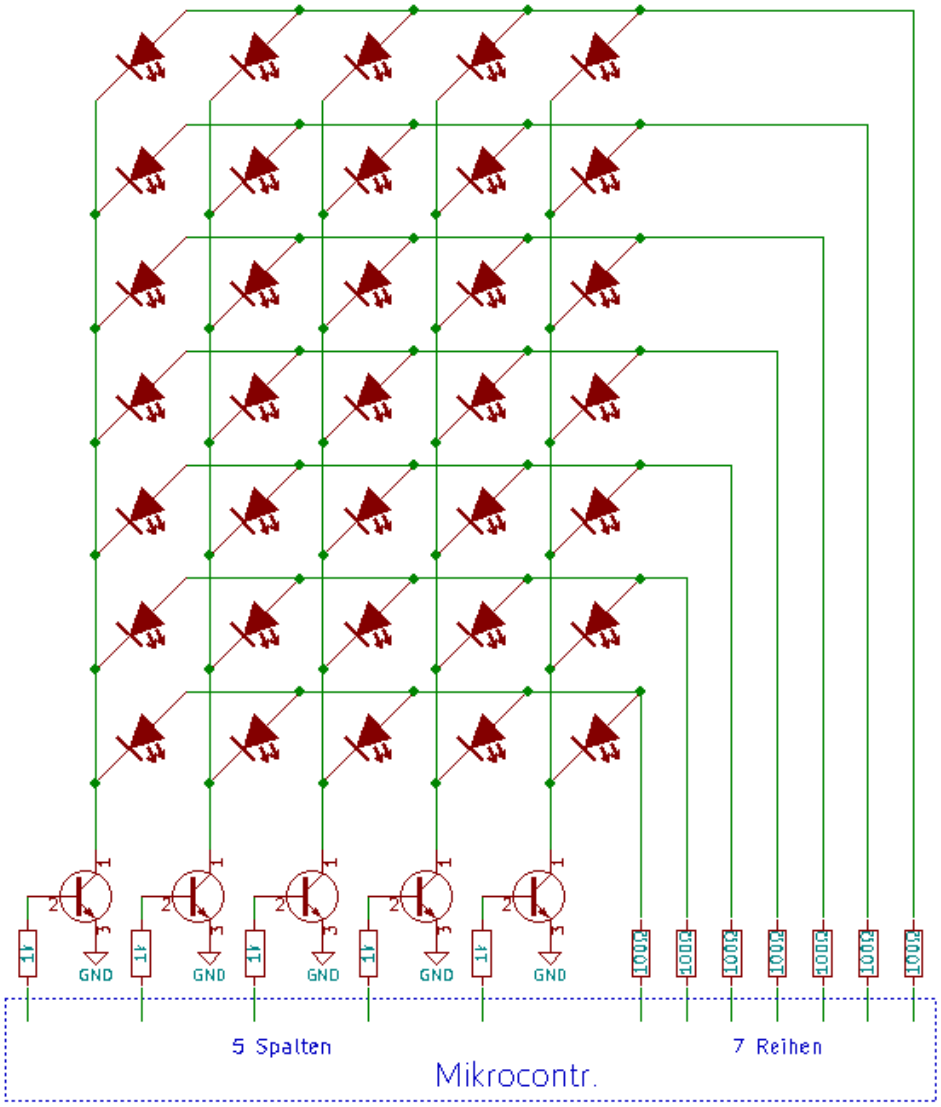


Das Prinzip der Matrix lässt sich auch auf Leuchtdioden übertragen.

Im Handel kann man fertige LED-Matrizen als Modul kaufen. Aus solchen Modulen stellt man Beispielsweise die Etagen-Anzeige in einem Aufzug her. Die Regionalzüge der Deutschen Bahn zeigen auf LED-Matrizen den Namen der nächsten Haltestelle an.

Diese Anzeigen gibt es in unterschiedlichen Größen. Ich rate allerdings von Modulen mit mehr als 5 Spalten ab, weil es sonst schwierig wird, sie ausreichend hell und ohne Flimmern anzusteuern.

LED Matrizen schließt man so an Mikrocontroller an:



Der Mikrocontroller steuert diese Anzeige ähnlich der Tastaturmatrix Spaltenweise an:

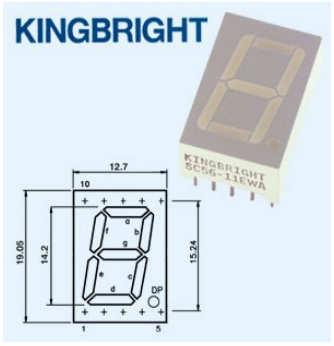
- Zuerst aktiviert er eine Spalte, indem der entsprechende Transistor eingeschaltet wird.
- Danach schaltet er die gewünschten LED‘s ein, indem er die entsprechenden Reihen-Ausgänge auf High legt.
- Dann wartet er ein bisschen.
- Dann schaltet er die Spalte wieder aus und steuert die nächste Spalte an. So werden alle Spalten nacheinander angesteuert. Nach der letzten Spalte kommt wieder die erste an die Reihe.

Achte darauf, den Mikrocontroller nicht zu überlasten. Er verträgt maximal 200 mA insgesamt! Wenn du die Vorwiderstände der LED‘s für 20 mA berechnest, bist du auf der sicheren Seite. (7 · 20 mA = 140 mA).

Wenn du mehr als eine dieser Anzeigen anzusteuern hast (was in realen Anwendungen wohl immer der Fall sein wird), benutze Schieberegister, um die Anzahl der I/O Pins des Mikrocontroller zu erhöhen und um den Laststrom auf mehrere Mikrochips zu verteilen.

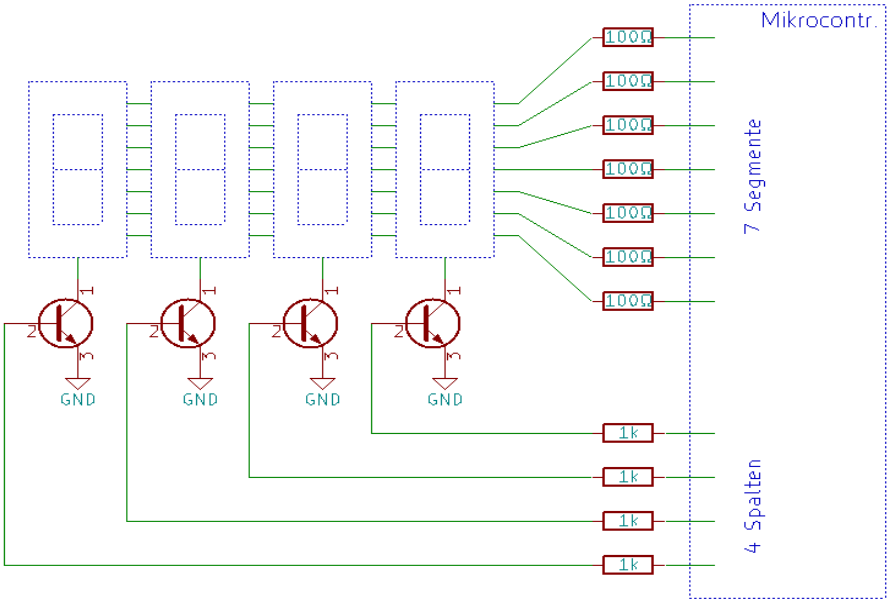
Beim Bau von großen LED Anzeigen solltest du allerdings stets berücksichtigen, das diese Anzeigen sehr viel Strom verbrauchen. Ich bevorzuge daher den Einsatz von beleuchteten LC-Displays.

9.4 7-Segment-Anzeige



Mit sieben Segment Anzeigen baut man numerische Displays auf. Verglichen mit Punkt-Matrix Anzeigen kommen sie mit viel weniger LEDs (nämlich eine pro Strich) aus und verbrauchen entsprechend weniger Strom.

Meistens kombiniert man mehrere solcher Anzeigen zu einer Matrix, wobei jede Spalte einer Ziffernposition entspricht und die Reihen den sieben Segmenten entsprechen.



Bei solchen Anzeigen werden die vier Ziffern immer abwechselnd angesteuert. Das kann man mit bis zu 6 Ziffern machen, von mehr würde ich allerdings wegen der sinkenden Leuchtkraft abraten.

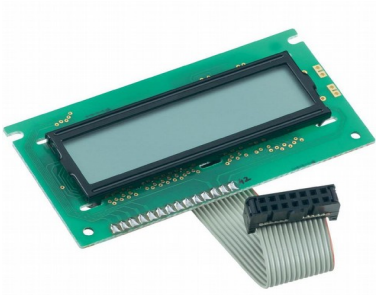
Diese Schaltung ist an einem AVR Mikrocontroller so gerade eben noch in Ordnung. Einen STM32 würde man damit überlasten. Bei STM32 müsstest du 220Ω Widerstände verwenden oder besser zusätzlich PNP Transistoren zur Verstärkung der oberen Ausgänge hinzufügen.

Bei umgekehrt gepolten Anzeigen (mit gemeinsamer Anode) müsstest du PNP Transistoren anstatt der oben gezeichneten NPN Transistoren verwenden.

9.5 LC-Display

Wesentlich moderner, als LED Anzeigen, sind LC-Displays. Sie sind kompakter und verbrauchen viel weniger Strom. Allerdings reagieren LC-Displays auf Temperatur-Änderungen. Je kälter es ist, umso schlechter ist der Kontrast. Bei Temperaturen unter 10 °C sind die meisten LC-Displays so blass, dass man sie kaum noch ablesen kann. Aber auch hohe Temperaturen beeinträchtigen die Lesbarkeit der Anzeige.

Darum bevorzugt man an Außen-Installationen und Fahrzeugen immer noch oft die alten LED Anzeigen. Recht preisgünstig sind die kleinen LC-Displays mit HD44780 Controller, wie dieses von Conrad Elektronik:



Diese Displays zeigen je nach Variante 1x16 bis 4x20 Teichen an. Die Schnittstelle zum Mikrocontroller besteht aus 8 oder 4 Daten-Leitungen, sowie zwei Steuerleitungen.

Auf der Webseite http://www.sprut.de/electronic/lcd/ findest du eine sehr detaillierte Beschreibung des HD44780 Chipsatzes, einschließlich Programmieranleitung.

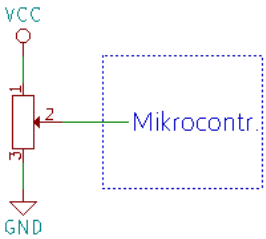
Im Fachhandel werden auch LC-Displays mit seriellen und I^2C Schnittstellen angeboten. Ich mag diese Displays, weil sie weniger Leitungen zum Mikrocontroller hin benötigen. Aber sie sind auch teurer.

Auf der Webseite http://www.lcd-module.de/ kannst du dir einen Eindruck verschaffen, was für tolle Display der Markt zu bieten hat. Dort gibt es sogar farbige Grafik-Displays mit eingebautem Flash-Speicher für Bilder und Videos, die man durch Befehle über den seriellen Port zur Anzeige aufruft.

Displays mit HD44780 Controller kann man fast immer mit 3,3 V betreiben, auch wenn in den Katalogen der Händler nur 5 V angegeben ist. Dann benötigt man allerdings für den Kontrast eine negative Hilfsspannung im Bereich -0,5 bis -2 Volt.

9.6 Potentiometer

Potentiometer sind komfortable analoge Eingabe-Elemente, die man in die Bedienfelder von elektronischen Geräten einbauen kann.



Sie bestehen aus einem Widerstand mit einem Schleifkontakt, den man durch Drehung der Achse positionieren kann. Der Drehbereich umfasst in der Regel etwa mehr als 180 Grad.

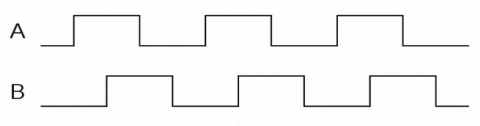
Für den direkten Anschluss an Mikrocontroller eignen sich Potentiometer mit 1 bis 10 kΩ. Schließe den linken Anschluss an GND an, den rechten an VCC (3,3 V) und den mittleren Anschluss an einen analogen Eingang des Mikrocontrollers. Je nach Drehung der Achse liefert der Potentiometer dann eine Spannung von 0 - 3,3 Volt, so dass der ADC des Mikrocontrollers Zahlenwerte von 0 bis 1023 liefert.

9.7 Drehimpulsgeber

Drehimpulsgeber liefern digitale Impulse, wenn man einer Achse dreht. In Bedienfeldern eignen sie sich als Ersatz für Potentiometer.

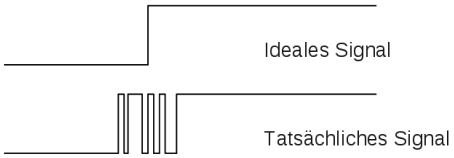
Äußerlich sehen diese Bauteile wie Potentiometer aus, man kann sie allerdings beliebig oft im Kreis drehen. Der mittlere Anschluss ist GND, die beiden anderen führen zum Mikrocontroller.

Die beiden Ausgänge liefern bei Drehung zeitlich versetzte Impulse:



Wenn die Achse anders herum gedreht wird, sind die Impulse anders herum versetzt.

In der Realität sieht das Signal aber nicht ganz so schön aus, denn wie bei allen Kontakten tritt auch hier das Sogenannte „Prellen“ auf. In dem Moment, wo ein Kontakt geschlossen oder geöffnet wird, geraten dessen Teile in mechanische Schwingungen. Diese Schwingungen bewirken beim Übergang von Low nach High (und zurück) ungewollte mehrfache Pegelwechsel.



Von diesem Effekt sind alle mechanischen Schalter und Taster betroffen.

Bei normalen Tastern benutze ich zum Entprellen manchmal einfach 100 nF Kondensatoren parallel zu den Kontakten. Die Kontakte von Drehgebern sind allerdings sehr empfindlich, sie vertragen den kurzzeitig hohen Entladestrom der Kondensatoren nicht.

Erfahrungsgemäß lässt sich das Prellen am besser per Software heraus filtern, indem man den Zustand der beiden Leitungen regelmäßig alle 1-2 ms abfragt und wartet, bis mehrmals der gleiche Wert (Low/High) vorlag. Das lässt sich gut mit einem Timer-Interrupt umsetzen. Da die Prellzeit zusammen mit der Alterung der Bauteile ansteigt, sollte man in der Software reichlich Reserve einplanen.

10 Webserver

Im Mikrocontroller-Umfeld schreibt man manchmal Mini-Webserver, zum Beispiel um den Status einer Maschine auf einem Smartphone anzuzeigen oder um ein Gerät komfortabel zu konfigurieren. Du kennst das von deinem Internet Router, und vielleicht hat dein Drucker auch einen eingebauten Webserver.

Um den Mikrocontroller als Webserver einzusetzen brauchst du ein Netzwerk-Interface. Verwende dazu vorzugsweise eins, wo die ganze Verarbeitungslogik des Netzwerkes (der TCP/IP Stack) schon eingebaut ist. Das vereinfacht die Programmierung kollosal.

Zum Beispiel eignet sich dazu ein Xport von der Firma Lantronix. Der Xport nimmt Verbindungen vom Web-Browser an und überträgt die Daten 1:1 über eine serielle Schnittstelle zum Mikrocontroller.

Oder du nimmst ein WLAN Modul mit ESP8266 Chip. Diese Teile unterstützen bis zu vier gleichzeitige Verbindungen, was praktisch ist, aber auch etwas schwieriger zu programmieren. Wenn es etwas mit Kabel sein soll, dann schau dir die Produkte von der Firma Wiznet an.

Zu jedem Netzwerkinterface gibt es reichlich Dokumentation, in der drin steht, wie man die Netzwerkparameter (z.B. IP Adresse und das WLAN Passwort) konfiguriert und wie man Verbindungen handhabt.

Was dort aber nicht drin steht ist, wie die Kommunikation zu einem Web-Browser funktioniert, konkret das HTTP Protokoll. In den vergangenen 25 Jahren wurden zahlreiche Erweiterungen für die Internet Protokolle vorgeschlagen. Manche wurden dauerhaft umgesetzt, andere inzwischen wieder verworfen. Für Anfänger ist es daher schwierig geworden, herauszufinden, wie Webserver nun wirklich mit dem Web Browser kommunizieren.

Bei diesem Aspekt möchte ich dir aushelfen. Ich erkläre dir die Aspekte des HTTP Protokolls, die für kleine Mini-Webserver notwendig sind. Die ganzen optionalen Erweiterungen lasse ich bewusst aus.

10.1 HTTP Protokoll

Web Browser nutzen das HTTP Protokoll, um mit Web Servern zu kommunizieren. Wenn du im Web-Browser eine Adresse in die Adressleiste eingibst, passiert folgendes:

1. Wenn du http://www.stefanfrings.de/index.html eingibst, findet der Web-Browser mit Hilfe eines DNS Servers heraus, welche IP-Adresse dahinter steckt.
2. Dann baut der Web-Browser eine sogenannte Socket Verbindung zu dieser IP-Adresse auf. Beim Verbindungsaufbau sendet er auch die Port Nummer 80 mit. Sie dient dazu zwischen Diensten (wie Email, Web, Fernwartung, Drucken) zu unterscheiden.
3. Der Web-Server nimmt die Verbindungsanfrage an und wartet dann auf eine Eingabe in Textform. Ab hier beginnt das eigentliche HTTP Protokoll.
4. Der Web-Browser sendet den Namen der Datei oder Seite, die er laden möchte. In diesem Fall sendet er:

   GET /index.html HTTP/1.1
   Host: stefanfrings.de

   gefolgt von zwei Zeilenumbrüchen.
5. Der Web-Server schaut nach, ob diese Seite existiert. Wenn das der Fall ist, sendet er folgende Antwort (Response), die aus zwei Teilen besteht: Einem Header und der angeforderten Datei.

HTTP/1.1 200 OK
Content-Type: text/html

<html>
<head>
<title>Stefan Frings, Willkommen</title>
</head>
...
</html>

Danach trennt der Server die Verbindung.

10.2 GET-Request Format

Für einen minimalen Webserver genügt es, nur auf GET-Requests zu antworten. Es gibt aber noch andere Request-Typen. Die erste Zeile eines GET-Requests sieht so aus:

GET /index.html HTTP/1.1

Hinter dem Wort GET kommt immer ein Dateiname. Der Dateiname beginnt immer mit einem Schrägstrich (Slash) und kann auch einen Pfad enthalten. Zum Beispiel so:

GET /kugelbahn/kugelbahn.jpg HTTP/1.1

Nach dem Dateinamen kommt die Versionsnummer des HTTP Protokolls. Anschließend kann der Web-Browser beliebig viele zusätzliche Informationen in beliebiger Reihenfolge senden. Aber nur eine Zeile ist Pflicht:

Host: stefanfrings.de

Hiermit zeigt der Browser an, welchen Server er anspricht. Das ist eigentlich doppelt gemoppelt, weil normalerweise jeder Server seine eigene IP-Adresse hat. Aber IP-Adressen sind knapp geworden und so hat man eine Möglichkeit geschaffen, dass mehrere Webserver sich eine Adresse teilen können.

Lass uns einmal nachschauen, was Web-Browser sonst noch alles so senden. Der folgende Ausdruck kommt von einem Firefox Browser unter Linux nach Eingabe von http://localhost.local:8080/index.html

```
GET /index.html HTTP/1.1
Host: stefanspc.local:8080
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:9.0.1) Gecko/20100101
Firefox/9.0.1
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
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
```

Mit **User-Agent** ist der Web-Browser so nett, sich zu identifizieren. Webserver nutzen diese Information, um ihre Inhalte ggf. an Eigenarten des Browser anzupassen.

Mit **Accept** teilt der Web-Browser mit, welchen Dateityp er erwartet. In diesem Fall bevorzugt er html, xhtml und xml Formate, aber alle anderen Dateiformate sind ihm auch recht (*/*).

Mit **Accept-Language** signalisiert der Web-Browser, welche Sprache der Benutzer bevorzugt. In diesem Fall ist deutsch die primäre Sprache, englisch aber auch Ok (das habe ich so in den persönlichen Einstellungen des Browser eingerichtet).

Mit **Accept-Encoding** gibt der Browser an, welche Kopressions-protokolle er unterstützt. Unabhängig von dieser Angabe kann jeder Browser auch unkomprimierte Seiten laden, und das ist auch in 90% der Fälle Praxis.

Für Mikrocontroller kommt Kompression mangels Rechenleistung und Speicher nicht in Frage. Aber gut zu wissen, das es so etwas gibt.

Mit **Accept-Charset** gibt der Browser an, welche Zeichensätze er bevorzugt. In diesem Fall möchte er gerne Seiten bekommen, die mit dem Zeichensatz ISO-8859-1 oder utf-8 codiert sind, was in Deutschland die üblichen Zeichensätze unter Linux sind.

Mit **Connection: keep-alive** berichtet der Web-Browser, dass er die Verbindung nach Übertragung der Webseite aufrecht erhalten möchte, wenn der Server es denn auch erlaubt.

Zum Vergleich zeige ich dir hier, wie die gleiche Abfrage unter Windows XP mit dem Internet Explorer 8 aussieht:

```
GET /index.html HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-flash, */*
Accept-Language: de
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)
Accept-Encoding: gzip, deflate
Host: stefanspc.local:8080
Connection: Keep-Alive
```

Nachdem der Web-Browser nun seinen Request gesendet hat, wartet er darauf, die angeforderte Seite oder Datei zu empfangen. Dies nennt man Response.

10.3 Response

Wenn die Angeforderte Seite nicht verfügbar ist, könnte die Antwort des Webservers ganz kurz ausfallen:

```
HTTP/1.1 404 Not found

Toll, was? Aber diese Seite gibt es hier nicht!
```

Im Web-Browser würde das so aussehen:

Die erste Zeile der Antwort ist der Header. Die möglichen Fehlercodes sind in diesem Dokument aufgelistet:

http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Für Mikrocontroller-Anwendungen genügt es, die folgenden Codes zu kennen:

- **200 Ok** – Die Seite oder Datei ist verfügbar und wird jetzt ausgeliefert.
- **403 Forbidden** – Du darfst diese Seite nicht aufrufen.
- **404 Not Found** – Die Datei existiert nicht.
- **500 Internal Server Error** – Der Webserver hat eine Störung.

Nach dem Status Code folgen beliebige viele weitere Kopfzeilen, dann eine Leerzeile und dann die angeforderte Datei (bei Code 200) oder eine Fehlermeldung, die der Browser im Fenster anzeigt. Wenn die Seite existiert und der Server auch nicht kaputt ist, könnte die Response im einfachsten Fall so aussehen:

```
HTTP/1.1 200 Ok
Content-Type: text/html

<html>
  <body bgcolor="green">
    Hallo, willkommen auf meiner ersten Webseite!
  </body>
</html>
```

In diesem Fall muss der Webserver anschließend die Verbindung beenden, um anzuzeigen, dass die Datei vollständig übertragen wurde. Diese Methode wendet man bei Mini-Webservern auf Mikrocontrollern meistens an, weil sie so schön einfach ist.

Das Ergebnis sieht so aus:

Wenn der Server etwas gesprächiger ist, teilt er dem Browser auch die Größe der Datei und weitere Informationen mit. Der Webserver von meiner Homepage antwortet so:

```
HTTP/1.1 200 OK
Date: Sat, 18 Feb 2012 21:17:08 GMT
Server: Apache
Last-Modified: Fri, 30 Dec 2011 20:58:50 GMT
ETag: "3c9478-470-4b5557ffce2b1"
Accept-Ranges: bytes
Content-Length: 1136
Content-Type: text/html

<html>
  <head>
    <title>Stefan Frings, Willkommen</title>
    ...
</html>
```

Interessant sind hier besonders die folgenden Header:

Mit **Content-Length** gibt der Webserver an, wie groß die Seite ist (in Bytes gezählt). So weiß der Web-Browser ganz genau, wie viele Daten er zu erwarten hat und kann bei zeitaufwändigen Ladevorgängen einen korrekten Prozent-Balken anzeigen. Außerdem ermöglicht dieser Header, die Verbindung länger aufrecht zu erhalten um sie für mehrere aufeinander folgende Requests wieder zu verwenden.

Mit **Content-Type** gibt der Webserver an, welchen Inhalt die angeforderte Datei hat. In diesem Fall ist es eine HTML Seite. Gängige Content-Typen sind:

- text/html Webseiten im HTML Format
- text/plain Einfacher unformatierter Text
- application/xml Daten im XML Format
- application/json Daten im JSON Format
- image/png Bilder
- image/jpeg Bilder
- image/gif Bilder

Nach jedem Request wartet der Web Browser immer auf eine Response. Die Response gilt als vollständig empfangen, wenn entweder die Anzahl der Bytes gemäß **Content-Length** erreicht wurden, oder wenn der Server die Verbindung beendet.

Es gibt noch einen wichtigen Header, der im obigen Beispiel nicht vorkommt:

Connection: close

Wenn der Web-Server diesen Header an den Web-Browser sendet, fordert er damit den Web-Browser auf, die Verbindung nach Übertragung der Response zu schließen. Der Web-Browser muss sich ggf. an diese Anweisung halten.

Das waren die Grundlagen des HTTP Protokolls. Mit diesem Wissen kannst du schon kleine Webserver programmieren. Wir beschränken uns im Rahmen dieses Buches darauf, das Protokoll einmal manuell auszuprobieren.

10.4 HTTP Protokoll manuell testen

Um das HTTP Protokoll manuell zu testen, besorge dir das Programm Netcat (nc). Dieses gibt es sowohl für Linux als auch für Windows.

Das Programm benutzt man an der Eingabeaufforderung (auch als DOS-Fenster oder cmd bekannt). So baust du eine Verbindung zu einem Webserver auf:

```
nc stefanfrings.de 80{Enter}
GET /index.html HTTP/1.1{Enter}
Host: stefanfrings.de{Enter}
{Enter}
```

Mit {Enter} meine ich, dass du die Enter-Taste (bzw. die Return-Taste) drücken sollst. Dann antwortet der Server ungefähr so:

```
HTTP/1.1 200 OK
Date: Sat, 18 Feb 2012 21:17:08 GMT
Server: Apache
…
```

Und so weiter. Du kennst da ja Protokoll ja schon.

Umgekehrt kannst du auch manuell den Server spielen. Wir benutzen jetzt allerdings den Port 8080 weil der Port 80 unter Windows schon für eine Komponente des Betriebssystems reserviert ist und unter Linux sind alle Ports unter 1024 für den root User reserviert.

```
nc -l -p 8080{Enter}
```

Starte nun einen Web-Browser und gebe in die Adressleiste http://localhost:8080/index.html ein. In Netcat siehst du dann den Request des Web-Browsers:

```
GET /index.html HTTP/1.1
Host: localhost:8080
…
```

Zum Schluss kommt eine Leerzeile, dann wartet der Web-Browser offensichtlich auf eine Antwort. Die Antwort kannst du in der Eingabeaufforderung manuell eintippen:

```
HTTP/1.1 200 Ok{Enter}
```

```
Content-Type: text/html{Enter}
{Enter}
<html><body>Hallihallo</html></body>{Enter}
{Strg-C}
```

Mit der Tastenkombination Strg-C beendest du netcat und ebenso die Verbindung. Danach erscheint im Web-Browser „Hallihallo“. So einfach geht das!

11 Programmieren in C (Fortsetzung)

Wenn du ein bisschen Programmiererfahrung gesammelt hast, solltest du dir die Aufsätze auf der Seite http://www.nongnu.org/avr-libc/user-manual/pages.html anschauen. Dort gibt es viele nützliche Hinweise zum Umgang mit Variablen und Interrupt-Prozeduren auf AVR Mikrocontrollern.

11.1 Konsole

Eine Konsole (engl. Terminal) besteht aus Tastatur und Bildschirm, die über eine serielle Leitung mit einem Computer verbunden sind. Man findet Konsolen heutzutage immer noch in Rechenzentren und als Bedienfeld für Maschinen. Klassische Konsolen können nur Text anzeigen, manche in Farbe, aber das ist längst nicht selbstverständlich.

Auch das Internet wurde ursprünglich über Konsolen benutzt, als es noch keine Web Browser gab. Web Browser sind prinzipiell immer noch Konsolen, aber mit der Fähigkeit, Bilder anzuzeigen und Multimediale Inhalte wiederzugeben. Sie zeigen die Ausgabe von Programmen an, die auf anderen Computern irgendwo auf der Welt laufen.

Dieses Prinzip lässt sich auch auf Mikrocontroller übertragen. Du hast schon bemerkt, dass man einen PC Monitor und eine normale PC Tastatur nicht ohne Weiteres an Mikrocontroller anschließen kann. Tatsächlich sind die Aufwände immens – zumindest, was den Bildschirm angeht. Denn du bräuchtest dazu eine Grafikkarte, die wiederum so komplex anzusteuern ist, dass es den Speicher des Mikrocontrollers schnell überfordert.

Aber mit Konsolen geht das ganz einfach, denn Konsolen erfordern nur eine einfache serielle Schnittstelle, und die hat der Mikrocontroller ja schon.

Der entscheidende Nutzen einer Konsole am Mikrocontroller besteht darin, dass du Text-Meldungen ausgeben kannst. Das Programm kann melden, was es gerade tut und warum es sich für die eine oder andere Aktion entschieden hat. Du kannst auch den Wert von wichtigen Variablen ausgeben. Auf diese Weise kannst du die korrekte Funktion viel leichter sicherstellen, als durch einen reinen „Blindflug“ ohne Bildschirmausgaben.

Du musst auf deinem Computer lediglich ein Terminal-Programm installieren und den Mikrocontroller seriell mit dem Computer verbinden, also über RS232 oder USB-UART oder Bluetooth. Ich empfehle dir dazu das Programm „Hammer Terminal“.

11.1.1 Konsole in C

In der Programmiersprache C spricht man Konsolen mit den gleichen Befehlen an, wie Dateien. Zu diesem Zweck sind zwei besondere Dateien reserviert:

- **stdin** für Eingaben (Tastatur)
- **stdout** für Ausgaben (Bildschirm)

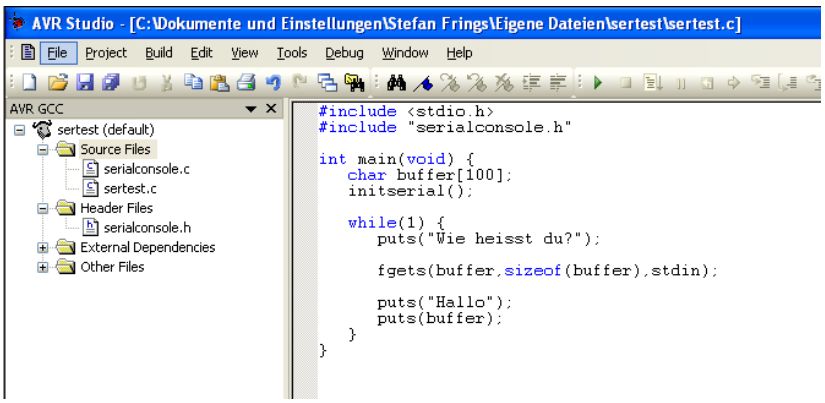
Wenn du also etwas auf dem Bildschirm ausgeben möchtest, dann schreibst du den Text in die Datei stdout. Und wenn du eine Eingabe von der Tastatur einlesen möchtest, dann liest du aus der Datei stdin.

11.1.2 Beispielprogramm

Allerdings sind die beiden Dateien stdin und stdout in der C Library zunächst leere Dummies ohne Funktion. Du musst ein paar Zeilen Code schreiben, um sie nutzen zu können.

Im Anhang des Buches findest du für AVR Mikrocontroller den Quelltext der beiden Dateien serialconsole.h und serialconsole.c. Wenn du diese Dateien in dein Projekt einbindest, bekommst du eine funktionierende Konsole über den seriellen Port des Mikrocontrollers.

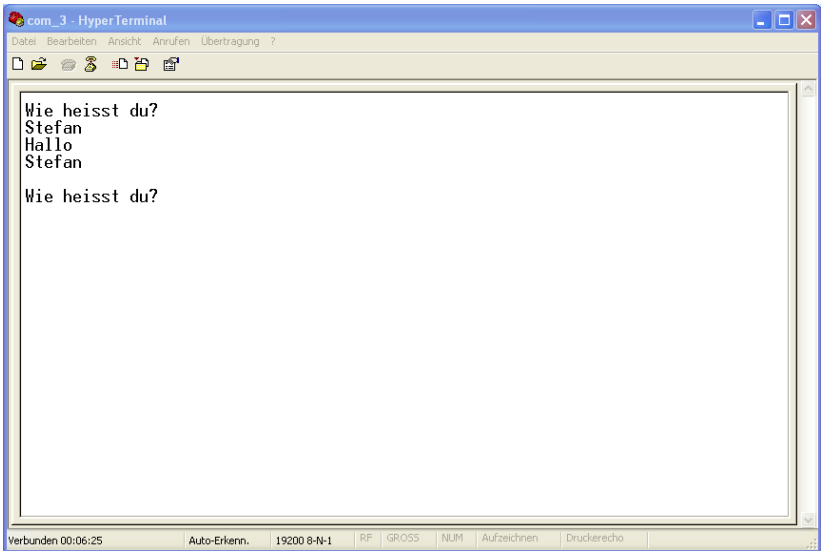
Mit dem folgenden Programm kannst du diese Konsole ausprobieren:



Compiliere das Programm und übertrage es in den Mikrocontroller. Starte dann ein Terminal-Programm. Öffne den seriellen Port, an den dein Mikrocontroller angeschlossen ist. In meinem Fall ist es COM3. Stelle folgende Parameter für die serielle Kommunikation ein:

- Bitrate: 19200 Baud (wie in serialconsole.h)
- Datenbits: 8
- Parität: Keine (N)
- Stop-Bits: 1
- Handshake: kein

Wenn die Verbindung aufgebaut ist, drücke den Reset-Knopf des Mikrocontroller oder die Enter-Taste auf dem PC. Und schon beginnt die Kommunikation.



11.1.3 Konfiguration

In der Datei **serialconsole.h** kannst du drei Einstellungen vornehmen:

- **TERMINAL_MODE 1**
  In der Programmiersprache C benutzt man normalerweise \n um einen Zeilenumbruch darzustellen. Die meisten Terminalprogramme stellen Zeilenumbrüche jedoch nur korrekt dar, wenn stattdessen \r\n gesendet wird. Genau dafür sorgt der Terminal Modus. Außerdem wirkt sich der Terminal-Modus auf den Empfang von Eingaben aus. Denn während die Programmiersprache davon ausgeht, dass die Enter-Taste eine Eingabe mit \n abschließt, senden die meisten Terminal-Programme stattdessen ein \r. Der Terminal-Modus ersetzt beim Empfang alle \r durch \n.
  In deinem Programm wirst du also immer nur ein einfaches \n als Zeilenumbruch

verwenden, so wie es die Programmiersprache vorsieht. Die Umsetzung auf die Eigenarten der Terminalprogramme übernimmt die serialconsole.

- **SERIAL_ECHO 1**
  Benutzer erwarten, dass alle Tastatureingaben während der Eingabe auf dem Bildschirm erscheinen. Genau dafür sorgt diese Option. Sie sendet jedes von der Tastatur empfangene Zeichen gleich wieder zurück an den Bildschirm.
- **SERIAL_BITRATE 19200**
  Stelle hier die gewünschte Übertragungsrate der seriellen Schnittstelle ein. Achte auch darauf, dass die Taktfrequenz in den Projekteinstellungen korrekt ist und dass die Fuse-Bits korrekt eingestellt sind. Denn sonst berechnet der Compiler die Bitrate falsch und es kommt keine vernünftige Übertragung zustande.

11.1.4 Funktionen aus der stdio.h

Die wichtigsten Funktionen in Zusammenhang mit der Konsole sind die folgenden:

Ein Zeichen empfangen:

```
char c=getchar();
```

Ein Zeichen senden:

```
putchar(c);
```

Eine Zeile empfangen:

```
char buffer[100];
fgets(buffer,sizeof(buffer),stdin);
```

Der Puffer enthält am Ende den Zeilenumbruch.

Eine Zeile senden:

```
puts("Hallo");
```

Ein Zeilenumbruch wird automatisch angehängt.

Eine Zeile senden und Variablen einfügen:

```
printf("Es wurden %i Ereignisse gezählt\n", anzahl);
printf("Du hast %s eingegeben\n", buffer);
printf("%i mal %i ergibt %i\n", 3, 4, 12);
```

Schau dir die Beschreibung dieser Funktionen in der Dokumentation der AVR C Library an. Vor allem bei printf (bzw. vfprintf) gibt es dort viel zu lesen.

Wenn du eine Funktion zum Empfangen verwendest, bleibt das Programm so lange stehen, bis es auch tatsächlich etwas empfangen hat. Möglicherweise möchtest du nicht immer warten, sondern lieber testen, ob schon ein Zeichen empfangen wurde und es erst dann auslesen – so dass das Programm nicht wartend hängen bleibt.

Das verhindert man so:

```
if (receivedChar())
{
   char c=getchar();
}
else
{
   … noch nichts empfangen
}
```

Die Funktion receivedChar() liefert ein true, wenn der serielle Port schon ein Zeichen empfangen hat, das noch nicht ausgelesen wurde.

11.2 Zeichenketten

Zeichenketten in der Programmiersprache C sind nichts anderes, als Arrays von Zeichen, die mit einem Null-Byte abgeschlossen werden. Diese Zeichenketten nennt man auch Strings.

```
char test[] = "Stefan";
```

Irgendwo im Speicher liegt eine Array-Variable mit dem Namen „test“. Sie ist sieben Bytes groß und enthält folgende Daten:

test[0]	test[1]	test[2]	test[3]	test[4]	test[5]	test[6]
'S'	't'	'e'	'f'	'a'	'n'	0

Das Array ist 7 Bytes lang, weil die Abschließende Null dazu gehört. Die Länge der Zeichenkette ist aber nur 6 Zeichen. Die Null kennzeichnet das Ende der Zeichenkette.

Wenn du versuchst, eine Zeichenkette ohne die abschließende Null zu verarbeiten, wird das Programm einfach abstürzen, denn es „denkt“, dass die Zeichenkette unendlich lang sei.

Um eine String-Variable zu löschen, schreibst du einfach eine Null hinein:

```
char meldung[]="Hallo";
meldung[0]=0;
```

Jetzt enthält die Variable „meldung“ eine leere Zeichenkette, weil sie gleich mit der Null beginnt. Ganz ähnlich kannst du lange Zeichenkette kürzen:

```
char meldung[]="Blumenkohlsuppe";
meldung[10]=0;
```

Jetzt enthält die Variable „meldung“ nur noch das Wort „Blumenkohl“, denn das 's' wurde durch eine Null ersetzt.

Anders als numerische Variablen kann man Strings nicht einfach so zuweisen:

```
char puffer[100];
puffer=“Hallo“;
```

Aus Sicht des C Compilers bedeutet diese Anweisung: Lege im RAM Speicher eine Zeichenkette mit dem Inhalt „Hallo“ an und dann soll puffer auf dessen Adresse zeigen. Aber puffer ist kein Zeiger, sondern ein Array, deswegen geht das so nicht. Um ein Array mit Zeichen zu befüllen, musst du die Zeichen kopieren:

```
char puffer[100];
strcpy(puffer,“Hallo“);
```

Oder du verwendest stattdessen einen Zeiger:

```
char* zeiger;
zeiger=“Hallo“;
```

Abgesehen davon erlaubt die Programmiersprache C weitgehend den gemischten Einsatz von Zeigern (Pointern) und Arrays in Ausdrücken und Parametern von Funktionen.

char meldung[]=“Hallo“;

meldung	liefert die Adresse des ersten Buchstabens im Speicher.
meldung[0]	liefert den ersten Buchstaben der Meldung ('H').
*meldung	liefern den Buchstaben, auf den der Zeiger zeigt, also ebenfalls den ersten Buchstaben.
meldung[1]	liefert den zweiten Buchstaben ('a').
*(meldung+1)	liefert den Buchstaben, der ein Byte hinter der Speicherzelle liegt, auf die die Variable meldung zeigt. Das ist der zweite Buchstabe.
meldung[5]	liefert den letzten Buchstaben ('o').
*(meldung+5)	liefert ebenfalls das ('o').
meldung[6]	liefert das abschließende Zeichen mit dem Wert 0.
*(meldung+6)	liefert ebenfalls das abschließende Zeichen mit dem Wert 0.

Das folgende Programm soll demonstrieren, wie man Arrays und Zeiger vermischt einsetzen kann.

```
#include <stdio.h>
#include „seialconsole.h“

function eins(char* meldung)
{
    puts(meldung);

    if (*meldung == 'D')
    {
        puts("Der erste Buchstabe ist ein D");
    }
```

```
    meldung+=1;
    if (*meldung == 'i')
    {
        puts("Der zweite Buchstabe ist ein i");
    }
}

function zwei(char meldung[])
{
    puts(meldung);

    if (meldung[0] == 'I')
    {
        puts("Der erste Buchstabe ist ein I");
    }

    if (meldung[1] == 'm')
    {
        puts("Der zweite Buchstabe ist ein m");
    }
}

function drei(char* meldung)
{
    puts(meldung);

    if (meldung[0] == 'R')
    {
        puts("Der erste Buchstabe ist ein R");
    }
}

function vier(char meldung[])
{
    puts(meldung);

    if (*(meldung+1) == 'u')
    {
        puts("Der zweite Buchstabe ist ein u");
    }
   }

int main(void)
{
    initserial();
    eins("Die Welt ist schön");
    zwei("Im Winter ist es kalt");
    drei("Rumpelstilzchen");
}
```

Die funktion eins() demonstriert, wie man Zeichenketten mit einem Zeiger übergibt, nämlich einen Zeiger auf den ersten Buchstaben der Zeichenkette.

Die Funktion zwei() demonstriert, wie man Zeichenketten als Array übergibt, nämlich ein Array aus Zeichen (char).

Die Funktion drei demonstriert, dass man Zeiger auf Zeichenketten sogar wie Arrays benutzen kann. Umgekehrt geht es ebenfalls, wie die Funktion vier() zeigt.

Offensichtlich ist eine Variable vom Typ char-Array identisch, mit einer Variable vom Typ char-Pointer. Bei der Übergabe der Meldung vom Hauptprogramm an die Funktionen passiert folgendes:

Meldung enthält die Adresse der Zeichenkette im Speicher, ganz egal, ob der Parameter als „char* meldung“ oder „char meldung[]“ deklariert wurde. In beiden Fällen bekommt die Funktion nur die Adresse der Zeichenfolge übergeben, nicht die Menge von Buchstaben. Und genau deswegen funktionieren drei() und vier() auch, auch wenn diese beiden Funktionen auf den ersten Blick fehlerhaft aus sehen.

11.2.1 Standardfunktionen für Strings

Die Programmiersprache C stellt einige Funktionen zur Verarbeitung von Zeichenketten bereit, und zwar in den Dateien stdio.h, stdlib.h und string.h.

Die am häufigsten verwendeten möchte ich hier kurz nennen:

- **sizeof(variable)**
  gibt die Größer der Variable in Bytes zurück. Bei Strings ist das Ergebnis nicht die Länger der Zeichenkette, sondern die größe des Arrays. Ein großes Array mit 400 Bytes kann auch viel kürzere Zeichenketten speichern.

  ```
  char puffer[400];
  strcpy(puffer,“Hallo“);   // kopiert „Hallo“ in den Puffer
  i=sizeof(puffer);         // ergibt 400
  i=strlen(puffer);         // ergibt 5
  ```

- **strlen("Hallo")**
  gibt die Länge der Zeichenkette zurück, ohne das abschließende Null-Byte mit zu zählen.
- **puts("Hallo")**
  gibt eine Meldung auf der Konsole aus, und hängt einen Zeilenumbruch an.
- ~~**gets(puffer)**~~
  liest eine Zeile von der Konsole ein und speichert sie in der Variable „puffer“, welche ein ausreichend großes Array von char sein muss. Der Puffer enthält keinen Abschließenden Zeilenumbruch.
  Das Problem ist: der Benutzer der Konsole kann eine beliebig lange Zeile eintippen. Wenn die Variable zu klein ist, werden andere Daten im RAM überschrieben, so das das Programm abstürzen wird. Benutze daher besser die nächste Funktion:
- **fgets(puffer, sizeof(puffer), stdin)**
  liest eine Zeile aus einer Datei ein und speichert sie in der Variable „puffer“. Der Puffer enthält den abschließenden Zeilenumbruch.
  Der zweite Parameter gibt an, wieviel Speicherplatz maximal zur Verfügung steht. Der dritte Parameter ist die Datei aus der gelesen wird, wobei stdin die Konsole ist.
- **printf("Die Variable k hat den Wert %i\n", k)**
  gibt eine Zeichenkette aus und fügt den Wert einer Variablen ein. In diesem Fall bedeutet %i, dass der Wert als Integer-Zahl ausgegeben werden soll. Mehr dazu weiter unten.
- **int i=atoi("124")**
  Wandelt die Zeichenkette in eine Integer-Zahl um.

- **float f=atof("14.99")**
  Wandelt die Zeichenkette in eine Fließkommazahl um. Achtung: Diese Funktion belegt viel Programm-Speicher, wie alle Fließkomma-Operationen.
- **strcpy(puffer, "Hallo")**
  kopiert den String „Hallo“ in den Puffer. Der Puffer muss ein ausreichend großes Array von chars sein. Der zweite Parameter kann auch eine String (char-Array) Variable sein.
- **strncpy(a , b, sizeof(a))**
  kopiert den String b in den Speicher von a, aber maximal so viele Zeichen, wie der dritte Parameter angibt. Auf diese Weise stellst du sicher, das der Speicherplatz von a nicht überschritten wird.
  Aber: Wenn die Zeichenkette b zu lang ist, fehlt am Ende in a das abschließende Byte mit dem Wert null! So macht man es richtig:

  char a[6];
  char b[]=“Fruchtgummis“;
  strcpy(a, b, sizeof(a));
  a[sizeof(a)-1]=0;

  a enthält nun die Zeichenkette „Fruch“ gefolgt vom abschließenden Null-Byte.
- **strcat(puffer, "Hallo")**
  hängt den String „Hallo“ an den bereits vorhandenen String im Puffer an. A muss ein ausreichend großes Array von chars sein. Der zweite Parameter kann auch eine String (char-Array) Variable sein.

11.2.2 Formatierung mit printf

Mit printf gibst du eine Zeichenkette auf der Konsole aus, und fügst den Wert einer oder mehrerer Variablen ein:

```
int k=7;
printf("Die Variable k hat den Wert %i", k);
```

%i gibt eine Integer-Zahl aus.

%u gibt eine Integer-Zahl ohne Vorzeichen aus.

%o gibt eine Zahl in oktaler Schreibweise aus.

%x gibt eine Zahl in hexadezimaler Schreibweise mit kleinen Buchstaben aus (z.B. a5).

%X gibt eine Zahl hexadezimaler Schreibweise mit großen Buchstaben aus (z.B. A5).

%p gibt den Wert eines Zeigers (also eine Adresse) in hexadezimaler Schreibweise aus.

Bei allen obigen Ausgabeformaten kannst du angeben, wie breit die lang mindestens sein soll: %6i gibt die Zahl 6-Stellig rechtsbündig aus. Wenn die Zahl kleiner ist, wird mit Leerzeichen aufgefüllt. %6i ergibt beim Wert 123 die Ausgabe „ 123“.

Linksbündige Ausgabe erreichst du, indem du vor die Längenangabe ein Minus Zeichen schreibst: %-6i ergibt beim Wert 123 die Ausgabe „123 „.

Wenn du eine Null vor die Längenangabe schreibst, wir mit Nullen anstatt mit Leerzeichen aufgefüllt: %06i ergibt beim Wert 123 die Ausgabe „000000123“.

Wenn du ein „+“ Zeichen in die Formatier-Anweisung schreibst, werden positive Zahlen mit einem + Zeichen ausgegeben: %+3i ergibt beim Wert 123 die Ausgabe „+123“.

Du kannst für positive Zahlen auch ein Leerzeichen an die Stelle ausgeben lassen, wo bei negativen Zahlen das minus wäre: % 3i ergibt beim Wert 123 die Ausgabe „ 123“ und bei -123 ergibt es die Ausgabe „-123“.

Ein zusätzliches l sagt aus, dass es sich um eine Long-Integer Variable (also 32 Bit) handelt, Zum Beispiel %li.

%c gibt ein Zeichen aus.

%s gibt eine Zeichenkette aus dem RAM aus.

%S gibt eine Zeichenkette aus dem Programmspeicher aus.

Bei Zeichenketten kannst du auch die Breite der Ausgabe festlegen, wie bei Zahlen.

Fließkommazahlen werden grundsätzlich in englischer Form (also mit Punkt statt Komma) ausgegeben und ohne Tausender-Trennzeichen.

%e gibt eine Fließkommazahl im englischen Format 1.9923e03 aus.

%f gibt eine Fließkommazahl in der Form 1992.3 aus.

%6g gibt eine Fließkommazahl wie ein Wissenschaftlicher Taschenrechner aus. Wenn die Zahl größer ist, als die angegebene Länge, wird auf die Form 1.9923e3 gewechselt. Ansonsten wird sie ganz normal mit Dezimalpunkt ausgegeben.

In Ergänzung zu den Integer Zahlen kannst du bei Fließkommazahlen zusätzlich angeben, wie viele Nachkommastellen du sehen willst:

%5.3f gibt eine Zahl mit 5 Stellen vor und 3 Stellen hinter dem Dezimalpunkt aus. Die Stellen vor dem Punkt werden ggf mit Leerzeichen aufgefüllt, die Stellen dahinter werden ggf. gerundet.

11.2.3 Strings im Programmspeicher

Die Programmiersprache C wurde ursprünglich für Computer entwickelt, bei denen sowohl das Programm als auch die Daten im RAM Speicher liegen. Alle Standard-Befehle für Zeichenketten gehen davon aus.

Bei Mikrocontrollern liegen Zeichenketten jedoch meistens im Programmspeicher und da sie nicht verändert werden, macht es auch keinen Sinn, sie zuerst in den den RAM zu kopieren, um anschließend die Standard-Funktionen für Zeichenketten zu benutzen.

Ich möchte dies am Beispiel von puts() vorführen. Normalerweise gibt man in C auf die folgende Weise Text auf die Konsole aus:

```
char message[]="Hallo";
puts(message);
```

oder einfach:

```
puts("Hallo");
```

Auf Personal Computern ist das die normale Vorgehensweise. Aber bei Mikrocontrollern führt diese Schreibweise dazu, dass die ganze Zeichenkette zunächst vom Programmspeicher in den Datenspeicher (RAM) kopiert wird und erst dann ausgegeben wird. So vergeudet man unnötig Speicher .

Darum enthält die C Library für AVR Mikrocontroller von fast allen String-Funktionen eine zweite Variante, die die Zeichenkette direkt aus dem Programmspeicher anstatt aus dem RAM liest. Diese Funktionen erkennt man am Suffix (Anhängsel) „_P“.

Außerdem muss man die Zeichenkette anders deklarieren, um dem Compiler anzuzeigen, dass diese Zeichenkette eben nicht ins RAM kopiert werden soll. Das macht man so:

```
char message[] PROGMEM = "Hallo";
puts_P(message);
```

oder so:

```
puts_P(PSTR("Hallo"));
```

Die beiden Schlüsselwörter PROGMEM und PSTR signalisieren dem Compiler, dass die angegebenen Zeichenketten im Programmspeicher verbleiben sollen, und dass sie nicht ins RAM kopiert werden sollen.

Natürlich ergibt sich daraus auch die Konsequenz, dass diese Zeichenketten unveränderlich sind. Die Variable „message“ ist also eigentlich gar keine Variable mehr, sondern eine Konstante.

Weitere Information dazu findest du übrigens in der Dokumentation der AVR libcLibrary auf dieser Seite: http://www.nongnu.org/avr-libc/user-manual/pgmspace.html

11.3 Multithreading

Manchmal wirst du Programme schreiben müssen, die mehrere Threads (Aufgaben) parallel abarbeiten.

Da AVR Mikrocontroller nur einen CPU Kern haben, können sie nicht mehre Threads gleichzeitig abarbeiten. Aber sie können sich immer abwechselnd ein bisschen um jeden Thread kümmern, so dass es von außen so aussieht, als ob alle Threads gleichzeitig ablaufen würden. Vergleiche dies mit einem Menschen, der beim Kochen gleichzeitig die Küche sauber hält. Er widmet sich diesen beiden Aufgaben immer abwechselnd.

11.3.1 Endlicher Automat

Eine gängiger Lösungsansatz dazu, ist das Modell des endlichen Automaten, auch Zustandsautomat oder State-Machine genannt. Ein konkretes Beispiel:

Thread 1 soll eine Leuchtdiode an Port B0 blinken lassen.

Thread 2 soll eine Taste an Port B1 abfragen bei Tastendruck eine zweiten Leuchtdiode an Port B2 umschalten.

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

void thread1(void)
{
    static uint8_t status=0;
    static uint16_t zaehler=0;

    if (++zaehler == 20000)
    {
        zaehler=0;

        switch (status)
        {

            case 0: // LED ist aus -> ein schalten
                PORTB |= 1;
                status=1;
                break;
```

```
            case 1: // LED is an -> aus schalten
                PORTB &= ~1;
                status=0;
                break;
        }
    }
}


void thread2(void)
{
    static uint8_t status=0;

    // Taste abfragen
    int gedrueckt= (PINB & 2) == 0;

    switch (status)
    {

        case 0: // LED ist aus, warte auf Tastendruck
            if (gedrueckt)
            {
                PORTB |= 4; // LED einschalten
                status=1;
            }
            break;

        case 1: // LED ist an, warte auf Loslassen
            if (!gedrueckt)
            {
                status=2;
            }
            break;

        case 2: // LED ist an, warte auf Tastendruck
            if (gedrueckt)
            {
                PORTB &= ~4; // LED aus schalten
                status=3;
            }
            break;

        case 3: // LED ist an, warte auf Loslassen
            if (!gedrueckt)
            {
                status=0; // zurück zum Anfang
            }
            break;
    }

}

int main(void)
{

    // PB0 und PB2 sind Ausgänge (für die LED)
    DDRB = 1+4;

    // Pull-Up Widerstand an Port B1 einschalten
    PORTB = 2;

    // Hauptschleife
```

```
    while (1)
    {
       thread1();
       thread2();
    }
}
```

Beide Threads sind als Endlicher Automat geschrieben.

Der thread1 zählt mit, wie oft er aufgerufen wird. Immer wenn er die 20.000 erreicht hat, schaltet er den Zustand der LED um und fängt wieder bei 0 an zu zählen.

Der thread2 schaltet eine LED bei Tastendruck immer abwechselnd an und aus. Kleine Anmerkung dazu: Hier wird davon ausgegangen, dass der Taster nicht prellt. Falls er es doch tut, kann man einen kleinen Kondensator (47 nF oder 100 nF) parallel schalten (es gibt bessere Methoden, aber diese hier zu erklären würde vom Thema ablenken).

Beide Threads nutzen statische Variablen, um den Status zu speichern. Der Unterschied zu normalen Variablen besteht darin, dass statische Variablen ihrem gespeicherten Wert zwischen mehreren Funktionsaufrufen nicht verlieren.

Die quasi-parallele Ausführung kommt zustande, indem die beiden Threads immer wieder abwechselnd aufgerufen werden. Nach dem gleichen Prinzip kannst du beliebige viele Threads quasi-parallel ausführen.

Achte darauf, dass die einzelnen Treads niemals stehen bleiben. Endlosschleifen sind innerhalb der Threads völlig tabu, ebenso Warteschleifen jeglicher Art, wie die _delay_ms() Funktion. Denn wenn ein Thread anhält, betrifft das auch alle anderen Threads. Sie würden sich gegenseitig behindern.

Wenn ein Programm zwischen den Statuswechseln eine gewisse Zeit abwarten muss, dann benutze einen Interrupt-gesteuerten Timer, der in regelmäßigen Intervallen eine Variable hoch zählt:

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

// Für ATtiny13
// Taktfrequenz = 1200000 Hz

volatile utint16_t systemTimer;

// Wird alle 1ms aufgerufen
ISR(TIM0_COMPA_vect)
{
    systemTimer++;
}

void start_timer(void)
{
    cli();
    TCCR0A = 2;  // CTC Modus
    TCCR0B = 3;  // Taktdurch 64 teilen
    OCR0A  = 19; // Beim jedem 19. Takt Interrupt auslösen
    TIMSK0 = 4;  // Interrupts erlauben
    sei();
}

void thread3(void)
{
    static utint16_t lastTime=0;
    static uint8_t status=0;

    switch (status)
```

```
    {

        case 0: // LED ist aus
            if (systemTimer-lastTime >= 300)
            {
                // ein schalten
                PORTB |= 1;
                lastTime=systemTimer;
                status=1;
            }
            break;

        case 1: // LED is an
            if (systemTimer-lastTime >= 300)
            {
                // aus schalten
                PORTB &= ~1;
                lastTime=systemTimer;
                status=0;
            }
            break;
    }
}


int main(void)
{

    // PB0 und PB2 sind Ausgänge (für die LED)
    DDRB = 1+4;

    // Pull-Up Widerstand an Port B1 einschalten
    PORTB = 2;

    // Timer starten
    start_timer();

    // Hauptschleife
    while (1)
    {
       thread1();
       thread2();
       thread3();
    }
}
```

Hierbei ist es wichtig, die verstrichene Zeit mit einer Subtraktion zu berechnen, weil das auch bei einem zwischenzeitlichen Überlauf der systemTimer Variable korrekt funktioniert.

Nachtrag: Ich habe dieses Thema inzwischen auf meiner Homepage ausführlicher beleuchtet: http://stefanfrings.de/multithreading_arduino/

12 Anhänge

Die folgenden Quelltexte passen zu fast allen klassischen AVR Mikrocontroller Modellen.

12.1 Quelltext Serielle Konsole

12.1.1 serialconsole.h

```
#ifndef __SERIALCONSOLE_H_
#define __SERIALCONSOLE_H_

#include <stdint.h>


// Serial bit rate
#define SERIAL_BITRATE 115200


// The  CPU clock frequency is usually set in AVR Studio
// #define F_CPU 20000000


// In the software, lines are always terminated with \n.
// If terminal mode is enabled, line breaks are converted
// automatically:
#define TERMINAL_MODE 1


// Enable echo on serial interface
#define SERIAL_ECHO 1


// Size of input buffer in chars (output is unbuffered)
#define SERIAL_INPUT_BUFFER_SIZE 80


// In case of ATmega targets, the following definition may be
// set to use the second serial port instead of the first one:
// #define USE_SERIAL2


// Initialize the serial port and bind it to stdin and stdout.
// Don't forget to set the TxD pin to output mode in case of ATxmega.
void initserial(void);


// Returns 1 if the serial port has received at least one character
// so that following calls to getc() do not block.
uint8_t receivedChar(void);


// Returns 1 if the serial port has received a line break
// so that following calls to gets() and scanf() do not block.
uint8_t receivedLine(void);


#endif //__SERIALCONSOLE_H_
```

12.1.2 serialconsole.c

```
#include "serialconsole.h"
#include <stdio.h>
#include <avr/io.h>
#include <stdint.h>
#include <avr/interrupt.h>


// Files for input and output through serial port.
static int serial_write(char, FILE *);
static int serial_read(FILE *);
static FILE serialPort = FDEV_SETUP_STREAM(serial_write, serial_read,
_FDEV_SETUP_RW);


// input ring buffer
static volatile char inputBuffer[SERIAL_INPUT_BUFFER_SIZE];


// index of first character in the buffer
static volatile uint8_t inputStart=0;


// point to the next free position in the input
// buffer (=inputStart in case of empty buffer)
static volatile uint8_t inputEnd=0;

// The previously received character
static volatile char prevChar=0;


// Check if the serial port has received at least one character
uint8_t receivedChar(void)
{
    return inputStart!=inputEnd;
}


// Check if the serial port has received a line break
uint8_t receivedLine(void)
{
    int i=inputStart;
    while (i!=inputEnd)
    {
        if (inputBuffer[i]=='\n')
        {
            return 1;
        }
        if (++i>=SERIAL_INPUT_BUFFER_SIZE)
        {
            i=0;
        }
    }
    return 0;
}


// Define aliases for ATmega controllers with multiple serial ports
// so they can be used with the same source code as controllers with
// only a single serial port.
#ifdef TXEN0            // Atmega with multiple serial ports
```

```
    #ifdef USE_SERIAL2  // configured to use the second serial port
        #define UCSRA UCSR1A
        #define UCSRB UCSR1B
        #define UCSRC UCSR1C
        #define UBRRH UBRR1H
        #define UBRRL UBRR1L
        #define UDRE  UDRE1
        #define UDR   UDR1
        #define RXC   RXC1
        #define RXEN  RXEN1
        #define TXEN  TXEN1
        #define UCSZ0 UCSZ10
        #define UCSZ1 UCSZ11
        #define U2X   U2X1
        #define RXCIE RXCIE1
        #ifdef USART_RX_vect
            #define USART_RXC_vect USART_RX_vect
        #endif
        #ifdef USART0_RX_vect
            #define USART_RXC_vect USART1_RX_vect
        #endif
    #else // Configured to use the first serial port
        #define UCSRA UCSR0A
        #define UCSRB UCSR0B
        #define UCSRC UCSR0C
        #define UBRRH UBRR0H
        #define UBRRL UBRR0L
        #define UDRE  UDRE0
        #define UDR   UDR0
        #define RXC   RXC0
        #define RXEN  RXEN0
        #define TXEN  TXEN0
        #define UCSZ0 UCSZ00
        #define UCSZ1 UCSZ01
        #define U2X   U2X0
        #define RXCIE RXCIE0
        #ifdef USART_RX_vect
            #define USART_RXC_vect USART_RX_vect
        #endif
        #ifdef USART0_RX_vect
            #define USART_RXC_vect USART0_RX_vect
        #endif
    #endif
#endif



    #define BAUD SERIAL_BITRATE
    #include <util/setbaud.h>


    // Write a character to the serial port
    static int serial_write(char c, FILE *f)
    {
        #if TERMINAL_MODE
            if (c=='\n')
            {
                // wait until transmitter is ready
                loop_until_bit_is_set(UCSRA, UDRE);
                UDR='\r';
            }
        #endif
```

```
    // wait until transmitter is ready
    loop_until_bit_is_set(UCSRA, UDRE);
    UDR = c;
    return 0;
}


// Read a character from serial port
static int serial_read(FILE *f)
{
    // wait until something is received
    while (inputStart==inputEnd);
    char c=inputBuffer[inputStart];
    if (++inputStart>=SERIAL_INPUT_BUFFER_SIZE)
    {
        inputStart=0;
    }
    return c;
}


// Initialize the serial port
void initserial(void)
{
    // set baudrate
    UBRRH = UBRRH_VALUE;
    UBRRL = UBRRL_VALUE;
    #if USE_2X
        UCSRA |= (1 << U2X);
    #else
        UCSRA &= ~(1 << U2X);
    #endif
    // framing format 8N1
    #ifdef URSEL
        UCSRC = (1<<URSEL) | (1<<UCSZ1) | (1<<UCSZ0);
    #else
        UCSRC = (1<<UCSZ1) | (1<<UCSZ0);
    #endif
    // enable receiver and transmitter
    UCSRB = (1<<RXEN) | (1<<TXEN) | (1 << RXCIE);
    sei();
    // Bind stdout stdin to the serial port
    stdout = &serialPort;
    stdin = &serialPort;
    inputStart=0;
    inputEnd=0;
}


// USART Receive interrupt
ISR ( USART_RXC_vect )
{
    char c=UDR;
    #if TERMINAL_MODE
        // If received a backspace character, then discard
        // the last received character
        if (c=='\b')
        {
            if (inputStart!=inputEnd)
            {
                if (--inputEnd<0)
                {
```

```
                    inputEnd=0;
                }
            }
            #if SERIAL_ECHO
                serial_write(c,0);
            #endif
            return;
        }
        // If received a line feed and the previous
        // character was a carriage-return
        // (or vice versa) then ignore the new character
        if ((c=='\n' && prevChar=='\r') || (c=='\r' && prevChar=='\n'))
        {
            return;
        }
        prevChar=c;
        // If received a carriage-return, then replace it by a
        // line-feed and send a line-feed as echo.
        if (c=='\r')
        {
            c='\n';
        }
    #endif
    // Buffer the received character
    inputBuffer[inputEnd]=c;
    if (++inputEnd>=SERIAL_INPUT_BUFFER_SIZE)
    {
        inputEnd=0;
    }
    // Discard oldest buffered character in case of buffer overflow
    if (inputEnd==inputStart)
    {
        if (++inputStart>=SERIAL_INPUT_BUFFER_SIZE)
        {
            inputStart=0;
        }
    }

    #if SERIAL_ECHO
        serial_write(c,0);
    #endif
}
```

12.2 I²C Funktion

```
// Sendet beliebig viele Bytes an den adressierten Slave und empfängt
// anschließend beliebig viele Bytes von dem Slave.

// slave_address ist die Adresse des Slave in 7bit Schreibweise (0-127).
// send_data zeigt auf die zu sendenden Daten, z.B. ein Array von Bytes.
// send_bytes gibt an, wieviele Bytes gesendet werden sollen.
// rcv_data zeigt auf einen Puffer, wo die empfangenen Daten abgelegt werden
// sollen, z.B. ein Array von Bytes.
// rcv_Bytes gibt an, wieviele Bytes empfangen werden sollen.
// Der Rückgabewert zeigt an, wie viele Bytes tatsächlich empfangen wurden.

// Wenn man nur senden will, gibt man rcv_data=0 und rcv_bytes=0 an.
// Wenn man nur empfangen will, gibt man send_data=0 und send_bytes=0 an.
// Es ist erlaubt, bei send_bytes und rcv_bytes Zeiger auf den selben Puffer
// zu übergeben.

uint8_t i2c_communicate(uint8_t slave_address, void* send_data,
    uint8_t send_bytes, void* rcv_data, uint8_t rcv_bytes)
{
    uint8_t rcv_count=0;

    // Adresse ein Bit nach links verschieben,
    // um Platz für das r/w Flag zu schaffen
    slave_address=slave_address<<1;

    if (send_bytes>0)
    {
        // Sende Start
        TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTA);
        while (!(TWCR & (1<<TWINT)));
        uint8_t status=TWSR & 0xf8;
        if (status != 0x08 && status != 0x10) goto error;

        // Sende Adresse (write mode)
        TWDR=slave_address;
        TWCR=(1<<TWINT) | (1<<TWEN);
        while (!(TWCR & (1<<TWINT)));
        if ((TWSR & 0xf8) != 0x18) goto error;

        // Sende Daten
        while (send_bytes>0)
        {
            TWDR=*((uint8_t*)send_data);
            TWCR=(1<<TWINT) | (1<<TWEN);
            while (!(TWCR & (1<<TWINT)));
            if ((TWSR & 0xf8) != 0x28) goto error;
            send_data++;
            send_bytes--;
        }
    }


    if (rcv_bytes>0) {
        // Sende START
        TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTA);
        while (!(TWCR & (1<<TWINT)));
        uint8_t status=TWSR & 0xf8;
        if (status != 0x08 && status != 0x10) goto error;

        // Sende Adresse (read mode)
```

```
        TWDR=slave_address + 1;
        TWCR=(1<<TWINT) | (1<<TWEN);
        while (!(TWCR & (1<<TWINT)));
        if ((TWSR & 0xf8) != 0x40) goto error;

        // Empfange Daten
        while (rcv_bytes>0)
        {
            if (rcv_bytes==1)
            {
                // das letzte Byte nicht mit ACK quittieren
                TWCR=(1<<TWINT) | (1<<TWEN);
            }
            else
            {
                // alle anderen Bytes mit ACK quittieren
                TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWEA);
            }
            while (!(TWCR & (1<<TWINT)));
            uint8_t status=TWSR & 0xf8;
            if (status!=0x50 && status != 0x58) goto error;
            *((uint8_t*)rcv_data)=TWDR;
            rcv_data++;
            rcv_bytes--;
            rcv_count++;
        }

    }

    // Sende STOP
    TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
    return rcv_count;

    error:
    // Sende STOP
    TWCR=(1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
    return 0;
}
```

12.2.1 Anwendungsbeispiele der I²C Funktion

```
// Bus-Taktfrequenz einstellen (=100kHz bei 8Mhz Quarz)
TWBR=75;


// Sende 0xFF an einen PCF8574 mit Array als Puffer
uint8_t buffer[1];
buffer[0]=0xFF;
i2c_communicate(0x20,buffer,1,0,0);


// Sende 0xFF an einen PCF8574 mit Byte als Puffer
uint8_t value=0xFF
i2c_communicate(0x20,&value,1,0,0);


// Empfange ein Byte vom PCF8574 mit Array als Puffer
uint8_t buffer[1];
i2c_communicate(0x20,0,0,buffer,1);


// Empfange ein Byte vom PCF8574 mit Byte als Puffer
uint8_t value;
i2c_communicate(0x20,0,0,&value,1);


// Sende drei Bytes an einen Chip mit Registern
uint8_t buffer[4];
buffer[0]=0;  // Nummer des ersten Registers
buffer[1]=10; // Erstes Byte → Register 0
buffer[2]=20; // Zweites Byte → Register 1
buffer[3]=30; // Drittes Byte → Register 2
i2c_communicate(0x70,buffer,4,0,0);


// Empfange bis zu 16 Bytes von einem Chip mit Registern
uint8_t buffer[16];
buffer[0]=3;        // Nummer des ersten Registers
uint8_t count=i2c_communicate(0x70,buffer,1,buffer,16);


// Empfange drei 16-Bit Werte von einem Chip mit Registern
uint8_t register=2; // Nummer des ersten Registers
uint16_t buffer[3]; // 3 16-Bit Werte, entsprechend 6 Bytes
uint8_t count=i2c_communicate(0x70,&register,1,buffer,6);
```