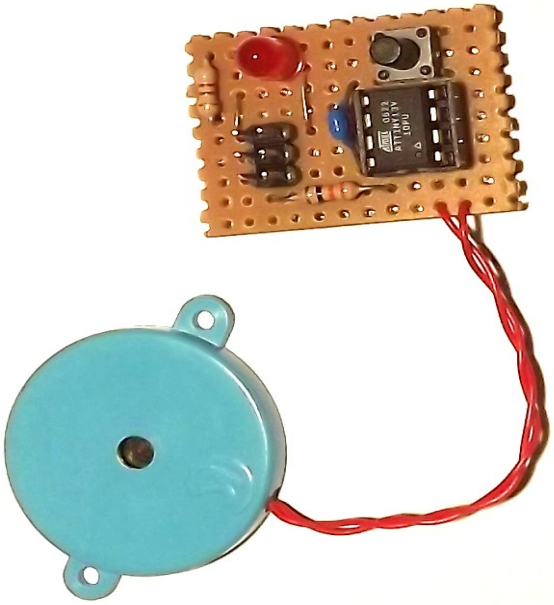
Einstieg in die Elektronik
mit Mikrocontrollern

Band 1
von Stefan Frings



Inhaltsverzeichnis

1 Einleitung

Mikrocontroller sind winzig kleine Computer mit faszinierenden Möglichkeiten. Mit Sicherheit befinden sich mehrere Mikrocontroller in deinem Haushalt, denn sie steuern Waschmaschinen, CD-Player, Fernseher und Mikrowellen-Öfen. Auch Taschenrechner, Armbanduhren und Handys enthalten Mikrocontroller.

In diesem Buch bringe ich dir auf unkonventionelle Weise den Umgang mit Mikrocontrollern bei. Du wirst gleich von Anfang an mit diesen Mikrochips experimentieren. Die nötigen Grundlagen wirst du dabei nach und nach aus diesem Buch lernen. Schon bald wirst du eigene kleine Computer bauen und ihre Funktionen ganz individuell nach deinen persönlichen Vorgaben programmieren.

Bevor du loslegst, besorge dir die Werkzeuge und Bauteile, die ich auf der letzten Seite des Buches aufgelistet habe. Du wirst sie für die Experimente benötigen. Arbeite das Buch der Reihe nach durch und führe alle Experimente aus, denn die Kapitel bauen aufeinander auf.

Die Downloads zum Buch findest du auf der Seite
http://stefanfrings.de/mikrocontroller_buch/index.html.

Die Schaltpläne in diesem Buch habe ich teilweise mit dem Programm KiCad erstellt.

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

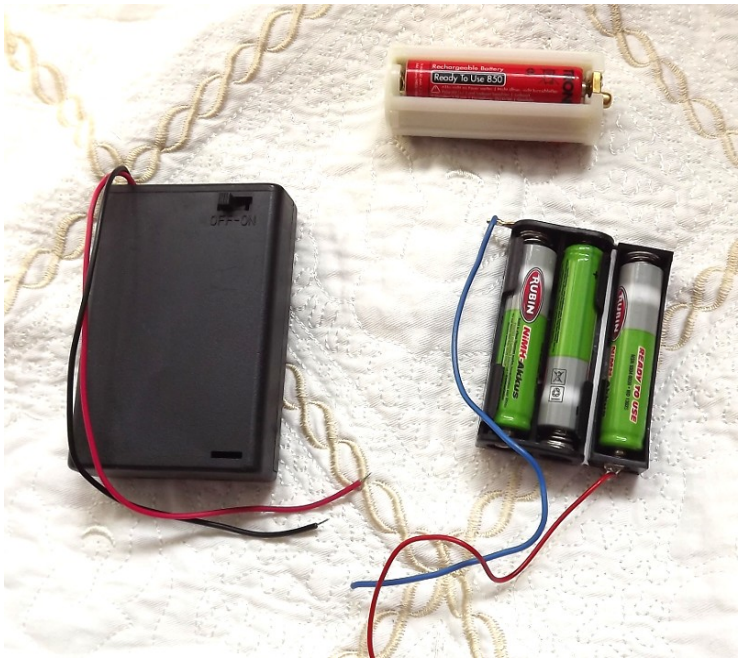
Stefan Frings, im Jahr 2014

2 Grundlagen

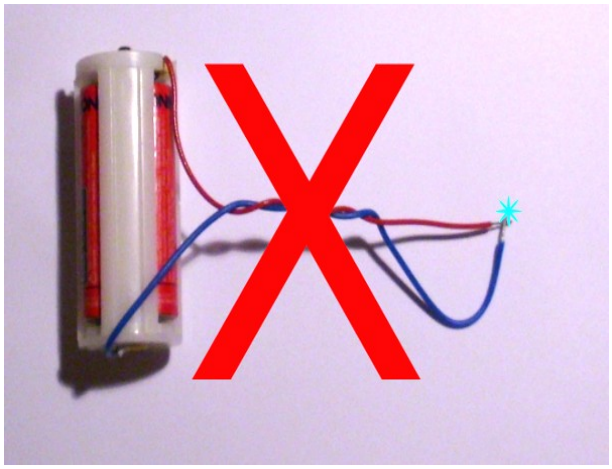
Dieses Kapitel vermittelt grundlegende Kenntnisse zu Bauteilen, die zum Umgang mit Mikrocontrollern notwendig sind. Du würdest die nächsten Kapitel ohne diese Grundkenntnisse nicht verstehen. Schon im nächsten Kapitel wirst du den ersten Mikrocomputer selbst aus Einzelteilen zusammenbauen.

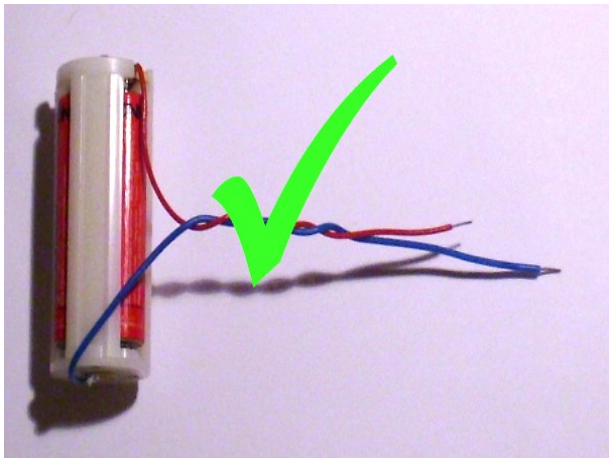
2.1 Stromversorgung

In diesem Buch benutzen wir drei kleine Akkus oder Einwegbatterien in Größe AA oder AAA in einem Batteriehalter. Zusammen liefern sie je nach Ladezustand 2,7 bis 4,8 Volt.



Von diesen kleinen Batterien geht keine Gefahr aus. Achte jedoch immer darauf, keinen Kurzschluss herbei zu führen, weil die Batterien dabei überhitzen, was zu einem Brand führen kann. Ein Kurzschluss entsteht, wenn man die beiden Anschlüsse der Batterien direkt miteinander verbindet.





Um Kurzschlüsse zu verhindern, kürze einen der beiden Anschlussdrähte von deinem Batteriehalter.

2.2 Was ist Elektrizität?

Das Strom Licht machen kann, Hitze erzeugen kann und auch mal schmerzhaft ist – das wissen wir alle. Strom bringt auch Dinge in Bewegung. Strom lässt uns sogar über eine unsichtbare Verbindung miteinander telefonieren.

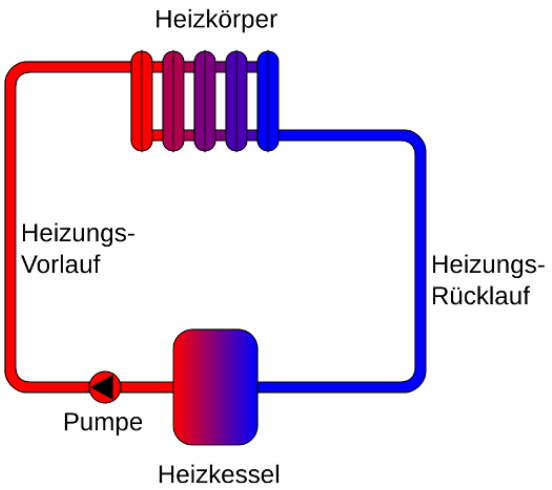
Der elektrische Strom besteht aus bewegten Elektronen. Elektronen sind so klein, dass man sie nicht sehen kann. Riechen kann man sie auch nicht, aber fühlen. Hast du schon einmal elektrischen Strom gefühlt? Wenn nicht, dann halte dir einmal die Anschlüsse von deinem Batteriehalter an die Zunge. Keine Angst, es tut nicht weh.

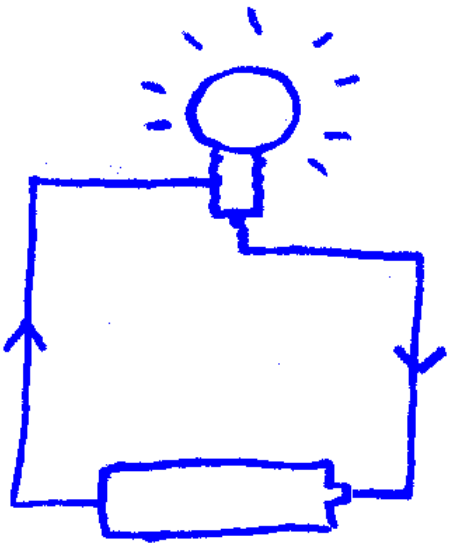
Ein derart kleiner Strom fühlt sich kribbelig an, ungefähr so, wie ein eingeklemmter Nerv. Das kommt daher, dass die Nerven in unserem Körper auch mit Elektrizität funktionieren. Sie senden elektrische Signale an das Gehirn. Durch die Batterie an deiner Zunge störst du die Funktion der Nerven, und dementsprechend fühlt es sich an.

Strom aus der Steckdose ist so stark, dass er nicht nur sehr schmerzhaft ist, sondern auch deine Haut verbrennen und dein Herz zum Stillstand bringen kann! Außerdem blockiert er deine Muskeln, so dass du womöglich nicht mehr loslassen kannst. Sogar die Atmung kann durch Strom blockiert werden.

2.2.1 Stromkreis

Im Stromkreis nutzt man die Bewegungsenergie von Elektronen aus. Der Vorgang ist mit einem Heiz-Kreislauf vergleichbar:





Schema eines Heizkreises (Quelle: Wikimedia)

Die Heizungsanlage im Keller pumpt heißes Wasser durch den Kreislauf. Das Wasser gibt die Wärme an den Heizkörper ab und fließt dann zurück zum Heizkessel. Das Wasser wird dabei nicht verbraucht, sondern nur bewegt.

Im elektrischen Stromkreis befinden sich freie Elektronen in der Leitung. Diese werden von der Batterie dazu angeregt, im Kreis zu fließen. Die Elektronen werden dabei nicht verbraucht, sie geben aber ihre Energie an den Verbraucher (hier: Glühlampe) ab.

In einer Eigenschaft verhält sich der elektrische Strom allerdings ganz anders als Wasser:

Wenn du eine Wasserleitung unterbrichst, fließt das Wasser heraus, denn Wasser kann sich durch die Luft bewegen. Bei Strom ist das anders. Die Elektronen fallen nicht aus den Löchern der Steckdose heraus auf den Fußboden. Elektrischer Strom kann nicht durch die Luft fließen. Außerdem will der Strom immer zu seiner Quelle zurück fließen, sonst fließt er nicht.

Die Elektronen kommen bei einer Batterie immer am Minus-Pol heraus und bewegen sich durch den Leiter zum Plus-Pol hin. In Schaltplänen zeichnet man jedoch in der Regel die technische Stromrichtung ein, und die geht von Plus nach Minus.

2.2.2 Leiter

Wenn ein Elektriker von einem Leiter spricht, meint er ein Kabel oder einen einzelnen Draht. Genau genommen sind alle Dinge, wo Elektronen hindurch fließen können, elektrische Leiter. Metalle können Strom leiten, allerdings unterschiedlich gut.

Silber	61
Kupfer	58
Gold	45
Aluminium	37
Wolfram	19
Eisen	10

Leitwerte einiger Metalle (Quelle: Wikimedia)

Silber ist der beste Leiter, aber teuer, deswegen stellt man Kabel fast immer aus Kupfer her, dem zweitbesten Leiter.

Das Gold nur der drittbeste Leiter ist, überrascht dich vielleicht. In der Elektronik beschichtet man Steckverbinder manchmal mit Gold, weil es toll aussieht. Wo es wirklich auf Qualität ankommt, beispielsweise im Auto), verwendet man andere Materialien.

Nicht nur Metalle leiten Strom, sondern auch Wasser, Kohle – ja sogar bestimmte Keramiken, wenn man sie kalt genug macht.

Wenn Strom durch einen sehr guten Leiter wie Silber fließt, verlieren die Elektronen kaum Energie. Gute Leiter bleiben kalt. Wenn Strom durch einen schlechten Leiter fließt, verlieren die Elektronen viel Energie. Schlechte Leiter werden dabei warm.

Das kann natürlich auch gewollt sein. In einem Föhn erzeugt ein schlechter Leiter (z.B. aus Eisen) die gewünschte Wärme. In einer Glühbirne wird ein sehr dünner Draht aus Wolfram so heiß, dass er nicht nur glüht, sondern hell leuchtet.

2.2.3 Isolator

Das Gegenteil vom Leiter ist der Isolator. Alles, was Strom nicht leiten kann, nennt man Isolator. Kabel werden mit einem Isolator umhüllt, damit man sie gefahrlos anfassen kann.

- Plastik
- Glas
- Keramik
- Papier
- Holz
- Luft

In elektrischen Geräten verwendet man sehr gerne Kunststoff, weil es leicht formbar ist. Aber Kunststoff wird im Laufe der Zeit brüchig und reagiert auf hohe Temperaturen empfindlich, deswegen verwenden die Stromversorger an ihren Überlandleitungen lieber Glas und Keramik.

Papier und Holz sind für Bastler gut geeignet, aber nur in trockener Form. Nasses Holz und nasses Papier sind nämlich leitfähig.

2.3 Elektrische Kenngrößen

Wenn du elektrische Schaltungen baust, musst du ab und zu etwas ausrechnen. Die wichtigsten Kenngrößen sind Spannung, Stromstärke und Leistung.

2.3.1 Spannung

Die Spannung ist mit dem Wasserdruck in einer Leitung vergleichbar. Ein Feuerwehr-Schlauch mit viel Druck wirkt wesentlich stärker, als ein einfacher Gartenschlauch mit weniger Druck.

Bei der elektrischen Spannung ist das ähnlich. Je höher die Spannung ist, umso schneller dreht sich ein Motor umso heller leuchtet eine Glühlampe.

Eine Wasserleitung wird bei zu hohem Druck platzen. Zu hohe elektrische Spannung führt dazu, dass Bauteile kaputt gehen. Im Extremfall bewirkt zu hohe Spannung, dass der Strom die Isolatoren durchdringt – das kann man zum Beispiel bei einem Gewitter beobachten, wenn Blitze durch die Luft gehen.

In mathematischen Formeln verwendet man für Spannungen den Buchstaben U, und ihre Maßeinheit ist Volt. Damit du ein Gefühl für diese Maßeinheit bekommst, habe ich mal einige Spannungen notiert:

- Die Oberleitungen der Bahn haben bis zu 25.000 Volt.
- Ein Zitteraal (Fisch) kann dich mit bis zu 500 Volt schocken.
- Aus der Steckdose kommen 230 Volt.
- Ein Handy-Akku hat ungefähr 3,7 Volt.

2.3.2 Stromstärke

Je mehr Strom fließt, umso größer ist die Stromstärke.

Die Stromstärke gibt an, wie viele Elektronen am Betrachter vorbei fließen, wenn man sie sehen könnte. Sie wird in Formeln mit I gekennzeichnet und in der Einheit Ampere gemessen.

Ein Ampere bedeutet konkret, dass etwa 6 Trillionen Elektronen pro Sekunde vorbei kommen. Das muss man sich nicht merken.

Du solltest jedoch eine ungefähre Vorstellung davon haben, welche Stromstärke viel und welche wenig ist.

- Beim Starten eines PKW mit Dieselmotor nimmt dessen Anlasser (Elektromotor) etwa 400 Ampere auf.
- Durch die Heizdrähte eines Toasters fließen etwa 5 Ampere.
- Ein Notebook nimmt etwa 2 Ampere aus der Batterie auf.
- Fahrrad-Scheinwerfer werden mit 0,4 Ampere betrieben.
- Ein Taschenrechner benötigt sogar weniger als 0,01 Ampere.

Du kannst dir also schon denken, dass wir beim Basteln meistens mit Stromstärken unter 1 Ampere zu tun haben. Stromstärken über ein Ampere kommen bei Hobby-Elektronikern eher selten vor. Elektroniker verwenden daher gerne die Einheit „Milliampere“. 1 mA entspricht 0,001 Ampere.

2.3.3 Leistung

Die dritte wichtige Kennzahl in der Elektronik ist die Leistung. Es ist kein Zufall, dass bei Glühlampen, Staubsaugern, Mikrowellen, Autos, etc. immer zuerst nach der Leistung gefragt wird. Die Leistung gibt an, wie viel Energie umgesetzt wird.

$$Leistung = Spannung \cdot Strom$$

Mathematische Formeln benutzen für Leistung das Symbol P. Die Leistung wird in der Einheit Watt angegeben. Bei der vergleichsweise kleinen Fahrrad-Glühbirne hat man zum Beispiel diese Werte:

Spannung U	6 Volt
Stromstärke I	0,4 Ampere (400 mA)
Leistung P	2,4 Watt

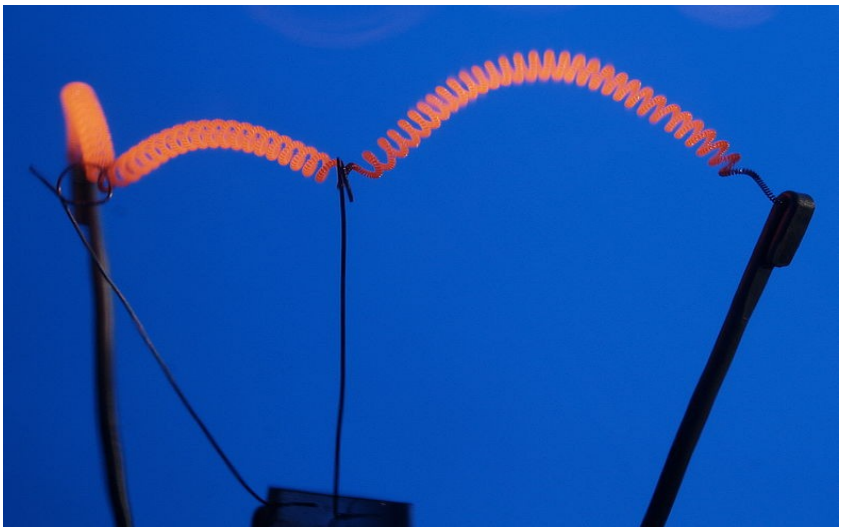
Auf einem Wasserkocher findet man diese Angaben:

Spannung U	230 Volt
Stromstärke I	8,6 Ampere
Leistung P	2000 Watt

Als Hobby-Elektroniker wirst du mit sehr unterschiedlichen Leistungen zu tun haben. Manche Bauteile setzen weniger als ein Milliwatt um, andere wiederum mehrere Watt. Du wirst jedoch nur selten mit Werten über 10 Watt zu tun haben.

2.3.4 Wärme-Wirkung

Stromfluss bewirkt Wärme, weil sich die Elektronen an den unbeweglichen Teilen des Leiters reiben. Wenn der Stromfluss durch einen Draht groß genug ist, dann glüht er sogar.



Blick ins Innere einer Glühlampe (Quelle: Wikimedia, Arnold Paul)

In der Elektronik wird elektrische Leistung zu annähernd 100% in Wärme umgewandelt. Wärme ist das ultimative Abfallprodukt der Elektronik. In der Elektronik gibt es nur wenige Bauteile, die neben Wärme noch etwas anderes produzieren:

- Gute Leuchtdioden setzen 50% der Energie in Licht um, der Rest ist Wärme.
- Glühlampen geben nur 10% als Licht ab, und 90% als Wärme.
- Gute Motoren können bis zu 90% der Energie in Bewegung umsetzen.

Alle elektronischen Geräte werden mehr oder weniger warm. Je mehr elektrische Leistung ein elektronisches Gerät aufnimmt, um so mehr Wärme gibt es ab. Wenn du ein Bauteil überlastest, wird es zu heiß – es brennt dann durch. Deswegen rechnen Elektroniker die Leistung oft nach, wenn sie etwas konstruieren.

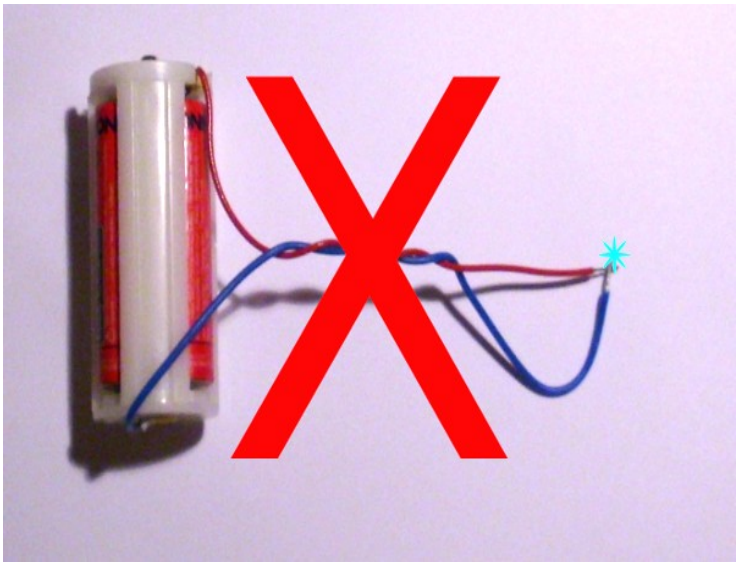
2.3.5 Wechselstrom

Aus der Steckdose kommt Wechselstrom. Wechselstrom ändert seine Fluss-Richtung in regelmäßigen Abständen. Bei der Steckdose ändert sich die Richtung alle 10 Millisekunden. So bewegen sich die Elektronen innerhalb einer Sekunde 50 mal vorwärts und 50 mal rückwärts. Immer hin und her.

Du wirst jedoch mit Gleichstrom basteln, wo der Strom immer in die selbe Richtung fließt.

2.3.6 Kurzschluss

Bei einem Kurzschluss knallt es und es geht etwas kaputt. Kurzschlüsse können sogar Feuer entfachen. Ein Kurzschluss entsteht, wenn du die Ausgänge einer Spannungsquelle (z.B. einer Batterie) direkt verbindest.



Beim Kurzschluss entlädt die Batterie ihr ganze Energie so schnell sie kann, denn der Strom fließt ungehindert direkt von einem Pol zum anderen. Dabei heizt sich nicht nur der Draht auf, sondern auch die Batterie selbst. Durch den großen Kurzschluss-Strom wird die Batterie innerhalb einiger Sekunden zerstört.

Manche Akkus brennen sogar ab, wenn man sie kurz schließt. Kurzgeschlossene Auto-Batterien explodieren und versprühen dabei ihre ätzende Säure.

Wo Kurzschlüsse für den Mensch gefährlich werden können, setzt man daher Sicherungen ein. Sicherungen enthalten einen dünnen Draht, der im Fall eines Kurzschlusses durchbrennt und somit den Stromkreis unterbricht.

Damit kein Feuer entstehen kann, sind Sicherungen in der Regel mit Glas, Keramik oder einem hitzebeständigen Kunststoff umhüllt.

2.4 Löten

Elektroniker müssen Löten. Beim Löten verbindet man Leiter mit geschmolzenem Zinn. Die betroffenen Teile müssen dazu auf etwa 300 Grad erhitzt werden. Das Werkzeug dazu ist der Lötkolben.

Löte niemals unter Spannung! Die Stromversorgung des Werkstücks muss beim Löten IMMER ausgeschaltet sein.

Für gelegentliches Arbeiten an elektronischen Schaltungen genügt so ein preisgünstiger ungeregelter Lötkolben mit ungefähr 30 Watt Leistung. Eine gute Lötspitze ist nützlicher, als aufwändige Temperatur-Regelung.

Wenn du einen neuen Lötkolben auspackst, dann benetze dessen Spitze mit Lötzinn, sobald er dazu warm genug ist. Viele Lötspitzen gehen nämlich kaputt, wenn sie trocken heiß laufen.

Beim Löten ist wichtig, dass die zu verlötenden Teile, und die Spitze des Lötkolben sauber und nicht oxidiert sind. Oxidierte Metalle verbinden sich nicht gut miteinander.

Elektroniker benutzen Lötdraht aus Zinn mit 0,5 bis 1 mm Durchmesser. Im Innern des Drahtes befindet sich eine geringe Menge Flussmittel, zum Beispiel Kolophonium (Baumharz). Das Flussmittel reduziert die Neigung des Zinns, Klumpen zu bilden.

Mit Lötzinn kannst du Kupfer, Bronze, Messing, Silber, Gold und Zink verbinden.

Beim Löten solltest du zum Schutz deiner Atemwege für gute Belüftung sorgen. Bei einfachen ungeregelten Lötkolben kann außerdem ein kleiner Ventilator hilfreich sein, die Temperatur zu reduzieren, insbesondere am Griff.

2.4.1 Verzinnen

Um den Umgang mit dem Lötkolben zu erlernen, fertigst du dir Verbindungskabel für das Steckbrett an. Zerlege ein altes Datenkabel in etwa 30 Einzelne Litzen zu je 20 cm. Dann entfernst du an den Enden der Litzen jeweils 1 cm von der Isolation. Ich ritze dazu die Isolation rundherum mit einem Messer an und ziehe das abgetrennte Ende dann vom Draht ab. Einfacher geht es mit einer sogenannten Abisolierzange, die es in unterschiedlichen Varianten gibt.

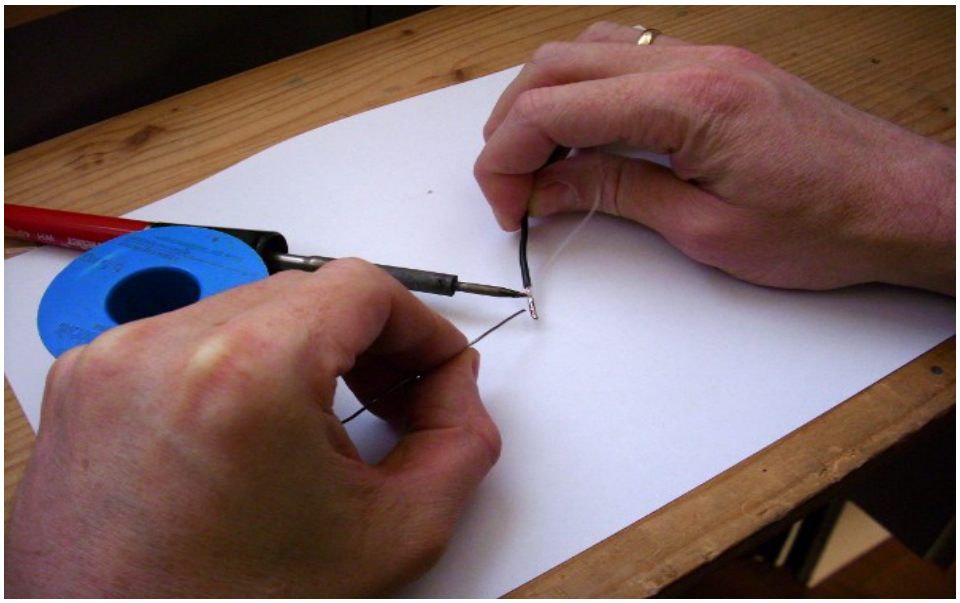
Nach dem Abisolieren verdrillst du die feinen Drähte. Verdrillen heißt, du packst die feinen Drähte zwischen zwei Finger und verdrehst sie Schraubenförmig umeinander, so dass sie nicht mehr auseinander stehen.

So soll es aussehen:

Dann verzinnst du das verdrillte Ende. Verzinnen bedeutet, die Oberfläche der Drähte mit wenig Zinn zu benetzen. Sinn der Sache ist, die Enden der Litzen steif zu machen, damit du sie später mühelos in das Steckbrett stecken kannst.

Verzinnen geht so: Du brauchst eine helfende Hand (oder eine Klemmvorrichtung), um entweder den Draht oder den Lötkolben zu halten. Stecke den Lötkolben in die Steckdose und warte 5 Minuten, damit er heiß genug wird.

Drücke die Spitze des Lötkolbens gegen den Draht und gebe sehr wenig Zinn (höchstens 1 mm) Zinn dazu. Dieses bisschen Zinn überträgt nun die Wärme des Lötkolbens auf den Draht. Nach etwa 2 Sekunden gibst du noch etwas Zinn hinzu, während du mit der Spitze des Lötkolbens am blanken Draht entlang streichst. Wenn es gut läuft, saugt der Draht das Zinn wie ein Schwamm auf.



Verwende wenig Zinn, sonst wird die Litze zu dick und passt nicht ins Steckbrett. Das Endergebnis sollte so aussehen:

Wenn du zu langsam arbeitest, verschmort die Kunststoff-Isolation der Litze. Falls dir das immer wieder passiert, versuche es mit einem anderen Kabel. Manche Hersteller verwenden sehr empfindliche Kunststoffe, die zum Löten nicht so gut geeignet sind.

Löten ist nicht einfach. Stelle dich darauf ein, dass du viele Versuche brauchst, bis die Ergebnisse gut werden.

2.4.2 Lötkolben reinigen

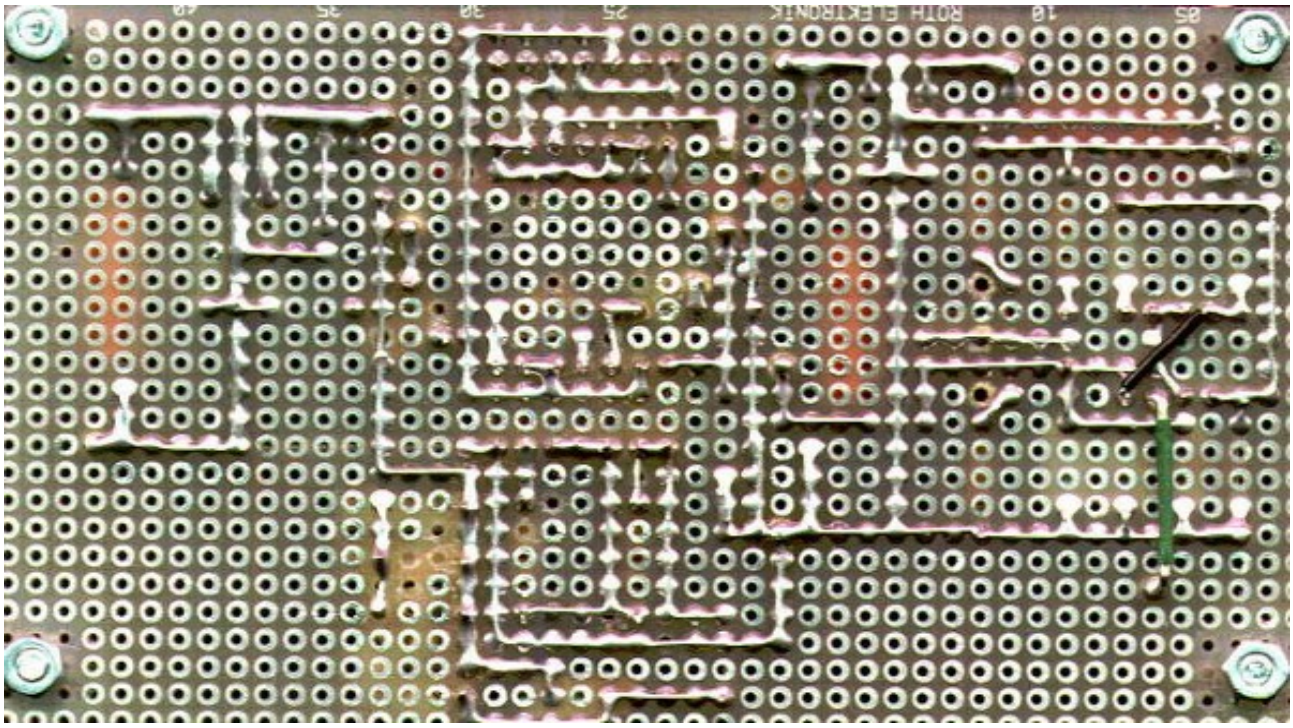
Die Spitze vom Lötkolben sollte vor der Benutzung gereinigt werden - nicht danach!

Lötkolben werden häufig zusammen mit einem hitzebeständigen Schwamm verkauft, den man feucht machen soll, um daran die Lötspitze zu reinigen. Alternativ dazu eignet sich Metallwolle oder ein Tuch aus reiner Baumwolle. Was man auf jeden Fall vermeiden sollte, ist heftiges Kratzen und Schaben an der Oberfläche. Denn wenn die Nickel-Beschichtung weg ist, kann man die Spitze nicht mehr verwenden.

Da Lötspitzen Verschleißteile sind, achte beim Kauf darauf, dass passende Lötspitzen erhältlich sind.

2.4.3 Elektronische Bauteile löten

Elektronische Bauteile werden auf Leiterkarten (umgangssprachlich: Platinen) gelötet. Für Hobbyelektroniker werden sogenannte Punktraster-Platinen angeboten, auch Lochraster-Platinen genannt. Man steckt die Bauteile durch die Löcher und verbindet sie mit Drähten auf der Rückseite der Platine:



Rückseite einer gelöteten Lochrasterplatine (Quelle: Ein anonymer Beitrag im Roboternetz Forum)

Da sich oxidierte Metalle schlecht löten lassen, eignen sich versilberte Drähte besonders gut - sie oxidieren wenig. Das klingt teurer als es ist. Oxidierte Platinen kann man vor dem Löten sehr gut mit einem blauen Radiergummi oder mit einem eingeseiften Topfreiniger aus Stahlwolle reinigen.

Das Verlöten von elektronischen Bauteilen erfordert viel Geschick, denn man muss es schnell machen. Die meisten elektronischen Bauteile vertragen Temperaturen über 100 Grad nur für wenige Sekunden.

Ich habe mir beim Löten von Platinen angewöhnt, die Zeit in Sekunden zu zählen.

1. Lötkolben an die Lötstelle drücken
2. Sehr wenig (maximal 1 mm) Lötzinn dazu geben, damit die es die Wärme überträgt.
3. 2 Sekunden warten
4. So viel Zinn dazu geben, wie die Lötstelle erfordert
5. wieder 2 Sekunden warten
6. Lötkolben weg nehmen und sicher ablegen

Länger darf es nicht dauern. Wenn der Lötvorgang bis dahin noch nicht abgeschlossen ist, musst du ihn abbrechen und dem Bauteil Zeit zum Abkühlen geben.

Die Plastik Gehäuse von Leuchtdioden schmelzen besonders schnell. Deswegen solltest du die Anschlussdrähte von Leuchtdioden möglichst lang lassen. Bei langen Drähten kommt die Hitze nicht so schnell am empfindlichen Ende an.

Das Flussmittel aus dem Lötdraht soll die zu verlötenden Metallflächen benetzen und in die Löcher hinein fließen. Schmelze den Lötdraht daher möglichst nahe an der Lötstelle ein.

Es wäre falsch, das Lot zuerst auf die heiße Lötspitze zu geben und danach an die Lötstelle zu führen. Bis dahin wäre das Flussmittel bereits verdampft und das Zinn würde Klumpen bilden.

Früher, als bleihaltiges Zinn üblich war, galt eine perfekt glänzende Oberfläche als Zeichen für eine gute Lötstelle. Bei dem heute üblichen bleifreiem Zinn ist das allerdings nicht mehr der Fall. Heute haben auch gute Lötstellen meistens eine matte Oberfläche.

2.4.4 Schutzlack

Nach dem Löten empfiehlt sich, die Platine mit einem speziellen Schutzlack zu überziehen. Denn oxidierte Platinen kannst du später im Fall des Falles kaum noch reparieren. Das Oxid selbst schadet den Bauteilen kaum, aber es erschwert spätere Lötvorgänge (Reparaturen) massiv.

Ich benutze dazu immer das Spray „Plastik 70“ von CRC Industries, welches jeder Elektronik-Händler vorrätig hat. Im Gegensatz zu gewöhnlichem Lack-Spray ist dieser sehr hitzebeständig. Zwar schmilzt er unter der Hitze des Lötkolben, aber er verbrennt oder verkohlt dabei nicht. Es bleibt also eine saubere Sache.

2.4.5 Entlötpumpe

Um ungewollte Zinn-Kleckse von Platinen zu entfernen, brauchst du eine Entlötpumpe.

Man benutzt sie so: Zuerst drückst du den Kolben in die Pumpe hinein. Dabei wird eine Feder im Innern der Pumpe gespannt. Der Kolben rastet am Ende ein.

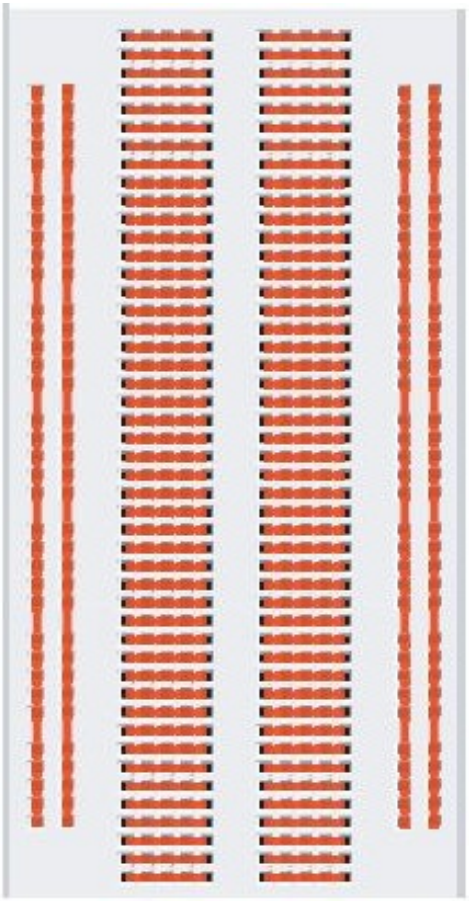
Dann bringst du das Zinn mit dem Lötkolben zum schmelzen. Wenn es soweit ist, drückst du die Spitze der Entlötpumpe sanft auf das flüssige Zinn und drückst den Auslöse-Knopf. Mit einem kräftigen Ruck schlürft die Pumpe dann das flüssige Zinn in sich hinein.

Anschließend drückst du den Kolben wieder in die Pumpe. Dabei kommt vorne aus der Spitze das erstarrte Zinn heraus. Ab in den Müll damit. Gebrauchtes Lötzinn kann man nicht nochmal verwenden.

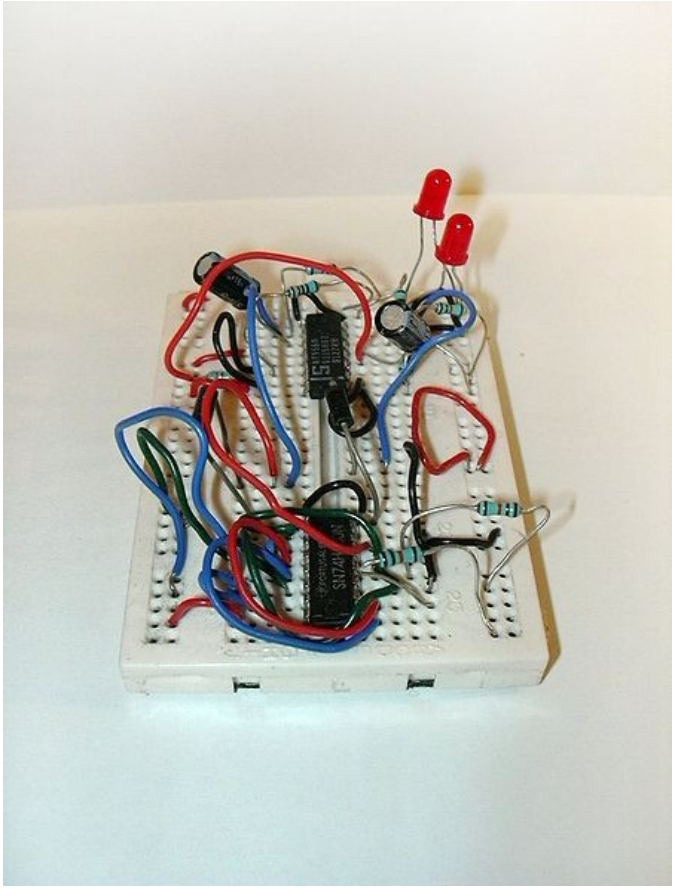
Manchmal verstopft oder verklemmt sich die Pumpe. Dann musst du sie auseinander schrauben und reinigen. Die weiße Spitze der Entlötpumpe ist ein Verschleißteil. Sie hält nicht sehr lange, darum kann man sie als Ersatzteil nachbestellen.

2.5 Steckbrett

Die meisten Experimente wirst du auf einem Steckbrett aufbauen. Steckbretter haben viele Löcher mit Kontaktfedern, die nach folgendem Muster miteinander verbunden sind:



Verbindungen im Steckbrett (Quelle: Strippenstrolch.de)



Wilder Aufbau auf einem Steckbrett (Quelle: Wikimedia, anonym)

Die langen vertikalen Verbindungen sind bei manchen Steckbrettern in der Mitte unterbrochen.

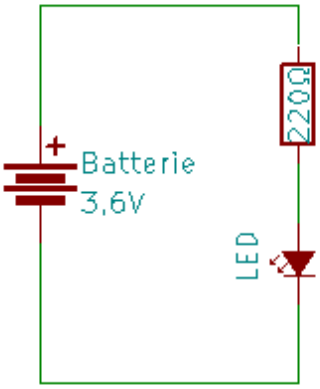
Wenn du mit einem Steckbrett arbeitest, achte darauf, dass die Drähte der Bauteile maximal 0,7 mm dick sind und keine Lötzinnreste anhaften. Dickere Drähte beschädigen die Kontaktfedern. Achte auch darauf, dass das Steckbrett nicht feucht wird.

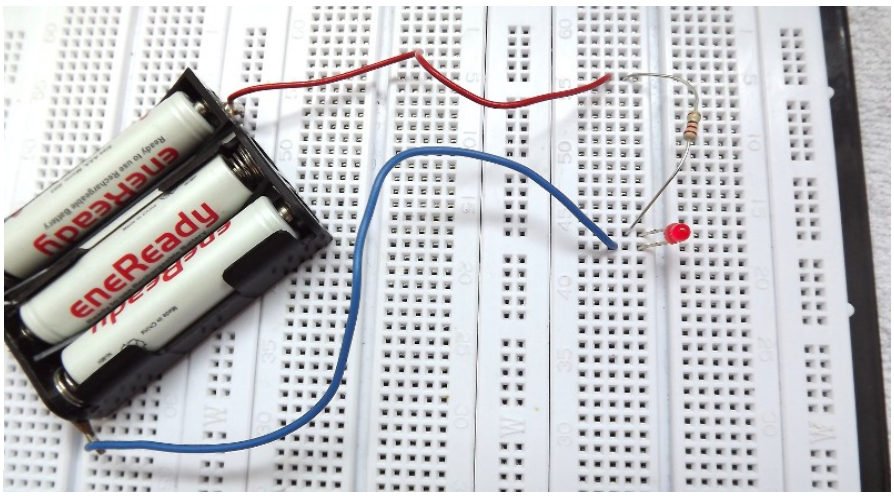
Nun bauen wir unsere erste elektronische Schaltung auf dem Steckbrett auf. Wir wollen eine Leuchtdiode zum Leuchten bringen.

Du benötigst folgende Bauteile:

- Einen 220 Ohm Widerstand (Farben: rot-rot-braun oder rot-rot-schwarz-schwarz)
- Eine Leuchtdiode (Farbe und Größe ist egal)
- Den Batterie-Halter mit 3 Batterien oder Akkus

Falls dein Batterie-Halter noch keine Anschlussdrähte hat, löte welche an. Ich verwende immer rot für den Plus-Pol und schwarz oder blau für den Minus-Pol.





Stecke die Teile wie im Foto gezeigt in das Steckbrett. Bei der Leuchtdiode kommt es darauf an, sie richtig herum einzubauen. Die abgeflachte Seite gehört in diesem Fall nach unten. Die beiden Bauteile werden in den nächsten Kapiteln weiter erläutert.

Die Elektronen kommen aus dem Minus-Pol der Batterie. Sie fließen durch die Leuchtdiode hindurch, dann durch den Widerstand und wieder zurück zur Batterie.

Das Schaltzeichen der Leuchtdiode erinnert an einen Pfeil, der allerdings in die umgekehrte Richtung zeigt. In elektronischen Plänen zeigen Pfeile immer von Plus nach Minus, während die Elektronen sich von Minus nach Plus bewegen. Mich hat das lange irritiert, aber so ist es halt.

Der Widerstand reduziert die Stromstärke. Ohne Widerstand würde ein viel zu starker Strom durch die Leuchtdiode fließen, nämlich soviel, wie die Batterie schafft. Die Leuchtdiode würde dann heiß werden und durchbrennen.

Du kannst die Bauteile auch umgekehrt anordnen, so dass der Strom zuerst durch den Widerstand fließt und dann durch die Leuchtdiode.

2.6 Messen

In diesem Kapitel wirst du lernen, das Digital-Multimeter zu benutzen. Als Elektroniker brauchst du auf jeden Fall ein Digital-Multimeter. Ein einfaches Modell für 10-20 Euro, wie das hier abgebildete genügt. Lies die Bedienungsanleitung!

Stecke das schwarze Kabel immer in die COM Buchse ein. In welche Buchse das rote Kabel gehört, hängt von der Einstellung des Dreh-Schalters ab. An dem Dreh-Schalter stellt du ein, was du messen willst:

- V= Gleichspannung
- V~ Wechselspannung
- A= Gleichstrom
- Ω Widerstand
- ⊣▶⊢ Dioden

Jede Hauptfunktionen ist in Messbereiche unterteilt (z. B. 2, 20, 200, 2000).

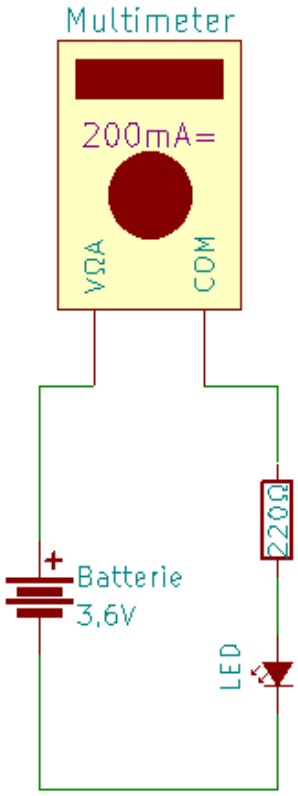
2.6.1 Stromstärke messen

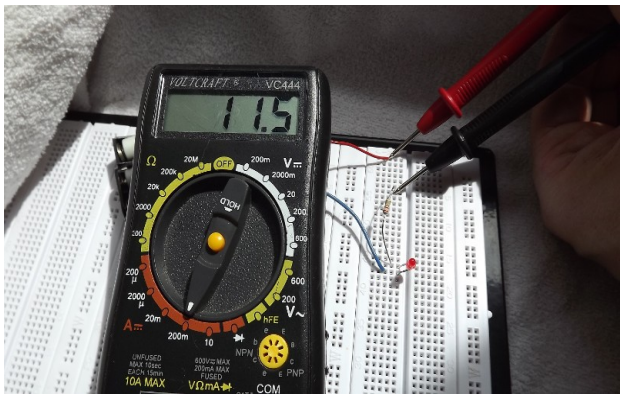
Die Stromstärke gibt an, wie viel Strom gerade durch eine Leitung hindurch fließt. Vergleiche das mit einer Wasseruhr – hat jeder im Keller:

Die Wasseruhr misst, wie viel Wasser durch sie hindurch fließt. Das Multimeter misst, wie viel Strom durch das Multimeter fließt. Dazu muss man die Leitung unterbrechen und den Strom durch das Messgerät leiten.

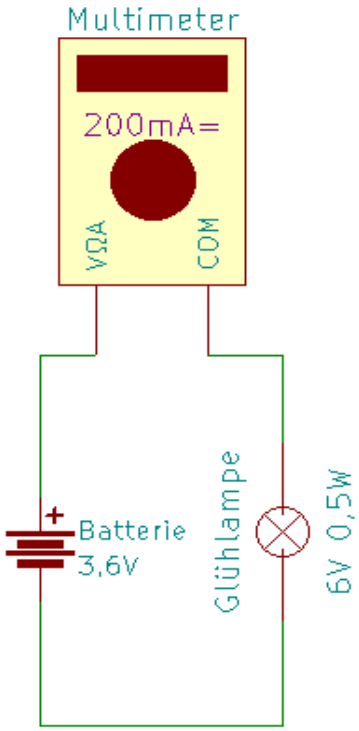
Stelle den Drehschalter des Multimeters auf 200 mA=. Stecke die schwarze Leitung in die COM Buchse und die rote Leitung in die Buchse, die für Strommessungen vorgesehen ist. Bei meinem Gerät ist sie mit „VΩmA“ beschriftet.

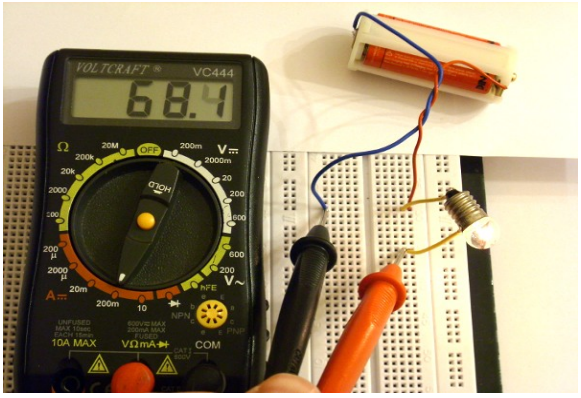
Nimm den Aufbau aus dem vorherigen Kapitel. Dort baust du zusätzlich das Multimeter so ein, dass der Strom durch das Messgerät hindurch fließen muss:





Das Messgerät zeigt an, dass die Stromstärke 11,5 mA beträgt. Zum Vergleich messen wir die Stromaufnahme einer Glühbirne (ohne Widerstand):

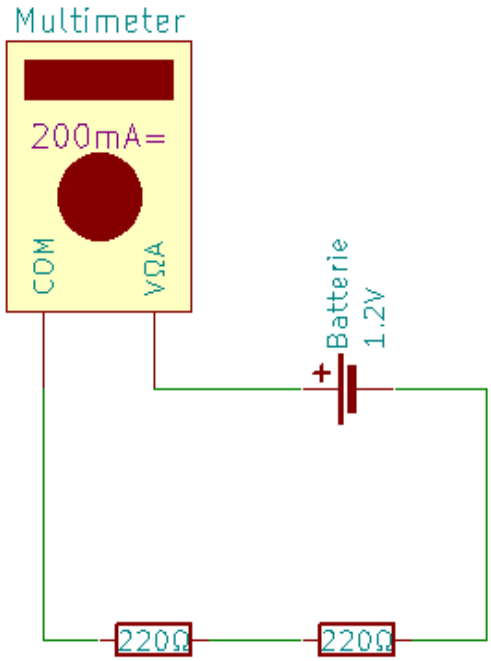


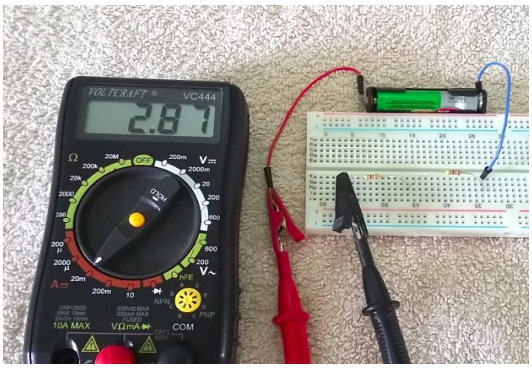


Die Glühbirne benötigt mehr Strom, als die Leuchtdiode. In diesem Fall sind es 68,1 mA.

Ein weiterer Versuch:

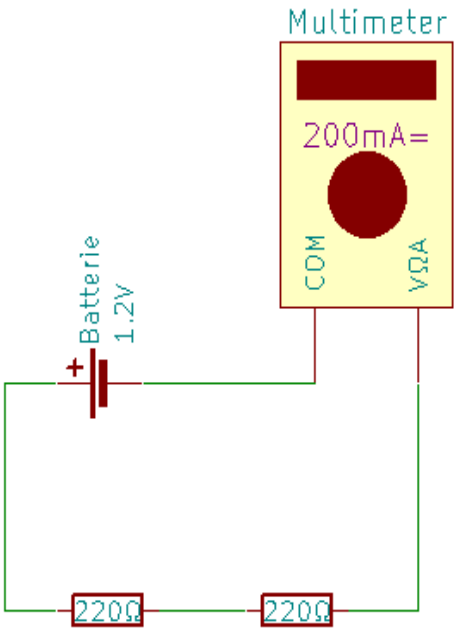
Stecke zwei 220 Ω Widerstände (rot-rot-braun oder rot-rot-schwarz-schwarz) so zusammen, dass sie in der Mitte miteinander verbunden sind. Schließe dann eine (oder drei) Batterien und ein Messgerät gemäß der folgenden Zeichnung an.

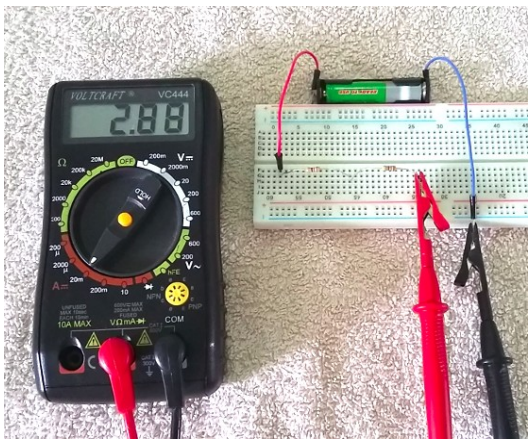




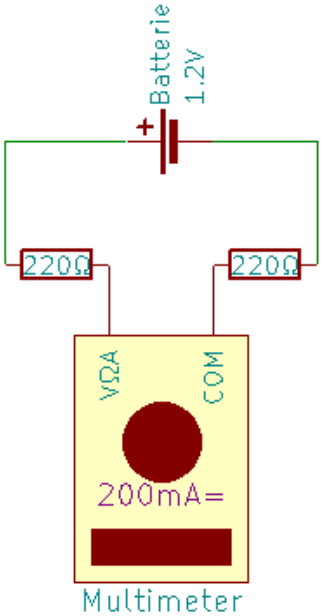
Das Messgerät zeigt an, dass am Plus-Pol der Batterie ein Strom von 2,87 mA fließt. Bei drei Batterien ist der Strom entsprechen größer.

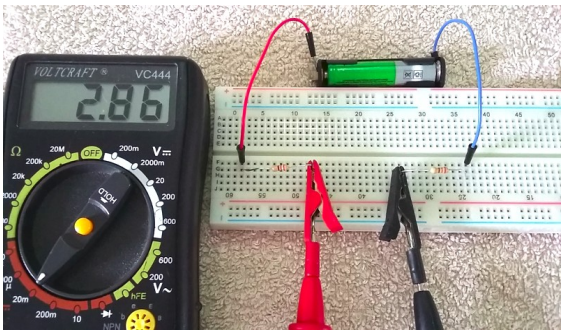
Wie viel Strom fließt wohl am Minus-Pol?





Am Minus-Pol fließt genau so viel Strom, wie am Plus-Pol. Nun messen wir die Stromstärke in der Mitte:





Du hast die Stromstärke an drei Stellen gemessen, und jedes mal (fast) den selben Wert erhalten. In den obigen Fotos waren es die Werte:

- 2,87 mA
- 2,88 mA
- 2,86 mA

Ein physikalisches Gesetz besagt, dass die Stromstärke in einem Stromkreis an allen Punkten immer genau gleich ist. Die leicht abweichenden oder gar schwankenden Anzeigen entstehen durch Wackelkontakte im Steckbrett und Ungenauigkeiten im Messgerät. Das ist völlig normal, es bedeutet keineswegs, dass das Messgerät mangelhaft ist.

Um dein Messgerät nicht versehentlich zu zerstören, solltest zu folgende Ratschläge bei der Strom-Messung beachten:

Das Messgerät muss bei der Strom-Messung (A) immer in eine bestehende Leitung eingefügt werden. Schließe das Messgerät niemals direkt an die Plus- und Minus-Pole der Batterie an, denn das käme einem Kurzschluss gleich. Dann brennt die Sicherung im Messgerät durch.

Bei einer unbekannten Stromstärke sollst du immer mit dem größten Messbereich anfangen. Wenn du dann siehst, dass der Strom gering genug ist, kannst du zu einem kleineren Messbereich wechseln.

Der 200 mA Bereich ist durch eine austauschbare Schmelzsicherung vor zu großen Stromstärken geschützt. Die anderen Bereiche können bei erheblicher Überlast kaputt gehen, denn sie sind nicht abgesichert.

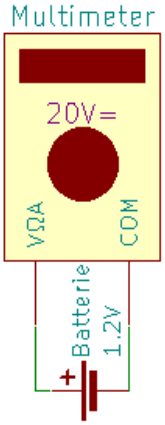
2.6.2 Gleichspannung messen

Bei der Spannungsmessung findest du heraus, wie viel Volt zwischen zwei Punkten anliegt. Das Messgerät „fühlt“ die Spannung an den Messspitzen. Man kann sie an beliebige Punkte einer elektrischen Schaltung dran halten.

Lass uns messen, wie viel Volt eine Batterie hat.

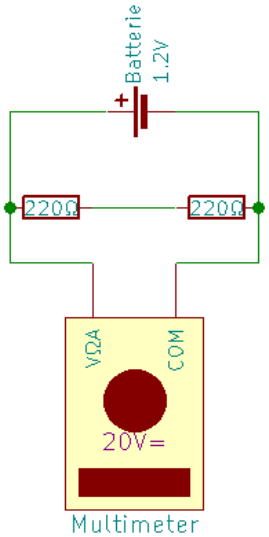
Stelle das Messgerät auf den 20 V= Bereich. Stecke die schwarze Leitung in die COM Buchse und die rote Leitung in die Buchse, die für Spannungsmessungen vorgesehen ist. Bei meinem Gerät ist sie mit „VΩmA“ beschriftet.

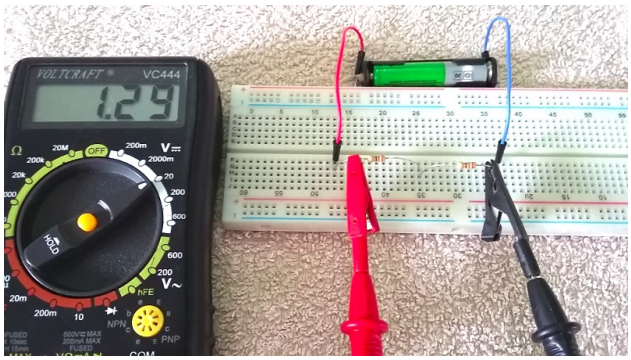
Dann hältst du einen Akku direkt an die Messleitungen. Rot gehört an den Plus-Pol und Schwarz gehört an den Minus-Pol.



Das Messgerät zeigt an, dass die Batterie gerade 1,41 V liefert, was für frisch geladene Akkus normal ist. Die Spannung wird bald auf etwa 1,2 V absinken.

Jetzt messen wir die Spannung in einer elektronischen Schaltung.



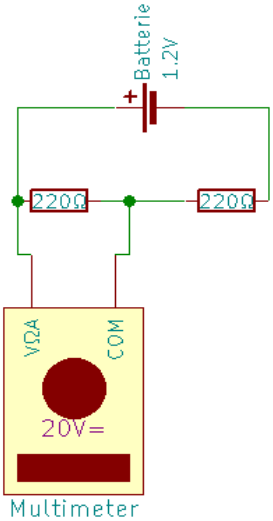


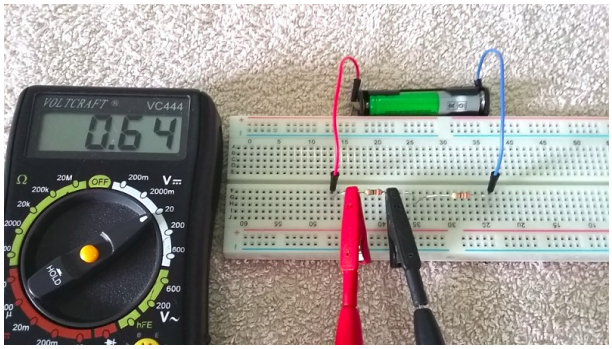
Stecke zwei 220 Ohm Widerstände (rot-rot-braun oder rot-rot-schwarz-schwarz) so zusammen, dass sie in der Mitte miteinander verbunden sind. Schließe dann eine Batterie und ein Messgerät gemäß der folgenden Zeichnung an.

Meine grüne Batterie hat 1,29 Volt, also ein bisschen weniger, als die weiße Batterie von vorhin. Der Strom fließt jetzt gleichzeitig über zwei Wege:

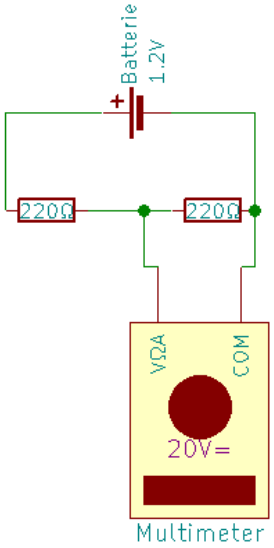
1. durch die beiden Widerstände, und
2. durch das Messgerät.

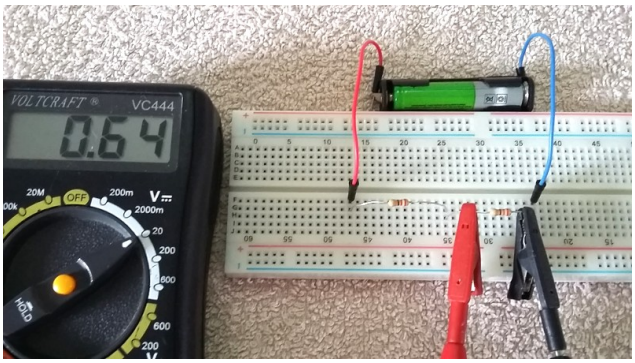
Nun messen wir, wie viel Spannung am linken Widerstand anliegt.





Das Messgerät zeigt nun 0,64 Volt an. Dieser Widerstand erhält also nur die Hälfte der Batteriespannung. Wo die andere Hälfte hängen geblieben ist, kannst du sicher erraten (am rechten Widerstand). Prüfe es nach!





Die Batteriespannung hat sich also an den Widerständen aufgeteilt. Eine Hälfte ist links, die andere Hälfte ist rechts.

Um dein Messgerät nicht versehentlich zu zerstören, solltest zu folgende Ratschläge beachten:

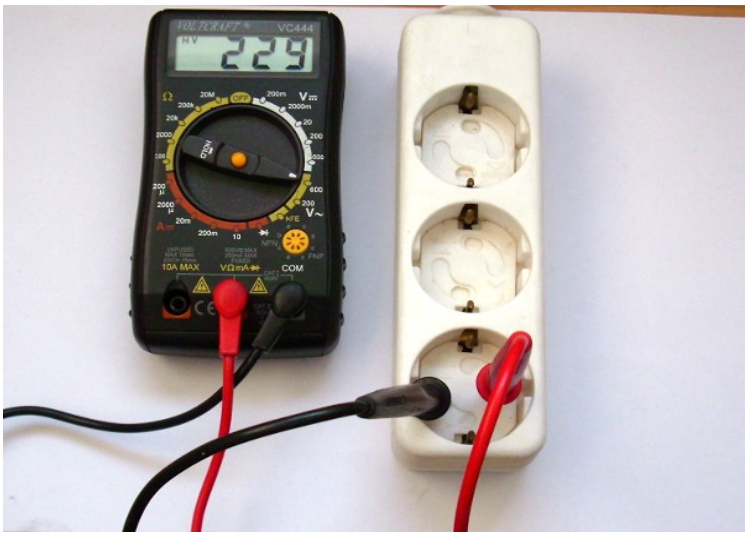
Wenn du keine Ahnung hast, wie viel Volt zu erwarten sind, dann stelle das Messgerät zunächst immer auf die höchst mögliche Spannung ein. Bei meinem Gerät wäre das der 600 V Bereich.

Wenn das Messgerät mit viel mehr Spannung belastet wird, als am Drehschalter eingestellt wurde, geht es kaputt. Deswegen solltest du vor allem den 200 mV (das sind 0,2 V) Bereich meiden. Du weißt ja jetzt schon, dass deine Batterien viel mehr als 200 mV liefern.

2.6.3 Wechselspannung messen

Das folgende Experiment ist gefährlich, darum sollst du es NICHT durchführen. Du sollst nur lesen, wie es geht. Auf gar keinen Fall sollst du es heimlich oder alleine machen. Selbst bei einfachsten Arbeiten an der Steckdose muss immer ein Erwachsener dabei sein, der im Notfall hilft. Das gilt auch für erfahrene Elektriker-Meister.

Zum Messen unbekannter Wechselspannungen stellt man den höchsten V~ Bereich ein und steckt die rote Messleitung in die V Buchse. Dann steckt man die Messspitzen in die Löcher der Steckdose und liest den Spannungswert vom Display ab:

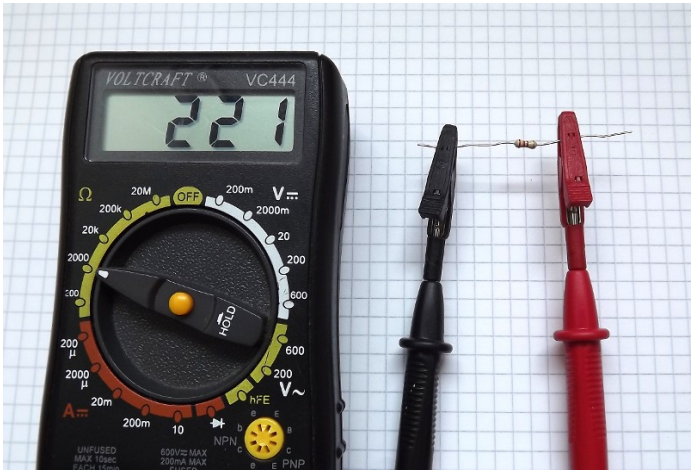


In diesem Fall haben wir 229 Volt. Und jetzt erkläre ich dir, was daran gefährlich ist:

Erstens besteht die Gefahr, dass du mit dem Finger eine der beiden Messspitzen berührst. Dann bekommst du einen Stromschlag, der tödlich verlaufen kann. Zweitens kann das Messgerät bei falscher Einstellung in Flammen aufgehen.

2.6.4 Widerstand messen

Widerstände bremsen den Stromfluss aus. Das Multimeter zeigt an, wie viel Widerstand das Bauteil dem Strom entgegen setzt. Stelle den Drehschalter auf 2 kΩ (oder 2000 Ω) . Stecke die schwarze Leitung in die COM Buchse und die rote Leitung in die Ω Buchse. Wenn du nun einen 220 Ohm Widerstand an die Messleitungen hältst, zeigt das Messgerät auch ungefähr 220 an.



Durch Fertigungstoleranzen beim Widerstand und beim Messgerät entstehen geringe Abweichungen von Soll-Wert. Die Anzeige in diesem Foto ist also in Ordnung.

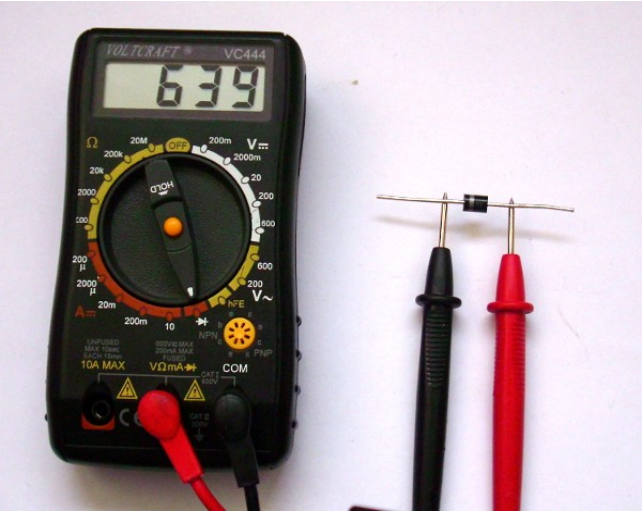
Bei der Messung fließt ein geringer Strom vom schwarzen Kabel aus durch den Widerstand und dann durch das rote Kabel zurück ins Messgerät. Das Messgerät ermittelt, wie sehr der Stromfluss durch den Widerstand behindert wird. Eine große Zahl bedeutet, dass der Strom stark behindert wird. Eine kleine Zahl bedeutet, dass der Strom wenig behindert wird.

- Großer Widerstands-Wert = wenig Stromfluss
- Kleiner Widerstands-Wert = viel Stromfluss

Widerstände kann man nur auf diese Art messen. Es ist nicht möglich, Widerstände in eingebautem Zustand zu messen. Versuche vor allem nicht, Widerstände in Geräten zu messen, die unter Strom stehen, denn dadurch kann das Messgerät zerstört werden.

2.6.5 Dioden testen

Alle Dioden lassen den Strom nur in eine Richtung fließen. Bei der Leuchtdiode hast du das bereits gesehen. Für das nächste Experiment verwenden wir eine nicht leuchtende Diode, die 1N 4001.



Der Strom fließt vom schwarzen Minus-Pol zum roten Plus-Pol. An der Diode ist der Minus-Pol durch eine silbernen Ring gekennzeichnet. Die Zahl 639 im Display bedeutet, dass an dieser Diode 639 Millivolt (also 0,639 Volt) verloren gehen. Auf diese Eigenschaft kommen wir später nochmal zurück.

Wenn du die Diode anders herum drehst, fließt kein Strom. Dann zeigt das Messgerät einen Strich an, oder eine Eins am linken Display-Rand. Dioden darfst du nur in ausgebautem Zustand messen oder wenn die elektronische Schaltung stromlos ist. Ansonsten kann das Messgerät kaputt gehen.

Leuchtdioden kann man leider nicht mit allen Multimetern messen, da sie mehr Spannung benötigen, als das Multimeter abgibt.

2.6.6 Regeln zum Messen

Das schwarze Kabel gehört in die COM Buchse. Das rote Kabel gehört in die Buchse, die entsprechend dem eingestellten Messbereich beschriftet ist.

Stromstärken (A) misst man in der laufenden Schaltung, indem man eine Leitung unterbricht und das Messgerät an dieser Stelle einfügt. Nur der 200 mA Bereich ist durch eine Sicherung vor Überlast geschützt, deswegen nutzen wir diesen Messbereich am liebsten.

Spannungen (V) kann man überall in der laufenden Schaltung messen, ohne Leitungen zu unterbrechen. Das Messgerät geht allerdings kaputt, wenn die Spannung viel größer ist, als der eingestellte Messbereich. Du wirst für alle Experimente in diesem Buch den 20 V= Bereich verwenden.

Widerstände (Ω) kann man nur in ausgebautem Zustand messen. Ein falsch gewählter Messbereich schadet nicht. Das Messgerät geht kaputt, wenn man Widerstände in eingebautem Zustand unter Spannung misst.

Dioden (⊣▶⊢) misst man normalerweise in ausgebautem Zustand. Und wenn nicht, dann muss die Schaltung wenigstens stromlos sein. Der angezeigte Messwert ist die Verlustspannung der Diode in Millivolt. Das Messgerät geht kaputt, wenn man Dioden in eingebautem Zustand unter Spannung misst.

Diese Regeln solltest du dir gut einprägen.

Schau dir dieses Video an: https://youtu.be/f5fwdGFBrwU

2.7 Bauteilkunde

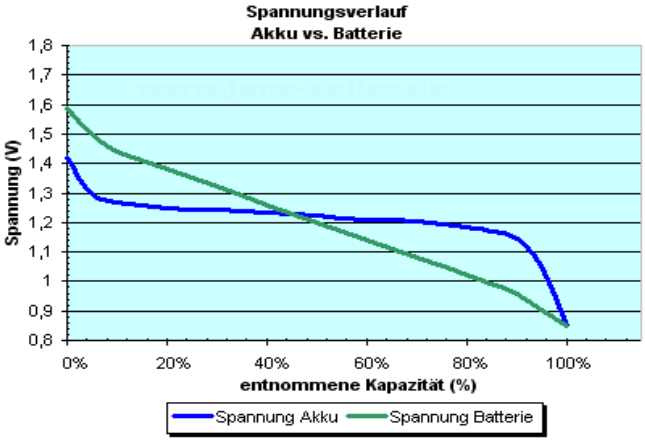
In diesem Kapitel wirst du die Bauteile kennen lernen, aus denen Mikrocomputer bestehen. Du wirst auch die offiziellen Symbole für Schaltpläne kennen lernen. Schau dir dazu einfach die Bilder an, dann wirst du bald auch eigene Schaltpläne zeichnen können.

2.7.1 Batterien

Die Experimente aus diesem Buch verwenden alkalische Einwegbatterien oder NiMh Akkus, darum beschränkt sich dieses Kapitel auf diese beiden Bauarten.

Alkalische Einwegbatterien liefern in frischem Zustand 1,6 V. Mit zunehmendem Verbrauch sinkt die Spannung merklich ab, bis auf etwa 0,9 Volt. Dann gilt die Batterie als verbraucht.

NiMh Akkus liefern bis kurz vor dem Erschöpfen eine ziemlich konstante Spannung von 1,2 bis 1,3 Volt. Erst wenn der Akku leer ist, sinkt die Spannung recht plötzlich ab. Das folgende Diagramm zeigt den Spannungsverlauf der beiden Batterietypen im Vergleich.



Quelle: www.jens-seiler.de

NiMh Akkus sind für Elektronik häufig besser geeignet, weil ihre Spannung fast die ganze Zeit lang annähernd konstant ist.

NiMh Akkus der alten Bauart entladen sich innerhalb von 1-2 Monaten von selbst. Doch aktuelle Modelle werden einsatzbereit verkauft und halten ihre Ladung typischerweise ein ganzes Jahr. Sie werden oft mit dem Vermerk „Ready to use“ gekennzeichnet.

Das Original ist die „Eneloop“ von Sanyo, aber es gibt längst gute preisgünstigere Alternativen, zum Beispiel „Eneready“ und die Hausmarke „Rubin“ von der Drogerie Rossmann.

Vermeide es, Akkus auf weniger als 0,9 V zu entladen. Denn dann findet eine Chemische Reaktion statt, die ihn zerstört.

2.7.1.1 Kapazität

Ein idealer Akku mit 2000 mAh (Milli-Ampere-Stunden) Kapazität kann eine Stunde lang 2000 mA liefern. Oder er kann 10 Stunden lang 100 mA liefern.

- Entladezeit = Kapazität : Entladestrom

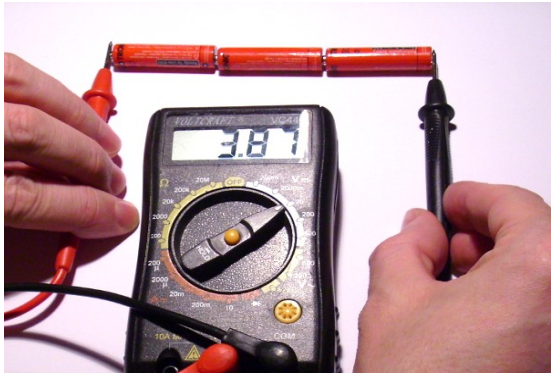
Wenn du den Stromverbrauch deines Gerätes kennst, kannst du ausrechnen, wie lange der Akku halten wird. Erinnere dich an die Schaltung mit der Leuchtdiode auf dem Steckbrett. Du hast nachgemessen, dass dort eine Stromstärke von etwa 2 mA fließt. Auf deinen Akkus steht wahrscheinlich 900 mAh (oder ein ähnlicher Wert) drauf.

Also rechnest du 900 mAh : 2 mA = 450 Stunden

Die Leuchtdiode wird demnach voraussichtlich 450 Stunden hell leuchten, und dann erst allmählich dunkel werden.

2.7.1.2 Reihenschaltung

Man schaltet Batterien in Reihe, um die Spannung zu erhöhen.



In deinem Batteriehalter sind drei Batterien oder Akkus in Reihe geschaltet. Die Gesamt-Spannung rechnest du so aus:

- Gesamt-Spannung = Summe aller Teilspannungen

Meine drei Akkus sind gerade frisch geladen, sie liefern daher 1,3 Volt. Zusammen (fast) 3,9 Volt, wie man am Foto sehen kann.

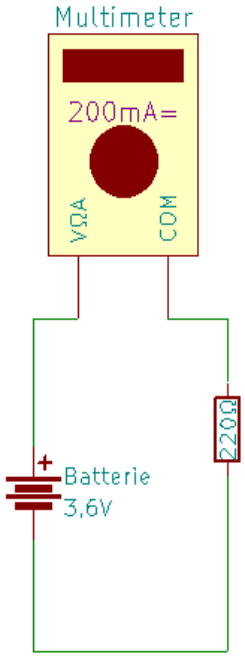
2.7.2 Widerstände

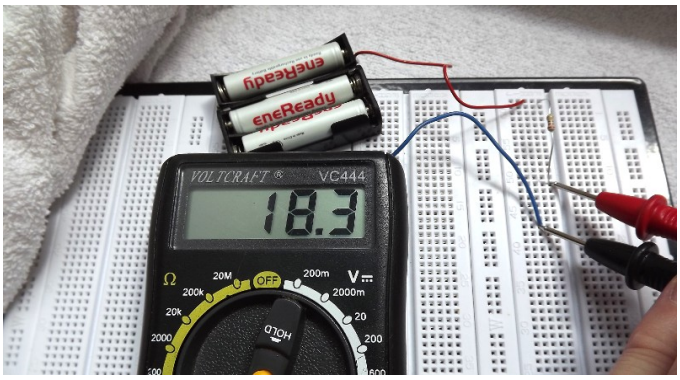
Beim ersten Experiment mit der Leuchtdiode hast du bereits einen Widerstand verwendet. In diesem Fall wurde der Widerstand verwendet, um die Stromstärke zu regulieren.

Widerstände bestehen typischerweise aus Kohle oder einem schlecht leitendem Metall. Sie bremsen die Elektronen aus, indem Sie einen Teil der Energie in Wärme umwandeln.

In mathematischen Formeln benutzt man für Widerstände das Symbol R und ihre Wert wird in Ohm (Ω) angegeben. Je größer der Widerstandswert ist, umso schlechter leitet er den Strom.

Im folgenden Experiment messen wir, wie viel Strom ein Widerstand fließen lässt. Stecke den Batterie-Pack, einen 220 Ohm Widerstand (Farben: rot-rot-braun oder rot-rot-schwarz-schwarz) und das Multimeter wie im folgenden Bild gezeigt zusammen:





Das Digital-Multimeter ist auf den 200 mA= Bereich eingestellt. Es zeigt an, dass durch den Stromkreis ein Strom von 18,3 Milliampere fließt.

2.7.2.1 Berechnung

Du kannst das Verhältnis zwischen Spannung, Strom und Widerstandswert nach folgenden Formeln berechnen:

- Stromstärke = Spannung : Widerstand
- Spannung = Stromstärke · Widerstand
- Widerstand = Spannung : Stromstärke

Meine frischen Batterien liefern zusammen 4 Volt. Der erwartete Strom ist daher:

- 4 V : 220 Ω = 0,018 A

Das Ergebnis stimmt mit der obigen Messung überein.

2.7.2.2 Farbcodierung

Kleine Widerstände werden mit bunten Ringen bedruckt, anstatt mit Zahlen. Dabei sind zwei Varianten üblich:

Codierung mit 3 Ringen

erster Ring = erste Ziffer
zweiter Ring = zweite Ziffer
dritter Ring = Anzahl der Nullen

Beispiele:

rot-rot-braun = 220 Ohm

rot-violett-orange = 27000 Ohm

0	schwarz
1	braun
2	rot
3	orange
4	gelb
5	grün
6	blau
7	violett
8	grau
9	weiß

Codierung mit 4 Ringen

erster Ring = erste Ziffer
zweiter Ring = zweite Ziffer
dritter Ring = dritte Ziffer
vierter Ring = Anzahl der Nullen

Beispiele:

rot-rot-schwarz-schwarz = 220 Ohm

rot-violett-schwarz-rot = 27000 Ohm

Nach dem Widerstandswert folgt eine etwas größere Lücke und dann mehr oder weniger zusätzliche Ringe, deren Bedeutung für unsere Experimente unwichtig ist.

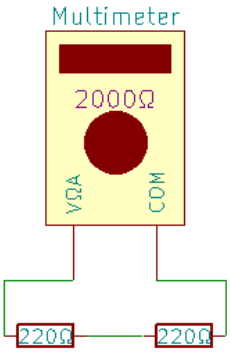
Manchmal ist nur schwer zu erkennen, wie herum man den Widerstand halten muss, um den Aufdruck zu lesen. In diesem Fall kannst du dein Multimeter benutzen, und einfach nach messen.

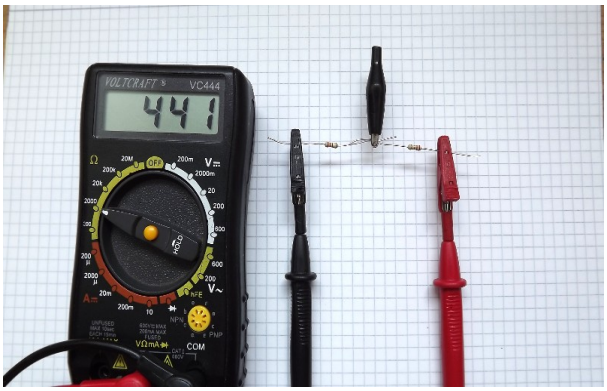
2.7.2.3 Reihenschaltung

Du hast bereits Batterien in Reihe geschaltet, um die Spannung zu erhöhen.

Und du hast bereits zwei Widerstände in Reihe geschaltet, und mit dem Multimeter herausgefunden, dass sich dann die Spannung auf beide Widerstände aufteilt.

Manchmal schaltet man Widerstände in Reihe, weil man einen bestimmten Wert nicht einzeln kaufen kann. 440 Ohm kann man zum Beispiel nicht kaufen, aber man kann stattdessen zwei mal 220 Ohm in Reihe schalten.





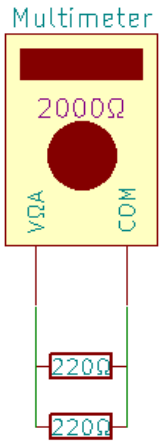
Der Gesamt-Widerstand wird nach dieser Formel berechnet:

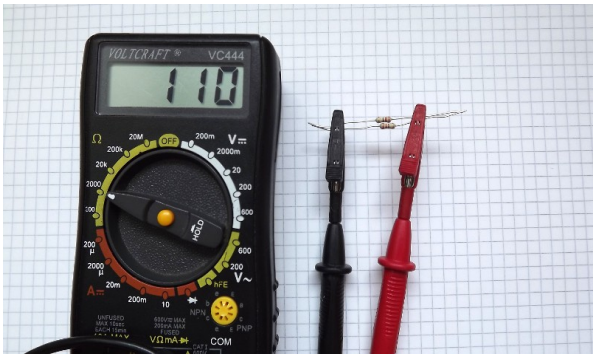
- Gesamt-Widerstand = Summe aller Teil-Widerstände

Zwei 220 Ohm Widerstände in Reihe haben also 440 Ohm. Drei 220 Ohm Widerstände in Reihe hätten 660 Ohm.

2.7.2.1 Parallelschaltung

Auch die Parallelschaltung ist geeignet, um auf Widerstandswerte zu kommen, die man einzeln nicht kaufen kann. Aber sie ist schwieriger zu berechnen.





Die Formel dazu lautet:

$$Rg = \frac{1}{\frac{1}{R1} + \frac{1}{R1} + \frac{1}{R3} \ldots}$$

Wobei R_g der Gesamtwiderstand ist, und R_1, R_2, R_3, … die Teilwiderstände sind. Zwei 220 Ohm Widerstände parallel ergeben nach dieser Formel 110 Ohm, also genau die Hälfte.

2.7.2.2 Leistung

Widerstände setzen elektrische Energie in Wärme um, die sie über ihre Oberfläche an die umgebende Luft abgeben. In Katalogen findest du stets eine Leistungsangabe in Watt, die besagt, welche Leistung der Widerstand bei Zimmertemperatur maximal verträgt.

Wenn du einem Widerstand zu viel Leistung zumutest, verschmort er oder brennt sogar ab.

Die Formel zur Berechnung der Leistung hatten wir schon gesehen:

- Leistung = Spannung · Stromstärke

Die meisten Widerstände in elektronischen Schaltungen sind bis zu ¼ Watt (also 0,25 Watt) oder ½ Watt (0,5 Watt) belastbar. Diese Widerstände sind die billigsten.

Für die Schaltung mit der Leuchtdiode rechnen wir nach:

Durch Messen mit dem Multimeter finden wir heraus, dass am Widerstand 2,53 Volt anliegen und dass dabei ein Strom von 11,5 Milliampere fließt.

- 2,53 V · 0,0115 A = 0,029 Watt

Das ist definitiv nicht zu viel für einen kleinen ¼ Watt Widerstand. Dein 220 Ohm Widerstand wird also ganz sicher nicht verschmoren.

2.7.3 Dioden

Nicht leuchtende Dioden verwendet man wegen ihrer Eigenschaft, den Strom nur in eine Richtung fließen zu lassen. Dioden gibt es in sehr unterschiedlichen Bauformen. Zwei Beispiele:

1N4001

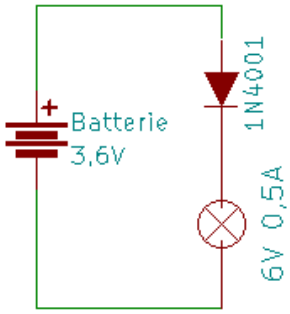
1N4148

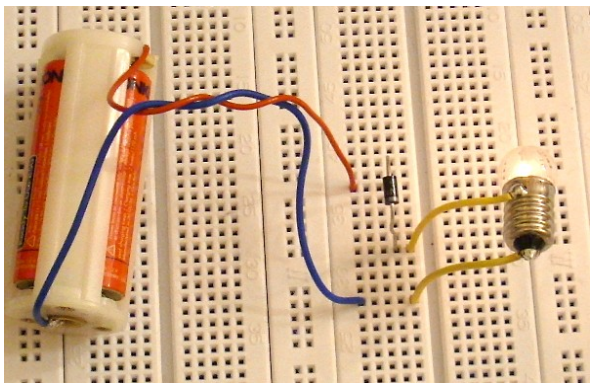
Anwendungsbeispiele:

- In Netzteilen wandelt man mit Dioden den Wechselstrom aus der Steckdose in Gleichstrom um.
- Man kann Dioden verwenden, um Geräte vor Verpolung (falsch herum angeschlossenen Batterien) zu schützen.
- Man verwendet Dioden oft als Schaltelement, um den Strom abhängig von seiner Flussrichtung über unterschiedliche Strecken fließen zu lassen.

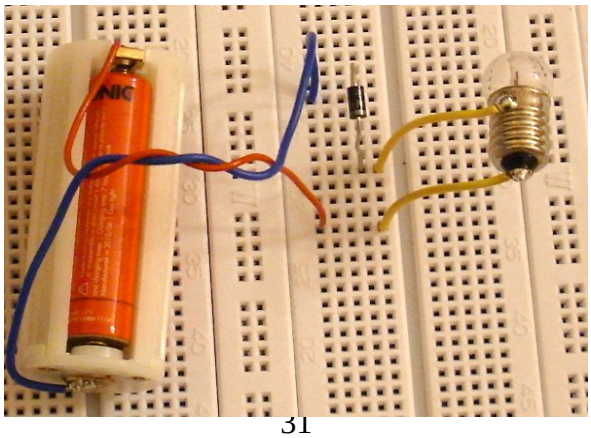
Der Plus-Pol heißt Anode und der Minus-Pol heißt Kathode. Die Kathode (-) ist durch einen farbigen Ring gekennzeichnet. Im Schaltzeichen deutet der Pfeil die technische Stromrichtung (von Plus nach Minus) an.

Mit dem folgenden Experiment kannst du eine Diode ausprobieren:





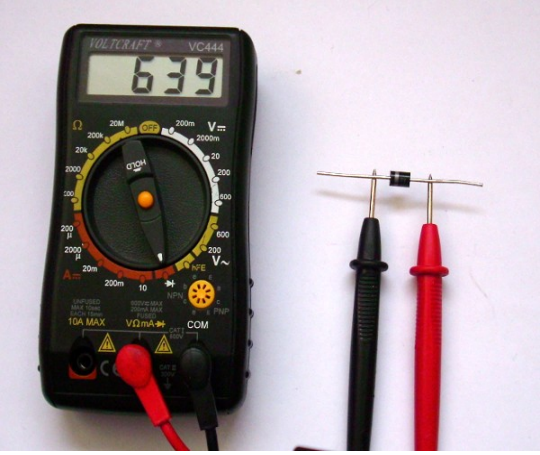
Wenn du die Batterie anders herum drehst, fließt kein Strom. Denn anders herum lässt die Diode keinen Strom fließen.



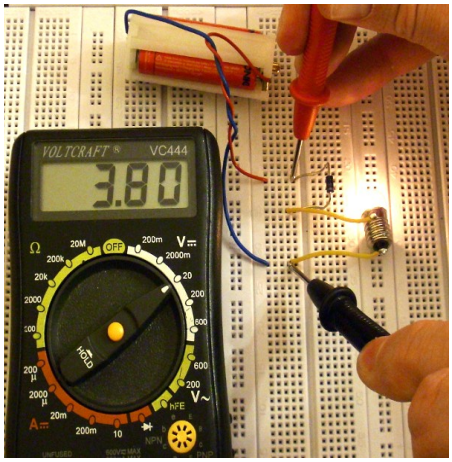
2.7.3.1 Schwellspannung

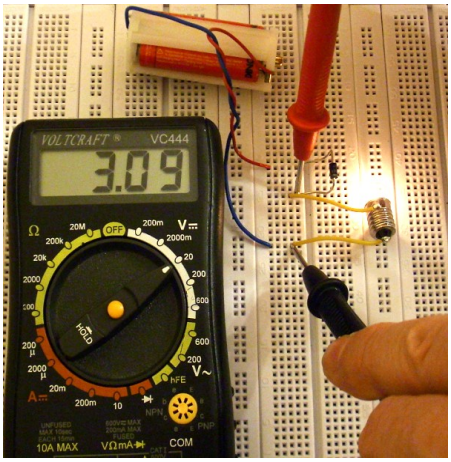
Dioden haben eine Schwellspannung, die so genannt wird, weil unterhalb dieser Spannung kein Strom fließt. Bei Silizium Dioden liegt die Schwellspannung bei ca. 0,7 Volt. Bei Shottky Dioden ist die Schwellspannung häufig aber nicht immer etwas geringer.

Mit dem Dioden-Test deines Digital-Multimeters kannst du diese Schwellspannung messen. Die Anzeige erfolgt in der Einheit Millivolt.



Wenn ein Strom durch die Diode fließt, geht an ihr ein bisschen Spannung verloren. Je mehr Strom fließt, umso höher ist die Verlustspannung, jedoch selten über 1 Volt.



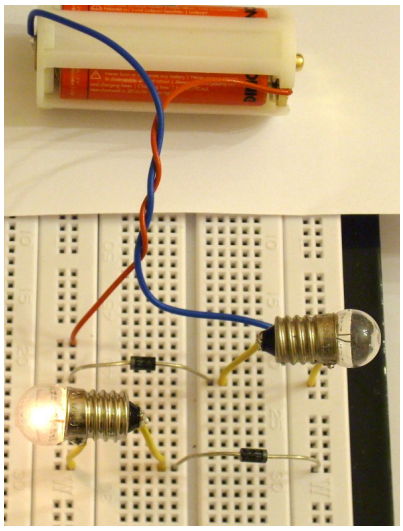


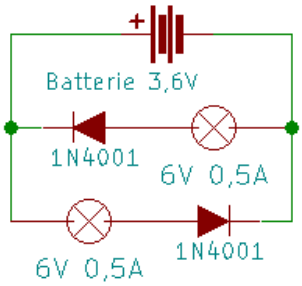
Meine Batterien liefern gerade 3,8 Volt. Davon bleiben hinter der Diode nur noch 3,09 Volt übrig. Also sind an der Diode 0,71 Volt verloren gegangen.

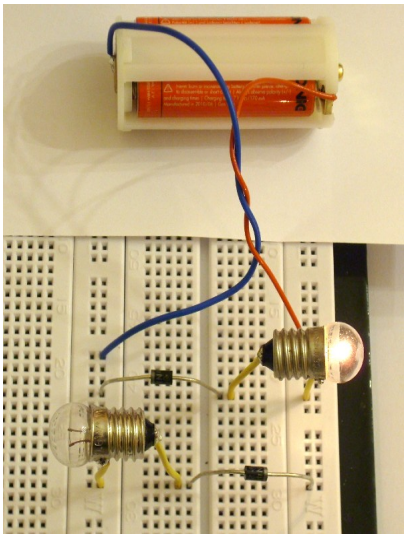
2.7.3.2 Diode als Schalt-Element

Dieses Experiment benutzt eine Diode als Schaltelement. Der Aufbau soll anzeigen, welcher Anschluss der Batterie der Plus-Pol ist.

.







Weil die Dioden unterschiedlich herum eingebaut sind, leuchtet immer nur eine Lampe. Welche das ist, hängt davon ab, wie Polung der Batterie ab. Wenn links der Plus-Pol ist leuchtet die linke Lampe. Wenn rechts der Plus-Pol ist leuchtet die rechte Lampe.

2.7.4 Leuchtdioden

Leuchtdioden (kurz LEDs) sind eine Sonderform der Dioden, weil sie leuchten. Es gibt Leuchtdioden in zahlreichen Formen und Farben.

Die gewöhnliche Leuchtdiode hat folgende technische Daten:

- Lichtfarbe: rot, gelb, grün, weiß, blau
- Durchmesser: 3 oder 5 mm
- Helligkeit: 400 bis 10.000 mcd
- Abstrahlwinkel: 20 bis 60 Grad
- Spannung: 1,6 bis 3,5 Volt
- Strom: max. 20 mA
- Leuchtdauer: über 50000 Stunden

Auf dem Foto befindet sich links die Anode (Plus-Pol) und rechts die Kathode (Minus-Pol). Auf Seite der Kathode ist das Gehäuse der Leuchtdiode abgeflacht und der Anschlussdraht ist kürzer. Du hast also drei Merkmale, die dir helfen, die Leuchtdiode richtig herum einzubauen.

Leuchtdioden geben im Gegensatz zu Glühlampen schon bei geringen Stromstärken unter 1 mA deutlich sichtbar Licht ab. Je höher der Strom ist, desto heller leuchten sie.

Bei normalen LEDs wird die Helligkeit in mcd (Milli-Candela) angegeben, das ist die Helligkeit, die man im Kern des Lichtkegels wahrnimmt. 1000 mcd soll ungefähr einer Kerzenflamme entsprechen. Bei stärkeren LEDs, die für Beleuchtungszwecke gedacht sind, wird die Helligkeit hingegen in lm (Lumen) angegeben. Das ist die Helligkeit, die insgesamt abgestrahlt wird, also unabhängig vom Betrachtungswinkel.

Das menschliche Auge reagiert logarithmisch auf Helligkeit. Dass heißt: Damit eine LED doppelt so hell aussieht, muss sie etwa 10 mal so viel mcd haben. Folglich ist der Unterschied zwischen z.B. 100 mcd und 200 mcd kaum wahrnehmbar. Leuchtdioden mit weniger als 100 mcd sind übrigens nicht mehr Zeitgemäß.

Beim Kauf sollte man aber nicht nur auf die Helligkeit achten, sondern auch auf den Abstrahlwinkel. Die meisten besonders hellen LEDs haben einen kleinen Abstrahlwinkel, so dass man sie schräg betrachtet nicht gut leuchten sehen kann. Für eine Taschenlampe mag das gerade richtig sein, für eine Anzeigetafel jedoch eher nicht.

Ich empfehle, mal einen Blick in die Shops von chinesischen Händlern zu werfen (Ebay, AliExpress). Dort bekommt man zu einem erstaunlich günstigen Preis größere Mengen von schön hellen Leuchtdioden.

2.7.4.1 Vorwiderstand

Wenn ein Widerstand dazu verwendet wird, den Strom durch ein einzelnes Bauteils zu regeln, nennt man ihn Vorwiderstand.

Leuchtdioden muss man IMMER mit Vorwiderstand benutzen, so wie du es bereits getan hast. Wenn du diese Regel missachtest, geht die Leuchtdiode nach kurzer Zeit kaputt (schon ausprobiert?).

Im Gegensatz zu Glühlampen werden Leuchtdioden nämlich nicht mit einer bestimmten Spannung betrieben, sondern mit einer bestimmten Stromstärke (normalerweise 5-20 mA). Die Spannung ergibt sich dann von selbst.

Der Widerstand hat die Aufgabe, die Stromstärke zu regulieren. Ohne Widerstand fließt viel zu viel Strom, was die Lebensdauer der LED drastisch reduziert.

Die Batteriespannung muss in jedem Fall höher sein, als die Betriebsspannung der Leuchtdiode, sonst fließt überhaupt kein Strom und sie leuchtet auch nicht.

Wir berechnen den Widerstand folgendermaßen: Von der Batteriespannung subtrahieren wir die Betriebsspannung der Leuchtdiode. Die übrige Spannung fällt am Widerstand ab. Wir teilen sie durch den gewünschten Strom und erhalten den erforderlichen Widerstandswert:

Rechenbeispiel für 3 Akkus mit 1,2 Volt, sowie eine LED mit 2V bei 10 mA:

3,6 V – 2 V = 1,6 V

1,6 V : 0,01 A = 160 Ohm

Bei alkalischen Batterien mit 3 mal 1,5 Volt wäre ein größerer Widerstand besser:

4,5 V – 2V = 2,5 V

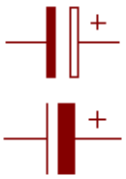
2,5 V : 0,01 A = 250 Ohm

Du wirst im Handel weder 160 Ohm noch 250 Ohm bekommen können – außer vielleicht zu einem besonders hohen Preis. Ich habe mich daher für den verfügbaren Wert 220 Ohm entschieden. Der eignet sich für beide Batterie-Typen, ohne dass wir die maximal zulässigen 20 mA überschreiten.

2.7.5 Kondensatoren

Kondensatoren Speichern elektrische Energie, ähnlich wie Akkus. Es gibt sie in sehr unterschiedlichen Bauformen und mit sehr unterschiedlichem Speichervermögen. Die Experimente dieses Buches verwenden die abgebildeten Bauformen.

Keramik-Kondensator



Elektroly-Kondensator (Elko)

In Katalogen ist immer die Speicherkapazität in der Einheit Farad angegeben, sowie die maximal zulässige Spannung in Volt.

Ein Farad bedeutet, dass die Spannung im Kondensator innerhalb einer Sekunde um 1 V steigt, wenn man ihn mit 1 A auflädt. Und umgekehrt sinkt seine Spannung um 1 V pro Sekunde, wenn die Stromstärke 1 A beträgt. Bei 2 A würde es nur eine halbe Sekunde dauern und bei 10 A nur 0,1 Sekunden.

Während man Akkus als Langzeit-Speicher für mehrere Stunden bis Monate einsetzt, verwendet man Kondensatoren für viel kürzere Speicher-Zeiten. Dafür kann man sie im Gegensatz zu Akkus jedoch beinahe beliebig oft auf- und entladen.

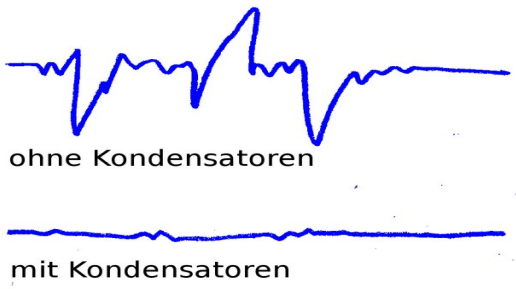
Kondensatoren mit weniger als 1 µF Kapazität haben meistens keine bestimmte Polarität. Bei den größeren Kondensatoren muss man jedoch auf richtigen Anschluss von Plus und Minus achten. In der Regel ist der Minus-Pol durch einen breiten aufgedruckten Balken gekennzeichnet.

Elektrolyt-Kondensatoren enthalten eine chemische Flüssigkeit, die vor allem bei Hitze irgendwann austrocknet. Sie verlieren dann Speicherkapazität. Aber immerhin halten sie bei guter Behandlung typischerweise mehr als 10 Jahre.

Kondensatoren sind sehr vielseitig einsetzbar, zum Beispiel:

- Als Alternative zu Akkus, um kurzzeitig Strom zu speichern, zum Beispiel im Fahrrad-Rücklicht.
- Man kann damit Zeitgeber bauen, zum Beispiel für das Licht im Treppenhaus.
- Als Puffer, um die Stromversorgung bei Last-Spitzen zu stabilisieren.

Die letzte genannte Funktion ist in elektronischen Schaltungen mit Mikrochips sehr sehr wichtig. Störungen in der Stromversorgung lösen Fehlfunktionen aus. Darum sind Computer mit solchen Kondensatoren regelrecht voll gestopft.



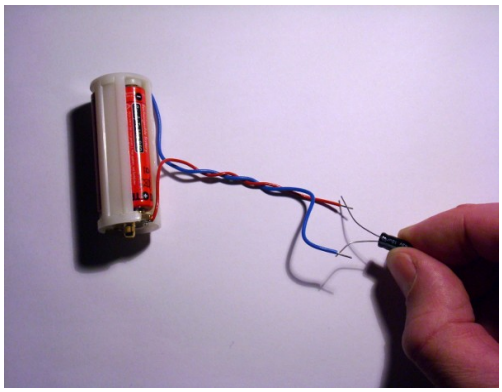
Störungen in der Stromversorgung werden in 99 % der Fälle durch simple Schaltvorgänge ausgelöst. Du hast sicher schon einmal beobachtet, wie manchmal alle Lampen in der Wohnung flackern, wenn man den Staubsauger an macht oder den Wasser-Kocher abschaltet.

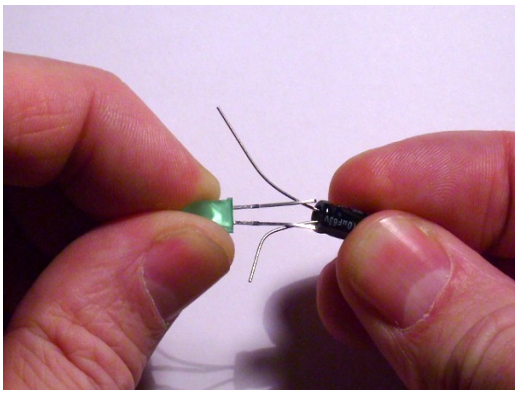
Schaltvorgänge kommen in gewöhnlichen Computern viele Millionen mal pro Sekunde vor. Ohne stabilisierende Kondensatoren, könnten diese Computer nicht einmal eine Sekunde lang fehlerfrei funktionieren.

Mit Kondensatoren stabilisieren wir die Stromversorgung, so wie ein Wasserturm die Wasserversorgung zu Stoßzeiten stabilisiert. Dazu eignen sich kleine Keramik-Kondensatoren besser als Elkos, weil sie weniger träge reagieren.

Ein Experiment:

Halte einen Kondensator mit einer Kapazität von 10 Mikro-Farad (10 µF) richtig gepolt an den Batteriekasten, damit er sich auflädt. Halte ihn danach an eine Leuchtdiode, ohne seine Anschlussbeinchen zu berühren. Die Leuchtdiode blitzt dann ganz kurz. Der Kondensator hat die in ihm gespeicherte Energie an die Leuchtdiode abgegeben.



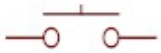


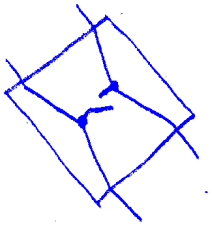
Du hast gesehen, dass der kleine Kondensator im Vergleich zu einer gleich großen Batterie nur sehr wenig Energie speichern kann – gerade genug um die LED blitzen zu lassen. Dafür ist der Kondensator aber viel länger haltbar. Ein normaler Akku ist spätestens nach 1000 Ladezyklen völlig abgenutzt, während du den Kondensator beliebig oft laden kannst.

2.7.6 Kurzhub-Taster

Du wirst einen Taster verwenden, um den Mikrocontroller manuell zurück zu setzen. Er startet dann neu durch, so als hättest du die Stromversorgung kurz aus geschaltet.

Taster schließen einen Stromkreis, wenn man sie drückt (Strom an). Wenn man sie los lässt, unterbrechen sie den Stromkreis wieder (Strom aus). Taster gibt es in unzähligen Bauformen. Wir verwenden Kurzhub-Taster, weil sie billig und gut sind.





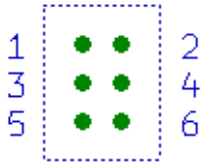
Kurzhub-Taster findest du in beinahe allen Geräten der Unterhaltungselektronik, vor allem hinter den Bedienfeldern. Diese Kurzhubtaster haben vier Anschlüsse, damit man sie gut befestigen kann. In der rechten Zeichnung siehst du, wie die Anschlüsse im Innern des Tasters miteinander verbunden sind.

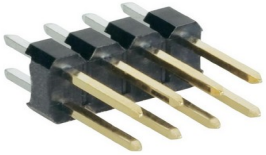
Manche Taster haben noch einen fünften Anschluss, der keine elektrische Funktion hat. Im obigen Foto habe ich einen roten Pfeil daran gemalt. Schneide diesen Anschluss ab, falls vorhanden. Er stört uns nur, weil er nicht in die Löcher der Platine passt.

Bevor du einen Kurzhub-Taster in dein Steckbrett stecken kannst, musst du die gewellten Anschlüsse mit einer Zange glatt drücken.

2.7.7 Stiftleiste

Die Stiftleiste wird als Anschluss für den ISP-Programmieradapter dienen. Über diesen Anschluss überträgst du später das Programm vom Computer in den Mikrocontroller-Chip.





Du kannst zu lange Stiftleisten mit einem scharfen Messer auf die gewünschte Länge kürzen. Du brauchst ein kleines Stück mit zwei Reihen zu je 3 Stiften. Den Rest hebst du für später auf.

Die Nummerierung der Stifte gilt für die Blickrichtung: von oben auf die Bauteilseite der Platine geschaut.

2.7.8 Integrierte Schaltkreise

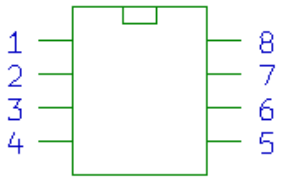
Integrierte Schaltkreise (kurz: Mikrochips oder IC) sind kompakte Bauteile, in denen sich hunderte bis tausende von mikroskopisch kleinen elektronischen Bauteilen befinden. Wenn du einen Mikrochip benutzt, musst du wissen, welche Funktionen er hat. Auf der Webseite des Chip-Herstellers kannst du dazu Datenblätter und ggf. weiterführende Dokumentationen finden.

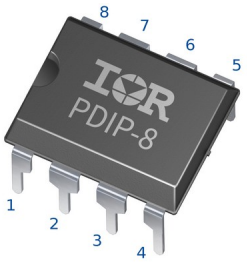
Wie die Funktionen im Innern des Mikrochips realisiert wurden, bleibt das Geheimnis des Herstellers. Dokumentiert wird nur, was herein geht und heraus kommt.

Es gibt Mikrochips für ganz unterschiedliche Aufgaben. Zum Beispiel:

- Mikroprozessor / Mikrocontroller zum Rechnen und Steuern
- Verstärker, um analoge Signale aufzubereiten
- Treiber, um größere Lasten anzusteuern, z. B. Motoren
- Spannungsregler sorgen in Netzteilen für eine konstante Ausgangsspannung
- Sensoren für alles Mögliche, z. B. Temperatur, Gas, Bewegung
- Logik-Gatter, zum realisieren logischer Verknüpfungen ohne Mikrocontroller (wie „und“, „oder“ und „nicht“, auch Zähler)
- Speicher, z. B. als Arbeitsspeicher in deinem Computer oder als Speicherkarte im Fotoapparat.

Wir verwenden Mikrochips in der Gehäuseform „PDIP“, auch „DIL“ oder „DIP“ genannt. Das folgende Foto zeigt einen kleinen Mikrochip in diesem Gehäuse. Es gibt auch größere PDIP Gehäuse, mit bis zu 40 Pins.





Die Anschluss-Pins von Mikrochips werden grundsätzlich gegen den Uhrzeigersinn (links herum) nummeriert. Man beginnt dabei immer links oben, wobei „oben“ durch eine Kerbe, ein Loch oder einen aufgedruckten Punkt gekennzeichnet ist. Manchmal befindet sich die Markierung auch genau in der Ecke, wo sich Pin 1 befindet.

Mikrochips darf man nicht falsch herum einbauen, sonst entsteht ein Kurzschluss, der den Chip sofort zerstört! Ebenso schädlich ist es, die Stromversorgung falsch herum anzuschließen.

2.7.9 Übungsfragen

Hast du gut aufgepasst? Dann beantworte für die folgenden Fragen. Die Lösungen findest du am Ende des Buches.

1. Wovon hängt die Stromstärke ab?

a) Von der Dicke des Kabels

b) von der Anzahl der Elektronen im Kabel

c) von der Anzahl der bewegten Elektronen im Kabel

2. Warum überwindet ein Blitz die eigentlich nicht leitfähige Luft?

a) Weil er eine hohe Spannung hat

b) Weil er einen hohe Stromstärke hat

3. Wenn jemand Strom „verbraucht“, verbraucht er dann die Elektronen?

a) Ja, volle Batterien enthalten viele Elektronen, leere Batterien enthalten keine Elektronen.

b) Nein, Elektronen sind immer da. Es kommt darauf an, ob sie sich bewegen.

4. Würde eine Leuchtdiode an einem einzelnen Akku funktionieren?

a) Sicher, solange der Akku nicht leer ist

b) Nein, ein einzelner Akku hat zu wenig Spannung

c) Nein, dazu wird noch ein Widerstand gebraucht, dann geht es

5. Warum verheizt man Strom in Widerständen?

a) Weil warme Geräte besser funktionieren

b) Weil der Strom dann besser fließt. Je mehr Widerstände, um so besser.

c) Weil sonst zu viel Strom fließen würde. Die Wärme ist ein notwendiges Übel.

6. Welchen Wert muss ein Widerstand haben, um eine LED im Auto an 12 Volt mit ca. 10 mA zu betreiben?

a) ungefähr 1000 Ohm

b) ungefähr 1200 Ohm

c) ungefähr 200 Ohm

7. Wie viel Strom fließt durch einen 1000 Ohm Widerstand an 9 Volt?

a) 9000 mA

b) 9 mA

c) 111,11 mA

8. Wie viel Energie speichert ein 10 µF Kondensator?

a) Es reicht gerade mal aus, um eine LED blitzen zu lassen

b) Es reicht aus, um eine Glühlampe kurz leuchten zu lassen

c) Damit könnte man einen MP3 Player stundenlang laufen lassen

9. Wenn ein Akku 2000 mAh Kapazität hat, wie lange kann man damit ein Gerät betreiben, das 1000 mA benötigt?

a) Eine halbe Stunde

b) Zwei Stunden

c) Vier Stunden

10. Wie heißt bei Mikrochips die Gehäuseform, die wir (für den Einsatz auf Steckbrett und Lochraster-Platine) verwenden werden?

a) PDIP

b) SMD
c) SOP
d) DIL
e) QFP
f) BGA

3 Der erste Mikrocomputer

3.1 Mikrocontroller

Kernstück deines Tisch-Computers ist vermutlich ein Mikroprozessor von Intel oder AMD. Wenn es noch kleiner sein soll, arbeitet man mit Mikrocontrollern. Schon ihr Name deutet an, dass Mikrocontroller weniger zum Rechnen gedacht sind, sondern eher, um Dinge zu Steuern. Zum Beispiel einen CD Player oder eine Waschmaschine.

In diesem Buch arbeiten wir mit Mikrocontrollern von der Firma Atmel, und zwar aus der AVR Serie. Im Allgemeinen spricht man von „AVR Mikrocontrollern“. Die ersten Experimente werden wir mit ganz kleinen Mikrocontrollern vom Typ ATtiny13-PU oder dem leicht verbesserten ATtiny13A-PU durchführen. Er hat nur 8 Pins und einen überschaubaren Funktionsumfang. Falls du den ATtiny13 nicht bekommen kannst, nimm alternativ einen ATtiny25, 45 oder 85.

Die größeren AVR Mikrocontroller haben mehr Speicher und mehr Anschlüsse, funktionieren jedoch nach dem gleichen Prinzip. Deswegen kannst du alles, was du mit dem ATtiny13 lernst, auch für die größeren Typen gebrauchen.

3.2 ISP-Programmieradapter

Programme für Mikrocomputer schreibt man auf gewöhnlichen Computern. Irgendwie musst du das fertige Programm von deinem Computer in den Mikrocontroller-Chip übertragen. Dazu brauchst du einen ISP-Programmer (auch Programmieradapter genannt).

Achte beim Kauf darauf, dass er das ISP Protokoll unterstützt und dass er die Spannungen seiner Signale automatisch an die Versorgungsspannung des Mikrocontrollers (oft „Target“ genannt) anpasst. Der unterstützte Spannungsbereich soll mindestens 2,7 bis 5 V sein. Das Anschlusskabel sollte möglichst 6 Polig sein, sonst brauchst du noch einen Adapter.

3.2.1 Diamex ISP Stick

Ein preisgünstiges Modell, das diese Anforderungen voll erfüllt, ist der „Diamex ISP Programmer Stick“:

Er kann auf Wunsch die Zielschaltung mit 3,3 V oder 5 V Spannung versorgen. Das ist für die ersten Experimente sehr praktisch und wir werden diese Funktion nutzen.

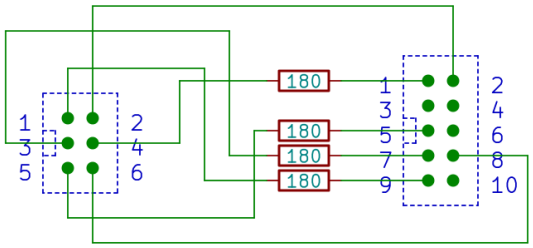
3.2.2 USBASP

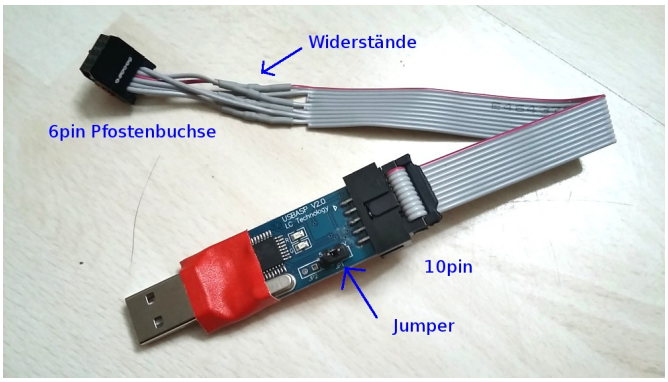
Wer es ganz billig haben will, nimmt einen Programmieradapter vom Typ USBASP. Davon gibt es viele Varianten. Die Bedienungsanleitung eines aktuellen Modells findest du hier:

http://eecs.oregonstate.edu/education/docs/ece375/USBASP-UG.pdf

Viele Modelle haben einen Jumper oder Schalter, der die Zielschaltung wahlweise mit 3,3 V oder 5 V versorgt. Für Zielschaltungen mit eigener Versorgung zieht man den Jumper ab. Modelle mit Schalter stellt man hingegen auf 3,3 V ein, wenn der Mikrocontroller eine eigene (eventuell auch höhere) Spannungsversorgung hat.

Allerdings haben die Signal-Leitungen unabhängig von diesem Jumper oft trotzdem 5 V Pegel, und das ist schlecht. Ich halte es für eine Fehlkonstruktion. Wenn du so einen USBASP Programmieradapter trotzdem verwenden möchtest, benötigst du ein Adapterkabel mit 180 Ohm Widerständen in den Signal-Leitungen. Die Widerstände beschützen deine Mikrocontroller davor, wegen Überspannung kaputt zu gehen.





Um den USB Port des Computers vor Kurzschlüssen zu beschützen, empfiehlt es sich, die Bauteile in der Nähe des USB Steckers mit Gewebeband abzudecken.

Für die Einstellung der Übertragungsgeschwindigkeit sind drei unterschiedliche Varianten im Umlauf:

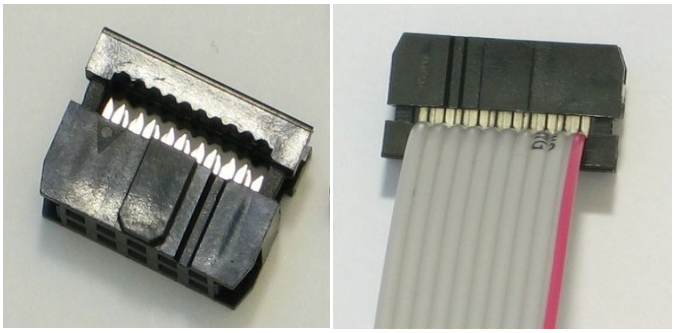
a) Die meisten USBASP Modelle aus China passen sich automatisch an die Taktfrequenz des Mikrocontrollers an. Sie haben häufig Lötpunkte für einen Jumper „JP3“, der jedoch nicht bestückt ist und auch nicht benötigt wird.

b) Die ursprüngliche Version der USBASP Programmieradapter hatte einen Jumper „JP3“, den man aufstecken musste, wenn der Mikrocontroller mit weniger als 8Mhz getaktet wird. Alle Experimente in diesem Buch sind davon betroffen, also musst du den Jumper „JP3“ schließen, falls vorhanden.

c) Weniger verbreitet sind Modelle, bei denen man die Taktfrequenz mit der Bedien-Software einstellen kann. Wie das geht, erkläre ich später im Kapitel Bedienung ISP-Programmer.

3.2.3 Umgang mit Pfostenbuchsen

Pfostenbuchsen dienen dazu, Flachkabel mit Platinen zu verbinden. Die Montage der Pfostenbuchsen geht so:



Quelle: wiki.lochraster.org

In dem Spalt, wo das Kabel eingelegt werden soll, befinden sich winzige Messer, die sich in die Isolation des Kabels herein drücken und so den Kontakt herstellen. Diese Schneid-Klemmen kann man nur einmal verwenden! Wenn du eine bereits zusammengedrückte Buchse wieder öffnest, gehen die Kontakte kaputt.

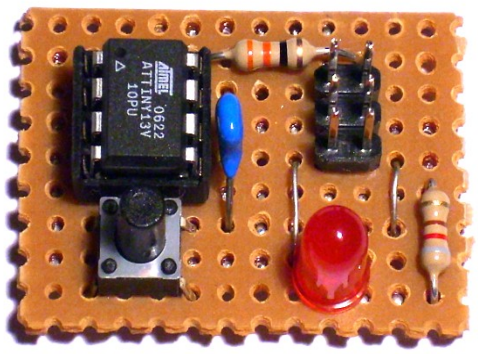
Die rote Seite des Flachkabels gehört auf die markierte Seite der Buchse - sie ist mit einem kleinen Pfeil gekennzeichnet. Lege das Kabel so tief in den Spalt ein, so dass es hinten noch einen Zentimeter über steht. Dann drückst du die Buchse mit einem Schraubstock langsam zusammen, bis sie mit einem hörbaren „Klick“ einrastet.

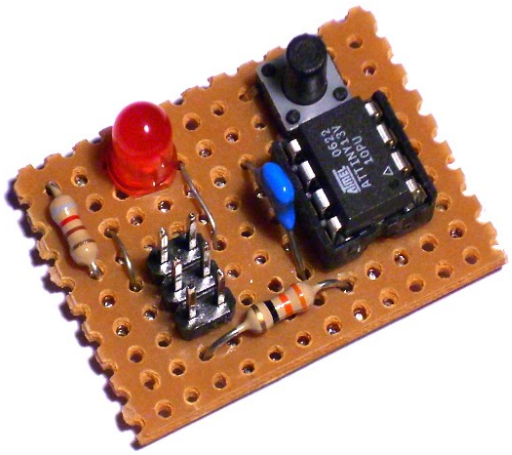
Anschließend schneidest du das überstehende Ende des Kabels mit einem scharfen Messer ab. Stelle sicher, dass dort keine losen Drahtenden heraus hängen.

3.3 Platine Löten

Den ersten Mikrocomputer löten wir direkt auf eine Lochraster-Platine. So kannst du ihn lange als Andenken aufbewahren. Außerdem eignet sich die Platine künftig als Programmierfassung.

So wird dein erster Mikrocomputer aussehen:





Er ist nur halb so groß, wie eine Streichholzschachtel. Also wirklich richtig „Mikro“. Wir werden später noch einen Lautsprecher hinzufügen, um das Teil interessanter zu machen.

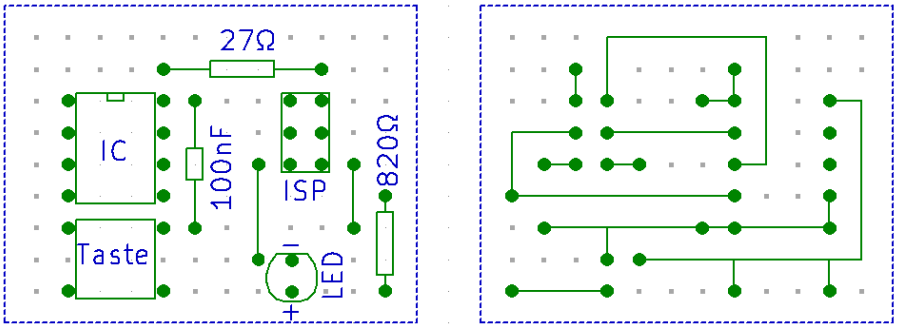
An die 6-polige Stiftleiste wirst du später (jetzt noch nicht) den ISP Programmieradapter stecken.

Schneide von der Lochraster-Platine ein Stück mit genau 9x12 Löchern ab. Du kannst die Platine mit einem Bastelmesser entlang einer Loch-Reihe an ritzen und dann durchbrechen. Hebe die restlichen Stücke für später in einer geschlossenen Plastiktüte auf, damit sie nicht Oxidieren.

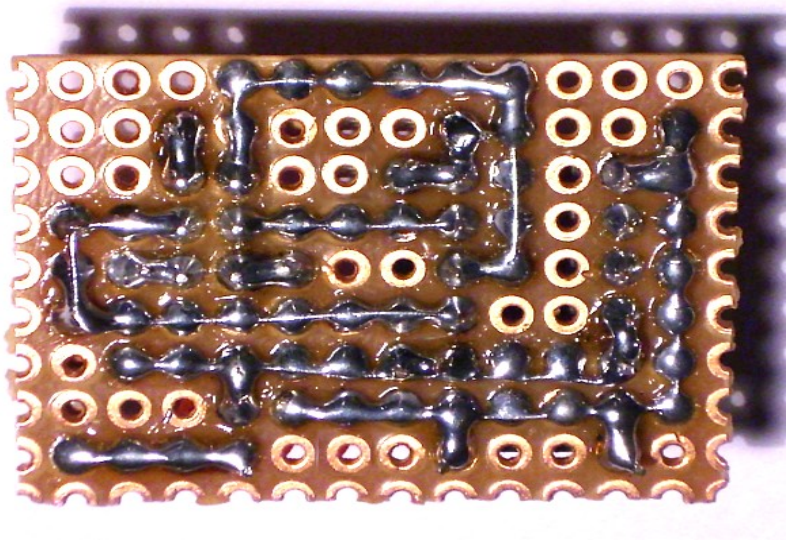
Lege folgende Bauteile bereit:

- Einen 8-poligen IC-Sockel, da kommt später der Mikrocontroller rein
- einen Kurzhub-Taster
- einen Keramik-Kondensator 100 nF
- eine rote Leuchtdiode
- einen 27 Ω Widerstand (rot-violett-schwarz)
- einen 820 Ω Widerstand (grau-rot-braun oder grau-rot-schwarz-schwarz)
- eine Stiftleiste mit 2x3 Stiften

Stecke die Bauteile nach und nach exakt nach folgendem Plan in die Platine und löte sie auf der Rückseite mit kleinen Zinn-Punkten fest. Die überstehenden Drahtenden schneidest du mit dem Elektronik-Seitenschneider ab. Du musst auch zwei Drahtstücke einlöten, die man „Drahtbrücken“ nennt. Bei der LED gehört die abgeflachte Seite mit dem kürzeren Draht nach oben.



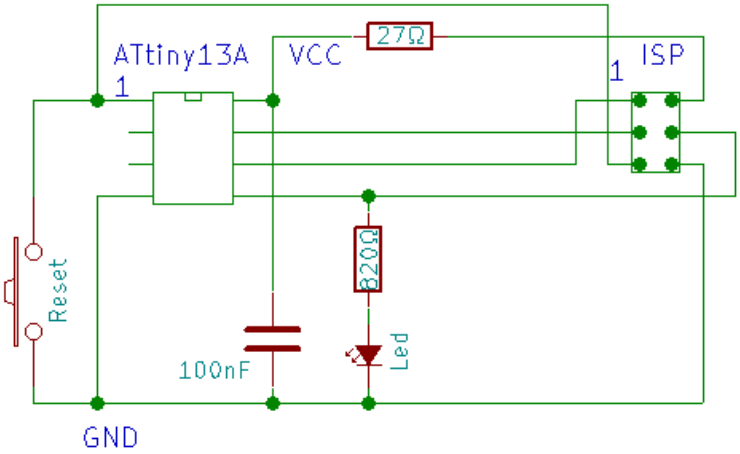
Erst wenn alle Teile richtig herum eingebaut sind, beginne damit, die Bauteile mit versilbertem oder verzinntem Draht auf der Rückseite der Platine zu verbinden.



Stecke den Mikrocontroller noch nicht in den Sockel und schließe die Platine noch nicht an den Computer an! Wir müssen erst prüfen, ob sie sicher ist, damit der ISP-Programmer und der USB Port nicht kaputt gehen.

3.4 Schaltplan

Zu jeder elektronischen Schaltung gehört ein Schaltplan. Der Schaltplan gibt darüber Auskunft, wie die Bauteile miteinander verbunden sind. Der Schaltplan von deinen ersten Mikrocomputer sieht so aus:

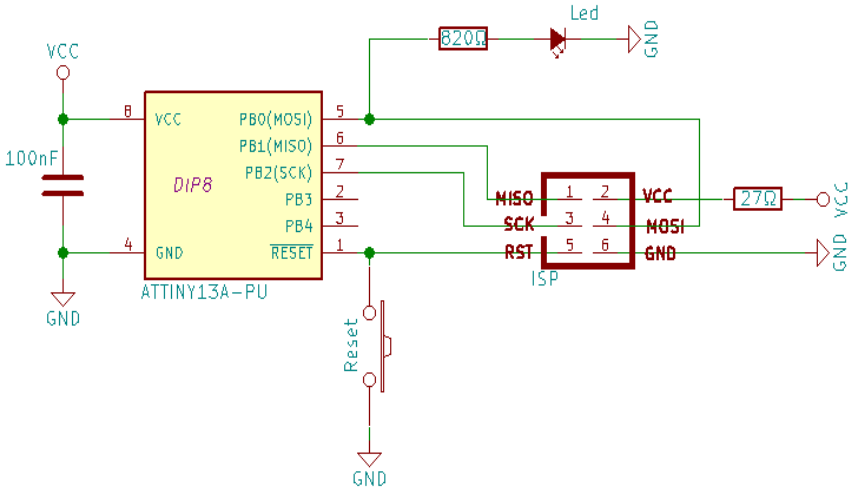


Vergleiche ihn mit deiner Platine. Du solltest jede einzelne Leitung nachvollziehen können.

Zwei ganz wichtige Leitungen habe ich blau beschriftet:

- VCC oder VDD ist die Stromversorgung der Schaltung. Wir versorgen sie mit 3,3 V oder 5 V aus dem Programmieradapter. Falls dein Programmieradapter keine Versorgungsspannung bereit stellt, kannst du auch 3 Akkus oder Batterien verwenden.
- GND (=Ground) ist der andere Anschluss der Stromversorgung. Diese Leitung hat per Definition immer 0 Volt. Normalerweise beziehen sich alle Spannungsangaben auf GND und alle Mikrochips müssen über GND miteinander verbunden sein. Häufig ist GND wortwörtlich mit der Erde verbunden.

Komplexere Schaltpläne zeichnet man allerdings in einem anderen Stil, damit sie übersichtlicher sind und man besser erkennen kann, welche Signale die Leitungen führen:

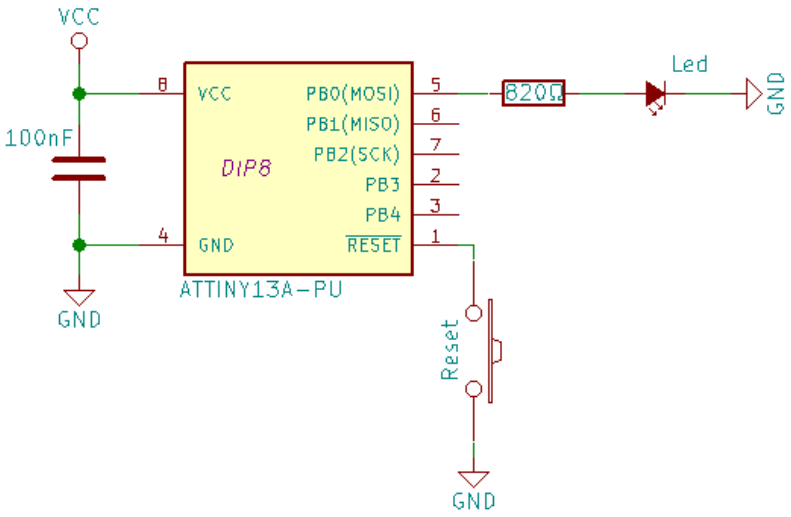


Die Leitungen der Stromversorgung wurden durch „VCC“ und „GND“ Symbole ersetzt, und beim Mikrochip wurden die Anschlüsse so angeordnet und beschriftet, dass ihre Funktion besser erkennbar ist. Mein Zeichenprogramm gibt Dreiecke für „GND“ vor. Manche andere Programme nutzen hingegen ein fettes „T“ das Kopfüber steht.

Der 27 Ω Widerstand dient übrigens als Schutz vor einem Kurzschluss durch falsch herum eingesetzten Mikrochip, indem er die Stromstärke begrenzt. Die Sache mit dem Widerstand funktioniert allerdings leider nur in ganz kleinen Schaltungen mit sehr wenig Stromverbrauch. Bei größeren Schaltungen würde am Widerstand zu viel Spannung verloren gehen.

3.5 Funktion der Schaltung

Den ISP Stecker lassen wir mal weg, weil er für die Funktion der Schaltung nicht relevant ist.



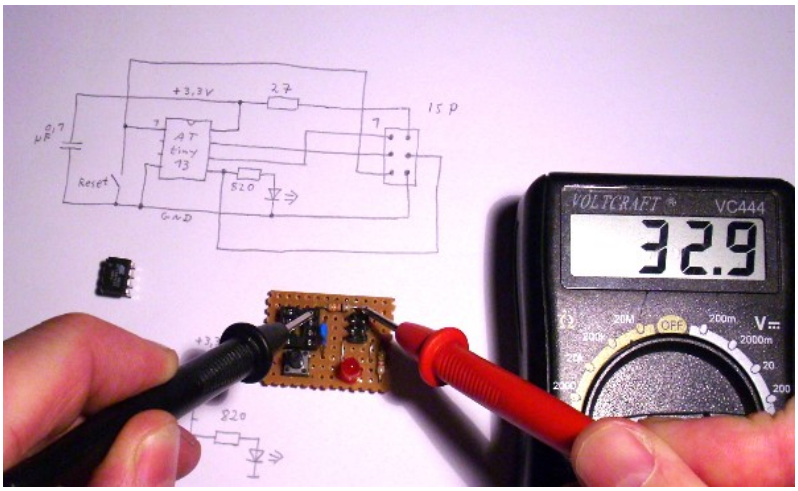
Der kleine 100 nF Keramik-Kondensator stabilisiert die Stromversorgung. Der Reset-Taster „zieht“ den Reset-Eingang des Mikrocontrollers auf Null Volt, und löst so einen Neustart des Mikrocontroller aus.

Pin 5 nutzen wir als Ausgang um eine Leuchtdiode anzusteuern. Der Widerstand vor der Leuchtdiode reguliert hier die Stromstärke. Pin 4 und 8 vom Mikrocontroller dienen der Stromversorgung. Du besitzt im Idealfall einen ISP Programmieradapter, der die schnuckelige Platine mit 3,3 V oder 5 V versorgen kann. Falls nicht, löte an Pin 4 und 8 den Batteriekasten an, in den du später (jetzt noch nicht!) Akkus oder Einwegbatterien einlegst.

3.6 Funktionskontrolle

Schau noch einmal genau hin, ob die Lötstellen alle sauber sind. Benachbarte Leitungen dürfen sich nicht berühren. Es dürfen keine falschen Zinn-Kleckse auf der Platine sein. Wenn du einen Fehler findest, benutze die Entlötpumpe. Wenn du die Platine versaut hast, fange nochmal von vorne an. Möglicherweise brauchst du zwei oder drei Anläufe, bis dir das Löten gelingt. Löten ist nicht einfach.

Die Platine muss jetzt noch Stromlos sein. Also nicht an den Programmieradapter anschließen und noch keine Batterien einlegen! Prüfe zuerst mit dem Multimeter nach, ob die „lebenswichtigen“ Leitungen korrekt verbunden sind.

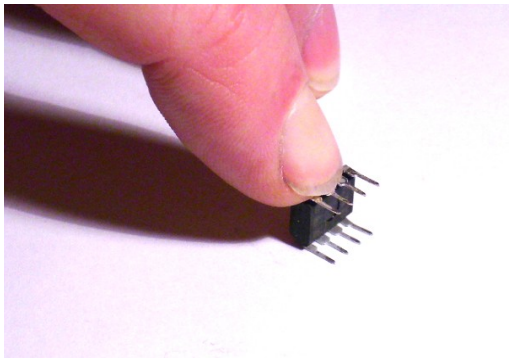


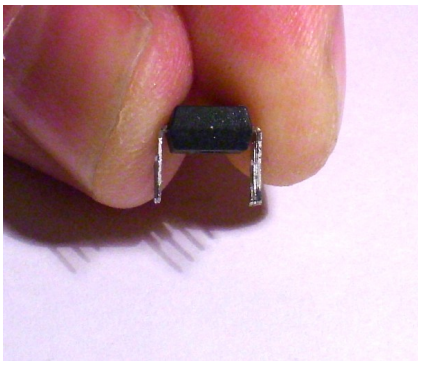
Stelle dein Digital-Multimeter auf dem 200 Ω Bereich ein und kontrolliere damit die folgenden Punkte:

1. Vom ISP-Stecker Pin 2 zum IC-Sockel Pin 8 muss dass Messgerät ungefähr 27 Ohm anzeigen (der Schutzwiderstand).
2. Vom ISP-Stecker Pin 6 zum IC-Sockel Pin 4 muss das Messgerät ungefähr 0 Ohm anzeigen (direkte Verbindung).
3. Vom IC-Sockel Pin 1 zum IC-Sockel Pin 4 muss das Messgerät zunächst nichts anzeigen. Drücke nun den Reset-Taster, dann muss das Messgerät ungefähr 0 Ohm anzeigen (der Taster verbindet diese beiden Leitungen).
4. Vom IC-Sockel Pin 4 zum IC-Sockel Pin 8 muss das Messgerät nichts anzeigen. Wenn es doch einen Widerstandswert anzeigt, hast du einen Kurzschluss in der Schaltung.
5. Vom IC-Sockel Pin 8 zu sämtlichen Pins vom ISP Stecker (außer Pin 2) darf das Messgerät nichts anzeigen, sonst hast du einen Kurzschluss in der Schaltung.

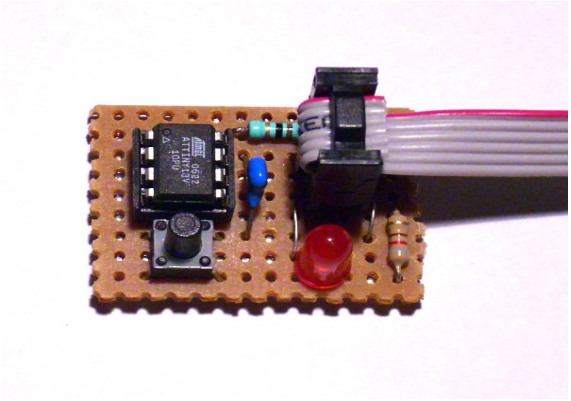
Korrigiere die Platine gegebenenfalls.

Nachdem nun die Verbindungen der Platine erfolgreich überprüft wurden, bist du bereit, den Mikrocontroller in den Sockel zu stecken. Aber er passt nicht so recht hinein, seine Beinchen stehen zu breit auseinander. Das ist normal, alle Mikrochips werden so geliefert. Am Besten biegst du sie vorsichtig gerade, indem du den Chip gegen die Tischplatte drückst:





Die Beinchen von Mikrochips sind sehr brüchig. Einmal um 20° hin und her gebogen brechen sie schon ab. Sei daher vorsichtig und langsam. Jetzt sollte der Mikrochip in den IC-Sockel passen. <u>Pin 1 gehört nach oben links</u>, wenn die Platine so liegt, wie das folgende Foto zeigt.



Wenn du ihn später wieder heraus nehmen musst, schiebe die Spitze eines kleinen Schlitz-Schraubendrehers zwischen Chip und Sockel. Mit dem Schraubendreher kannst du den Chip vorsichtig aushebeln ohne seine Beinchen zu verbiegen.

Stelle nun deinen ISP Programmieradapter auf 3,3 V oder 5 V ein. Falls die Stromversorgung hingegen aus Batterien kommt, lege jetzt die Batterien in den Batteriekasten. Dann schließt du den ISP-Programmer richtig herum an.

Beim Programmieradapter „Atmel AVR ISP mkII“ müsste die Kontrollleuchte nun von rot auf grün wechseln. Alle anderen Programmieradapter haben meines Wissens nach keine solche Kontrollleuchte.

Tatsächlich ist der AVR Mikrocontroller schon in Betrieb und durch den ISP-Programmer ansprechbar. Damit die Leuchtdiode auf der Platine an geht, muss er noch programmiert werden. Das Programm sagt dem Mikrocontroller, was er tun soll.

3.7 Programmier-Software

Das erste Programm soll die Leuchtdiode einfach nur einschalten. Das Programm ist simpel, dabei wirst du den Umgang mit der nötigen PC-Software lernen.

3.7.1 Programmiersprache

Wir programmieren AVR Mikrocontroller in der Programmiersprache „C“. Die Programmiersprache definiert Fachwörter und Sonderzeichen, aus denen dein Programm zusammengesetzt wird. Zum Beispiel:

```
int alter=14;
printf(„Ich bin %i Jahre alt“,alter);
```

Hier gibt der Computer die Meldung „Ich bin 14 Jahre alt“ auf dem Bildschirm aus. Der geschriebene Text eines Computer-Programms heißt „Quelltext“, auf englisch: „Source“ oder „Source-Code“.

3.7.2 Compiler

Der Quelltext wird von dem sogenannten Compiler in eine Folge numerischer Anweisungen übersetzt, den man „Byte-Code“ nennt. Der Mikrocontroller kann diesen Byte-Code lesen und abarbeiten. Der Byte-Code für das obige Programm sieht ungefähr so aus:

```
09C021C020C01FC01EC01DC01CC01BC0151AC019C011241FBECFE9CDBF10E0A0E661B0E0E4E
1F6E002C005900D92A637B1071A …
```

Mikroprozessoren führen nicht den Quelltext aus, sondern den daraus erzeugten Byte-Code. Dieses Verfahren gilt für Großrechner genau so wie für Notebooks und auch für die ganz kleinen Mikrocontroller. Der Compiler für AVR Mikrocontroller heißt „avr-gcc“. Das ist ein kostenloses und quell-offenes Programm, welches von der Free Software Foundation bereitgestellt wird. Damit der Compiler nutzbar ist, braucht man noch einige andere Hilfsprogramme und die C-Library. All das zusammen gepackt nennt man „Toolchain“.

3.7.3 Entwicklungsumgebung

Programmierer benutzen häufig eine Entwicklungsumgebung (kurz: IDE), um Quelltexte zu schreiben. Die Entwicklungsumgebung funktioniert im Grunde wie eine Textverarbeitung, enthält jedoch zusätzliche Funktionen, die das Entwickeln von Programmen vereinfacht. Für AVR Mikrocontroller nutzen wir die kostenlose Entwicklungsumgebung „AVR Studio 4.19“ zusammen mit der Toolchain „WinAVR 2010“. Beide Programme kannst du von meiner Homepage http://stefanfrings.de/avr_ide/index.html herunterladen.

Im Band 3 erkläre ich, wie du deine Programme ohne AVR Studio auf der Kommandozeile compilieren kannst.

3.7.3.1 Simulator

Der Simulator ist eine Teilfunktion von AVR Studio, welche die inneren Funktionen von AVR Mikrocontrollern simuliert. Man kann damit in gewissem Maße das Programmieren üben, ohne einen echten Mikrocontroller-Chip zu benutzen. Da die Simulatoren nicht mit „echter“ Hardware verbunden werden können, ist ihr Nutzen jedoch sehr begrenzt.

3.7.3.2 Debugger

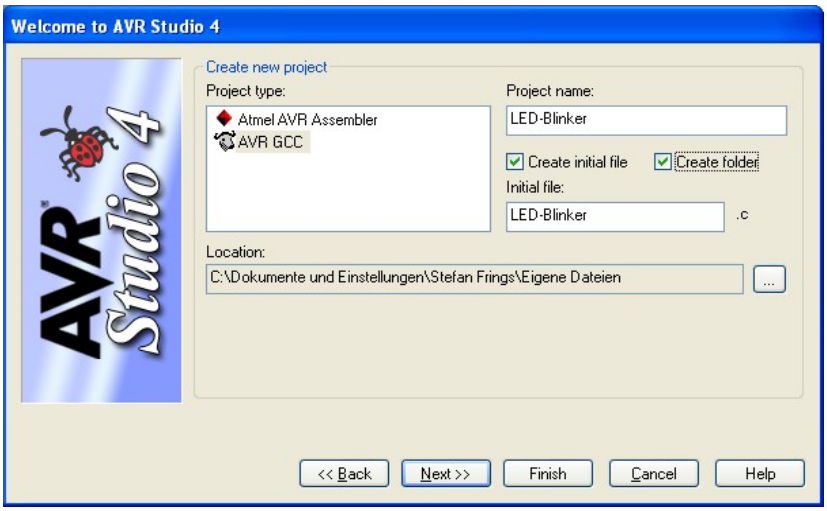
Mit dem Debugger des AVR Studios kann man ein Programm pausieren und dann den Status des Rechenkerns einsehen, sowie den Inhalt des Speichers, und auch den Status aller Ein/Ausgabe Leitungen.

Der Debugger kann zusammen mit dem Simulator benutzt werden, aber auch mit den meisten echten AVR Mikrocontrollern. Dazu sind die Mikrochips je nach Größe entweder mit einer JTAG Schnittstelle oder mit einem Debug Wire (DW) ausgestattet. Um diese Schnittstellen benutzen zu können, benötigt man ein spezielles Gerät, nämlich den „Amtel Dragon“ oder den „Atmel ICE“. Aus Kostengründen verzichtet dieses Buch darauf.

3.7.4 Bedienung AVR Studio

Du hast den ersten Mikrocomputer zusammen gelötet, die nötige Software installiert, und dein ISP-Programmer ist auch einsatzbereit. Jetzt kannst du endlich dein allererstes Programm schreiben.

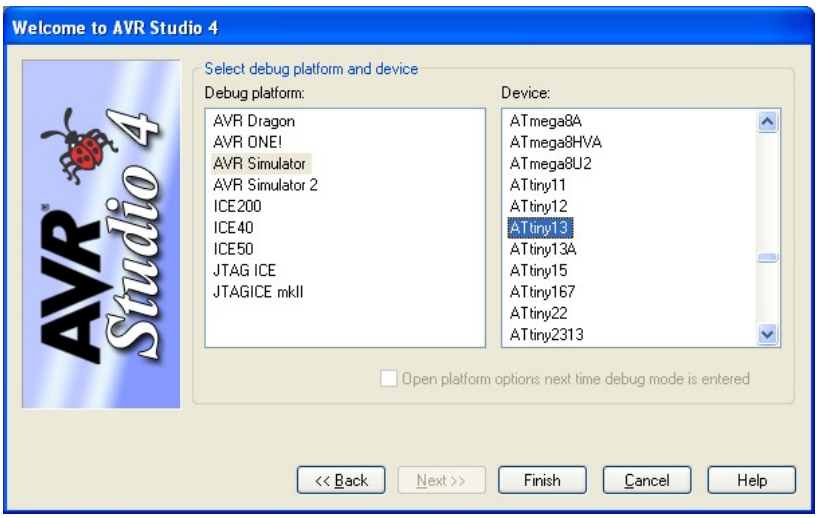
Starte das AVR Studio. In dem Willkommen-Dialog kannst du neue Software-Projekte anlegen oder vorhandene Projekte öffnen. Klicke auf „New Project“.



Im Feld „Project Type“ wählst du „AVR GCC“ aus. Assembler ist eine andere Programmiersprache, die ich in diesem Buch nicht erkläre.

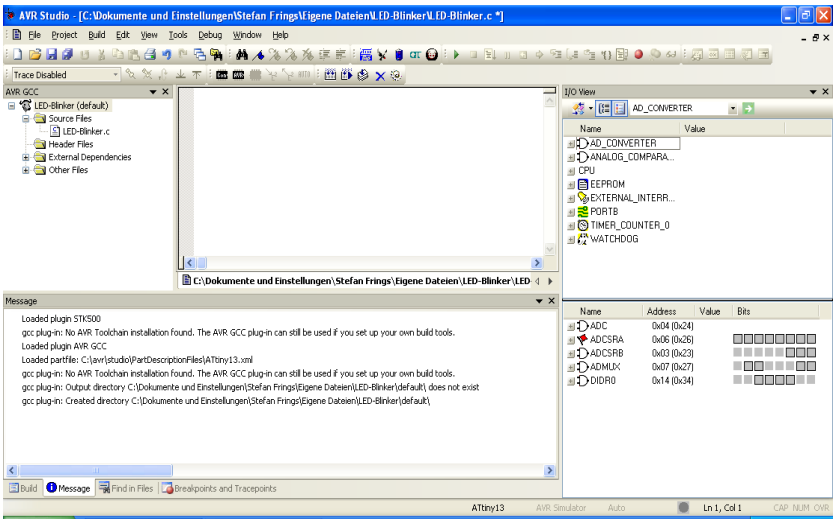
Gib in das Feld „Project Name“ einen Namen für dein Projekt ein, und zwar ohne Leerzeichen. Zum Beispiel „LED-Blinker“. Markiere die beiden Checkboxen darunter.

Das Feld „Location“ zeigt dir an, wo die Dateien des Projektes abgelegt werden. In meinem Fall ist es der Ordner „Eigene Dateien“. Den Ordner kannst du nach Belieben ändern. Klicke auf „Next“.



Links wählst du aus, welchen Debugger oder Simulator du verwenden wirst. Da du keinen Debugger hast, wählst du den „AVR Simulator“ oder den „AVR Simulator 2“. Beide sind für dieses Projekt gleich gut geeignet. Auf der rechten Seite stellst du ein, welchen AVR Mikrocontroller du verwendest, also den „ATtiny13“. Klicke dann auf „Finish“.

Das Hauptfenster von AVR Studio sieht so aus:



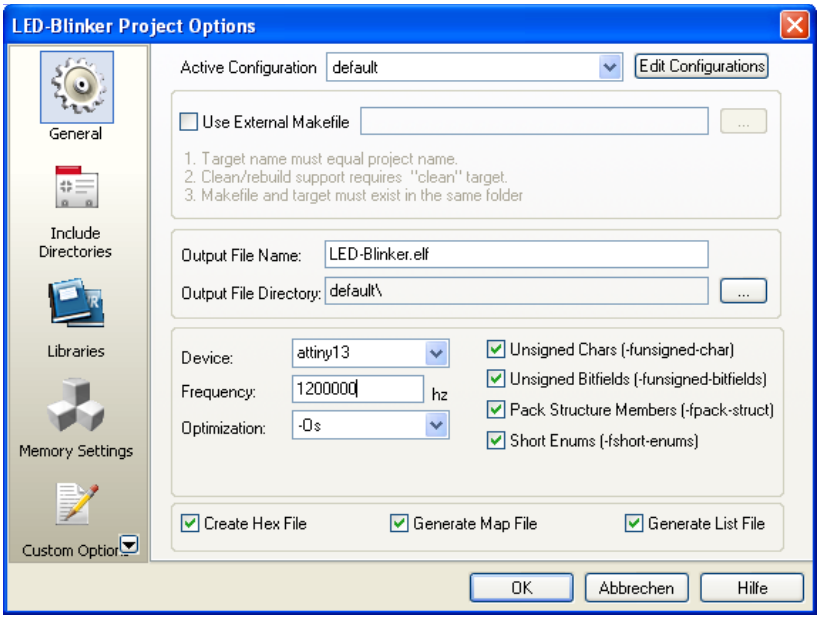
- Im linken Bereich listet die Entwicklungsumgebung die wichtigsten Dateien deines Projektes auf. Neue Projekte bestehen zunächst nur aus einer einzigen Quelltext-Datei. Das kann sich später ändern.
- Im unteren Bereich befinden sich mehrere Fenster mit Meldungen vom Programm. Dort findest du alle Fehlermeldungen.
- Im rechten Bereich zeigt der Simulator (oder der Debugger) an, was im Innern des Mikrocontroller passiert.
- In der Mitte befindet sich der Quelltext-Editor. Hier schreibst du das Programm.

Wenn dein Bildschirm klein ist, kannst du im Menü View / Toolbars einige Sachen ausblenden, die du vorläufig nicht brauchst: MDI Tabs, AVRGCCPLUGIN, STK500, TRACEBAR, I/O, PROCESSOR.

Schreibe das folgende Programm ab:

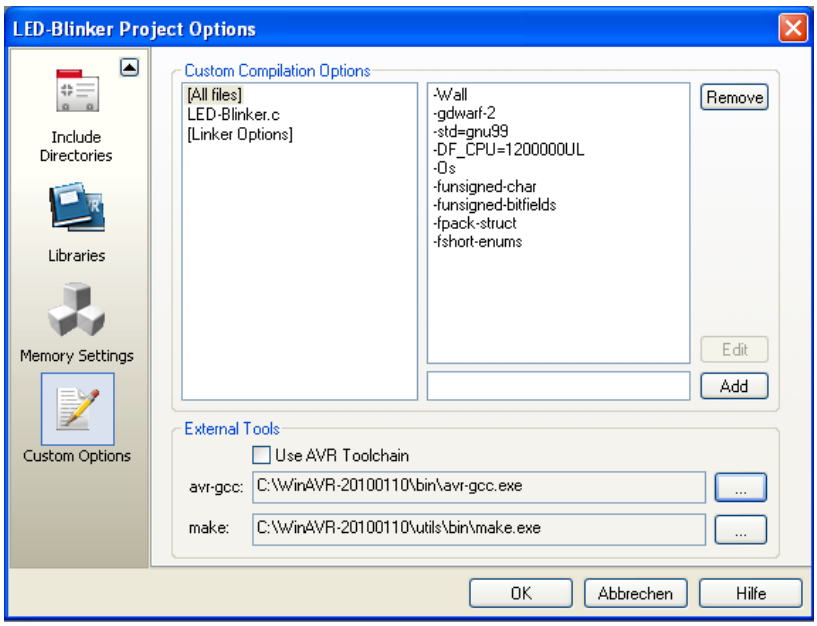
```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
    DDRB = 1;
    while (1) {
        PORTB = 1;
        _delay_ms(500);
        PORTB = 0;
        _delay_ms(500);
    }
}
```

Was das alles bedeutet, wirst du später lernen. Zuerst sollst du lediglich lernen, das AVR Studio zu bedienen. Klicke auf den Speicher-Button oder drücke Strg-S, um die Datei abzuspeichern. Klicke auf den Menüpunkt Project/Configuration Options.



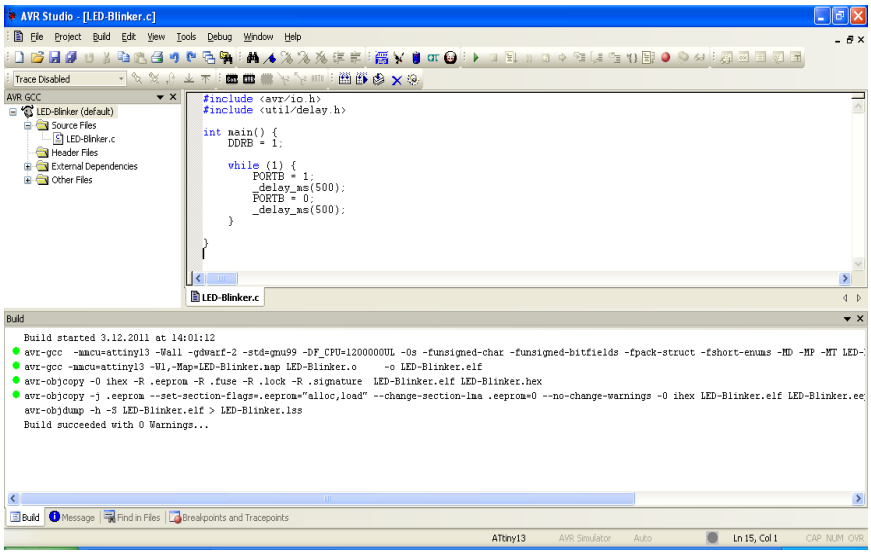
Gib in das leere „Frequency“ Feld die Zahl 1200000 (bzw. 1000000 beim ATtiny25, 45 oder 85) ein, das ist die Taktfrequenz des Mikrocontrollers.

Gehe dann links unten auf „Custom Options“.



Schalte die Checkbox „Use AVR Toolchain“ aus, denn diese Einstellung funktioniert nicht. Wir verwenden stattdessen das Programmpaket „WinAVR“. Klicke auf die Buttons mit den drei Punkten unten rechts und suche die Dateien avr-gcc.exe und make.exe. Klicke zum Schluss Ok.

Zurück im Hauptfenster drücke F7 oder benutze den Menüpunkt Build/Build, um den Quelltext zu kompilieren. Wenn alles richtig eingestellt ist, erhältst du die folgenden Meldungen:



Der Quelltext ist jetzt fertig kompiliert. Nun überträgst du ihn mit dem ISP-Programmer in den Mikrocomputer.

3.7.5 Bedienung ISP-Programmer

Stelle die Schalter oder Steckbrücken des ISP-Programmer so ein, dass er das Target (Ziel) mit 3,3 Volt oder 5 Volt Spannung versorgt. Falls dein Programmieradapter keine Spannungsversorgung bereitstellen kann (und nur dann!), schließt du deinen Batteriehalter mit drei Akkus an VCC und GND an.

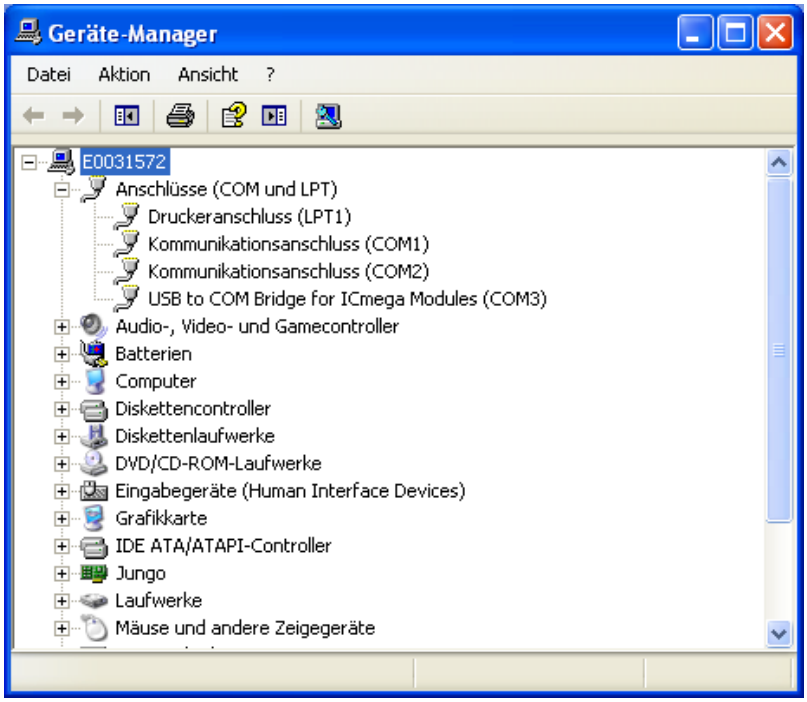
Ich weiß, das habe ich bereits zweimal geschrieben. Es ist aber wirklich wichtig, denn wenn du das falsch machst, riskierst du, den USB Port des Computers zu zerstören.

Stecke den Programmieradapter in den USB Port deines Computers und schließe die Platine mit dem Mikrocontroller richtig herum an den Programmieradapter an.

3.7.5.1 Programmer mit seriellem Port

Viele USB Programmieradapter (z.B. der Diamex Stick) kommunizieren zum Anwendungsprogramm über einen virtuellen seriellen Port. Wenn man sie zum ersten mal einsteckt, installiert Windows den entsprechenden Treiber in der Regel vollautomatisch. Den Namen des seriellen Ports findest du so heraus:

- Windows XP:
  Start / rechte Maustaste auf Arbeitsplatz / Eigenschaften / Hardware / Geräte-Manager
- Windows Vista und Windows 7 bis 10:
  Auf das Windows-Logo in der Task-Leiste klicken und dann in das Suchfeld „Geräte-Manager“ eingeben.



In diesem Fall ist es der Anschluss "COM3".

- Linux:
  Gib unmittelbar nach dem Einstecken in einem Terminalfenster „dmesg“ ein. Dann erscheinen ziemlich viele Meldungen auf dem Bildschirm, von denen nur die letzten paar Zeilen von Interesse sind. Eine davon sollte darauf hinweisen, welcher serielle Port dem gerade erkanntem Gerät zugewiesen wurde.

```
stefan@STEFANSPC: ~
[ 2043.740562] usb 2-1.2: new full-speed USB device number 4 using ehci-pci
[ 2043.835026] usb 2-1.2: New USB device found, idVendor=10c4, idProduct=ea60
[ 2043.835038] usb 2-1.2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 2043.835045] usb 2-1.2: Product: CP2102 USB to UART Bridge Controller
[ 2043.835051] usb 2-1.2: Manufacturer: Silicon Labs
[ 2043.835056] usb 2-1.2: SerialNumber: 0001
[ 2044.879109] usbcore: registered new interface driver usbserial
[ 2044.879150] usbcore: registered new interface driver usbserial_generic
[ 2044.879183] usbserial: USB Serial support registered for generic
[ 2044.887135] usbcore: registered new interface driver cp210x
[ 2044.887355] usbserial: USB Serial support registered for cp210x
[ 2044.887419] cp210x 2-1.2:1.0: cp210x converter detected
[ 2044.889136] usb 2-1.2: cp210x converter now attached to ttyUSB0
stefan@STEFANSPC:~$
```

In diesem Fall ist es der Anschluss "ttyUSB0". Der Name fängt immer mit tty an.

3.7.5.2 Programmer ohne seriellen Port

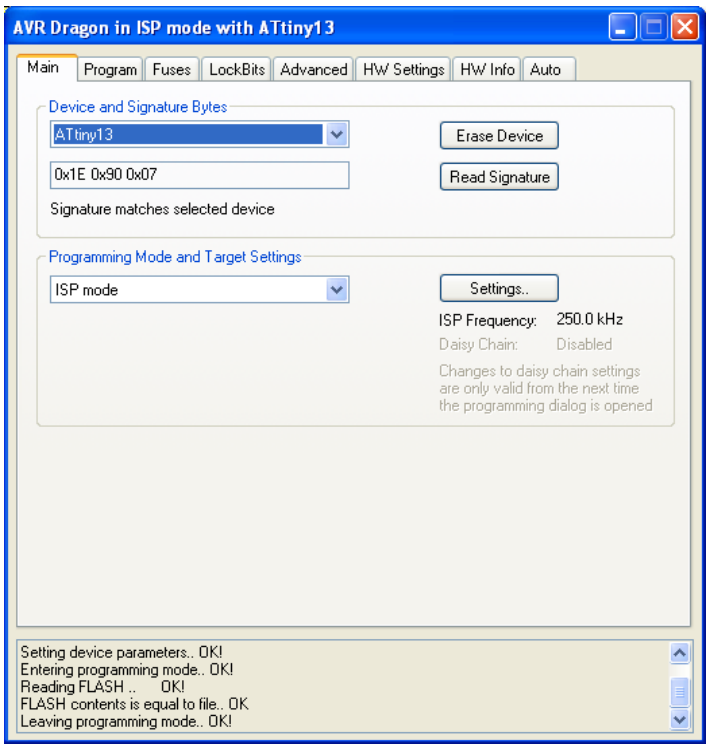
Programmieradapter ohne (virtuellen) seriellen Port werden mit Hilfe eines USB Treibers angesprochen. Das AVR Studio nutzt dazu den mitgelieferten **Jungo** Treiber, die meisten anderen Programme (z.B. avrdude) nutzen hingegen den **libusb** Treiber. Jungo und libusb dürfen gleichzeitig installiert sein, sie stören sich nicht gegenseitig.

Windows 8 und neuere Versionen verweigern normalerweise das Laden dieser beiden Treiber, weil sie nicht von Microsoft signiert sind. Eine Anleitung, wie man Windows dazu bringt, solche Treiber zu laden, findest du auf

http://stefanfrings.de/isp_programmieradapter/libusb.html

3.7.5.3 Atmel kompatible Programmer

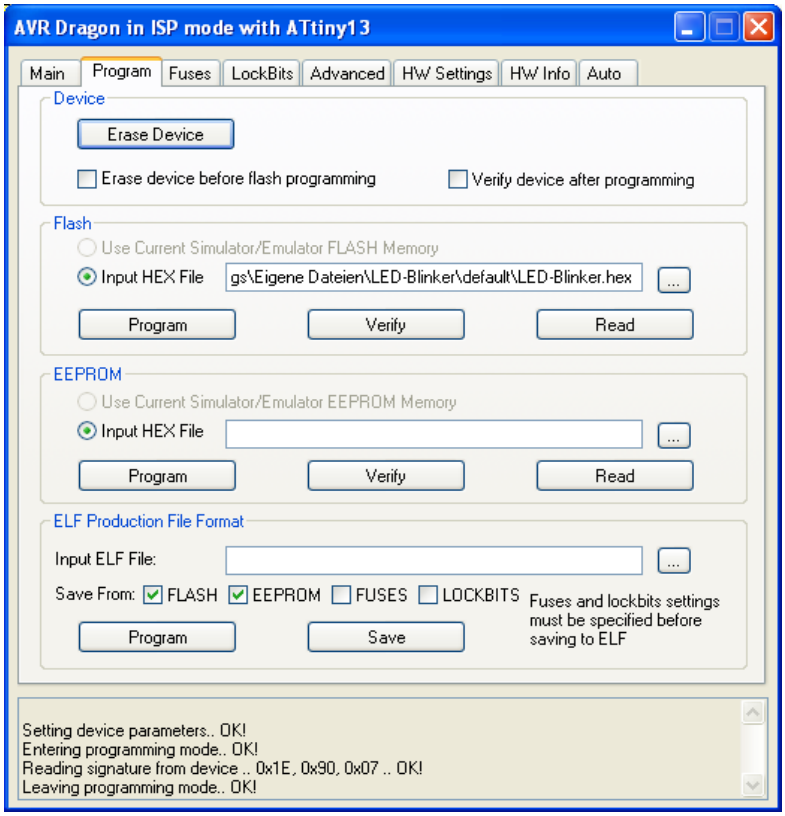
EInige Programmieradapter von Atmel und serielle Programmieradapter mit dem STK500 Protokoll (z.B. der Diamex Stick) werden direkt vom AVR Studio Unterstützt. Starte es über den Menüpunkt Tools / Program AVR / Connect. Gehe zuerst in den „Main“ Tab:



Stelle den „Programming Mode“ auf „ISP mode“. Dann klickst du auf den Settings Knopf. Dort stellst du ein, wie schnell die Kommunikation zum Mikrocontroller ablaufen soll. Mit mehr als 250kHz wird es nicht funktionieren, weil die Taktfrequenz deines Mikrocontroller 1,2MHz ist und die Übertragungsrate kleiner als ¼ der Taktfrequenz sein muss.

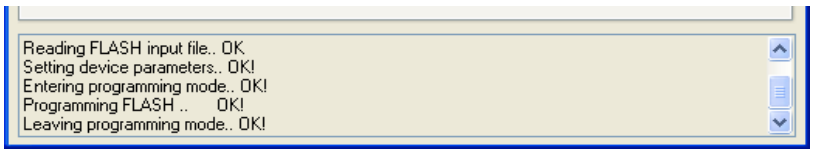
Klicke dann auf den Knopf „Read Signature“. Das Programm kommuniziert nun mit deinem AVR Mikrocontroller. Es fragt die Signatur des Mikrochips ab, und zeigt dann dessen Namen „ATtiny13“ oben links an. Durch die automatische Erkennung werden auch einige Einstellungen angepasst, die später wichtig sind. Jedes mal, wenn du das Programm neu startest oder den ISP-Programmer neu verbindest, solltest du auf den „Auto-Detect“ Knopf klicken.

Wechsle nun zum „Program“ Tab.



Löschen den Mikrocontroller durch Klick auf den Knopf „Erase Device“. Der Mikrocontroller muss immer vorher gelöscht werden.

Klicke in dem „Flash“ Bereich auf den Knopf mit den drei Punkten und suche die Datei LED-Blinker/default/LED-Blinker.hex. Übertrage dann das Programm durch Klick auf den Knopf „Program“. Der Erfolg des Vorgangs wird durch diese Meldungen bestätigt:

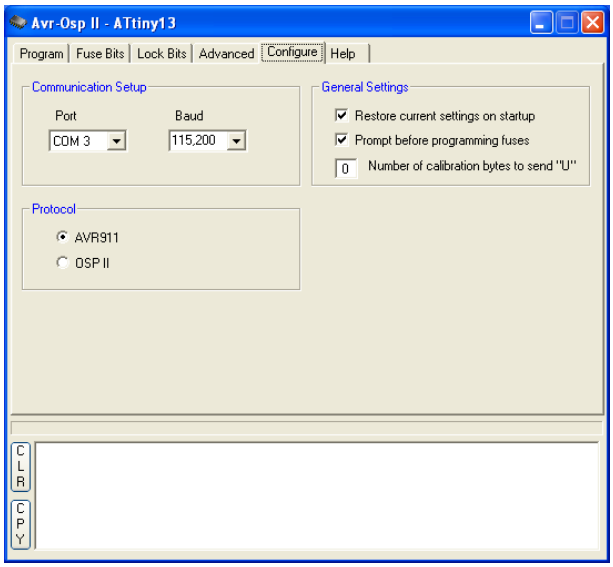


Die LED sollte jetzt blinken. Falls sie nicht blinkt, klicke auf den „Verify“ Knopf, um zu prüfen, ob das Programm korrekt übertragen wurde. Falls nicht, lösche den Chip nochmal und übertrage das Programm erneut. Möglicherweise hast du auch einen Fehler im Quelltext. Nach jeder Korrektur musst du ihn neu Kompilieren (Taste F7), dann den Mikrocontroller löschen und dann das Programm erneut übertragen.

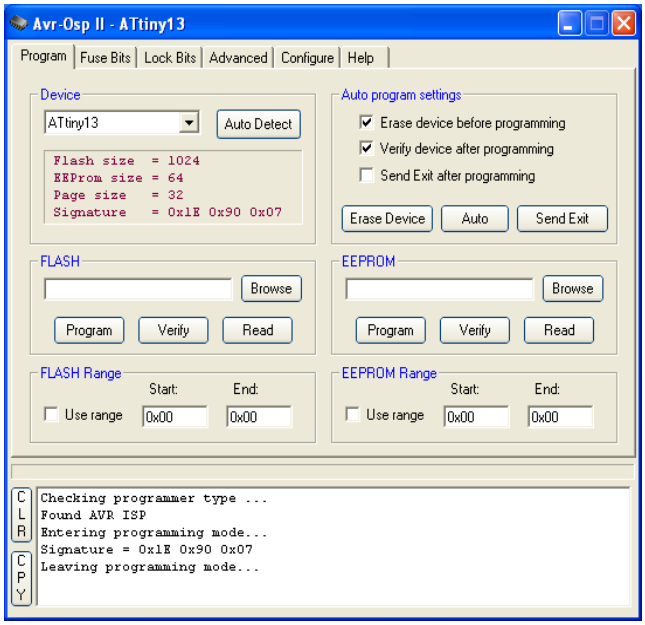
3.7.5.4 AvrOspII

Unter Windows kannst du serielle Programmieradapter mit dem AVR910, AVR911, AVRISP und OSP II Protokoll mit dem Programm AvrOspII bedienen. Dieses Programm eignet sich zum Beispiel für die ICprog Programmieradapter von der Firma In-Circuit.

Im Tab „Configure“ stellst du den COM-Port und die Bitrate ein, die dein Gerät benötigt. Stelle darunter das richtige Protokoll ein.



Für AVR910, AVR911 und AVRISP kompatible Geräte stellst du das Protokoll auf AVR911 ein. Für OSP II kompatible Geräte stellst du das Protokoll auf OSP II ein. Dann gehst du auf den „Program“ Tab und klickst auf den „Auto-Detect“ Knopf.



Das Programm kommuniziert nun mit deinem AVR Mikrocontroller. Es fragt die Signatur des Mikrochips ab, und zeigt dann dessen Namen „ATtiny13“ oben links an. Durch die automatische Erkennung werden auch einige Einstellungen angepasst, die später wichtig sind.

Jedes mal, wenn du das Programm neu startest oder den ISP-Programmer neu verbindest, solltest du auf den „Auto-Detect“ Knopf klicken.

Lösche den Mikrocontroller durch Klick auf den Knopf „Erase Device“. Der Mikrocontroller muss immer vorher gelöscht werden. Klicke auf den „Browse“ Knopf und suche die Datei LED-Blinker/default/LED-Blinker.hex. Übertrage dann das Programm durch Klick auf den Knopf „Program“.

Der Erfolg des Vorgangs wird durch diese Meldungen bestätigt:

```
Checking programmer type ...
Found AVR ISP
Entering programming mode...
Reading HEX input file for flash operations...
Programming Flash contents... 0x0000 TO 0x0051
Leaving programming mode...
```

Die LED sollte jetzt blinken. Falls sie nicht blinkt, klicke auf den „Verify“ Knopf, um zu prüfen, ob das Programm korrekt übertragen wurde. Falls nicht, lösche den Chip nochmal und übertrage das Programm erneut. Möglicherweise hast du auch einen Fehler im Quelltext. Nach jeder Korrektur musst du ihn neu Kompilieren (Taste F7), dann den Mikrocontroller löschen und dann das Programm erneut übertragen.

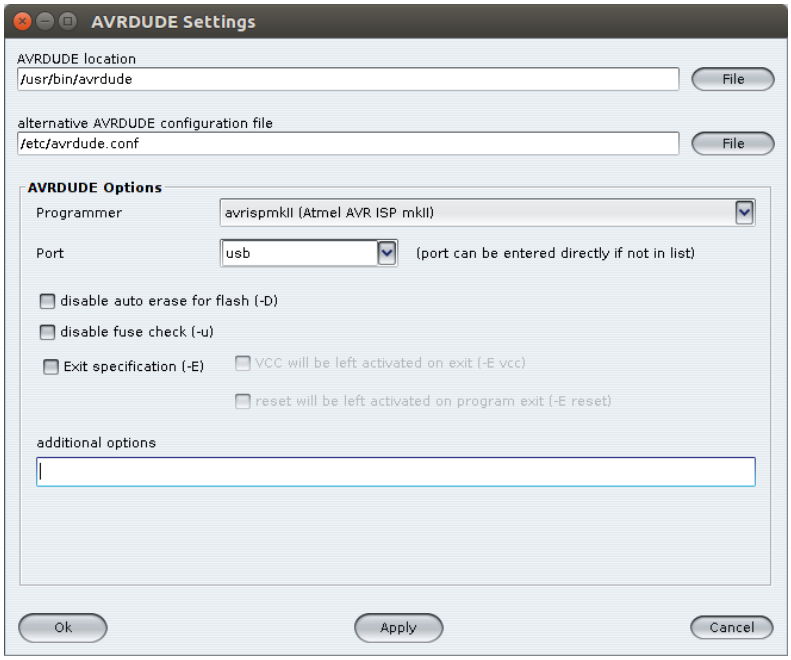
3.7.5.5 Avrdude und Burn-O-Mat

Falls du einen Programmieradapter hast, der von den beiden zuvor genannten Programmen nicht unterstützt wird, kannst du „avrdude“ verwenden. Avrdude gibt es für Linux, Windows und Mac OS.

Es handelt sich dabei um ein Kommandozeilenprogramm, das dazu gedacht ist, im Befehlsfenster benutzt zu werden. Dazu passend erhält man eine grafische Benutzeroberfläche, indem man das Java Programm Burn-O-Mat installiert.

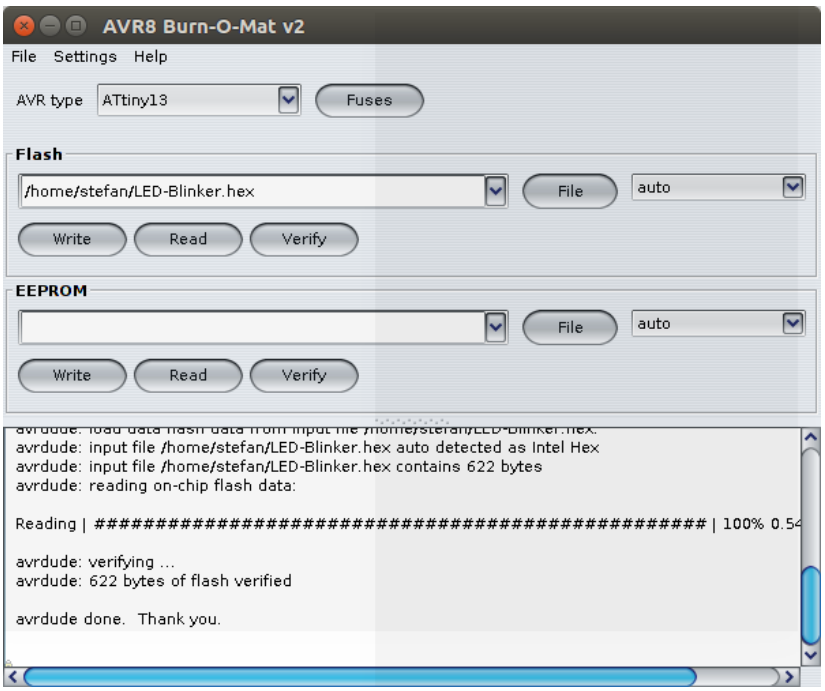
Installiere also avrdude, Burn-O-Mat und eine Java Runtime (JRE), sowie ggf. den libusb Treiber für Geräte, die keinen virtuellen seriellen Port bereitstellen.

Nach dem ersten Programmstart gehst du im Burn-O-Mat ins Menü Settings/AVRDUDE um die Pfade zum avrdude Programm und dessen Konfigurationsdatei einzustellen. Beispiel für Ubuntu Linux:



In diesem Dialog stellst du auch den Typ deines Programmieradapters ein, und an welchen Anschluss er angeschlossen ist. Für Geräte ohne seriellen port ist „usb“ die richtige Einstellung.

Im Hauptfenster vom Burn-O-Mat stellst du nun ganz oben den richtigen Mikrocontroller Typ ein. Danach kannst du nun mit dem „File“ Knopf das compiliertes Programm (*.hex Datei) laden und dann mit dem „Write“ Knopf in den Mikrocontroller übertragen.



Die LED sollte danach blinken. Falls sie nicht blinkt, klicke auf den „Verify“ Knopf, um zu prüfen, ob das Programm korrekt übertragen wurde. Möglicherweise hast du auch einen Fehler im Quelltext. Nach jeder Korrektur musst du ihn neu Kompilieren (Taste F7), und dann das Programm erneut übertragen.

Falls der Programmieradapter eine Kommunikationsstörung zum „Target“ meldet, oder eine falsche „Device Signature“ moniert, obwohl alle deine Einstellungen ganz sicher richtig sind, muss der Programmieradapter eventuell dazu gebracht werden, langsamer mit dem Mikrocontroller zu kommunizieren.

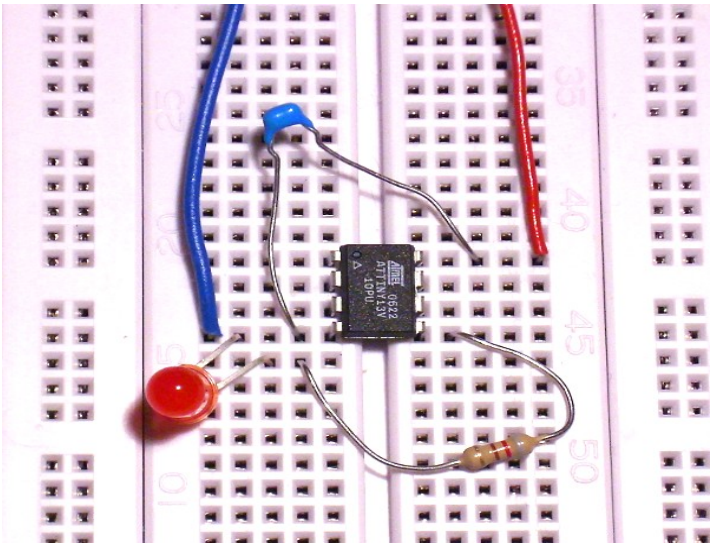
Manche Programmieradapter haben dazu einen Schalter oder eine Steckbrücke, mit der Beschriftung „Slow Clock“ oder so ähnlich. Bei den anderen kannst du die Geschwindigkeit mit der Option “-B20“ herabsetzen. Diese gibst du im Dialog Settings/AVRDUDE bei „additional options“ ein. Falls das auch nicht klappt, lass dir von jemandem helfen, der mit avrdude erfahren ist. Wenn du ein Bildschirmfoto von der Fehlermeldung im mikrocontroller.net Forum postest, wird dir sicher schnell geholfen.

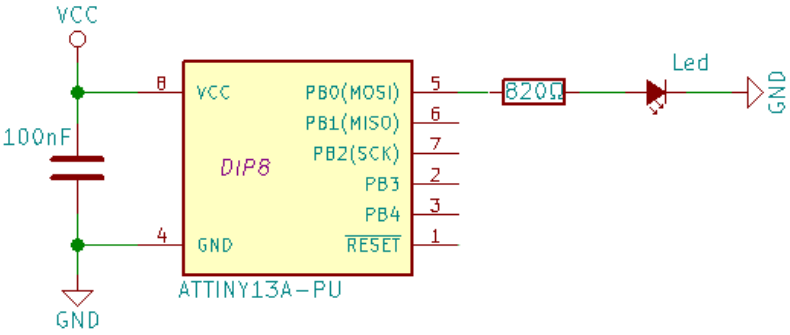
Warnung:

Lass die Finger von den Fuses und den Lock-Bits, solange du dich damit noch nicht auskennst. Durch falsche Einstellung kannst du deinen AVR Mikrocontroller unbrauchbar machen. Ich erkläre diese Sachen später.

3.8 Aufbau auf dem Steckbrett

Der erste gelötete Mikrocomputer läuft nun. Wir wollen ihn nun auf das Steckbrett übertragen, damit wir einfacher an ihm herum experimentieren können. Baue die folgende Schaltung nach, wobei du den bereits programmierten Mikrocontroller aus der gelöteten Platine verwendest.





Zur Stromversorgung dient jetzt unser Batteriekasten mit drei Akkus, die zusammen etwa 3,6V liefern, oder Einwegbatterien die in frischem Zustand 4,5 V liefern. Den ISP-Stecker und den Schutzwiderstand lassen wir weg. Der Keramik-Kondensator stabilisiert wie gehabt die Spannung der Stromversorgung und der Widerstand begrenzt die Stromstärke der Leuchtdiode. Alles nichts Neues.

Sobald die Batterien angeschlossen sind, blinkt die Leuchtdiode.

3.9 Übungsaufgaben

Du hast nun einen ganz wichtigen Punkt erreicht. Du hast den ersten Mikrocomputer gebaut und in Betrieb genommen. Und er funktioniert!

Mache eine mehrtägige Pause und teile deine Freude mit Menschen, die dir lieb sind. Danach solltest du versuchen, die folgenden Übungsaufgaben zu lösen.

1. Wie viele Pins hat ein ISP Stecker?
   - **a)** zehn
   - **b)** sechs
   - **c)** acht
2. Warum darfst du Mikrochips niemals falsch herum einstecken?
   - **a)** Weil dann ein Kurzschluss entsteht, der nicht nur den Chip, sondern auch den ISP-Programmer und den USB-Port des Computers zerstören kann.
   - **b)** Weil dann der Chip kaputt geht – ohne weitere Auswirkungen.
   - **c)** Weil nicht klar ist, was dann passiert.
3. Was ist ein Mikrocontroller?
   - **a)** Ein Prozessor von Intel, wie er in vielen Notebooks steckt.
   - **b)** Ein Mikroprozessor mit mindestens 8 Gigabyte eingebauten Speicher.
   - **c)** Ein Mikroprozessor mit Speicher, der speziell für kleine Steuerungs-Aufgaben gedacht ist.
4. Wozu dient der rote Streifen auf Flachkabeln?
   - **a)** Wenn er sichtbar wird, ist die Rolle fast aufgebraucht.
   - **b)** Er kennzeichnet die erste Ader, die mit Pin1 des Steckers verbunden ist.
   - **c)** Rot ist immer der Plus-Pol.
5. Ist es wichtig, dass gelötete Platinen „sauber“ aussehen?
   - **a)** Ja, unsauber verarbeitete Platinen enthalten Fehler. Eventuell sogar Kurzschlüsse, die zu Schäden führen.
   - **b)** Nein, die Optik ist dem Strom egal.
6. Was bedeutet es, wenn in einem Schaltplan eine Linie mit einem umgedrehten „T“ endet?
   - **a)** Alle so gekennzeichneten Punkte sind miteinander verbunden und werden an den Minus-Pol der Stromversorgung angeschlossen.
   - **b)** Diese Leitungen sind mit nichts verbunden. Das Symbol bedeutet „Sackgasse“.
   - **c)** Hier wird die Erdung oder der Blitzableiter angeschlossen.
7. Warum platziert man grundsätzlich neben jeden Mikrochip einen 100 nF Kondensator?
   - **a)** Das ist Spielerei, hat sie die Lobby der Bauteile-Hersteller so ausgedacht.
   - **b)** Sie stabilisieren die Versorgungsspannung. Ohne diese Kondensatoren muss man mit sporadischen Fehlfunktionen rechnen.
   - **c)** Sie überbrücken kurze Ausfälle der Stromversorgung, z.B. wenn man die Batterien auswechselt.
8. Wie ist Pin 1 eines Mikrochips gekennzeichnet?
   - **a)** Die Nummern der Pins sind auf dem Chip aufgedruckt.
   - **b)** Man zählt im Uhrzeigersinn. Der Anfang ist durch einen Punkt oder eine Kerbe markiert.
   - **c)** Man zählt gegen den Uhrzeiger-Sinn. Oben ist irgendwie markiert.

9. Wenn man fertig gelötet hat, kann man dann den Strom einschalten?

a) Nein, zuerst ist eine Sichtkontrolle angemessen. Danach sollte man zumindest die Leitungen der Spannungsversorgung durch messen, um unsichtbare Kurzschlüsse zu erkennen.

b) Sichtkontrolle ist quatsch, man kann den Strom sowieso nicht sehen. Man sollte alle Leitungen durch Messen prüfen, bevor man zum ersten mal den Strom einschaltet.

c) Ja sicher, für Notfälle hat man ja einen Sicherungskasten im Keller.

10. Was überträgt man in den Mikrocontroller?

a) Den Quelltext des Programms.

b) Den Byte-Code des Programms.

c) Den Compiler von Atmel.

11. Worin unterscheiden sich Simulator und Debugger?

a) Der Simulator läuft nur auf dem Computer. Er hat keine Verbindung zur externen Hardware. Der Debugger lässt das Programm auf der externen Hardware laufen und überwacht sie.

b) Der Simulator simuliert einen kompletten Computer. Der Debugger untersucht Programme auf dem Computer.

c) Das ist das Gleiche.

12. Mit welcher Taktfrequenz läuft dein erster Mikrocomputer?

a) 3,3 Gigahertz

b) 250 Kilohertz

c) 1,2 Megahertz

13. Was darf man nicht vergessen, wenn man ein Programm in einen AVR Mikrocontroller überträgt?

a) Man muss vorher das laufende Programm stoppen.

b) Man muss vorher den Programmspeicher löschen.

c) Man muss das alte Programm vorher deinstallieren.

4 Programmieren in C

Das AVR Studio unterstützt zwei Programmiersprachen: Assembler und C. Beide Programmiersprachen haben ihre Vor- und Nachteile. Die Programmierung in Assembler ist näher an der Hardware, da jeder Assembler Befehl genau einem Byte-Code entspricht. Dementsprechend sind Assembler Programme immer nur für einen ganz bestimmten Mikroprozessor geschrieben.

Die Programmierung in C findet auf einer abstrakteren Ebene statt. Der C Programmierer muss den Byte-Code und Aufbau des Mikroprozessors nicht so genau kennen. C Programme werden vom Compiler zuerst optimiert und dann Byte-Code übersetzt. Die automatische Optimierung durch den Compiler ist ganz besonders nützlich. Ein Beispiel:

Wenn du in deinem Programm „a=b*2“ schreibst, dann macht der Compiler daraus „a=b+b“, weil der Compiler weiß, dass der AVR Mikrocontroller schneller addieren kann, als multiplizieren.

In diesem Kapitel wirst du lernen, AVR Mikrocontroller in C zu programmieren. Wir werden dazu den Simulator verwenden, du kannst die elektronischen Teile also erst mal weg legen.

4.1 Grundgerüst für jedes Programm

Lege ein neues Projekt in AVR Studio an, mit dem Namen „Test“. Die Vorgehensweise kennst du schon:

- Klicke im Menü auf Project/New Project
- Project Type = AVR GCC
- Project Name = Test
- „Create Initial File“ und „Create Folder“ einschalten
- Location: ist egal
- Dann auf „Next“ klicken
- Debug Platform = AVR Simulator
- Device = ATtiny13
- Klicke auf „Finish“

AVR Studio legt automatisch die (leere) Datei test.c an und öffnet sie im Editor.

- Klicke im Menü auf Project/Configuration Options
- Frequency = 1200000 bei ATtiny13, oder 1000000 bei ATtiny25, 45oder 85
- Klicke am linken Rand auf „Custom Options“
- „Use AVR Toolchain“ aus schalten
- avr-gcc = c:\<Pfad zur Toolchain>\avr-gcc.exe
- make = c:\<Pfad zur Toolchain>\make.exe
- Dann auf „Ok“ klicken.

Im Editor gibst du das folgende Programmgerüst ein:

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

int main(void)
{
    .............
}
```

Du wirst jedes C-Programm genau auf diese Weise beginnen.

4.1.1 Include

Die include Anweisung sagt dem Compiler, dass er den Inhalt einer weiteren Datei einbeziehen soll. Wenn der Dateiname in spitze Klammern (<...>) eingeschlossen ist, sucht der Compiler die Datei in dem include-Verzeichnis der Toolchain. Wenn der Dateiname in Anführungsstriche ("...") eingeschlossen ist, sucht der Compiler die Datei im Projektverzeichnis.

Die Toolchain enthält neben dem GNU C Compiler auch die AVR C Library. Das ist eine Sammlung hilfreicher Codestücke, die du in dein Programm einbinden kannst. In deinem ersten Programm mit der blinkenden LED hast du bereits eine Funktion aus der AVR C Library benutzt, nämlich die Funktion _delay_ms().

Die Funktionen der AVR C Library sind in mehrere Include-Dateien aufgeteilt. In diesem Buch verwenden wir folgende Include-Dateien:

- avr/io.h
  enthält alle Register des Mikrocontrollers. Diese Datei brauchst du immer.
- avr/eeprom.h
  enthält Funktionen zum Zugriff auf den EEPROM Speicher.
- avr/power.h
  enthält Funktionen zur Energieverwaltung.
- avr/sleep.h
  enthält Funktionen, mit denen man den Mikrocontroller zum Einschlafen bringen kann.
- avr/wdt.h
  enthält Funktionen zur Steuerung des Watch-Dog.
- util/delay.h
  enthält Funktionen für Warte-Pausen.

Die originale Dokumentation der Library findest du auf der Webseite http://www.nongnu.org/avr-libc oder durch eine Suche nach „avr-libc". Diese Seiten gibt es leider nur auf englisch.

4.1.2 Main-Funktion

Beim Einschalten der Stromversorgung startet das Programm immer mit der main() Funktion:

```
int main(void)
{
    .............
}
```

Die geschweiften Klammern markieren Anfang und Ende der Funktion. Dazwischen platziert man die Anweisungen (oder Befehle) des Programms.

4.2 Syntax

Die meisten Regeln zur Schreibweise eines Quelltextes sind ganz einfach und logisch. Du wirst sie beim Lesen der Programm-Beispiele von selbst erkennen. Dieses Kapitel beschränkt sich daher auf die wenigen Regeln, die nicht offensichtlich sind.

Du musst sie nicht gleich auswendig lernen, aber es wäre sicher nicht schlecht, ein Lesezeichen in diese Seite zu stecken, damit du sie bei Bedarf schnell wieder findest. Wir werden all diese Regeln durch Übungen im Simulator ausprobieren.

Ein gewisses Maß an langweiliger Theorie ist für dieses Hobby leider notwendig. Das Lesen von Anleitungen ist genau so wichtig, wie der viel spannendere Umgang mit Bauteilen.

4.2.1 Funktionen

C-Programme bestehen in der Regel aus vielen Funktionen. Sogar das Hauptprogramm ist eine Funktionen, und die heißt main(). Funktionen sind mehr oder weniger kleine Blöcke von Befehlen, aus denen das ganze Programm zusammengesetzt wird.

Funktionen haben in der Regel Eingabe-Parameter, auch „Argumente“ genannt. Zum Beispiel hat die Funktion _delay_ms() einen Eingabeparameter:

```
_delay_ms(500);
```

Die Zahl 500 ist in diesem Fall der Eingabeparameter, was 500 Millisekunden bedeutet. Bei Funktionen mit mehreren Eingabeparametern trennt man die Werte durch Kommata:

```
delete(3,9);
```

Die meisten Funktionen berechnen irgend etwas und liefern das Ergebnis als Rückgabewert ab. Zum Beispiel erhältst du so den Sinus von 1 in der Variable x:

```
x=sin(1);
```

4.2.2 Whitespaces

Wahrscheinlich hast du schon bemerkt, dass es dem Compiler ganz egal ist, wie viele Leerzeichen und Leerzeilen du schreibst. Auch die Tabulator-Zeichen zählen dazu, das sind die „breiten“ Leerzeichen, die erscheinen, wenn du die Tabulator-Taste drückst. Diese Zeichen nennt man „Whitespace“:

- Leerzeichen
- Tabulator
- Zeilenumbruch

Für den Compiler haben alle drei Whitespace Zeichen die gleiche Bedeutung und es ist egal, wie viele Whitespace Zeichen aufeinander folgen. Eins genügt, mehr sind aber auch Ok. Du erhältst so die Möglichkeit, den Quelltext deines Programms übersichtlich zu gestalten.

Die beiden folgenden Quelltexte funktionieren genau gleich:

Variante 1:

```
int main(void)
{
    warte_minuten(2);
    int x = 4*12;
    if (x == 48)
    {
        richtig();
    }
    else
    {
        falsch();
    }
}
```

Variante 2:

```
int main(void){warte_minuten(2);int x=4*12;if (x==48) {richtig();} else
{falsch();}}
```

Der Compiler erzeugt aus beiden Quelltexten exakt den gleichen Byte-Code. Du wirst deine Programme wie die erste Variante schreiben, damit sie gut lesbar sind.

4.2.3 Semikolons und geschweifte Klammern

Jede Anweisung muss mit einem Semikolon beendet werden. Man gruppiert mehrere Anweisungen mit Geschweiften Klammern. Die korrekte Platzierung von geschweiften Klammern ist vor allem bei if-Ausdrücken wichtig:

```
if (person == erwachsen)
{
    eintritt_kassieren(5.00);
    herein_lassen();
}
```

Hier soll eine Person abkassiert und herein gelassen werden, wenn sie erwachsen ist (z.B. in einer Diskothek). Wenn die Klammern vergessen werden, entsteht ein ganz falscher Programmablauf, der durch die veränderte Einrückung deutlich wird:

```
if (person == erwachsen)
    eintritt_kassieren(5.00);
herein_lassen();
```

Mit diesem fehlerhaften Programm würden wir alle Personen herein lassen, aber nur die Erwachsenen müssten Bezahlen.

4.2.4 Runde Klammern

Runde Klammern benutzt man, um Funktionsaufrufe zu Kennzeichnen:

```
eintritt_kassieren(5.00)
```

und um mathematische Ausdrücke zu formulieren:

```
x = (1+2)*5
```

und um Bedingungen zu formulieren:

```
if (Bedingung)
 dann tu etwas;
else
 tu etwas anderes;
```

4.2.5 Konstanten

Konstanten sind Zahlen oder Texte, die man direkt in den Quelltext schreibt. Man schreibt sie so:

- Zeichenketten in Anführungsstriche: “Hallo Welt!“
- Einzelne Zeichen in Hochkommata: 'A'
- Zahlen schreibt man englisch, mit Punkt statt Komma und ohne Tausender-Trennzeichen: 14000.99
- Negative Zahlen schreibt man mit Minus direkt vor den Ziffern: `-18`

Alternative Zahlensysteme:

- Zahlen, die mit einer Null beginnen, werden oktal interpretiert: 010 entspricht 8 dezimal.
- Binäre Zahlen beginnen mit „0b“: 0b11111111 entspricht 255 dezimal. Sie sind für Mikrocontroller-Programmierung sehr praktisch, weil viele Register bitweise verwendet werden. Binärzahlen sind im C-Standard leider nicht enthalten, aber der GNU C Compiler unterstützt sie dennoch.

- Hexadezimal-Zahlen: 0xFF entspricht 255 dezimal. Programmierer verwenden sie gerne, weil man sie leicht in binäre Zahlen umrechnen kann. Hexadezimal-Zahlen gehören zum C-Standard.

4.2.6 Makros

Makros definieren einfache Textersetungen. Im folgenden Beispiel steht das Markos PI für die Zahl 3.14159. Anschließend kannst du an beliebigen Stellen das Schlüsselwort PI verwenden. Der Compiler ersetzt es stets durch 3.14159.

```
#define PI 3.14159

int main(void)
{
    float radius=8;
    float umfang=2*PI*radius;
}
```

Das Prinzip funktioniert nicht nur mit Zahlen, sondern mit fast allem, was man so in Quelltexten schreiben kann. Beispiel:

```
#define FORMEL a+b

int main(void)
{
    int a=3;
    int b=4;
    int c=FORMEL;
}
```

Auch das funktioniert:

```
#define ENDE while(1)

int main(void)
{
    …
    ENDE;
}
```

Wenn ein Quelltext solche Definitionen enthält, dann ersetzt der Compiler zuerst alle entsprechenden Zeichenfolgen und compiliert den so erzeugen Text dann in einem zweiten Durchlauf. Definitionen schreibt man in der Regel in Großbuchstaben, um sie besonders auffällig machen.

Die Datei avr/io.h definiert auf diese Weise alle Register des Mikrocontrollers, zum Beispiel DDRB und PORTB.

4.2.7 Zuweisungen

Die Zuweisung setzt eine Variable auf einen bestimmten Wert. Zuweisungen schreibt man mit dem Gleichheitszeichen, wie in der Mathematik:

```
x = 3;
```

Hier wird die Variable x auf den Wert drei gesetzt.

4.2.8 Variablen und Typen

Variablen sind Platzhalter für Werte, wie in der Mathematik. Technisch gesehen belegt jede Variable Platz im RAM Speicher. Der Platzbedarf einer Variable hängt von ihrem Typ ab:

Typ	**Speicherbedarf**	**Werte von**	**Werte bis**
uint8_t	1 Byte	0	255
int8_t	1 Byte	-128	127
uint16_t	2 Bytes	0	65535
int16_t oder int	2 Bytes	-32768	32767
uint32_t	4 Bytes	0	4294967295
int32_t	4 Bytes	-2147483648	2147483647
float oder double	4 Bytes	$-3{,}40 \cdot 10^{38}$	$3{,}40 \cdot 10^{38}$
char	1 Byte	Exakt ein Zeichen	
Zeiger	2 Bytes	0	65535

Fließkommazahlen, die länger als 7 Stellen sind, werden nur mit ihrem ungefähren Wert gespeichert. Die Genauigkeit von float entspricht der eines billigen Taschenrechners.

Variablen müssen vor ihrer Verwendung deklariert werden. Beispiele:

```
char meinBuchstabe;     // Deklaration
meinBuchstabe = 'A';    // Zuweisung eines Wertes

char noch_ein_Buchstabe = 'Z';

uint8_t kiste_bier;
kiste_bier = 4*5;
float preis = 4.99;
```

Diese Variablen könnten so im Speicher abgelegt sein:

Speicherzelle	45	37	34	51-54
Typ der Variable	char	char	uint8_t	float
Name der Variable	meinBuchstabe	noch_ein_buchstabe	kiste_bier	preis
Wert	'A'	'Z'	20	4.99

Variablen sind immer in dem Code-Block erreichbar, in dem sie deklariert wurden.

```
int main(void)
{
    uint8_t a;
    benutze(a);
    if (a>3)
    {
        uint8_t b;
        benutze(a);
        benutze(b);
    }
    benutze(b); // geht nicht
}
```

Die Variable a kann in der gesamten main() Funktion benutzt werden. Die Variable b kann jedoch nur innerhalb des inneren Code-Blockes verwendet werden, weil sie dort deklariert wurde.

Globale Variablen (die außerhalb aller Code-Blöcke deklariert sind) werden übrigens automatisch mit Nullen initialisiert. Variablen innerhalb von Code-Blöcken haben jedoch keinen definierten Anfangswert, es sei denn, du gibst einen an.

4.2.9 Zeiger

Zeiger sind besondere Variablen, denn sie zeigen auf Speicherzellen, in denen ein einzelner Wert oder eine Folge von Werten gespeichert ist. Bezugnehmend auf die obige Tabelle hätte ein Zeiger auf die Variable „meinBuchstabe“ den Wert 45, weil die Variable „meinBuchstabe“ in der Speicherzelle 45 liegt. Zeiger verwendet man so:

```
char* zeiger;
zeiger = &meinBuchstabe;
char wert = *zeiger;
```

In der ersten Zeile wird eine Zeigervariable mit dem Namen „zeiger“ deklariert. Das Wor char* bedeutet „Zeiger auf ein char“.

In der zweiten Zeile wird dem Zeiger die Adresse von „meinBuchstabe“ zugewiesen. Das Wort „&meinBuchstabe“ bedeutet „Adresse von meinBuchstabe“.

In der dritten Zeile holen wir uns den Wert aus der Speicherzelle, auf die der Zeiger zeigt. Dazu dient das *-Zeichen vor dem Name des Zeigers. Man nennt diesen Vorgang „Dereferenzierung“.

Zeiger benutzt man am Häufigsten in Zusammenhang mit Arrays und Funktionen.

4.2.10 Arrays

Arrays speichern eine Folge von gleichartigen Werten. Arrays haben immer eine fest definierte Größe. Beispiele:

```
char buchstaben[5];
buchstaben[0] = 'H';
buchstaben[1] = 'a';
buchstaben[2] = 'l';
buchstaben[3] = 'l';
buchstaben[4] = 'o';
```

In der ersten Zeile wird ein Array für maximal fünf Zeichen deklariert. In den Zeilen darunter wird das Array mit Werten gefüllt. Beachte, dass die Nummer der ersten Zelle immer Null ist.

4.2.11 Zeichenketten

Weil man relativ oft mit Zeichenketten programmiert, bietet die Programmiersprache C eine alternative Methode an, char-Arrays zu füllen:

```
char buchstaben[6];
buchstaben = “Hallo“;
```

Bei dieser Schreibweise muss das Array jedoch mindestens eine Zelle größer sein, als die Länge der Zeichenkette. Denn an Zeichenketten hängt der Compiler einen Null-Wert an, um deren Ende zu Kennzeichnen. Im Speicher sieht das so aus:

Zelle	0	1	2	3	4	5
Wert	'H'	'a'	'l'	'l'	'o'	0

Computer sind numerische Maschinen. Sie wandeln absolut alles in Zahlen um – auch Buchstaben. Das Wort „Hallo“ besteht in Wirklichkeit aus der Zahlenfolge 72,97,108,108,111,0. Als Programmierer müssen wir diese Zahlencodes aber nicht auswendig lernen, denn der Compiler erledigt die Umrechnung für uns.

Der englische Fachbegriff für solche Zeichenketten lautet „Strings“. Strings sind eine Folge von Buchstaben, deren Ende durch eine Null gekennzeichnet ist.

4.2.12 Kommentare

Quelltexte sollte man umfangreich kommentieren, damit andere Programmierer und man selbst später gut nachvollziehen kann, was man sich dabei gedacht hat. Im vorherigen Kapitel habe ich bereits Kommentare vorgeführt.

- Einzeilige Kommentare beginnen mit zwei Schrägstrichen
- Mehrzeilige Kommentare beginnen mit /* und enden mit */

Kommentare werden vom Compiler ignoriert. Sie erzeugen keinen Byte-Code. Beispiel:

```
/*
Hier berechnen wir, wie teuer eine einzelne Frikadelle ist.
Dazu dividieren wir den Preis der Packung durch die Anzahl der
Frikadellen.
*/

float gesamt_preis = 2.99; // Preis der Packung
uint8_t anzahl = 6;        // Frikadellen pro Packung
float einzelpreis = gesamt_preis/anzahl;
```

4.2.13 Mathematische Ausdrücke

Mathematische Ausdrücke hast du bereits gesehen. Hier werden teilweise andere Symbole verwendet, als du in der Schule im Mathematik-Unterricht gelernt hast:

- + Addition
- - Subtraktion
- * Multiplikation (aus der Schule bekannt als: × oder ·)
- / Division (aus der Schule bekannt als: ÷ oder :)
- % Modulus (=Rest einer Division)

Multiplikation und Division haben Vorrang vor Addition und Subtraktion:

- 1+2*3 ergibt 7
- (1+2)*3 ergibt 9
- 10%3 ergibt 1 (denn 10 kann man 3 mal durch 3 teilen, der Rest ist 1)

Mathematische Ausdrücke können mit Zuweisungen kombiniert werden:

- x += 3 ist identisch mit x=x+3. Entsprechend sind auch folgende Ausdrücke möglich:
- x -= 3
- x *= 3
- x /= 3
- x %= 3

4.2.14 Bedingungen

Bedingungen verwendet man, um Befehle nur dann auszuführen, wenn eine Aussage zutrifft. Für den Compiler trifft alles zu, was „wahr“ ist oder nicht Null ist.

```
if (x>100)
    y=0;
```

Nur wenn x größer als 100 ist, wird die Variable y auf Null gesetzt. Auch dieser Ausdruck ist erlaubt:, bewirkt aber etwas anderes:

```
if (x)
    y=0;
```

Hier wird die Variable y auf Null gesetzt, wenn x nicht Null ist. Wahlweise kann man einen Befehl für den Fall angeben, dass die Bedingung nicht erfüllt ist:

```
if (x>100)
    y=0;
else
    y=1;
```

Bedingungen kann man auch verketten:

```
if (x>100)
    y=0;
else if (x>50)
    y=1;
else if (x>10)
    y=2;
else
    y=3;
```

Wenn mehrere aufeinander folgende Befehle an eine Bedingung geknüpft werden sollen, verwendet man geschweifte Klammern:

```
if (x>100)
{
    tu etwas;
    tu noch etwas;
    und noch etwas;
}
else
{
    tu etwas anderes;
}
```

4.2.15 Vergleiche

Vergleiche benutzt man, um Bedingungen zu formulieren. Beispiele:

- if (a==b)… Wenn a den gleichen Wert wie b hat, dann …
- if (a!=b)… Wenn a nicht den gleichen Wert wie b hat, dann …
- if (a<b)… Wenn a kleiner als b ist, dann …
- if (a>b)… Wenn a größer als b ist, dann …
- if (a<=b)… Wenn a kleiner oder gleich wie b ist, dann …
- if (a>=b)… Wenn a größer oder gleich wie b ist, dann …
- if (a)… Wenn a nicht Null ist, dann …
- if (!a)… Wenn a Null ist, dann

Achtung: Schreibe nicht (a=b), denn das wäre eine Zuweisung. Du würdest dann den Wert von der Variable b in die Variable a kopieren, anstatt sie miteinander zu vergleichen.

Mehrere Bedingungen kann man durch logische Operatoren miteinander verknüpfen:

- (a || b) bedeutet: a oder b

- (a && b) bedeutet: a und b

Beispiele:

```
uint8_t gross = (zentimeter>90); // geht auch ohne Klammern
uint8_t riesig = (zentimeter>200);

if (gross && !riesig)
{
   schreibe("Er ist gross aber nicht riesig");
}

uint8_t x=12;
uint8_t y=3;

if (x==3 || y==3)
{
    schreibe("Eine der beiden Variablen hat den Wert 3");
}
```

4.2.16 Boolesche Werte

Boolesche Werte können nur zwei Zustände haben, nämlich

- 1 oder 0, bzw. wahr oder falsch

Jede Bedingung und jeder Vergleich ist ein boolescher Wert. Die Programmiersprache C kennt leider keinen eigenen Datentyp für diese Werte. Sie speichert boolesche Werte daher stets in Integer-Variablen.

- (1>2) ist falsch, also 0
- (2>1) ist wahr, also 1

Wenn in den Klammern ein numerischer Wert steht (anstatt ein Vergleich), interpretiert die Programmiersprache den Wert Null als „falsch“ und alle anderen Werte als „wahr“.

- (5) ist wahr
- (3+5) ist wahr
- (0) ist falsch
- (4-2-2) ist falsch
- aber: (4-2-2==0) ist wahr

4.2.17 Schleifen

While-Schleifen wiederholen Befehle mehrmals, solange eine Bedingung zutrifft:

```
uint8_t i=0;
while (i<100)
{
    bingbong();
    i=i+1;
}
```

In diesem Fall macht der Computer 100 mal BingBong. Dabei zählt er in der Variable i die Wiederholungen mit. Bei jedem Schleifendurchlauf wird die Variable i um 1 erhöht.

Das kann man auch anders machen:

```
for (uint8_t i=0; i<100; i=i+1;)
{
```

```
    bingbong();
}
```

Die For-Schleife funktioniert ganz ähnlich. In den runden Klammern sind drei Ausdrücke erforderlich, die man jeweils mit Semikolon abschließt:

1. Initialisierung einer Variablen, wird vor der ersten Durchlauf ausgeführt. (i=0)
2. Bedingung, wird vor jedem Schleifendurchlauf kontrolliert. (i<101)
3. Modifikation, wird nach jedem Schleifendurchlauf ausgeführt. (i=i+1)

Programmierer bevorzugen im Allgemeinen die For-Schleife, wenn dabei eine Variable kontinuierlich hochgezählt wird, wie im obigen Beispiel. Für alle anderen Fälle bevorzugen sie While-Schleife. Programmierer verwenden für Zählvariablen gerne die Buchstaben i, j und k. Wenn das alle so machen, fällt es leichter, fremde Quelltexte zu lesen.

Im Gegensatz zur While-Schleife prüft die Do-While-Schleife ihre Bedingung erst nach dem ersten Durchlauf. Sie wird also immer mindestens einmal durchlaufen, auch wenn die Bedingung von Anfang an nicht zutrifft.

```
uint8_t i=0;
do
{
    bingbong();
    i=i+1;
}
while (i<100);
```

Alle drei Schleifen (while, for und do-while) kann man mit den folgenden Befehlen abbrechen:

- break; bricht die ganze Schleife ab. Als nächstes wird der erste Befehl hinter der Schleife ausgeführt.
- continue; bricht den aktuellen Schleifendurchlauf ab. Es wird mit dem nächsten Durchlauf weiter gemacht (wird selten verwendet).

4.2.18 Increment und Decrement

Weil man in Schleifen häufig eine Variable hoch oder runter zählt, bietet die Programmiersprache dazu eine besonders auffällige Scheibweise an:

- `i++`
- `i--`
- `++i`
- `--i`

Mit `i++` erhöhst du den Wert der Variable i, es wirkt sich also aus, wie `i=i+1` oder auch `i+=1`. Der Unterschied zwischen i++ und ++i liegt in dem Zeitpunkt, wann die Variable verändert wird:

```
if (i++ == 3)
    klingeling();
```

Hier wird zuerst geprüft, ob i den Wert 3 hat. Dann wird i um eins erhöht und dann klingelt es, wenn die Bedingung erfüllt war (also wenn i VOR der Erhöhung den Wert 3 hatte).

```
if (++i == 3)
    klingeling();
```

In diesem Fall wird die Variable i zuerst um eins erhöht und dann erst mit 3 verglichen. Noch ein anderes Beispiel:

```
uint8_t i=4;
while (--i)
{
  klingeling();
}
```

Das Programm wird dreimal klingeln, denn die Variable i wird vor jedem Schleifendurchlauf um eins verringert. Beim vierten mal erreicht sie den Wert Null, dann wird die Schleife nicht mehr ausgeführt (denn Null wird als „falsch“ interpretiert). Anders verhält es sich, wenn du i-- schreibst:

```
uint8_t i=4;
while (i--)
{
  klingeling();
}
```

Dieses mal klingelt das Programm viermal, denn der Wert von i wirst zuerst geprüft (ob er ungleich Null ist) und erst danach wird er verringert.

4.2.19 Switch-Case Anweisung

Die switch-case Anweisung ist eine Alternative für verkettete If-Bedingungen.

```
switch (x)
{
    case 1:
        befehle …;
        break;
    case 2:
        befehle …;
        break;
    case 3:
    case 4:
        befehle …;
        break;
    default:
      befehle …;
}
```

Die Switch Anweisung vergleicht die angegebene Variable (in diesem Fall x) mit den case-Werten und führt die Befehle des passenden Case-Blocks aus. Jeder Case-Block muss mit dem break Befehl beendet werden, sonst würden die Befehle des folgenden Case-Blockes auch noch ausgeführt werden.

Im obigen Beispiel gibt es einen gemeinsamen case-Block für den Fall, dass x==3 oder x==4 ist. Der default-Block wird ausgeführt, wenn kein case-Wert zutrifft. Der default-Block ist optional, man darf ihn weg lassen.

4.2.20 Bitweise Operatoren

Die Programmiersprache C enthält einige Operatoren, die einzelne Bits aus Variablen bearbeiten. Für Mikrocontroller sind diese Operatoren enorm wichtig, da fast jedes Register bitweise programmiert wird.

4.2.20.1 Was ist ein Bit?

Ein Bit ist die kleinste denkbare Speichereinheit. Ein Bit kann nur zwei Werte speichern: 1 und 0, oder anders gesagt „an“ und „aus“. Wenn man acht Bits zusammen fügt, hat man ein Byte.

Bits	7	6	5	4	3	2	1	0
Byte								

AVR Mikrocontroller nennt gehören zur Klasse der 8-Bit Prozessoren, weil jede Speicherzelle aus acht Bits besteht. Sie verarbeiten (fast) immer 8-Bits gleichzeitig.

Die Programmiersprache C kennt folgende bitweise Operatoren:

4.2.20.2 Und

```
x = a & b
```

Im Ergebnis sind nur die Bits auf 1 gesetzt, die in beiden Operanden auf 1 stehen:

Operand a	1	0	0	1	1	1	0	1
Operand b	0	0	1	1	0	1	1	0
Ergebnis x	0	0	0	1	0	1	0	0

4.2.20.3 Oder

```
x = a | b
```

Im Ergebnis sind alle Bits auf 1 gesetzt, die in wenigstens einem Operand auf 1 stehen:

Operand a	1	0	0	1	1	1	0	1
Operand b	0	0	1	1	0	1	1	0
Ergebnis x	1	0	1	1	1	1	1	1

4.2.20.4 Exklusiv-Oder

```
x = a ^ b
```

Im Ergebnis sind alle Bits auf 1 gesetzt, die in genau einem Operand auf 1 stehen:

Operand a	1	0	0	1	1	1	0	1
Operand b	0	0	1	1	0	1	1	0
Ergebnis x	1	0	1	0	1	0	1	1

4.2.20.5 Negation

```
x = ~a
```

Im Ergebnis sind alle Bits umgekehrt:

Operand a	1	0	0	1	1	1	0	1
Ergebnis x	0	1	1	0	0	0	1	0

4.2.20.6 Schieben

```
x = a >> n
y = a << n
```

Die Schiebe-Operatoren verschieben alle Bits um n Positionen nach links oder rechts. Die entstehende Lücke wird mit Nullen aufgefüllt.

Beispiel für x = a >> 2:

Operand a	1	0	0	1	1	1	0	1
Ergebnis x	0	0	1	0	0	1	1	1

Beispiel für x = a << 2:

Operand a	1	0	0	1	1	1	0	1
Ergebnis x	0	1	1	1	0	1	0	0

4.3 Umgang mit dem Simulator

Das soll jetzt erst einmal genug Theorie sein. Wahrscheinlich raucht dir jetzt der Kopf und du hast höchstens 20 % vom vorherigen Kapitel behalten. Das ist OK. Mit der Zeit wirst du von ganz alleine die Regeln der Programmiersprache C auswendig lernen. Mache erst einmal mindestens einen Wochenende lang Pause, bevor du weiter lernst.

...

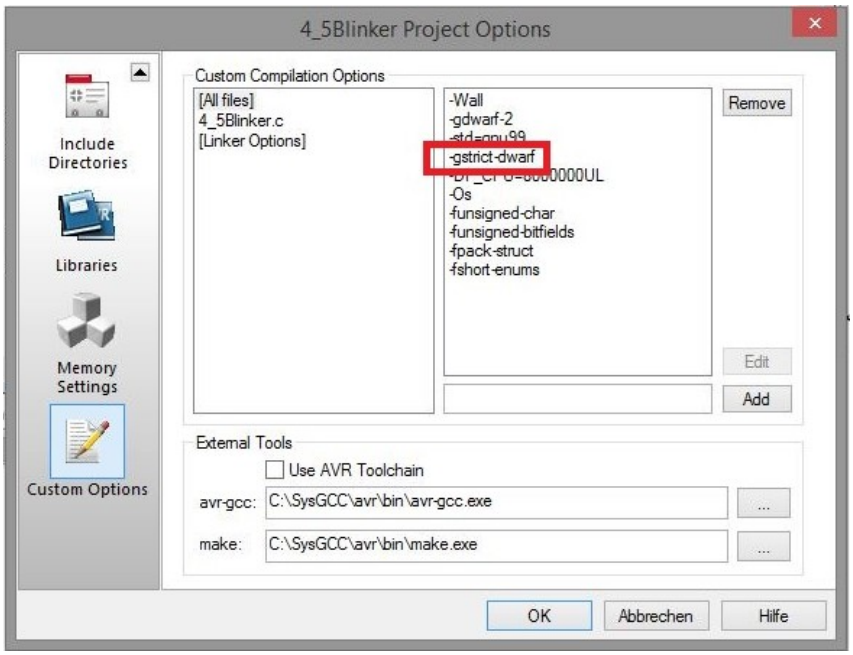
Du hast ein Grundgerüst für das Projekt „Test“ angelegt. Die Quelltextdatei Test.c sieht so aus:

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

int main(void)
{
    …
}
```

Dieses Programm wollen wir jetzt mit einigen konkreten Anwendungsbeispielen füllen und im Simulator austesten. Danach werden wir ein zweites Programm für den „echten“ Mikrocontroller schreiben.

Anmerkung:

Falls der Simulator/Debugger bei den folgenden Arbeiten häufig abstürzt, könnte folgende Einstellung helfen:



Sollte das nicht helfen, empfehle ich die Verwendung von WinAVR als Toolchain, damit hatte ich nämlich noch nie Probleme.

4.3.1 Seiteneffekte durch Compiler-Optimierung

Normalerweise konfrontiert man Anfänger nicht schon während der ersten Lernstunden mit Problemen. In diesem Fall möchte ich eine Ausnahme machen, um zu vermeiden, dass du deinen Computer voreilig auseinander nimmt und denkst, er sei kaputt.

Beim Testen mit dem Simulator passieren manchmal „seltsame“ Dinge, weil der Compiler das Programm stärker verändert, als erwartet. Probiere das aus:

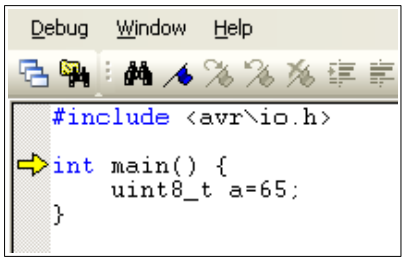
```
#include <avr/io.h>

int main(void)
{
    uint8_t a=65;
}
```

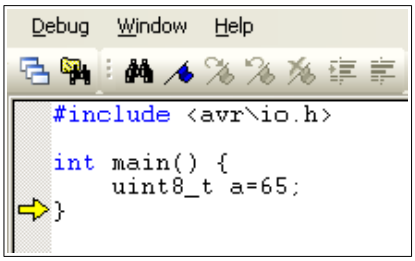
Das Programm deklariert eine Variable mit Namen a und weist ihr den Wert 65 zu. Wir müssten also eine Speicherzelle mit diesem Wert wieder finden, wenn das Programm ausgeführt wird. Drücke F7, um den Quelltext in Byte-Code zu übersetzen. Der Compiler warnt uns:

```
Build started 11.12.2011 at 17:52:17
avr-gcc  -mmcu=attiny13 -Wall -gdwarf-2 -Os -std=c
../Test.c: In function 'main':
../Test.c:4: warning: unused variable 'a'
```

Was er damit meint, ist: Du hast der Variablen a zwar einen Wert zugewiesen, aber benutzt sie danach nicht mehr. Der Compiler vermutet, dass das Programm unvollständig ist (womit er Recht hat), und darum warnt er dich. Wir wollen das Programm trotzdem im Simulator ausprobieren.

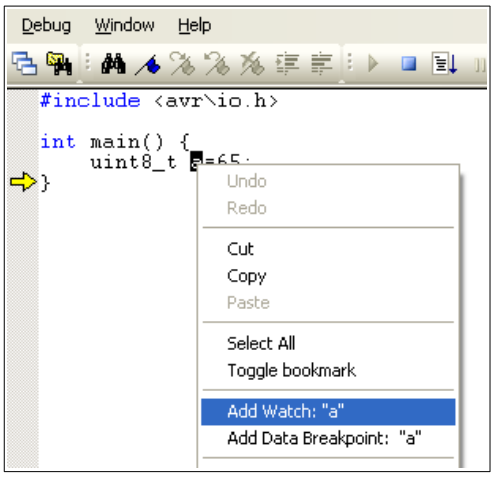
Klicke auf den Menüpunkt Debug/Start Debugging. Es erscheint ein gelber Pfeil, der anzeigt an welcher Stelle das Programm startet. Drücke nun die Taste F11, um den ersten Befehl auszuführen:

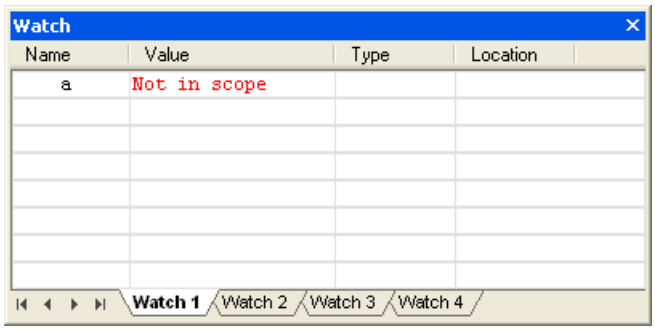




Schon steht der gelbe Pfeil am Programm-Ende. Die Zuweisung a=65 hat er offenbar einfach übersprungen. Lass uns überprüfen, welchen Wert die Variable a jetzt hat.

Klicke dazu mit der rechten Maustaste auf die Variable a und wähle den Menüpunkt „Add Watch“:



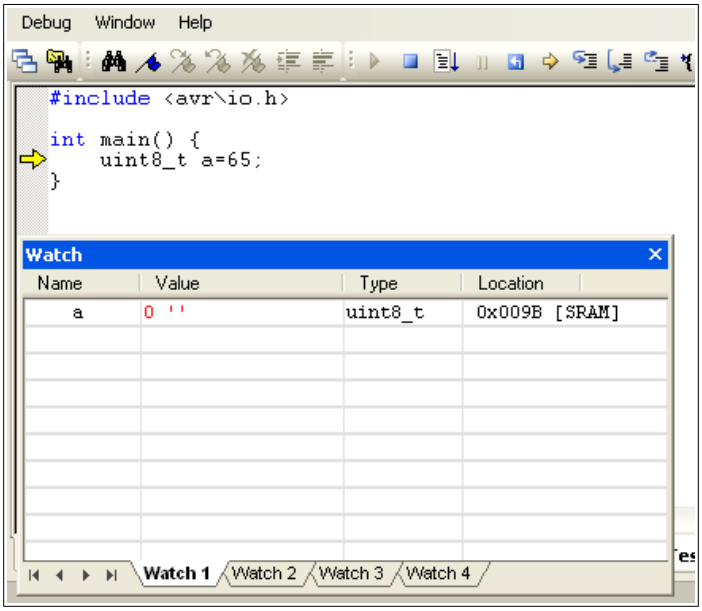


Es erscheint ein neues Fenster, in dem drin steht, dass die Variable „Not in scope“ (nicht im Anwendungsbereich) ist. Mit anderen Worten: die Variable existiert nicht. Wie kann das sein?

Der Compiler hat bemerkt, dass dein Programm den Inhalt der Variable a nirgends auswertet. Alles, was nicht verwendet wird, entfernt der Compiler im Rahmen der automatischen Optimierung. Die ganze vierte Zeile „uint8_t a=65;“ wurde durch den Compiler entfernt. Darum hat der Simulator sie übersprungen und darum können wir den Wert der Variablen auch nicht einsehen.

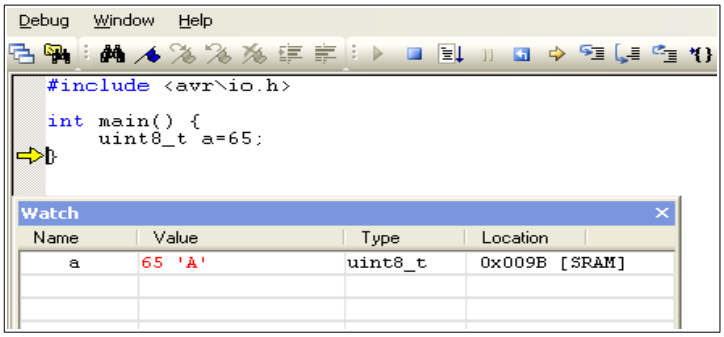
Damit unsere Übungen simulierbar werden, deaktivieren wir also die Optimierung: Klicke auf den Menüpunkt Project/Configuration Options. Stelle die Optimization auf „-O0“ um. Zurück im Quelltext-Editor drücke F7, um den Quelltext erneut in Byte-Code zu übersetzen. Starte den Simulator wieder mit dem Menüpunkt Debug/Start Debugging und drücke F11, um den ersten Befehl auszuführen.

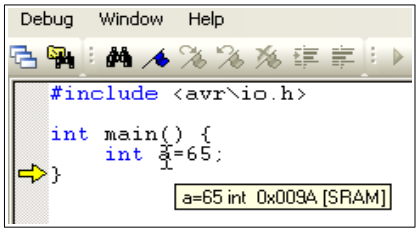
Dieses mal wird die Zeile 4 nicht mehr übersprungen.



Du siehst im Watch-Fenster, dass die Variable a zunächst den Wert 0 hat und dass sie in der Speicherzelle 0x009B liegt.

Drücke F11, um die Zuweisung a=65 auszuführen. Im Watch-Fenster kannst du sehen, dass die Variable a nun den Wert 65 hat. Der Buchstabe 'A' erscheint daneben, weil der numerische Wert von 'A' der Zahl 65 entspricht. 'B' hat den Wert 66, 'C' hat den Wert 67, usw.





Variablen kann man auch ohne das Watch-Fenster einsehen, und zwar durch einfaches Zeigen. Zeige auf die Variable a und warte einen Moment. Dann erscheint ein hell-gelber Kasten:

Stoppe den Simulator durch den Menüpunkt Debug/Stop Debugging.

4.3.2 Mathematische Ausdrücke

Lass uns einige Mathematische Ausdrücke ausprobieren. Wir erweitern das vorhandene Programm:

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

int main(void)
{

    // Variable und Zuweisung
    uint8_t  a = 65;

    // Mathematische Ausdrücke
    float    b = 10/3;
    uint8_t  c = 10/3;
    uint8_t  d = 1+2*4;
    uint8_t  e = (1+2)*4;
    uint8_t  f = 100*100;
    uint32_t g = 12000*10;
    uint32_t h = 350000*10;
}
```

Drücke wieder F7, um den Quelltext in Byte-Code zu übersetzen. Starte dann den Simulator durch den Menüpunkt Debug/Start Debugging.

Drücke die Taste F11 so oft, bis der gelbe Pfeil in der letzten Zeile (mit der geschweiften Klammer) steht. Schau dir dann die Inhalte der Variablen a, b, c, d, e, f , g und h an:

a = 65
b = 3
c = 3
d = 9
e = 12
f = 16
g = 4294956224
h = 3500000

Einige Werte sind offensichtlich falsch. Warum?

Die Variable b hat nicht den Wert $3.\overline{33}$, weil der Compiler einen falschen Algorithmus eingesetzt hat. Die Zahlen 10 und 3 sind beides Integer Zahlen (ohne Nachkomma-Stellen), darum hat der Compiler nicht erkannt, dass ein Algorithmus für Fließkommazahlen benötigt wird. Um diesen Fehler zu korrigieren, hast du zwei Möglichkeiten:

- Schreibe „(float) 10/3“, um dem Compiler anzuzeigen, dass er für die Division einen Algorithmus für Fließkommazahlen verwenden soll.
- Schreibe „10.0/3“ oder „10/3.0“, denn der Compiler verwendet dann für die Division einen Algorithmus für Fließkommazahlen weil einer der beiden Operaden eine Fließkommazahl ist.

Die Variable c=3 ist richtig, denn die Variable c ist ein Integer-Typ, der sowieso keine Nachkommastellen speichern kann. Der Rest wird einfach abgeschnitten.

Die Variable f hat einen falschen Wert, denn 100·100 müsste 10000 ergeben. Der Wert 10000 liegt allerdings nicht im erlaubten Bereich für den Datentyp uint8_t. Das lässt sich so korrigieren:

- Ändere den Datentyp der Variablen auf uint16_t.

Die Variable g hat einen völlig falschen Wert. 12000·10 ist 120000 und nicht 4294956224. Dieser falsche Wert kommt zustande, weil der Compiler einen 16-Bit Algorithmus verwendet hat, das Ergebnis aber für 16-Bit Variablen zu groß ist. Lösung:

- Teile dem Compiler mit, dass er einen 32-Bit Algorithmus verwenden soll, indem du „(uint32_t) 12000*10“ schreibst.

Das probieren wir gleich aus:

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

int main(void)
{

    // Variable und Zuweisung
    uint8_t  a = 65;

    // Mathematische Ausdrücke
    float    b = 10/3;
    uint8_t  c = 10/3;
    uint8_t  d = 1+2*4;
    uint8_t  e = (1+2)*4;
    uint8_t  f = 100*100;
    uint32_t g = 12000*10;
    uint32_t h = 350000*10;

    // Korrigierte mathematische Ausdrücke
    float    b2 = (float) 10/3;
    float    b3 = 10.0/3;
    uint16_t f2 = 100*100;
    uint32_t g2 = (uint32_t) 12000*10;
}
```

Führe auch dieses Programm im Simulator aus und schaue dir die Variablen b2, b3 und f2 an:

b2 = 3.3333333
b3 = 3.3333333
f2 = 10000

3.3333333 immer noch nicht ganz korrekt, denn es müsste $3.\overline{33}$ heißen. Doch Variablen vom Typ float haben nur eine begrenzte Genauigkeit von 8 Stellen. Die Programmiersprache C enthält keine Algorithmen für absolut genaue Fließkommazahlen.

Einerseits optimiert der Compiler den Programmcode auf sehr clevere Weise, andererseits stellt er sich bei der Auswahl von Algorithmen für Mathematische Ausdrücke überraschend dumm an. Woran liegt das?

Der Grund ist ganz einfach. Zu dem Zeitpunkt, wo eine Formel compiliert wird, kann der Compiler nicht wissen, wie groß das Ergebnis werden wird. Denn er berechnet die Formel nicht, sondern er erzeugt lediglich ein Programm, dass sie berechnen wird.

Die Wahl des Algorithmus macht der Compiler von den Operanden ab, und zwar in dieser Reihenfolge:

1. Ist ein Operand eine Fließkommazahl, verwendet er einen float-Algorithmus.
2. Ansonsten: Ist ein Operand ein Integer, der größer als 16 Bit ist, verwendet er den dazu passenden Integer Algorithmus.
3. Ansonsten verwendet der Compiler einen 16-Bit Integer Algorithmus.

Schauen wir uns dieses Regelwerk für den Ausdruck 12000·10 an. 12000 passt in einem 16-Bit Integer, darum verwendet der Compiler gemäß der 4. Regel einen 16-Bit Algorithmus. Wenn das Programm ausgeführt wird, versagt der Algorithmus jedoch, denn das Ergebnis von 12000·10 ist für 16-Bit zu groß.

In diesen Fällen kann man manuell den Notwendigen Datentyp angeben, so dass der Compiler den richtigen Algorithmus wählt: (uint32_t) 12000*10

4.3.3 Schleifen im Simulator

Um Schleifen zu üben, kannst du die bisherigen Tests löschen. Wir fangen wieder mit einem Leeren Grundgerüst an. Lass uns folgendes versuchen: Eine Variable x soll von Null bis 10 hochgezählt werden und dann wieder herunter auf Null.

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

int main(void)
{
    uint8_t x=0;

    do
    {
        x=x+1;
    }
    while (x<10);

    do
    {
        x=x-1;
    }
    while (x>0);
}
```

Hier wird zuerst eine 8-Bit Integer Variable deklariert und mit dem Wert Null initialisiert. Die erste Schleife wird so oft durchlaufen, bis x=10 ist. Dann wird die zweite Schleife so oft durchlaufen, bis x=0 ist.

Ich möchte dir jetzt zeigen, wie man das noch kürzer schreiben kann:

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

int main(void)
{
    uint8_t x=0;
```

```
    while (x<10)
    {
        x++;
    }

    while (x>0)
    {
        x--;
    }
}
```

Anstelle der do-while Schleife kommt dieses mal eine while Schleife zum Einsatz. In diesem Anwendungsfall ist sie gleichermaßen geeignet. Anstelle von x=x+1 wird der Inkrement-Operator verwendet. Es geht aber noch kürzer:

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

int main(void)
{
    uint8_t x=0;

    while (++x < 10);
    while (--x > 0);
}
```

In der ersten Schleife wird die Variable zuerst um eins erhöht und dann wird die Schleife wiederholt, wenn x kleiner als 10 ist. Die zweite Schleife verringert x und prüft dann, ob x den Wert Null hat. Wenn nicht, wird sie wiederholt.

Diese dritte Variante ist nicht so gut lesbar. Programmierer sind in der Regel Schreibfaul, man sollte es aber nicht übertreiben, sonst blickt man ein paar Monate später durch seinen eigenen Quellcode nicht mehr durch.

Übrigens erzeugen alle drei Varianten praktisch den gleichen Byte-Code, sofern die Optimierung durch den Compiler eingeschaltet ist. Weniger Tippen bedeutet nicht zwangsläufig, dass das Programm kleiner oder effizienter wird.

Wir haben die for-Schleife noch nicht ausprobiert:

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

int main(void)
{
    for (uint8_t x=0; x<10; x++);
    for (uint8_t x=10; x>0; x--);
}
```

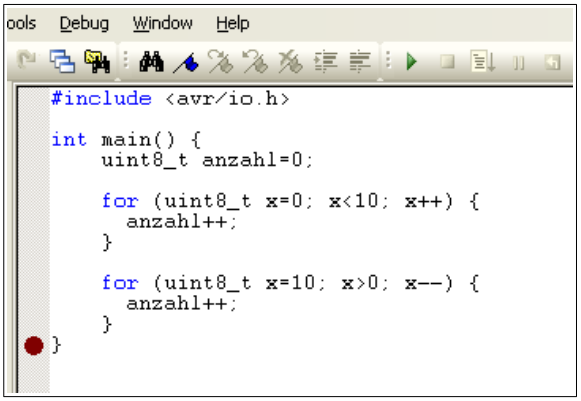
Das ist natürlich ein bisschen Unsinnig. Normalerweise würden die for-Schleifen jeweils einen Code-Block enthalten, der 10 mal ausgeführt wird. Zum Beispiel so etwas:

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

int main(void)
{
    uint8_t anzahl=0;

    for (int x=0; x<10; x++)
    {
        anzahl++;
    }
```

```
        for (int x=10; x>0; x--)
        {
            anzahl++;
        }
    }
```

Jetzt wird die Variable x zuerst von 0 auf 10 hochgezählt, und dann wieder herunter. Dabei zählt die Variable „anzahl“ die Schleifendurchläufe.

Wir wollen dieses Programm anders simulieren, als bisher. Setze einen Unterbrechungspunkt in die letzte Zeile, indem du mit der rechten Maustaste auf den grauen Rand auf Höhe der letzten Zeile klickst (wo im folgenden Foto der rote Punkt ist) und dann den Menüpunkt „Toggle Bookmark“ wählst:



Starte nun den Debugger wie gehabt durch den Menüpunkt Debug/Start Debugging. Der gelbe Pfeil steht zuerst wie gewohnt in der ersten Zeile der main() Funktion. Drücke jetzt F5, oder den Menüpunkt Debug/Run. Das Programm wird dann bis zu dem roten Punkt durchlaufen.

Du kannst auf diese Weise beliebig viele Unterbrechungspunkte setzen. Jedesmal, wenn du F5 drückst, wird das Programm bis zum nächsten Unterbrechungspunkt ausgeführt.

Schau dir jetzt den Inhalt der Variable „anzahl“ an: sie hat den Wert 20. Du hast das Programm dieses mal nicht mehr Zeilenweise ausgeführt, aber am Wert der Variablen „anzahl“ kannst du dennoch ablesen, dass es korrekt funktioniert hat.

4.4 Andere Zahlensysteme

Um eigene Programme zu schreiben, genügt es in der Regel, mit dezimalen und binären Zahlen vertraut zu sein. Erfahrene Programmierer benutzen darüber hinaus oft hexadezimale Zahlen.

4.4.1 Binäres Zahlensystem

Wenn man Zahlen so schreibt, dass jede Ziffer einem Bit entspricht, spricht man von Binärzahlen. Zum Beispiel ist 10111001 eine Binärzahl. Bei Binärzahlen kann jede Ziffer nur die Werte 0 und 1 haben. 0 bedeutet „aus“ und 1 bedeutet „an“.

Die Bits eines Bytes werden von rechts nach links nummeriert, beginnend mit 0. Das ganz rechte Bit (also das Bit 0) hat den geringsten Wert. Das ganz linke Bit hat den höchsten Wert. Um Binärzahlen manuell in gewöhnliche Dezimalzahlen umzurechnen, musst du die Werte der Bits kennen:

Bit Nr.	7	6	5	4	3	2	1	0
Wert	128	64	32	16	8	4	2	1

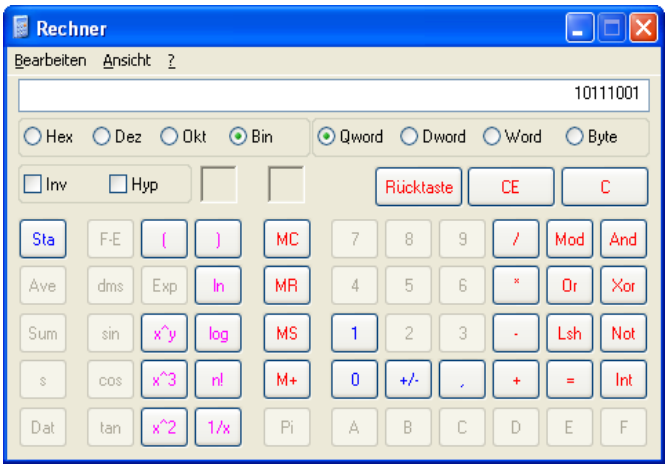
Die Binärzahl 10111001 rechnest du so in eine Dezimalzahl um:

Summe von	1 mal 128	0 mal 64	1 mal 32	1 mal 16	1 mal 8	0 mal 4	0 mal 2	1 mal l

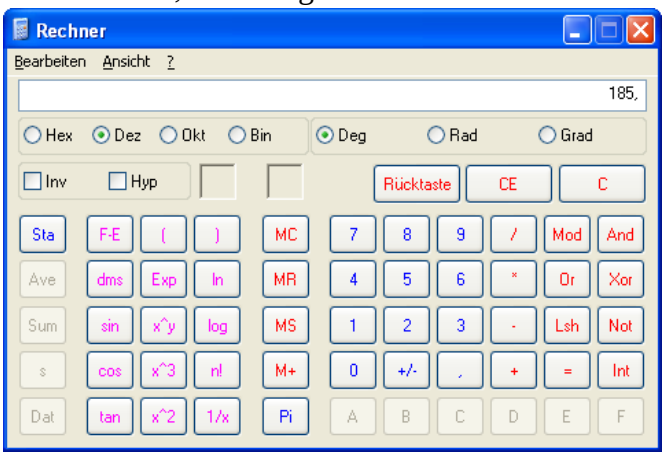
128 + 32 + 16 + 8 + 1 = 185

Diese Rechnung ist Mühsam, daher benutzen wir einen wissenschaftlichen Taschenrechner.

Der Taschenrechner von Windows kann das auch. Schalte im Ansicht-Menü auf die wissenschaftliche oder Programmierer-Ansicht um. Aktiviere den „Bin“ Modus und gib die Zahl 10111001 ein:



Schalte dann in den „Dez“ Modus um, dann zeigt der Rechner diese Zahl in dezimaler Form an:



Der GNU C Compiler verlangt, dass man Binärzahlen ein „0b“ voran stellt, damit er sie als Binärzahl erkennt. In einem C-Programm musst du die obige Zahl also als 0b10111001 schreiben.

4.4.2 Hexadezimales Zahlensystem

Wenn man Zahlen so schreibt, dass jede Ziffer aus genau vier Bits besteht, spricht man von hexadezimalen Zahlen. Zum Beispiel ist B9 eine hexadezimale Zahl, die dem dezimalen Wert 185 entspricht. Jede Hexadezimale Ziffer kann 16 Werte darstellen:

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

Jede Hexadezimale Ziffer besteht aus genau vier Bits. Die folgende Tabelle hilft bei der Übersetzung von hexadezimalen in dezimale oder binäre dezimale Ziffern:

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dez	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Da eine hexadezimale Ziffer aus genau vier Bits besteht, stellen zwei hexadezimale Ziffern ein Byte dar.

Um zweistellige hexadezimale Zahlen ins dezimale System umzustellen, multiplizierst du den dezimalen Wert der linken Ziffer mit 16 und addierst den dezimalen Wert der rechten Ziffer. Beispiel für B9:

B: 11·16 = 176

9: = 9

Summe: 185

Hexadezimale Zahlen sind in der Mikrocontroller-Programmierung beliebt, weil man sie anhand der obigen Tabelle leicht in binäre Zahlen übersetzen kann.

Hex	Binär	Dezimal
B9	1011 1001	176
C2	1100 0010	194
10	0001 0000	16

Der GNU C Compiler verlangt, dass man hexadezimalen Zahlen ein „0x“ voran stellt, damit er sie erkennt. In einem C-Programm musst du die obige Zahl also als 0xB9 schreiben.

4.5 Blinkmuster programmieren

Das erste Programm für deinen „echten“ Mikrocomputer hattest du einfach abgeschrieben, ohne genau zu verstehen, wie es funktioniert. Nun sollst du ein weiteres Programm schreiben. Tippe es dieses mal nicht einfach nur ab, sondern versuche, dessen Funktion nachzuvollziehen.

Aufgabe: Die Leuchtdiode soll 5 mal langsam blinken, dann eine Weile lang sehr schnell blinken und dann soll das Programm wieder von vorne beginnen.

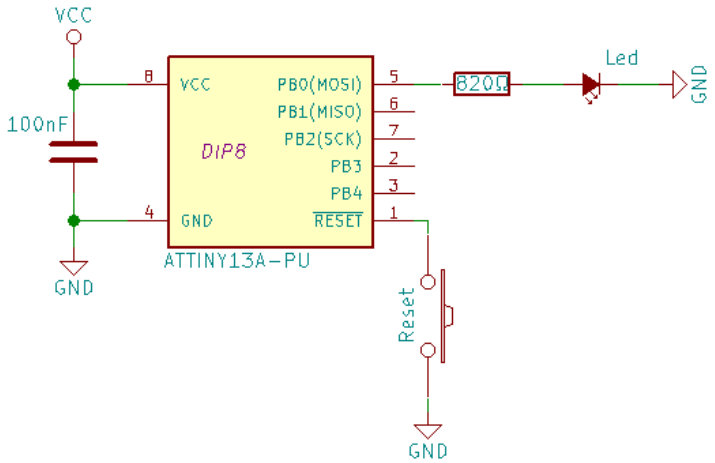
```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRB=1;
    while(1)
    {
        for (uint8_t i=0; i<5; i++)
        {
            PORTB=0;
            _delay_ms(500);
            PORTB=1;
            _delay_ms(500);
        }
        for (uint8_t i=0; i<30; i++)
        {
            PORTB=0;
            _delay_ms(50);
            PORTB=1;
            _delay_ms(50);
        }
    }
}
```

Stelle sicher, dass die Optimierung durch den Compiler wieder aktiviert wird und dass die Taktfrequenz korrekt eingestellt ist, sonst funktioniert die delay-Funktion nicht richtig (das steht in der Dokumentation der avr-libc Library):

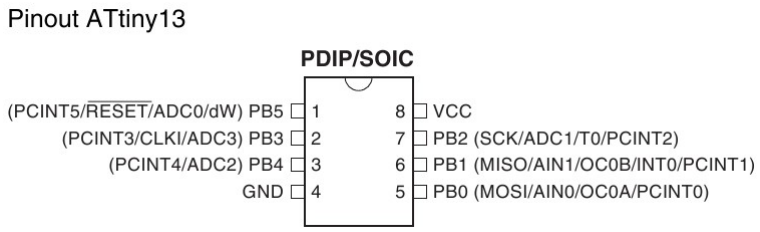
- Menü Project/Configuration Options
- Frequence= 1200000 Hz bei ATtiny13 oder 1000000 Hz bei ATtiny25, 45 und 85.
- Optimization = -Os

Compiliere das Programm mit der Taste F7 und übertrage dann die erzeugte *.hex Datei mit deinem ISP Programmer in den Mikrocontroller. Unmittelbar nach dem Übertragen des Programms sollte die LED immer abwechselnd 5 mal langsam und dann 30 mal schnell blinken.

Um nachzuvollziehen, wie das Ganze funktioniert, brauchst du das Datenblatt (Datasheet) des Mikrocontrollers. Du findest auf auf der Webseite von Atmel. An dem Schaltplan der Platine kannst du ablesen, dass die Leuchtdiode an Pin 5 angeschlossen ist:



.



Der Anschluss von Pin 5 heißt „Port B0“. Die anderen Namen in Klammern gelten, wenn bestimmte Sonderfunktionen aktiviert sind, was bei diesem Programm jedoch nicht der Fall ist. Standardmäßig ist bei allen AVR Controllern lediglich die RESET Funktion aktiviert. Außerdem sind standardmäßig alle Pins als Eingang geschaltet, so dass kein Strom heraus kommt.

Das Programm muss also den Port B0 als Ausgang umschalten. Das macht der Befehl „DDRB=1“. DDRB ist eine Register-Variable. Im Gegensatz zu normalen selbst deklarierten Variablen sind alle Register-Variablen fest vorgegeben, und zwar in der include Datei avr/io.h. Register-Variablen speichern nicht einfach nur irgendwelche Werte, sondern sie dienen dazu, Ein-/Ausgabefunktionen zu steuern. Im Datenblatt des Chips sind alle Register detailliert beschrieben.

Das Register DDRB bestimmt, welche Anschlüsse als Ausgang verwendet werden. Für jeden Anschluss gibt es ein entsprechendes Bit in diesem Register:

Register DDRB	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Port	nicht verwendet		B5	B4	B3	B2	B1	B0

Wenn du den Wert 1 (= binär 00000001) in dieses Register schreibst, dann wir der Port B0 als Ausgang konfiguriert und alle anderen Pins bleiben Eingänge. Der Wert 63 (=binär 00111111) würde alle Pins als Ausgang konfigurieren. Spätestens jetzt dürfte klar sein, warum ein wissenschaftlicher Taschenrechner zu den Arbeitsmitteln eine Mikrocontroller-Programmierers gehört.

Nachdem der Port B0, an dem die Leuchtdiode hängt, konfiguriert ist, kommt eine endlose while() Schleife. Sie wird endlos wiederholt, weil die Bedingung in den Klammern einfach eine eins ist. Eins ist nicht Null daher ist diese Bedingung immer wahr. Die Schleife wird niemals enden.

Innerhalb dieser Endlos-Schleife, haben wir zwei for() Schleifen, die nacheinander ausgeführt werden. Die erste lässt die Leuchtdiode langsam blinken, denn sie enthält lange Verzögerungen mittels _delay_ms(500). Die Leuchtdiode geht 500ms lang an, dann 500ms lang aus, und das Ganze wird fünf mal wiederholt. Die zweite Schleife verwendet viel kürzere delay's und mehr Wiederholungen.

Das Einschalten der Leuchtdiode übernimmt der Befehl PORTB=1. Ähnlich, wie das DDRB Register hat auch das PORTB Register für jeden Anschluss ein Bit. Der Wert 1 schaltet den Ausgang auf die Versorgungsspannung (z.B. 3,6V) und der Wert 0 schaltet ihn auf Null Volt. Folglich geht die LED bei DDRB=1 an, und bei DDRB=0 geht sie wieder aus.

4.6 Umgang mit dem Simulator (Fortsetzung)

Dieses Programm kannst du im Simulator nicht ausführen, weil:

1. die _delay_ms() Funktion verlangt, dass du die Optimierung des Compiler aktivierst, was der Simulator nicht mag.
2. der Simulator die Taktfrequenz nicht simulieren kann. Auf meinem Computer ist der Simulator etwa 4x langsamer, als der echte Mikrocontroller.
3. Der Simulator den Zustand des PORTB nicht anzeigt, während das Programm läuft, sondern nur, wenn man es durch anhält (pausiert). Du wirst also kein blinkendes Bit in PORTB sehen können.

Wenn du einmal ein Programm schreibst, das unerwartet falsch funktioniert, dann analysiere den Fehler so:

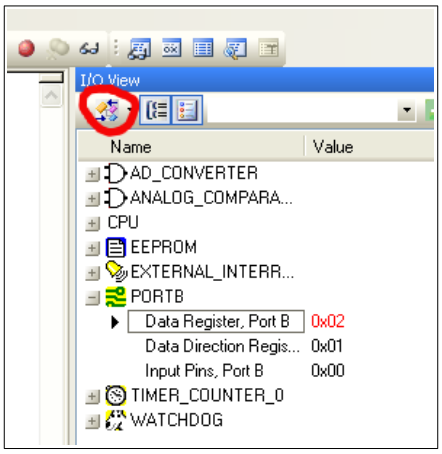
- Reduziere das Programm auf so wenig Code, wie möglich. Es sollte nur der fehlerhafte Teil untersucht werden.
- Kommentiere alle delay Aufrufe aus.
- Deaktiviere die Optimierung des Compilers
- Führe dann den reduzierten Programmcode im Debugger/Simulator aus

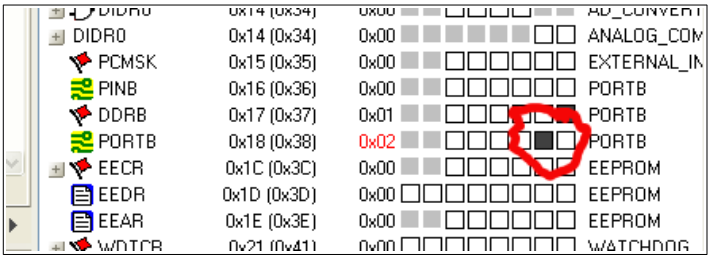
Lass uns einen möglichen Programmierfehler debuggen. Angenommen, du hast in der ersten for-Schleife versehentlich PORTB=2 anstatt 1 geschrieben, was dir aber noch nicht aufgefallen ist. Die LED wird 5 Sekunden dunkel bleiben, dann 3 Sekunden lang schnell blinken, dann wieder 5 Sekunden dunkel bleiben. Bei diesem Sympton ist klar, dass die zweite for() Schleife ihren Sinn korrekt erfüllt, die erste aber nicht. Daher reduzierst du das Programm auf den fehlerhaften Teil:

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

int main(void)
{
    DDRB=1;
    while(1)
    {
        for (uint8_t i=0; i<5; i++)
        {
```

```
            PORTB=0;
            //_delay_ms(500);
            PORTB=2;
            //_delay_ms(500);
        }
        //for (uint8_t i=0; i<30; i++)
        //{
        //    PORTB=0;
        //    _delay_ms(50);
        //    PORTB=1;
        //    _delay_ms(50);
        //}
    }
}
```

Deaktiviere die Optimierung des Compilers. Kommentiere die delay Aufrufe und die entsprechende include-Anweisung aus, weil sie generell im Simulator nicht funktionieren. Jetzt kannst du das Programm mit F7 Compilieren und dann im Simulator schrittweise mit F11 ausführen. Wenn der Befehl PORTB=2 ausgeführt wurde, schaue dir den Zustand von Register PORTB an:

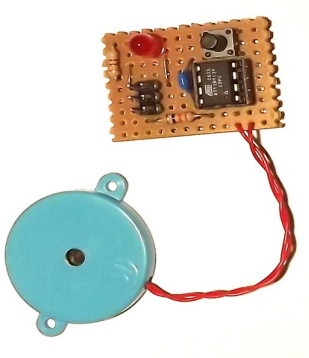


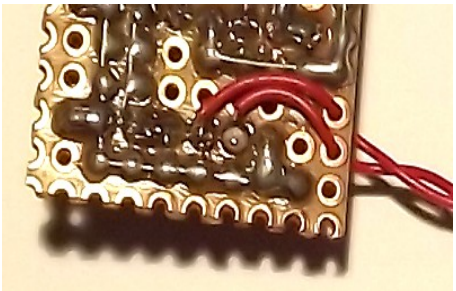


Du siehst hier, dass das Register den hexadezimalen Wert 0x02 hat, was dezimal eine 2 ist. Es sollte aber eine 1 sein. Der Schreibzugriff auf dieses Register ist also fehlerhaft. Du kannst die Darstellung dieser Ansicht ändern, indem du auf den rot eingekreisten Knopf klickst. Die Register werden dann bitweise angezeigt. In dieser Ansicht ist noch deutlicher zu sehen, dass von Port B das Bit 1 anstatt das erwartete Bit 0 , an dem die LED hängt, eingeschaltet wurde.

4.7 Töne erzeugen

Erweitere den kleinen Mikrocomputer mit dem Piezo-Schallwandler, so dass er Töne von sich geben kann:





Der Schallwandler gehört an die Pins 2 und 3 vom Mikrocontroller. Da angelötete Litzen bei

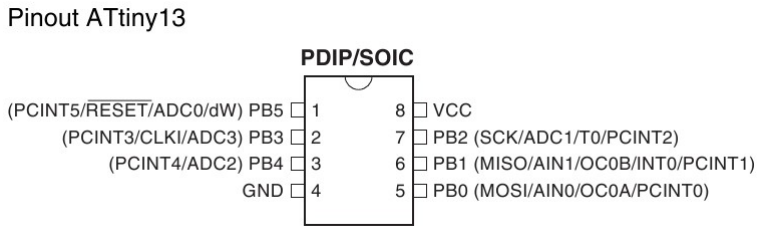
Bewegung zum Abbrechen neigen, solltest du sie irgendwie fixieren. Zum Beispiel mit einem Bindfaden, oder mit Heißkleber, oder indem du sie wie im obigen Foto durch die Löcher steckst – sofern sie dazu dünn genug sind.

Der Piezo-Schallwandler besteht aus einem Piezo-Kristall, der auf ein rundes Blech geklebt ist und in ein Plastik Gehäuse eingebaut ist. Der Kristall verformt sich geringfügig, wenn man eine elektrische Spannung anlegt, so dass man ihn als Lautsprecher verwenden kann. Er kann umgekehrt auch als Mikrofon verwendet werden, denn er gibt Spannung ab, wenn man ihn durch äußere Kräfte (Schallwellen) verformt. Für Experimente ist der Piezo-Schallwandler besonders gut geeignet, weil er nicht kaputt geht, wenn man eine Dauer-Spannung anlegt.

Übertrage das folgende Programm in den Mikrocontroller:

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

int main(void)
{
    DDRB=24;
    while(1)
    {
        PORTB=8;
        _delay_us(500);
        PORTB=16;
        _delay_us(500);
    }
}
```

Beachte, dass dieses mal die Funktion _delay_us() anstatt von _delay_ms() verwendet wird. Der Mikrocontroller sollte nun einen andauernden Piepton von sich geben. Das Programm funktioniert so: Der Schallwandler wurde an die Pins 2+3 angelötet. Laut Datenblatt sind das die Ports PB3 und PB4.



Mit DDRB=24 werden diese beiden Pins als Ausgang konfiguriert, denn 24 ist 16+8.

Bit Nr.	7	6	5	4	3	2	1	0
Wert	128	64	32	16	8	4	2	1

In der endlosen while() Schleife werden die beiden Ausgänge immer wieder abwechselnd eingeschaltet. Dies bewirkt, dass der Piezo-Kristall hin und her Schwingt – es entsteht ein Ton. Die Frequenz des Tons hängt davon ab, wie schnell diese Wechsel stattfinden. Zuerst ist PB3 für 500 Mikro-Sekunden an, danach ist PB4 für 500 Mikro-Sekunden an. Zusammen gerechnet ergibt das 1000 Mikro-Sekunden (=0,001 Sekunden). Die Frequenz berechnet man nach der Formel:

Frequenz in Hertz = 1 : Zeit in Sekunden

In unserem Fall ergibt 1:0,001 den Wert 1000, also 1000 Hertz oder 1 Kilo-Hertz.

4.7.1 Übungsaufgaben

1. Verändere das Programm so, dass es den Kammerton A (=440 Hz) abspielt.
2. Verändere das Programm so, dass es einen Intervall-Ton mit 2000 Hz abspielt, so wie ein Wecker.
3. Finde heraus, bei welchen Frequenzen der Schallwandler am lautesten klingt und programmiere einen möglichst aufdringlichen Alarm-Ton.

4.8 Eigene Funktionen

Bei den Übungsaufgaben hast du vermutlich mehrere Male identischen oder fast identischen Quelltext wiederholt. Funktionen dieses dazu, solche Wiederholungen zu reduzieren. Da Mikrocontroller ziemlich kleine Programmspeicher haben, muss man dort unnötige Wiederholungen von Programmcode vermeiden.

Schau dir an, wie meine Lösung für den Alarm-Ton mit Hilfe von Funktionen besser umgesetzt werden kann:

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

void ton1(void)
{
    for (int i=0; i<100; i++)
    {
        PORTB=8;
        _delay_us(1000000/3000/2);
        PORTB=16;
        _delay_us(1000000/3000/2);
    }
    _delay_ms(20);
}

void ton2(void)
{
    for (int i=0; i<100; i++)
    {
        PORTB=8;
        _delay_us(1000000/3500/2);
        PORTB=16;
        _delay_us(1000000/3500/2);
    }
    _delay_ms(20);
}

void ton3(void)
{
    for (int i=0; i<100; i++)
    {
        PORTB=8;
        _delay_us(1000000/4000/2);
        PORTB=16;
        _delay_us(1000000/4000/2);
    }
    _delay_ms(20);
}

int main(void)
{
    DDRB=24;
    while(1)
```

```
    {
        ton1();
        ton2();
        ton3();
        ton2();
    }
}
```

Dieses Programm enthält drei Funktionen mit den Namen „ton1“, „ton2“ und „ton3“. Im Hauptprogramm werden die Funktionen aufgerufen. Auf diese Weise ist zumindest das Hauptprogramm übersichtlicher geworden.

Das Programm ist außerdem etwas kleiner geworden, denn es enthält nun nicht mehr vier for() Schleifen, sondern nur noch drei. Die Schleife für den Ton mit 3500 Hz existiert nun nur noch einmal, wird aber zweimal aufgerufen.

Funktionen deklariert man so:

```
void name(void)
{
    …
}
```

In die Klammern schreibt man void, wenn die Funktion keine Aufruf-Parameter hat. Vor den Namen schreibt man „void“, wenn die Funktion kein Ergebnis zurück liefert. Andernfalls gibt man an dieser Stelle an, welchen Datentyp die Funktion zurück liefert. Beispiel:

```
uint8_t gibMirFuenf(void)
{
    …
    return 5;
}
```

Wenn eine Funktion das Ergebnis berechnet hat, gibt sie es mit der „return“ Anweisung zurück. Die Funktion ist damit beendet. Falls noch irgendwelche Befehle hinter der return Anweisung kommen, werden diese nicht mehr ausgeführt.

Funktionen, die nichts zurück liefern, können auch eine return Anweisung enthalten, allerdings ohne Rückgabe-Wert:

```
void tuWas(void)
{
    …
    if (abbruchbedingung)
    {
        return;
    }
    weiter machen;
    …
}
```

Funktionen benötigen oft Aufruf-Parameter, zum Beispiel:

```
float drittel(float betrag)
{
    return betrag/3;
}
```

Diese Funktion dividiert den übergebenen Betrag durch drei und liefert das Ergebnis zurück. Man ruft sie so auf:

```
float ergebnis=drittel(100);    // ergibt 33,333333
```

Funktionen können mehrere Übergabe-Parameter haben, aber nur ein Ergebnis:

```
uint8_t summe(uint8_t a, uint8_t b, uint8_t c)
{
    return a+b+c;
}

uint8_t ergebnis=summe(12,45,7);
```

4.9 Divisionen sind teuer

Mit diesem neuen Wissen fällt uns gleich eine Möglichkeit ein, das Töne-Programm noch weiter zu verkürzen:

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

void ton(uint16_t frequenz)
{
    for (int i=0; i<100; i++)
    {
        PORTB=8;
        _delay_us(1000000/frequenz/2);
        PORTB=16;
        _delay_us(1000000/frequenz/2);
    }
    _delay_ms(20);
}

int main(void)
{
    DDRB=24;
    while(1)
    {
        ton(3000);
        ton(3500);
        ton(4000);
        ton(3500);
    }
}
```

Das sieht doch sehr elegant aus! Dummerweise passt das Programm nun nicht mehr in den Speicher. Die große Frage ist: Wieso ist der Byte-Code des Programms jetzt größer geworden, obwohl wir die Anzahl der Quelltext-Zeilen erheblich reduziert haben?

Das ist ein ganz besonderes Problem bei der Mikrocontroller Programmierung, wo nur ganz wenig Speicher verfügbar ist. Vergleiche: Ein einfaches Notebook hat typischerweise eine Millionen mal mehr Speicher, als ein Mikrocontroller.

Das Problem entsteht durch Divisionen, die der Compiler dieses mal nicht mehr weg optimieren kann. Divisionsaufgaben sind viel komplizierter, als Additionen und erzeugen eine beträchtliche Menge an Programmcode. Divisionen kommen beim Aufruf der delay() Funktionen vor, aber auch innerhalb der delay() Funktion.

Die delay() Funktionen vertrödeln etwas Zeit, indem Sie leere Wiederhol-Schleifen ausführen. Die Dauer der Verzögerung hängt davon ab, wie oft diese Schleifen wiederholt werden.

Wenn du beim Aufruf einer delay() Funktion eine konstante Zahl angibst, kann der Compiler im voraus ausrechnen, wie oft die Verzögerungs-Schleifen wiederholt werden müssen. Der Befehl

```
_delay_ms(20);
```

ist also kein Problem. Problematisch sind aber die Aufrufe, die Variablen enthalten:

```
_delay_us(1000000/frequenz/2);
```

Was 1000000/2 ergibt, kann der Compiler noch vor-berechnen, aber dann bleibt immer noch sowas übrig:

```
_delay_us(500000/frequenz);
```

Da die Variable „frequenz“ jeden beliebigen Wert haben kann, kann der Compiler diesen Ausdruck nicht weiter optimieren. Auch innerhalb der delay() Funktion, deren Quelltext wir nicht gesehen haben, gibt es eine Division.

Angenommen, die Frequenz wäre 1000Hz, dann muss der Mikrocontroller den Ausdruck 500000/1000=500 berechnen. Die delay() Funktion muss berechnen, wie viele Verzögerungsschleifen nötig sind, um 500 Mikrosekunden zu vertrödeln. Dazu dividiert sie die Taktfrequenz (1,2MHz) durch die gewünschte Zeit.

Wir können das Problem nur lösen, indem wir entweder einen anderen Mikrocontroller mit mehr Speicher verwenden, oder das Programm so umschreiben, dass alle delay() Aufrufe konstante Übergabeparameter haben. Zum Beispiel so:

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

void ton(char tonlage)
{
    for (int i=0; i<100; i++)
    {
        PORTB=8;
        switch (tonlage)
        {
            case 'A':
                _delay_us(1000000/3000/2);
                break;

            case 'B':
                _delay_us(1000000/3500/2);
                break;

            case 'C':
                _delay_us(1000000/4000/2);
                break;
        }
        PORTB=16;
        switch (tonlage)
        {
            case 'A':
                _delay_us(1000000/3000/2);
                break;

            case 'B':
                _delay_us(1000000/3500/2);
                break;
```

```
            case 'C':
                _delay_us(1000000/4000/2);
                break;
        }
    }
    _delay_ms(20);
}

int main(void)
{
    DDRB=24;
    while(1)
    {
        ton('A');
        ton('B');
        ton('C');
        ton('B');
    }
}
```

Jetzt passt das Programm wieder in den Speicher des Mikrocontrollers hinein.

4.9.1 Übungsaufgabe

4. Verändere das Programm so, dass es die Tonleiter C-D-E-F-G-A-H-C spielt.

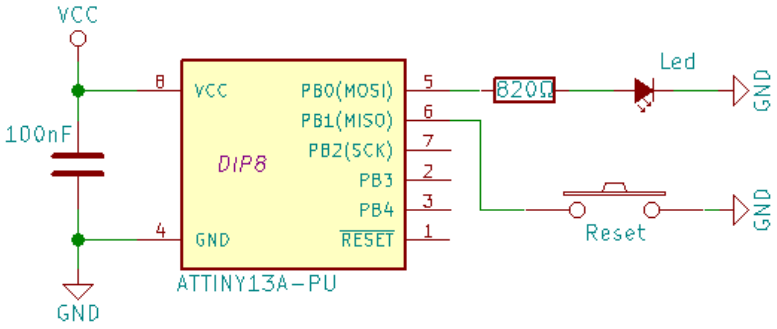
Die Frequenzen der Dur-Tonleiter sind folgende:

C = 523,25 Hz
D = 587,33 Hz
E = 659,26 Hz
F = 698,46 Hz
G = 783,99 Hz
A = 880 Hz
H = 987,77 Hz
C = 1046,50 Hz

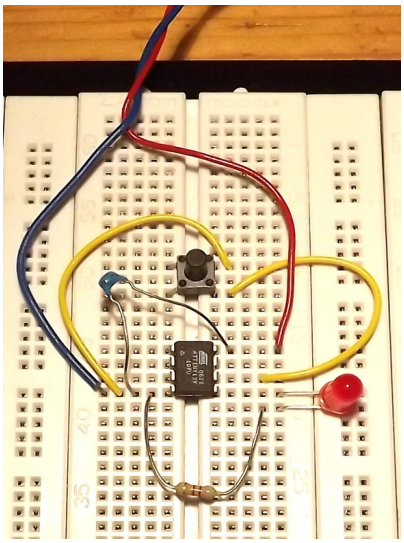
Um Speicherintensive Fließkommazahl-Arithmetik zu vermeiden, runde diese Werte auf Integer-Werte (ohne Nachkommastellen) auf bzw. ab.

4.10 Eingänge Abfragen

Ich möchte dir nun erklären, wie man Eingänge abfragt. In diesem Zusammenhang wirst du noch einige Informationen über bitweise Programmierung erhalten. Baue die folgende Schaltung auf dem Steckbrett auf:



Der Taster ist dieses mal nicht mit dem Reset-Eingang verbunden, sondern mit PB1.



Um das Programm in den Chip zu übertragen verwendest du entweder die Platine vom ersten Mikrocomputer oder lose Kabel, die vom ISP Programmieradapter zum Steckbrett führen.

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

int main(void)
{
    DDRB  = 0b00000001; // PB0 (LED) ist Ausgang
    PORTB = 0b00000010; // Pull-Up von PB1 (Taster) aktivieren
    while (1) {
        if (PINB & 0b00000010)
        {  // Wenn PB1 High ist
            PORTB |= 0b00000001;  // LED an
        }
        else
        {
            PORTB &= 0b11111110;  // LED aus
        }
    }
}
```

Wenn du das Programm ausführst, sollte folgendes passieren: Zunächst ist die LED an. Wenn du den Taste drückst, geht die LED aus. Wenn du den Taster loslässt, geht sie wieder an. Das Programm verwendet die folgenden Register:

- **DDRB**
  jedes gesetzte Bit konfiguriert den entsprechenden Pin von Port B als Ausgang. Das kennst du schon.
- **PORTB**
  Bei Ausgängen steuert dieses Register den Ausgang an. 1=High, 0=Low. Bei Eingängen aktiviert jedes gesetzte Bit den Pull-Up Widerstand des entsprechenden Eingangs. Ich erkläre später, was ein Pull-Up ist.
- **PINB**
  Benutzt man, um den Status der Eingänge abzufragen.

Neu ist in diesem Programm auch die Verwendung von Binären Zahlen. Mit dezimalen Zahlen würde es auch funktionieren, in diesem Fall finde ich binäre Zahlen jedoch praktischer.

4.10.1.1 Erklärung

Zuerst wird der Anschluss PB0 als Ausgang konfiguriert. Alle anderen Pins sollen Eingänge sein.:

```
DDRB  = 0b00000001;
```

Dann wird der Pull-Up Widerstand des Eingang aktiviert, wo der Taster angeschlossen ist. Für Taster und Schalter braucht man Pull-Up's, mehr musst du dazu momentan nicht wissen.

```
PORTB = 0b00000010;
```

Es folgt eine Endlosschleife, in der der Eingang immer wieder abgefragt wird:

```
if (PINB & 0b00000010)
{
    tu etwas;
}
else
{
    tu etwas anderes;
}
```

Mit PINB lesen wir den Status der Eingänge ein. Diese verknüpfen wir mit der bitweisen Und-Funktion, um nur das eine gewünschte Bit heraus zu filtern. Wenn der Taster gedrückt ist, haben wir folgende Situation:

PINB	x[1]	x	x	x	x	x	0	x
Operand	0	0	0	0	0	0	1	0
Ergebnis	0	0	0	0	0	0	0	0

Wenn der Taster losgelassen ist, haben wir diese Situation:

PINB	x	x	x	x	x	x	1	x
Operand	0	0	0	0	0	0	1	0
Ergebnis	0	0	0	0	0	0	1	0

Das obere Ergebnis 0b00000000 wird als „falsch“ interpretiert, weil es Null ist. Das untere Ergebnis 0b00000010 wird als „wahr“ interpretiert, weil es nicht Null ist. Die Bitweise Und-Verknüpfung ist ein gängiges Mittel, um aus einem ganzen Register mit 8 Bits ein einzelnes Bit heraus zu filtern.

Wenn der Eingang auf High steht, also der Ausdruck „wahr“ ist, dann soll die Leuchtdiode an gehen. Da die Programmiersprache C keine Befehle kennt, um einzelne Bits anzusteuern, müssen wir hier einen Trick einsetzen:

```
PORTB |= 0b00000001;
```

Das ist eine Bitweise Oder-Funktion mit Zuweisung. Sie bedeutet: Verwende den aktuellen Wert von PORTB, mache eine bitweise Oder-Verknüpfung mit 0b00000001 und schreibe das Ergebnis in PORTB zurück. Dadurch wird dieses eine Bit auf High gesetzt, alle anderen Bits bleiben so, wie sie vorher waren.

Um die LED wieder aus zu schalten, wird ein ähnlicher Kunstgriff verwendet:

```
PORTB &= 0b11111110;
```

Das ist eine bitweise Und-Funktion mit Zuweisung. Sie bedeutet: Verwende den aktuellen Wert von PORTB, mache eine bitweise Und-Verknüpfung mit 0b11111110 und schreibe das Ergebnis in PORTB zurück. Dadurch wird das eine Bit auf Low gesetzt, alle anderen Bits bleiben so, wie sie vorher waren.

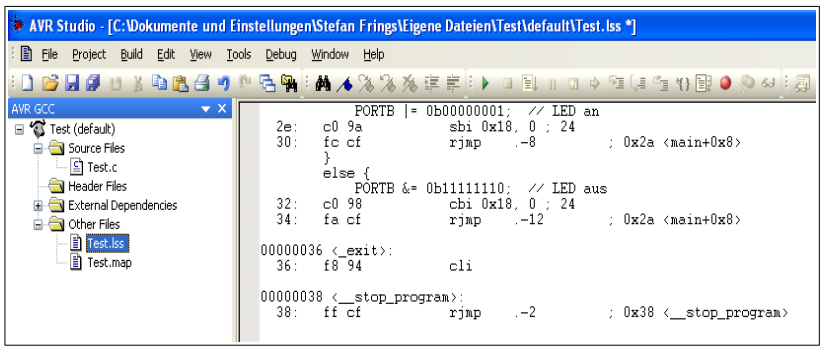
4.11 Wie der Compiler Einzel-Bit Befehle optimiert

Obwohl die Programmiersprache C eigentlich nicht zur Verarbeitung einzelner Bits gedacht ist, kann der Compiler dennoch Code erzeugen, der tatsächlich nur einzelne Bits auf einem Port anspricht. Ein Ausdruck wie:

```
PORTB &= 0b11111110;
```

Wird tatsächlich auf einen simplen Assembler-Befehlt umgesetzt, der ein einzelnes Bit löscht. Der Mikrocontroller muss hier gar keine UND-Verknüpfung durchführen. Gleiches gilt auch für die das Setzen eines Bits mittels ODER-Verknüpfung. Wir schauen uns das mal in der lss-Datei an, sie befindet sich im Projektverzeichnis.

1 Das x bedeutet Egal



Diese Datei zeigt uns, den Assembler- und Maschinencode, den der Compiler erzeugt hat. Das ist also der Code, der vom Mikrocontroller letztendlich ausgeführt wird. Fast am Ende der Datei wirst du diesen Abschnitt finden:

```
PORTB |= 0b00000001;  // LED an
2e: c0 9a             sbi   0x18, 0     ; 24
```

Die erste Zeile informiert darüber, um welchen Teil aus dem C-Quelltext es geht. Darunter folgt der vom Compiler erzeugte Byte-Code:

- **2e** ist die Position im Programmspeicher
- **c0 9a** ist der Byte-Code des Befehls
- **sbi 0x18, 0** ist der Byte-Code übersetzt in die Programmiersprache Assembler. Das ist der Teil, der uns hier interessiert.
- **; 24** ist ein Kommentar.In diesem Fall ist das der dezimale Wert von 0x18.

Der Befehl sbi bedeutet „Set Bit Immediate“. Der erste Operand „0x18“ ist die Nummer von PORTB, der zweite Operand „0“ ist die Nummer des Bits, das gesetzt werden soll. Also das Bit 0 von Port B. Von der ODER-Verknüpfung ist hier nichts mehr zu sehen.

Schauen wir uns an, wie die LED ausgeschaltet wird:

```
PORTB &= 0b11111110;  // LED aus
32: c0 98             cbi   0x18, 0     ; 24
```

Der Befehl cbi bedeutet „Clear Bit Immediate“. Von der UND-Verknüpfung ist hier nichts mehr zu sehen.

Du siehst, dass der Compiler einen äußerst kompakten Byte-Code erzeugt, obwohl die Programmiersprache C uns zu weniger hübschen Kunstgriffen zwingt, weil sie keine bitweisen Befehle wie „Setze ein Bit 0 in Port B auf 1“ kennt.

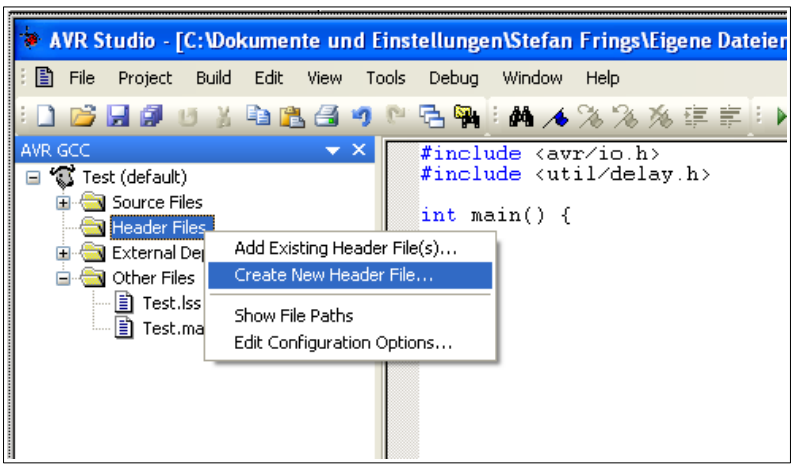
4.11.1 Übungsaufgaben

1. Wozu war nochmal der Kondensator gut?
2. Gestalte den C-Quelltext durch Verwendung von #define übersichtlicher.
3. Erkläre in eigenen Worten, warum man Bitweise Verknüpfungen benötigt, wenn man den Taster abfragt und die LED ansteuert.

4.12 Programm auf mehrere Dateien aufteilen

Ab einer gewissen Größe wird es der Übersicht halber nötig sein, das Programm in mehrere Dateien aufzuteilen. An einem Beispiel erkläre ich, wie man das macht.

Wir schreiben ein ganz einfaches Programm, das lediglich die Leuchtdiode einschalten wird. Dabei soll eine Funktion verwendet werden, die sich in einer zweiten Quelltext-Datei befindet. Erstelle im AVR Studio ein neues Projekt. Klicke dann im linken Fenster mit der rechten Maustaste auf „Header Files“ und wähle dann den Befehl „Create New Header File“.



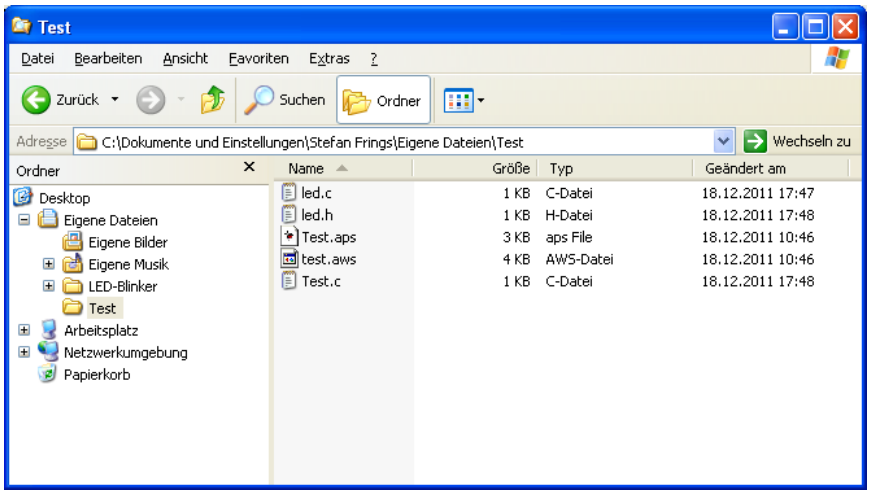
Nenne die neue Header-Datei „led.h“. Sie soll folgenden Inhalt bekommen:

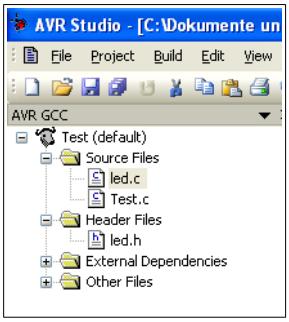
```
void led_an(void);
```

Lege jetzt auf die gleiche Weise eine Source-Datei an, mit dem Namen „led.c“. Sie soll folgenden Inhalt bekommen:

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

void led_an(void)
{
    DDRB=1;
    PORTB=1;
}
```

Die Funktion led_an() wird die Leuchtdiode einschalten, wenn sie aufgerufen wird. Du hast nun folgende Verzeichnis-Struktur:





Ich habe das Projekt „Test“ genannt, darum befindet sich mein Hauptprogramm in der Datei Test.c. Das Hauptprogramm soll so aussehen:

```
#include "led.h"
```

```
int main(void)
{
    led_an();
    while (1);
}
```

Die Funktion led_an() hast du in eine zweite Source-Datei ausgelagert. Jede Source-Datei kann beliebig viele Funktionen und Variablen enthalten. Durch die Anweisung

```
#include "led.h"
```

sagst du dem Compiler „Mache mir alle Funktionen aus led.h verfügbar. Ich verspreche dir, dass sie in irgendeiner *.c Datei vollständig definiert sind.“. In der Header Datei befindet sich lediglich die Kopfzeile der Funktion:

```
void led_an(void);
```

Die vollständige Definition kann sich theoretisch in jeder beliebigen *.c Datei des Projektes befinden. Normalerweise heißt die *.c Datei jedoch immer genau so, wie die *.h Datei, damit man nicht durcheinander kommt.

Zusammenfassung:

- Die Include-Anweisung macht alle Funktionen verfügbar, die dort definiert sind.
- Die includierte Header-Datei (*.h) deklariert die Kopfzeilen von Funktionen.
- Eine normalerweise gleichnamige Source-Datei (*.c) definiert die gesamten Funktionen, mit Kopfzeile und Code-Block.

4.12.1.1 Mehrfaches Inkludieren verhindern

In größeren Projekten mit komplexen Abhängigkeiten zwischen den Dateien kann es vorkommen, dass der Compiler mit folgender Meldung ab bricht:

- multiple definition of 'xyz'

Wobei xyz eine Definition, eine Funktion oder auch eine Variable sein kann. Das Problem entsteht durch mehrfache #include Anweisungen mit dem gleichen Dateinamen und es ist manchmal unvermeidbar. Den Konflikt kann man jedoch lösen, indem man um den gesamten Inhalt der betroffenen Header-Datei herum diese Anweisungen schreibt:

```
#ifndef DATEINAME_H
#define DATEINAME_H

…

#endif
```

Hier passiert folgendes: Die #ifndef Anweisung sagt dem Compiler, dass er den Inhalt dieser Datei nur dann verarbeiten soll, wenn das angegebene Schlüsselwort noch nicht definiert ist. Innerhalb dieses bedingten Blockes wird dann das Schlüsselwort (ohne konkreten Wert) definiert. Der bedingte Block endet mit #endif.

Wenn der Compiler diese Header-Datei nun ein zweites mal verarbeitet, dann ist das Schlüsselwort bereits definiert, so dass der gesamte Inhalt der Datei übersprungen wird.

Alle Header-Dateien der avr-libc Library wenden diese Methode an. Ich empfehle, dass du es ebenfalls immer so machst, unabhängig davon ob es wirklich notwendig ist.

5 AVR - Elektrische Eigenschaften

Die Grundlagen der Programmiersprache C hast du gelernt und einige Programme zum Laufen gebracht. Wenden wir uns nun primär dem Mikrocontroller selbst zu. Du lernst in diesem Kapitel, welche elektrischen Eigenschaften AVR Mikrocontroller haben.

5.1 Digitale Signale

Digitale Schaltungen arbeiten mit digitalen Signalen. Im Gegensatz zu analogen Signalen, die unendlich viele unterschiedliche Werte annehmen können, kennen digitale Signale nur zwei gültige Werte, nämlich 0 und 1 oder Low und High.

Digitale Schaltungen verhalten sich wie Lichtschalter, die such nur zwei Zustände kennen, nämlich „Aus“ und „An“. So etwas wie „ein bischen an“ gibt es in digitalen Schaltungen nicht.

- **Low oder 0** ist eine niedrige Spannung (nahe Null Volt)
- **High oder 1** ist eine hohe Spannung (nahe der Versorgungsspannung)

Alle Spannungen zwischen Low und High sind im digitalen Umfeld nicht zulässig.

5.2 Stromversorgung

Alle Spannungsangaben beziehen sich auf GND, das heißt auf englisch: Ground, auf deutsch: Masse. Der Name „Masse“ erinnert an Kraftfahrzeuge. Dort ist der Minuspol der Batterie an das eiserne Chassis angeschlossen, also die große Masse des Fahrzeuges.

- **GND** ist Null Volt, in Schaltplänen oft durch das Symbol ⊥ oder ↓dargestellt.
- **VCC** (oder VDD) ist die Versorgungsspannung

Aktueller Industriestandard für digitale Schaltungen ist eine Versorgungsspannung von 3,3 Volt. Im zwanzigsten Jahrhundert wurden Mikrochips standardmäßig noch mit 5 Volt betrieben.

AVR Mikrocontroller stellen nur einfache Ansprüche an die Stromversorgung. Die Spannung darf sich während des Betriebes langsam ändern, aber nicht sprunghaft. Sie muss frei von Aussetzern sein. Die Höhe der Versorgungsspannung bestimmt die maximale Taktfrequenz, den Stromverbrauch, und die Belastbarkeit der Ausgänge.

Spannung	Frequenz	Stromverbrauch	Belastbarkeit
2,7 Volt	1 MHz	0,4 mA	15 mA
2,7 Volt	10 MHz	3 mA	15 mA
3,3 Volt	1 MHz	0,5 mA	20 mA
3,3 Volt	10 MHz	4 mA	20 mA
5 Volt	1 MHz	1 mA	30 mA
5 Volt	10 MHz	6 mA	30 mA
5 Volt	20 MHz	11 mA	30 mA

Die alten Modelle Atmega 8, 8515, 8535, 16 und 32 laufen nur bis 8 Mhz bei 3,3 Volt und 16MHz bei 5 Volt. Alle anderen schaffen 10 Mhz bei 3,3 Volt und 20MHz bei 5 Volt.

Bei 3,3 Volt Versorgungsspannung und 1 MHz Taktfrequenz verbrauchen AVR Mikrocontroller weniger als 2 Milli-Watt Strom. Wegen diesem geringen Stromverbrauch sind AVR Mikrocontroller sehr gut für Batteriebetrieb geeignet.

Der Stromverbrauch hängt direkt von der Taktfrequenz ab. Bei doppelter Taktfrequenz ist der Stromverbrauch auch ungefähr doppelt so hoch. Bei zehnfacher Taktfrequenz ist der Stromverbrauch ungefähr zehnmal so hoch.

Die Stromaufnahme ist (fast) Null ist, wenn der Takt angehalten wird. Ich habe vor 5 Jahren ein elektronisches Spiel gebaut, dass mit einer Knopfzelle betrieben wird. Die Batterie ist heute immer noch gut.

Mehr als 5,5 Volt vertragen die AVR Mikrocontroller nicht.

5.3 Eingänge

Digitale Eingänge kennen nur zwei gültige Signale: High und Low. Welche Spannung als High oder Low erkannt wird, hängt von der Versorgungsspannung ab. Im Datenblatt findet man konkrete Angaben unter „Electrical Characteristics“, welche Spannungen als High und Low gültig sind:

- Low ist höchstens 0,2 mal VCC
- High ist mindestens 0,6 mal VCC

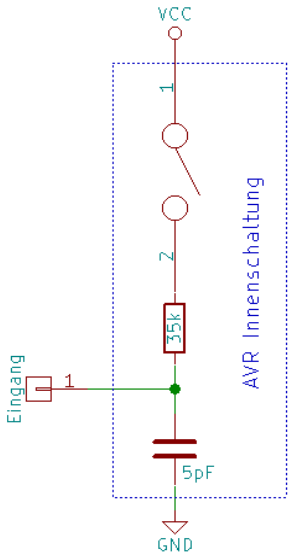
Darüber hinaus gilt grundsätzlich die Regel, dass die Spannung an allen Anschlüssen nicht unter -0,5 Volt und nicht mehr als +0,5 V über VCC liegen darf, denn sonst geht der Mikrocontroller kaputt. Die konkrete Werte sind demnach:

VCC	Low	High
2,7 Volt	-0,5...0,54 Volt	1,62...3,2 Volt
3,3 Volt	-0,5...0,66 Volt	1,98...3,8 Volt
3,6 Volt	-0,5...0,72 Volt	2,10...4,1 Volt
5,0 Volt	-0,5...1,00 Volt	3,00...5,5 Volt

Falls du einmal Mikrochips der 74er Reihe mit AVR Mikrocontroller kombinieren möchtest, dann verwende 74HCxx oder 74HCTxx Typen. Die 74LSxx Typen sind ungeeignet weil ihre Ausgangs-Signale nicht der obigen Tabelle entsprechen.

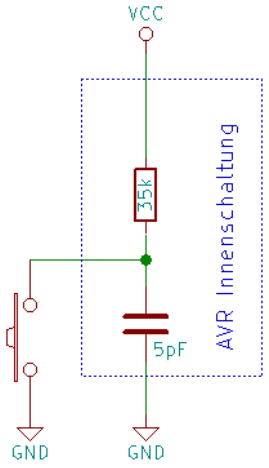
Die Eingänge des AVR Mikrocontrollers wirken von außen betrachtet wie ein kleiner Kondensator mit einigen Pikofarat Kapazität (Atmel gibt im Datenblatt keinen konkreten Wert für die Kapazität an), sowie einem zuschaltbaren Widerstand.

Immer, +wenn sich die Spannung am Eingang ändert, fließt für einen kurzen Moment lang ein Strom, nämlich so lange, bis der Kondensator sich auf die Eingangsspannung umgeladen hat. Je höher die Signalfrequenz ist (also je mehr Low/High Wechsel pro Sekunde), um so mehr Strom fließt. Je geringer die Signalfrequenz ist, umso geringer ist der Stromfluss. Wenn sich die Spannung am Eingang gar nicht ändert, fließt kein Strom.



Der Chip-Interne Widerstand heißt Pull-Up. Er kann per Software eingeschaltet werden. Er wird häufig in Kombination mit Tastern verwendet.

Wenn der Taster gedrückt wird, zieht er den Eingang auf Low. Dabei fließt ein Strom von etwa 0,1 Milliampere durch den Widerstand und den Taster. Wenn er losgelassen wird, zieht der Pull-Up Widerstand den Eingang auf High. Ohne Widerstand hätten wir keinen definierten Signalpegel, wenn der Taster losgelassen wird.

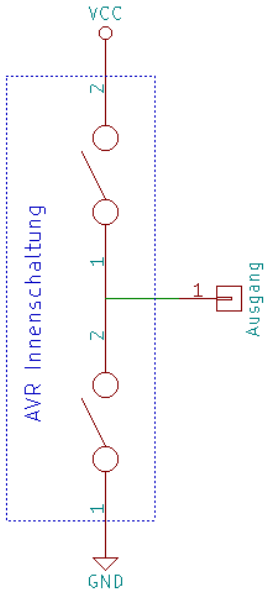


5.4 Ausgänge

Die digitalen Ausgänge werden intern durch Schalt-Transistoren immer entweder mit VCC oder mit GND verbunden. Es ist immer einer der beiden Schalter geschlossen.

Die Schalt-Transistoren sind nicht beliebig hoch belastbar. Für jeden Ausgang gelten diese Grenzwerte:

Versorgungsspannung	Höchster zu erwartender Kurzschluss-Strom	Zulässige Belastung in Betrieb
2,7 Volt	20 mA	15 mA
3,3 Volt	30 mA	20 mA
5,0 Volt	40 mA	30 mA

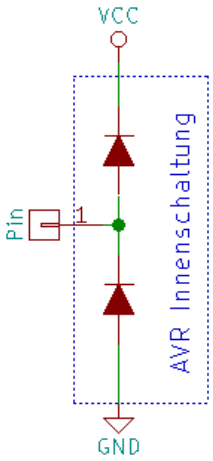


Der angegebene Betriebsstrom berücksichtigt die für Low und High Pegel verlangten Spannungen. Wenn man die Ausgänge höher belastet, geht im Chip zu viel Spannung verloren, so dass das Signal dann im undefinierten Bereich liegt und unbrauchbar wird.

Man kann einzelne Ausgänge kurzschließen, ohne dass der Chip kaputt geht. Aber insgesamt dürfen nicht mehr als ca. 200 mA fließen. Wenn man also viele Ausgänge kurzschließt, kann der Chip doch zerstört werden.

5.5 Schutzdioden

Grundsätzlich gilt die Regel, dass die Spannung an allen Anschlüssen nicht wesentlich unter Null Volt und nicht wesentlich über VCC liegen darf.



Ohne Schutzschaltung kann schon eine einfache Berührung mit dem Finger zur Zerstörung des Mikrochips führen, da der Mensch manchmal elektrostatisch aufgeladen ist. Statische Ladungen haben typischerweise einige tausend Volt.

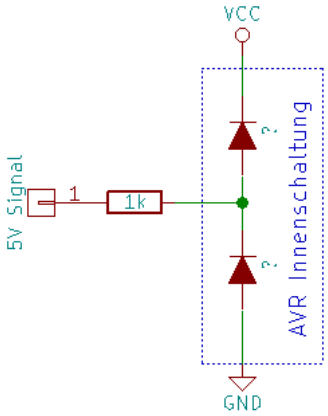
Darum haben fast alle Mikrochips interne Schutzdioden eingebaut. Sie sollen sowohl Überspannung als auch Unterspannung ableiten.

Wenn die Signalspannung mehr als 0,6 Volt unter Null geht, wird die untere Diode leitend. Wenn die Signalspannung mehr als 0,6 Volt über der Versorgungsspannung liegt, wird die obere Diode leitend. Diese Dioden leiten die Überspannung wirksam an die Stromversorgung ab, sofern sie überlastet werden.

Eine Anwendungsnotiz von Atmel sagt, dass man diese Dioden mit bis zu 2 mA belasten darf. In den Datenblättern der Mikrcocontroller findet man dazu jedoch keine konkrete Angabe.

5.6 Schutzwiderstände

Man kann die Schutzdioden in das Schaltungsdesign einbeziehen. Mit Hilfe eines zusätzlichen Widerstandes kann man den Ausgang eines 5 V Mikrochips mit einem Eingang verbinden, der nur 3,6 Volt verträgt.



Wenn das Signal 5 V beträgt, fließt ein Strom durch den Widerstand und die Diode. Die Diode verhindert, dass die Spannung am Eingang des AVR zu groß wird. Der Widerstand begrenzt dabei die Stromstärke.

Spannung am Widerstand = 5 V – 3,6 V – 0,6 V = 0,8 V

Stromstärke = 0,8 V : 1000 Ohm = 0,0008 Ampere also 0,8 mA

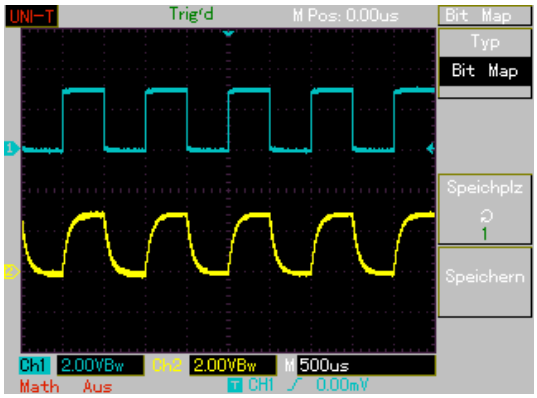
Ohne Widerstand hätten wir keine Strombegrenzung, so dass ein Kurzschluss mit zerstörerischer Wirkung entstehen würde.

Wenn man zwei elektronische Geräte miteinander verbindet, sollte man solche Schutzwiderstände noch aus einem anderen Grund vorsehen. Und zwar kann es vorkommen, dass das sendende Gerät zuerst eingeschaltet wird, während das Empfangende Gerät noch Stromlos ist. Der Widerstand verhindert dann, dass durch die Schutzdioden des Empfängers übermäßig viel Strom fließt. Laut einer „Application Note“ von Atmel soll man diese Dioden mit maximal 2 mA belasten.

5.7 Kapazitive Belastung

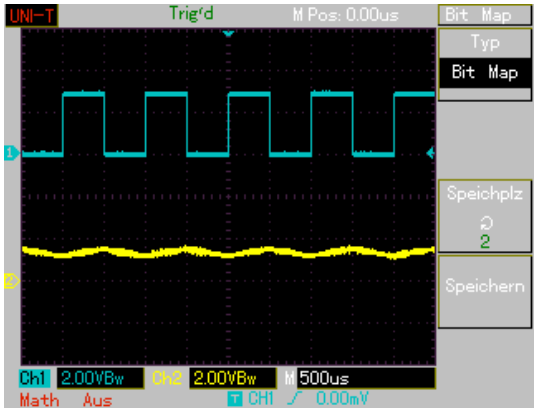
Die Kapazitäten in den Eingängen der Mikrochips und zwischen den Leitungen belasten, verzögern und verzerren das Signal.

Das folgende Bild zeigt ein digitales Signal im zeitlichen Verlauf von links nach rechts. Ich habe es mit einem sogenannten Oszilloskop erstellt, das ist ein Messgerät zur Bildlichen Darstellung von elektrischen Signalen.



In der Farbe türkis siehst du das ursprüngliche (fast) ideale Signal, das aus dem Ausgang eines Mikrochips heraus kommt, wenn er nur nicht belastet wird. Darunter in gelb siehst du, wie das Signal durch die kapazitive Belastung verzerrt wird. Bei einem Wechsel von Low nach High steigt die Spannung nicht mehr sprunghaft an, sondern langsamer. Und beim Wechsel von High nach Low tritt der selbe Effekt auf.

Wenn man das Signal zu stark belastet oder die Frequenz des Signals zu hoch ist, sieht das Ergebnis so aus:



Das Signal ist bis zur Unkenntlichkeit verzerrt. So kann die Schaltung nicht funktionieren.

Die Schwelle zwischen „geht noch“ und „geht nicht mehr“ nennt man Grenzfrequenz. Computer werden oft knapp unter der Grenzfrequenz betrieben, um die höchst mögliche Rechenleistung zu erreichen. Durch einen großzügigen Abstand zur Grenzfrequenz erreicht man jedoch eine geringe Ausfallrate.

Die Berechnung der Grenzfrequenz ist kompliziert und hängt von sehr vielen Faktoren ab. Wer Nachrichtentechnik studiert, lernt solche Sachen zu beherrschen. Merke dir für den Anfang einfach mal diese Faustregel:

- Je länger die Leitung ist, umso niedriger muss die Frequenz des Signals sein.
- Bei 1 Mhz sind Leitungen bis zu 30 cm Ok
- Bei 100 kHz sind Leitungen bis 3 Meter Ok

6 AVR - Funktionseinheiten

Dieses Kapitel ist ist eine Zusammenfassung der wesentlichen Module aus denen AVR Mikrocontroller zusammengesetzt sind. Ich beschreibe die Eigenschaften, die mir persönlich am wichtigsten erscheinen.

Das Kapitel soll dich darauf vorbereiten, die Datenblätter des Herstellers zu verstehen, mit dem du ab jetzt arbeiten wirst. Besorge dir also einen Ausdruck vom Datenblatt des ATtiny13 und vergleiche meine Angaben mit dem Datenblatt.

Beachte bitte, dass die Programm-Beispiele auf den ATtiny13 bezogen sind. Bei den größeren Modellen heißen die Register und die Bits teilweise ein bisschen anders.

6.1 Prozessor-Kern

Der Prozessor-Kern eines Mikrocontrollers entspricht der CPU eines normalen Computers. Er bestimmt maßgeblich die Leistungsfähigkeit des Computers.

Assembler Programmierer müssen den Aufbau des Prozessor-Kerns sehr genau kennen und bei jeder einzelnen Programm-Zeile berücksichtigen. Denn Assembler Befehle werden 1:1 in Byte-Code übersetzt und vom Prozessor-Kern ausgeführt. Der Kern bestimmt, welche Befehle der Mikrocontroller kennt und ausführen kann.

Für C Programmierer sind die Komponenten des Prozessor-Kerns jedoch weniger bedeutend, weil der Compiler den Befehlssatz der Programmiersprache C in Assembler übersetzt.

Dennoch kann es hilfreich sein, zu wissen, woraus so ein Prozessor-Kern besteht. Er besteht aus folgenden Komponenten:

6.1.1 ALU

ALU bedeutet „Arithmetisch Logische Einheit“. Das ist der innerste Kern jedes Computers. Die ALU führt Befehle aus, indem sie rechnet, verknüpft und Bedingungen prüft. Wenn der Computer zum Beispiel zwei Zahlen miteinander multiplizieren soll, dann weiß die ALU, wie das geht und berechnet das Ergebnis.

Übrigens: Die ALU's aller AVR Mikrocontroller können Subtrahieren, Addieren, bitweise Verschieben und Verknüpfen. Die großen Atmega Modelle können auch multiplizieren, aber keiner kann Dividieren. Dennoch sind Divisionsaufgaben lösbar, und zwar mit genau der Methode, die du in der Grundschule gelernt hast. Glücklicherweise erzeugt der C Compiler den notwendigen Byte-Code automatisch, so dass du einfach Ausdrücke wie a=b/3 schreiben kannst.

6.1.2 Programmzähler

Der Programmzähler ist direkt mit dem Taktgeber verbunden. Er zeigt stets auf das Wort im Programmspeicher, das als nächstes ausgeführt wird und wird normalerweise bei jeden Takt um eins erhöht.

Es gibt allerdings Befehle, die den Programmzähler auf andere Werte Springen lassen. In der Programmiersprache C sind dies die Schleifen, Bedingungen und Funktionsaufrufe.

Der Programmzähler heißt „PC“.

6.1.3 Stapelzeiger

Der Stapelzeiger zeigt stets auf die nächste freie Position des Stapelspeichers.

- Nachdem ein Byte auf den Stapel gelegt wurde, wird 1 vom Zeiger subtrahiert.
- Bevor ein Byte vom Stapel gelesen wird, wird 1 zum Zeiger addiert.

Der Stapelzeiger heißt „SP“.

6.1.4 Allgemeine Register

AVR Mikrocontroller haben 32 allgemeine Register mit 8-Bit Größe, in denen Daten abgelegt werden können. Sie unterscheiden sich zum RAM Speicher darin, dass die ALU schneller auf sie zugreifen kann. Diese Register heißen „R0“ bis „R31“.

Darüber hinaus gibt es noch die drei Register X, Y und Z, die in Kombination mit bestimmten Assembler-Befehlen verwendet werden.

6.1.5 Status Register

Das Status Register heißt SREG. Es speichert das Ergebnis des vorherigen Befehls bitweise.

- Bit 1 heißt „**Zero Flag**“ (kurz: Z)
  Es zeigt an, ob das Ergebnis des vorherigen Befehls eine Null ergeben hat.
- Bit 2 heißt „**Negative Flag**“ (kurz: N)
  Es zeigt an, ob das Ergebnis des vorherigen Befehls negativ war.
- Bit 0 heißt „**Carry Flag**“ (kurz: C)
  Es zeigt an, ob der vorherige Befehl einen Übertrag ausgelöst hat, also das Ergebnis größer als 8 Bit ist (z.B. nach einer Addition).
- Bit 3 heißt „**Overflow Flag**“ (kurz: V)
  Es zeigt an, ob in einer Zweier-Komplementären Operation ein Überlauf stattgefunden hat. Dieses Flag wird von einigen wenigen Befehlen speziell verwendet und ist im Assembler-Handbuch zusammen mit den entsprechenden Befehlen genauer beschrieben.
- Bit 4 heißt „**Sign Bit**“ (kurz: S)
  Das Sign Bit ist eine Exklusiv-Oder Verknüpfung der Bits N und V. Auch dieses Bit hat eine besondere Bedeutung für nur wenige Assembler Befehle.
- Bit 5 heißt „**Half-Carry Flag**“ (kurz: C)
  Dieses Flag zeigt an, ob bei dem letzten Befehl ein Übertrag von Bit 4 nach Bit 5 aufgetreten ist.

Zwei weitere Bits sind keine richtigen Status-Flags, aber dennoch im gleichen Register untergebracht:

- Bit 6 heißt „**Bit Copy Storage**“ (kurz: T)
  Dieses eine Bit kann als temporärer Zwischenspeicher für ein einzelnes Bit verwendet werden. Es gibt spezielle Assembler Befehle, die einzelne Bits aus anderen Registern hierhin kopieren oder umgekehrt von hier in andere Register kopieren.
- Bit 7 heißt „**Global Interrupt Enable**“ (kurz: I)
  Dieses Bit kann dazu verwendet werden, um Programm-Unterbrechungen aufgrund von Hardware-Ereignissen zu erlauben.

Die Status Flags werden im Debugger innerhalb vom AVR Studio besonders angezeigt. Für den Prozessor-Kern sind sie sehr wichtig, denn alle bedingten Befehle fragen die Status-Flags ab. So gibt es in der Assembler-Sprache zum Beispiel eine bedingte Sprunganweisung, die nur dann ausgeführt wird, wenn das Zero-Flag gesetzt ist.

Für C-Programmierer ist in diesem Register lediglich das Global Interrupt Flag relevant, weil man damit die gesamte Interrupt-Logik ein und aus schaltet. Dazu gibt es zwei besondere Funktionen:

- **sbi()** Setzt das Global Interrupt Flag auf 1.
- **cli()** Löscht das Global Interrupt Flag, also auf 0.

6.2 Fuse-Bytes

Mit den Fuse-Bytes konfiguriert man die Eigenschaften des Mikrocontrollers, die bereits vor dem Start des Programms festgelegt sein müssen. Je nach Modell gibt es zwei bis drei Fuse-Bytes. Sie werden mit dem ISP-Programmer eingestellt.

Aufgepasst: Ein Fuse-Bit gilt als „programmiert“ oder „aktiviert“, wenn es den Wert 0 hat! Bei manchen ISP Programmen bedeutet ein ausgefülltes Kästchen, das das Bit auf 1 steht. Bei anderen wiederum bedeutet es genau umgekehrt, dass das Bit 0 ist. Vergleiche die Anzeige deines Programms mit dem Datenblatt, um herauszufinden, wie dein Programm sie anzeigt.

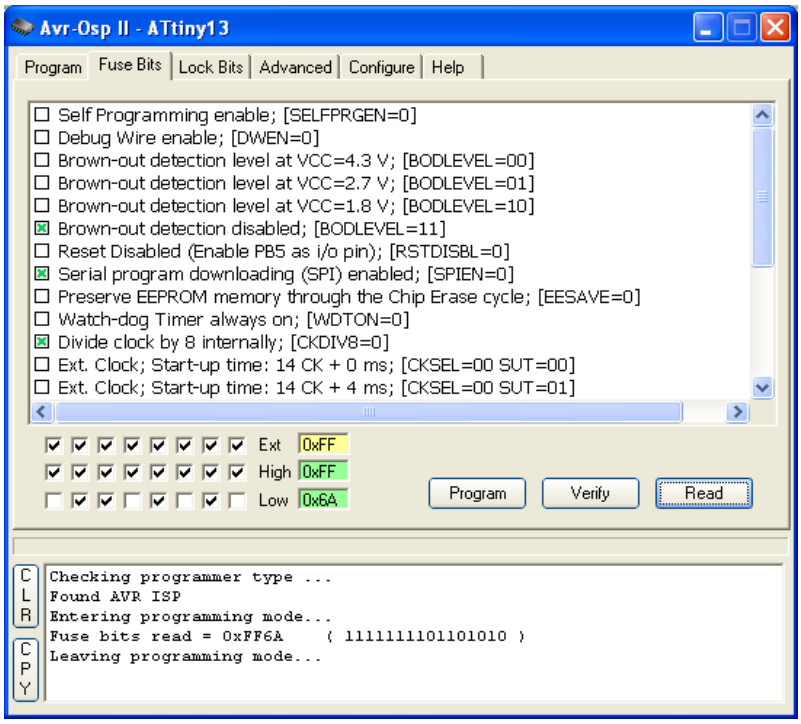
Achtung: Einige falsche Einstellungen der Fuse-Bits führen dazu, dass der Mikrochip nicht mehr per ISP Schnittstelle ansprechbar ist. Du kannst dann auch die Fuse-Bits nicht mehr ändern, jedenfalls nicht ohne einen speziellen High-Voltage Programmer.

Auf jeden Fall solltest du immer zuerst die Fuse-Bytes auslesen (Read), bevor du sie veränderst. Ansonsten besteht ein sehr hohes Risiko, sie falsch einzustellen.

Die Fuse-Bytes haben je nach AVR Modell unterschiedliche Bedeutung! Schau ggf. in das jeweilige Datenblatt.

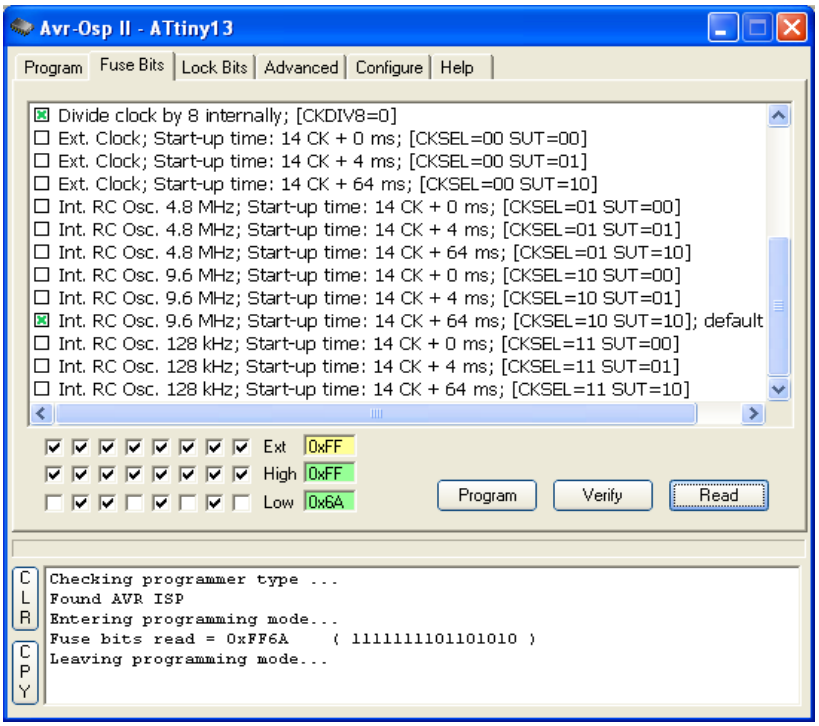
6.2.1 Fuse-Bytes des ATtiny13

Der ATtiny13 hat zwei Fuse-Bytes (auch Fuses genannt) mit den Namen „Low“ und „High“. Im Auslieferungszustand zeigt das Programm AVROSPII sie so an:



- **Self Programming**
  Wenn diese Funktion aktiviert ist, dann kann das laufende Programm den Programmspeicher verändern. Aktiviere diese Funktion sicherheitshalber nur, wenn du sie wirklich benötigst. Auf die Programmierung „von außen“ via ISP Schnittstelle hat diese Einstellung keinen Einfluss.
- **Debug Wire**
  Der Debug Wire ist die Schnittstelle zum JTAG Debugger. Physikalisch gesehen handelt es sich dabei um den Reset-Anschluss. Wenn du den Debug Wire aktivierst, hast du keinen Reset Pin mehr und auch die ISP Schnittstelle ist dann deaktiviert.
  Wenn der Debug Wire aktiviert ist, kann man den Chip nur noch mit einem Debugger programmieren und konfigurieren. Aktiviere den Debug Wire daher nur, wenn du auch ein Debugger besitzt.

- **Brown Out Detection**
  Der Brown-Out Detektor überwacht die Spannungsversorgung und löst einen Reset aus, wenn er einen Aussetzer erkennt. Er kann auf unterschiedliche Schwellwerte eingestellt werden.
- **Reset Disabled**
  Du kannst den Reset-Pin so um konfigurieren, dass er als normaler Port arbeitet, nämlich PB5. Dann ist der Mikrocontroller aber nicht mehr programmierbar, denn die ISP Schnittstelle benötigt einen Reset-Pin. Aktiviere diese Funktion daher niemals!
- **Serial program downloading**
  Diese Funktion aktiviert die ISP Schnittstelle. Du brauchst die ISP Schnittstelle, um den Chip zu programmieren. Deaktiviere die ISP Schnittstelle daher niemals!
- **Preserve EEPROM memory**
  Normalerweise wird der EEPROM Speicher mit gelöscht, wenn du den Programmspeicher löschst. Durch Aktivierung dieser Funktion wird der Inhalt des EEPROM vom Löschvorgang ausgeschlossen.
- **Watchdog Timer always on**
  Der Watchdog löst einen Reset aus, wenn das laufende Programm aufhört, regelmäßig ein „ich lebe noch“ Signal abzusetzen. Normalerweise wird der Watchdog von der Software aktiviert, sobald sie dazu bereit ist. Durch setzen dieses Bits kann man dafür sorgen, dass er immer aktiviert ist. Dann muss das Programm allerdings gleich nach dem Start damit beginnen, die „ich lebe noch“ Signale zu abzusetzen.
- **Divide clock by 8**
  Durch diese Funktion wird die Frequenz des Taktgebers zuerst durch 8 geteilt, bevor das Signal an die einzelnen Funktionseinheiten weiter gereicht wird. Standardmäßig wird der ATtiny13 durch einen 9,6MHz Oszillator getaktet, der durch 8 geteilt wird. Also erhält man eine wirksame Taktfrequenz von 1,2 MHz.



Unterhalb von „Divide clock by 8 internally“ zeigt das Programm AVROSP die Werte von mehreren Fuse-Bits kombiniert an. Hier wählt man aus, welche Quelle den Takt liefern soll und wie lange der Mikrocontroller nach dem Einschalten der Stromversorgung warten soll, bis er das Programm startet.

Man verwendet eine große Wartezeit, wenn die Spannungsversorgung direkt nach dem Einschalten zunächst instabil ist. Generell kann es nicht schaden, immer die größtmögliche Zeit zu wählen.

Benutze die Einstellung „Ext. Clock“ nur, wenn deine Schaltung auch einen externen Taktgeber enthält. Denn ohne Taktsignal funktioniert die ISP Schnittstelle nicht. Eine falsche Einstellung der Taktquelle macht den Chip unbrauchbar.

6.3 Flash Programmspeicher

Der Programmspeicher enthält das Programm, das der Mikrocontroller ausführen soll. Er ist nicht flüchtig. Einmal programmiert, behält er seinen Inhalt auch ohne Stromversorgung, so wie einen Memory-Stick.

Der Programmspeicher wird in der Regel über die ISP-Schnittstelle beschrieben. Bis auf den ATtiny26 können alle AVR Mikrocontroller ihn außerdem durch das laufende Programm beschreiben.

Obwohl es sich um einen 8-Bit Prozessor handelt, ist der Programmspeicher 16-Bit breit. Wegen der Breite sagt man, dass jede Speicherzelle ein Wort ist (bei 8-Bit wäre es ein Byte). Fast alle Befehle sind ein Wort groß.

Die Adresse des ersten Wortes ist 0x0000. Die höchst mögliche Adresse ist 0xFFFF. Daraus ergibt sich, dass der Programmspeicher prinzipiell maximal 65.536 Wörter, entsprechend 128 Kilobytes groß sein kann. Der ATtiny13 hat einen Programmspeicher von 1 Kilobyte Größe, also 512 Wörter.

Der Programmspeicher kann nur block-weise beschrieben werden, wobei jeder Block 16 Wörter groß ist. Laut Datenblatt verträgt der Programmspeicher bis zu 10.000 Schreibzugriffe, bevor mit Verschleiß-Erscheinungen zu Rechnen ist.

6.4 RAM Speicher

Der RAM (Random Access Memory) nimmt variable Daten vom laufenden Programm auf und er beherbergt außerdem den Stapelspeicher.

Der Stapelspeicher funktioniert so ähnlich wie ein Stapel Spielkarten – daher der Name. Wenn man vom Stapelspeicher liest, erhält man zuerst die Daten, die oben liegen. Allerdings kann der Mikrocontroller auch mogeln und sich über diese Reihenfolge hinweg setzen.

Der Stapelspeicher wird von der Programmiersprache C für drei Aufgaben benutzt:

- Wenn eine Funktion aufgerufen wird, wird die Rücksprung-Adresse auf den Stapel gelegt. Das ist die Stelle im Programm, wo es nach Beendigung der Funktion weiter gehen soll.
- Wenn eine Funktion Übergabeparameter hat, werden diese über den Stapel zwischen dem aufrufendem Programmteil und der Funktion ausgetauscht.
- Wenn eine Funktion innere Variablen hat liegen diese physikalisch auf dem Stapel.

Während globale Variablen den RAM Speicher von unten beginnend füllen, hängt der Stapelspeicher quasi kopfüber unter der Decke – am oberen Ende des Speichers. Er beginnt am oberen Ende des Speichers und wächst nach unten hin.

Der ATtiny13 hat einen kleinen RAM Speicher mit nur 64 Byte Größe. Es gibt auch AVR Mikrocontroller, die gar keinen RAM haben – die sind allerdings für die Programmiersprache C ungeeignet und kommen daher in diesem Buch nicht vor.

6.5 Taktgeber

Alle derzeit existenten Computer arbeiten mit einem Taktgeber. Der Taktgeber bestimmt, wie schnell der Mikroprozessor arbeitet.

AVR Mikrocontroller dürfen beliebig langsam getaktet werden. Aber nach oben hin gibt es Grenzen, die vom Modell und von der Spannungsversorgung abhängt.

Über die Fuse-Bytes wählst du eine der folgenden Taktquellen aus:

- Interner R/C Oszillator, kann je nach Modell auf unterschiedliche Frequenzen eingestellt werden. Die Frequenz weicht typischerweise bis zu 5% vom Soll ab.
- Externer R/C Oszillator. Nicht besser als der interne, daher recht uninteressant.
- Externer Schwing-Quartz. Liefert eine sehr genaue Taktfrequenz mit weniger als 0,0001% Abweichung zum Soll. Elektrische Uhren beziehen ihren Takt aus einem Quartz.
- Externer Keramik Resonator (ist so was ähnliches wie ein Quartz, nur weniger präzise).
- Externer Taktgeber. In diesem Fall erwartet der AVR Mikrocontroller ein sauberes digitales Signal von beliebiger Quelle. Wird oft verwendet, wenn man mehrere Mikrocontroller mit einem gemeinsamen Taktgeber Synchron betreibt.

Der ATtiny13 unterstützt nicht alle dieser Möglichkeiten, sondern nur:

- Interner R/C Oszillator mit wahlweise 128 kHz, 4,8 MHz oder 9,6 MHz.
- Externer Taktgeber.

Standardmäßig wird der interne R/C Oszillator mit 9,6 MHz verwendet und diese Frequenz durch 8 geteilt.

6.6 Reset

Die Reset-Einheit sorgt dafür, dass der Mikrocontroller beim Einschalten der Stromversorgung in einen definierten Anfangs-Zustand versetzt wird:

- Alle Anschlüsse werden als Eingang ohne Pull-Up Widerstand konfiguriert.
- Der Stapelzeiger zeigt auf die letzte Speicherzelle des RAM.
- Alle weiteren Funktionen werden so initialisiert, dass sie deaktiviert sind bzw. sich neutral verhalten.
- Die Programmausführung beginnt an Adresse Null.

Die Reset-Einheit hat keinen Einfluss auf den Inhalt des RAM Speichers. Nach dem Einschalten der Stromversorgung enthält das RAM lauter Zufallswerte.

Die Reset-Einheit kann recht flexibel konfiguriert werden. Zum einen kann man einstellen, wie lange nach dem Einschalten der Stromversorgung gewartet werden soll, bis das Programm startet. Zum anderen kann man mit den Fuse-Bytes den „Brown Out Detector“ aktivieren, der Aussetzer in der Stromversorgung erkennt und dann ggf. einen Reset auslöst.

Weiterhin kann ein Reset auch von außen ausgelöst werden, nämlich durch den Reset-Eingang. Dein erster Mikrocontroller ist dazu mit einem entsprechenden Taster ausgestattet. Der Reset wird aktiviert, indem man den Eingang auf Low zieht. Er hat einen internen Pull-Up Widerstand.

Der Reset-Eingang hat beim Programmieren noch zwei besondere Funktionen:

- Bei kleinen Mikrocontrollern dient der Reset-Pin als Debug-Wire, also die Verbindung zum Debugger.
- Solange der Reset-Pin auf Low gezogen wird, ist die ISP Schnittstelle aktiviert. Ohne Reset-Pin kannst du den Mikrocontroller also nicht programmieren.

6.7 Ports

Ports sind gewöhnliche Eingänge und Ausgänge. Per Software kannst du den Status der Eingänge abfragen und bestimmen, welche Signale (Low/High) an den Ausgängen heraus kommen sollen.

Bei AVR Mikrocontrollern heißen die Ports: Port A, Port B, Port C und so weiter. Jeder Port hat bis zu acht Anschlüsse, die von 0 bis 7 durchnummeriert sind. Der Anschluss 0 von Port B wird kurz PB0 genannt.

Zur Programmierung hat jeder Port genau drei Register, die bitweise arbeiten:

- **DDRx**
  Das „Data Direction Register“ konfiguriert die Daten-Richtung. 0=Eingang, 1=Ausgang.
- **PORTx**
  Bei Ausgängen bestimmt dieses Register, welches Signal (Low/High) aus dem Ausgange heraus kommt. Bei Eingängen schalten Bit mit dem Wert 1 die internen Pull-Up Widerstände ein.
- **PINx**
  Bei Eingängen liest man über dieses Register die Signale ein. Bei Ausgängen hat dieses Register keine definierte Bedeutung.

Viele Anschlüsse lassen sich für unterschiedliche Sonderfunktionen konfigurieren. So können sie z.B. das Ausgangssignal eines Timers liefern oder als serielle Schnittstelle dienen. Nach einem Reset sind diese Sonderfunktionen zunächst nicht aktiviert. Dann arbeiten alle Anschlüsse (mit Ausnahme des Reset-Eingangs) als normaler „Port“.

6.7.1 Programmierung

Die Befehle

```
DDRB=13;
DDRB=1+4+8;
DDRB=1|4|8;
DDRB=0b00001101;
DDRB=(1<<0) + (1<<2) + (1<<3);
DDRB=(1<<0) | (1<<2) | (1<<3);
```

konfigurieren alle gleichermaßen PB0, PB2 und PB3 als Ausgang, während alle anderen Pins von Port B Eingänge sind. Dies sind alles alternative Schreibweisen, von denen du je nach Geschmack eine auswählen kannst.

- Bei den ersten drei Ausdrücken musst du die Werte der einzelnen Bits auswendig kennen und zusammen addieren oder verknüpfen.
  - Bit 0 = 1
  - Bit 1 = 2
  - Bit 2 = 4
  - Bit 3 = 8
  - Bit 4 = 16
  - Bit 5 = 32
  - Bit 6 = 64
  - Bit 7 = 128
- Bei dem vierten Ausdruck verwendest du binäre Zahlen, die der GNU C Compiler zwar versteht, aber in der Spezifikation der Programmiersprache C nicht vorgesehen sind.
- Die letzen beiden Ausdrücke benutzen geklammerte Schiebe-Befehle um mit Bit-Nummern anstatt mit deren Werten zu arbeiten. (1<<n) ist der Wert von Bit n.

Letztendlich erzeugen alle sieben Ausdrücke exakt den gleichen Byte-Code.

Um nun die Ausgänge PB2 und PB3 auf High zu schalten und alle anderen Bits auf Low zu schalten, brauchst du einen dieser Befehle:

```
PORTB = 12;
PORTB = 4+8;
PORTB = 4|8;
PORTB = 0b00001100;
PORTB = (1<<2) + (1<<3);
PORTB = (1<<2) | (1<<3);
```

Beachte, dass du hier alle acht Bits von Port B ansprichst. Es wird ein neuer Wert in das Register PORTB geschrieben, ohne den vorherigen Zustand des Registers zu beachten.

Es kommt allerdings viel öfters vor, dass nur ein einzelnes Bit oder mehrere Bits (aber eben nicht alle auf einmal) verändert werden sollen.

6.7.2 Einzelne Bits setzen

Um einen einzelnen Ausgang (sagen wir Bit 3) auf High zu setzen, benutzt man die bitweise Oder-Verknüpfung mit Zuweisung:

```
PORTB |= 8;
PORTB |= 0b00001000;
PORTB |= (1<<3);
```

Das gleiche Prinzip funktioniert auch mit mehreren Ausgängen. Die folgenden Befehle schalten Bit 2 und 3 auf High während alle anderen Bits unverändert bleiben:

```
PORTB |= 4+8;
PORTB |= 0b00001100;
PORTB |= (1<<2) + (1<<3);
PORTB |= (1<<2) | (1<<3);
```

Wenn man PORTB mit 12 Oder-Verknüpft, bleiben alle Bits unverändert, außer die Bits 2 und 3, die den Wert 1 haben:

PORTB	x	x	x	x	x	x	x	x
Oder Operand	0	0	0	0	1	1	0	0
Ergebnis	x	x	x	x	1	1	x	x

6.7.3 Einzelne Bits löschen

Um einzelne Bits zu löschen (auf Low setzen), verwendet man die Negation zusammen mit einer bitweisen Und-Verknüpfung:

```
PORTB &= ~(4+8);
PORTB &= ~0b00001100;
PORTB &= ~((1<<2) + (1<<3));
PORTB &= ~((1<<2) | (1<<3));
```

Erklärung: Die Ausdrücke auf der rechten Seite haben wie bereits gesagt letztendlich alle den gleichen Wert, nämlich 12. Durch die Negation werden alle Bits umgekehrt:

12 = 0b00001100

~12 = 0b11110011

Wenn nun der Port B mit !12 Und-Verknüpft wird, bleiben alle Bits unverändert, außer die Bits 2 und 3, wie die den Wert Null haben:

PORTB	x	x	x	x	x	x	x	x
Und Operand	1	1	1	1	0	0	1	1
Ergebnis	x	x	x	x	0	0	x	x

6.7.4 Einzelne Bits umschalten

Um einzelne Bits umzuschalten (Toggeln) verwendet man die bitweise Exklusiv-Oder Verknüpfung mit Zuweisung.

```
PORTB ^= 4+8;
PORTB ^= 0b00001100;
PORTB ^= (1<<2) + (1<<3);
PORTB ^= (1<<2) | (1<<3);
```

Wenn man PORTB mit 12 Oder-Verknüpft, bleiben alle Bits unverändert, außer die Bits 2 und 3, die im Operand den Wert 1 haben:

PORTB	x[2]	x	x	x	0	1	x	x
Exklusiv-Oder Operand	0	0	0	0	1	1	0	0
Ergebnis	x	x	x	x	1	0	x	x

6.7.5 Eingänge abfragen

Um Eingänge abzufragen, verwendest du das PINx Register, also PINB wenn es Port B sein soll. PINB liefert eine 8-Bit Integer Zahl entsprechend der acht Eingänge PB0...PB7.

Meistens soll aber nur ein einzelnes Bit abgefragt werden. Dazu benutzt man eine Und-Verknüpfung. Die folgenden Ausdrücke bewirken alle das Gleiche, sie fragen das Bit 1 ab, also den Eingang PB2.

```
if (PINB & 2) {…}
if (PINB & 0b00000010) {…}
if (PINB & (1<<2)) {…}
```

Die Bedingung ist erfüllt, wenn das abgefragte Bit auf 1 gesetzt ist, also wenn der entsprechende Eingang auf High liegt. Denn dann ergibt der Ausdruck einen Wert, der größer als Null ist.

6.8 Analog/Digital Wandler

An die analogen Eingänge kannst du Sensoren anschließen, die kontinuierlich veränderliche Werte liefern, zum Beispiel Temperaturfühler. Der Analog/Digital Wandler (kurz ADC) misst analoge Spannungen wie ein Digital-Multimeter und liefert als Ergebnis einen numerischen Wert im Bereich 0...1023.

- 0, wenn die Spannung Null Volt ist
- 512, wenn die Spannung halb so hoch ist, wie die Referenzspannung
- 1023, wenn die Spannung größer oder gleich der Referenzspannung ist

Die Referenzspannung ist je nach AVR Modell auf mehr oder weniger unterschiedliche Werte einstellbar:

- 1,1 Volt
- 2,56 Volt
- VCC

2 Das "x" bedeutet Egal

Der ADC benötigt eine Taktfrequenz im Bereich 50kHz bis 200kHz. Sie wird mit Hilfe eines Teilers aus der Taktfrequenz des Prozessor-Kerns gewonnen. Wenn der Prozessor-Kern mit 1,2Mhz getaktet wird, dann stellst du den Teiler beispielsweise auf 16 ein, das ergibt dann für den ADC eine Taktfrequenz von 1.200.000:16=75.000 Hz.

Jede Messung (bis auf die erste) benötigt 13 Takte. Wenn du die Taktfrequenz durch 13 teilst, erhältst du die Anzahl von Messungen pro Sekunde.

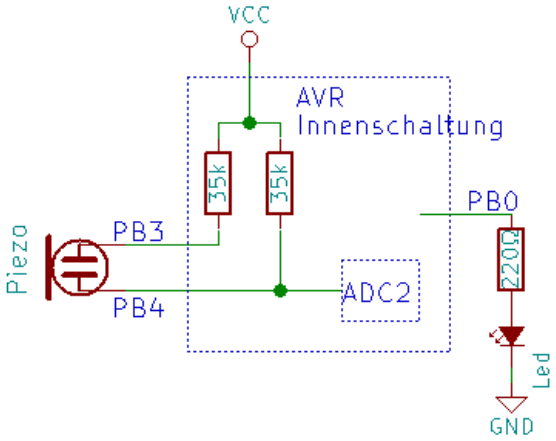
Der Analog/Digital Wandler hat mehrere Eingänge mit Namen ADC0...ADC7. Der Wandler kann aber immer nur einen Eingang gleichzeitig messen. Man muss also vor jeder Messung einstellen, welchen Eingang man benutzen will.

6.8.1 Programmierung

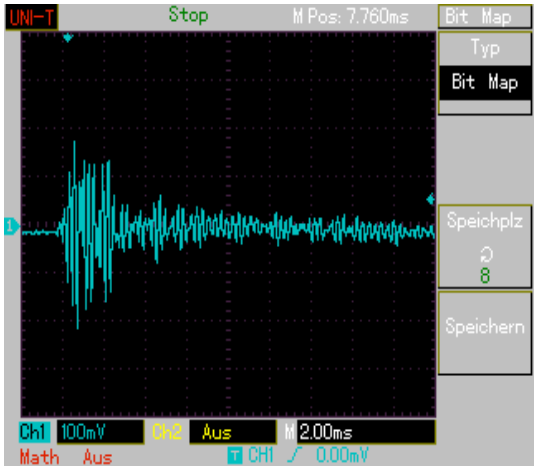
Um den ADC zu programmieren, musst du auf seine Register zugreifen. Das Datenblatt beschreibt diese Register im Kapitel „Analog to Digital Converter“. Schlage im Datenblatt das Kapitel für den ADC auf und überprüfe meine Angaben.

Ich erkläre die Programmierung des ADC anhand eines Experimentes. Wir wollen einen Klatsch-Schalter bauen. Er schaltet ein Licht an, wenn man laut klatscht. Mit dem Reset-Taster kann man das Licht wieder aus schalten.

Der Piezo-Schallwandler deines Mikrocomputers wird dieses mal als Mikrofon eingesetzt. Bei dieser Gelegenheit zeige ich dir das Schaltzeichen des Piezo-Schallwandlers. Es ist das Teil ganz links im folgenden Block-Schaltbild:



Der Schallwandler ist an die Anschlüsse PB4 (=ADC2) und PB3 angeschlossen. Wir aktivieren die Pull-Up Widerstände, um an beiden Eingängen im Ruhezustand eine definierte Eingangsspannung zu erhalten. Wenn der Schallwandler ein lautes Geräusch empfängt (z.B. Klatschen), gibt er eine Spannung ab, etwa so:



Die Spannung am analogen Eingang verändert sich zusammen mit der Schallwelle um einige Millivolt. Diese Veränderung soll das folgende Programm erkennen und mit einer roten Leuchtdiode anzeigen.

Wenn das Mikrofon wie im obigen Schaltplan gezeichnet angeschlossen wird, dann ruht der Engang des ADC auf Höhe der Versorgungsspannung (was dem Zahlenwert 1023) entspricht. Das laute Geräusch bewirkt, dass sich diese Spannung für mehrere kurzen Momente verringert. Wir müssen also einfach nur fortlaufend den ADC auslesen und kontrollieren, ob er einen Zahlenwert von deutlich weniger als 1023 liefert.

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

int main(void)
{
    DDRB  = 0b00000001;
    PORTB = 0b00011000;
    ADMUX = 2;
    ADCSRA = (1<<ADEN)+(1<<ADSC)+(1<<ADATE)+(1<<ADPS2);

    _delay_ms(1000);

    while (1)
    {
        while ((ADCSRA & (1<<ADIF))==0);
        if (ADC<1014)
        {
            PORTB |= 0b00000001;
            _delay_ms(1000);
        }
        ADCSRA |= (1<<ADIF);
    }
}
```

Zuerst konfigurieren wir den Anschluss PB0 als Ausgang, weil dort die Leuchtdiode angeschlossen ist:

```
DDRB  = 0b00000001;
```

Dann schalten wir die beiden Pull-Up Widerstände ein:

```
PORTB = 0b00011000;
```

Wir stellen den ADC so ein, dass er den Eingang ADC2 misst. Außerdem wird die Referenzspannung auf VCC gestellt, also der Versorgungsspannung. Mit ADC3 würde es auch gehen:

```
ADMUX = 2;
```

Und dann starten wir den ADC:

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

Spätestens jetzt solltest du in das Datenblatt schauen, um heraus zu finden, was hier passiert.

ADEN ist eine Definition mit dem Wert 7. Der Ausdruck (1<<ADEN) entspricht also (1<<7), was wiederum 128 ergibt. Das ist der Wert des Bits mit dem Namen ADEN. Auf diese Weise greift man in C Programmen üblicherweise auf benannte Bits in Registern zu.

Laut Datenblatt haben die Bits folgende Bedeutung:

- ADEN schaltet den ADC ein
- ADSC startet die erste Messung
- ADATE aktiviert automatische Wiederholung der Messung. In diesem Zusammenhang bestimmt das Register ADCSRB, welche Signalquelle den Start der folgenden Messungen auslösen soll. ADCSRB hat standardmäßig den Wert 0b00000000, was dem „Free running mode“ entspricht. Der ADC wiederholt die Messungen in diesem Modus fortlaufen von selbst so schnell er kann.
- Wenn von den ADPS Bits nur das Bit ADPS2 gesetzt ist, dann wird die Taktfrequenz des Prozessorkerns durch 16 geteilt und damit der ADC getaktet. 1.2000.000:16=75.000. Der ADC wird also mit 75 Kilohertz getaktet. Da er pro Messung 13 Takte benötigt, kommen wir auf ungefähr 5769 Messungen pro Sekunde.

Dann wartet das Programm eine Sekunde, bevor es die Messergebnisse auswertet. Diese Wartezeit ist wichtig, damit der Klatsch-Schalter nicht gleich beim Programmstart falsch auslöst. Denn in dem Moment, wo wir die Pull-Up Widerstände einschalten, entsteht ein großer Spannungsprung, der den Schallwandler kurzzeitig zum Schwingen anregt. Diese Schwingung soll nicht als Klatschen fehl-interpretiert werden.

Nach einer Sekunde Wartezeit können wir ziemlich sicher sein, dass die Schwingungen aufgehört haben. Wahrscheinlich würden 0.1 Sekunden auch genügen. Du kannst ja mal ausprobieren, wie viel Wartezeit wirklich nötig ist.

In der folgenden Endlosschleife liest das Programm immer wieder den gemessenen Wert des ADC aus und schaltet die LED an, wenn ein lautes Geräusch gemessen wurde:

```
while ((ADCSRA & (1<<ADIF))==0);
if (ADC<1014)
{
    PORTB |= 0b00000001;
}
ADCSRA |= (1<<ADIF);
```

Die erste Zeile ist eine leere Warteschleife. Sie wird so lange wiederholt, bis das Bit ADIF auf 1 geht. Mit diesem Bit signalisiert der ADC, dass er eine Messung fertig gestellt hat. Das Ergebnis kann dann als 16-Bit Wert aus dem Register ADC ausgelesen werden (oder als zwei 8-Bit Werte aus ADCL und ADCH).

Wir erwarten folgende Messwerte:

- Wenn völlige Stille herrscht, liegt der Eingang des ADC auf Höhe der Versorgungsspannung, weil der Pull-Up Widerstand ihn auf VCC zieht. Der Eingang liegt also gleichem Niveau mit der Referenzspannung. Folglich liefert der ADC den Zahlenwert 1023.
- Wenn ein lautes Geräusch kommt, wird die Spannung zeitweise unter der Versorgungsspannung liegen. Wie weit, wissen wir noch nicht, sicher ist aber, dass wir dann Zahlenwerte kleiner als 1023 bekommen werden.

Das Programm soll also prüfen, ob der Wert deutlich kleiner als 1023 ist, und wenn ja, dann soll die LED eingeschaltet werden.

Mit diesem Wissen habe ich unterschiedliche Werte ausprobiert und bin zu dem Schluss gekommen, dass mein Klatsch Schalter mit dem Wert 1014 zufriedenstellend arbeitet. Du verwendest einen anderen Piezo-Schallwandler, deswegen ist bei dir wahrscheinlich ein etwas anderer Zahlenwert besser.

Wir können jetzt ausrechnen, wie viel Volt das eigentlich sind. Angenommen deine Batterien liefern 3,6 Volt, dann gilt folgende Rechnung:

Die Spannung für den Wert 1 entspricht 3,6 Volt : 1024, das ergibt etwa 3,5 Millivolt. Mein Klatsch-Schalter reagiert, wenn die Spannung kleiner als 1014 ist, also mindestens 10 weniger als im Ruhezustand. 10 · 3,5 mV = 35 mV. Mein Klatsch-Schalter reagiert also bei einer Signalspannung ab 35 mV.

Es gibt noch eine Programmzeile zu erklären:

```
ADCSRA |= (1<<ADIF);
```

Wir haben gewartet, bis das ADIF Bit auf 1 geht, um zu erkennen, dass eine Messung fertig ist. Nun müssen wir das Bit auf 0 stellen, damit wir erneut auf das Ergebnis der nächsten Messung warten können. Laut Datenblatt setzt man das ADIF Flag zurück, indem man eine 1 hinein schreibt (nicht eine 0)! Das hast du sicher nicht erwartet. Lies also immer aufmerksam das Datenblatt, bevor du eine Funktion verwendest, mit der du noch nicht vertraut bist.

6.8.2 Übungsaufgaben:

1. Finde heraus, bei welchen Zahlenwert dein Klatsch-Schalter am besten reagiert. Berechne anschließend die Schwellenspannung.
2. Ändere das Programm so, dass die LED nach drei Sekunden von selbst wieder aus geht.

Zu diesen Übungsaufgaben gibt es keinen Lösungsvorschlag. Von jetzt an sollst du deine Lösungsansätze selbst erarbeiten und bewerten.

6.9 Analog Vergleicher

Eine kleine Nebenfunktion des ADC ist der „Analog Comparator“. Er vergleicht die Spannung der beiden Eingänge AIN0 und AIN1 (nicht zu Verwechseln mit ADC0 und ADC1). Wenn die Spannung an AIN0 höher ist, als die Spannung an AIN1, dann ist das Ergebnis „1“. Ansonsten ist das Ergebnis „0“.

Weiterhin kann der Analog Vergleicher anstelle von AIN1 die Eingänge des ADC verwenden, sofern der ADC nicht aktiviert ist. Man stellt dann im Register ADMUX ein, welcher ADC-Eingang verwendet werden soll.

6.9.1 Programmierung

Der Analog Vergleicher ist standardmäßig eingeschaltet und sofort Einsatzbereit. Das Ergebnis befindet sich im Bit ACO im Register ACSR.

Um die Eingänge AIN0 und AIN1 miteinander zu vergleichen, geht man folgendermaßen vor:

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

int main(void)
{
    DDRB  = 0b00000001; // PB0 (LED)=Ausgang
    while (1)
    {
        if (ACSR & (1<<ACO))
        {
            PORTB |= 0b00000001; // LED an
        }
        else
        {
            PORTB &= !0b00000001; // LED aus
        }
    }
}
```

Hier wird das Ergebnis des Vergleichers direkt im Rahmen einer if(...) Entscheidung verwendet. Anders als beim ADC ist es nicht notwendig, auf das Ergebnis des Vergleiches zu warten.

6.10 Interrupts

Interrupts sind ein Mittel, das laufende Programm beim Auftreten besonderer Ereignisse zu unterbrechen. Interrupts treten zum Beispiel auf, wenn der ADC eine Messung beendet hat.

Wenn ein Interrupt auftritt, wird das laufende Hauptprogramm kurzzeitig unterbrochen. Stattdessen wird die Interrupt-Funktion ausgeführt. Wenn diese fertig ist, wird das Hauptprogramm fortgesetzt.

Interrupt-Funktionen können nicht durch weitere Interrupts unterbrochen werden. Wenn ein zweiter Interrupt ausgelöst wird, während noch eine andere Interrupt-Funktion aktiv ist, dann wird die zweite Funktion erst nach der ersten ausgeführt.

Fast jede Funktionseinheit des AVR Mikrocontrollers kann Interrupts auslösen. Sogar die Ports können Interrupts auslösen, wenn sie einen Signal-Wechsel erkennen.

Im Datenblatt gibt es ein Kapitel mit dem Namen „Interrupts“. Dort sind die Interrupt Vektoren als Tabelle aufgelistet. Diese Tabelle brauchst du, um deine Interrupt-Funktionen korrekt zu deklarieren.

6.10.1 Programmierung

Um die Programmierung von Interrupts auszuprobieren, benutzen wir nochmal den kleinen Mikrocomputer als Klatsch-Schalter. Wir hatten bisher in einer Endlos-Schleife immer wieder darauf gewartet, dass der ADC mit seiner Messung fertig ist, bevor wir das Messergebnis ausgelesen haben. Nun wollen wir genau dieses Warten durch eine Interrupt-Funktion ersetzen.

Der Interrupt-Vektor des ADC heißt laut Datenblatt „ADC“ – wie praktisch. Die Funktion muss daher „ISR(ADC_vect)“ heißen.

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

ISR(ADC_vect)
{
    if (ADC<1014)
    {
```

```
        PORTB |= 0b00000001;
    }
}

int main(void)
{
    DDRB  = 0b00000001;
    PORTB = 0b00011000;
    ADMUX = 2;
    ADCSRA = (1<<ADEN)+(1<<ADSC)+
        (1<<ADATE)+(1<<ADIE)+(1<<ADPS2);

    _delay_ms(1000);
    sei();

    while (1) {}
}
```

Ich habe die neuen Teile hervorgehoben.

Die neue Interrupt-Funktion wird das Hauptprogramm immer dann unterbrechen, wenn der ADC einen Interrupt ausgelöst hat. Dann können wir den gemessenen Wert direkt verarbeiten, ohne warten zu müssen. Auch brauchen wir das Interrupt-Flag nicht mehr zurück setzen, weil der ADC das beim Aufruf der Interrupt-Funktion schon automatisch macht (steht so im Datenblatt).

Damit der ADC überhaupt Interrupts auslöst, setzen wir beim Konfigurieren zusätzlich das ADIE-Bit. Außerdem müssen wir das globale Interrupt-Flag im Status Register des Prozessor-Kerns setzen, sonst würde die Interrupt-Funktion niemals aufgerufen werden. Auf das Status Register können wir in der Programmiersprache C nicht direkt zugreifen, daher gibt es zwei spezielle Funktionen:

- sei() „Set Interrupt-Flag“ setzt das Interrupt-Flag im Status Register
- cli() „Clear Interrupt-Flag“ löscht das Interrupt-Flag im Status Register

Wir setzen das Flag allerdings erst nach der Sekunde Wartezeit, damit unsere Interrupt Funkton nicht zu früh aufgerufen wird. Wir wollen die Messergebnisse des ADC ja erst nach Ablauf der Wartezeit auswerten.

Das Hauptprogramm ist nun ziemlich klein geworden. Es besteht nur noch aus einer leeren Endlosschleife. Das ist übrigens oft so. Viele Mikrocontroller-Programme sind komplett Interrupt-Gesteuert und ohne „richtiges“ Hauptprogramm.

6.11 Externe Interrupts

Mit externen Interrupts sind Signale von außerhalb des Mikrocontrollers gemeint, die Interrupt auslösen. Jeder Port-Pin kann einen Interrupt auslösen, wenn sich ein Signal von Low nach High oder umgekehrt ändert. Das funktioniert sowohl bei Eingängen als auch bei Ausgängen.

Darüber hinaus haben AVR Mikrocontroller spezielle Eingänge mit Name INT0, INT1, usw. bei denen man etwas feiner konfigurieren kann, wann sie einen Interrupt auslösen sollen. Und zwar hat man dort diese Auswahlmöglichkeiten:

- Interrupt wird wiederholt ausgelöst, solange der Eingang auf Low ist.
- Ein Signalwechsel löst den Interrupt aus (wie bei jeden anderen Port-Pin).
- Interrupt wird ausgelöst, wenn der Eingang von High nach Low wechselt.
- Interrupt wird ausgelöst, wenn der Eingang von Low nach High wechselt.

6.11.1 Programmierung

Das folgende Programm kannst du sehr schön im Debugger ausprobieren.

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

ISR(PCINT0_vect)
{
    PORTB |= 0b00000001;  // LED einschalten
}

int main(void)
{
    DDRB = 0b00000111;  // PB0, PB1 und PB2 sind Ausgang

    PCMSK = 0b00000110; // Wählt Pin PB1 und PB2 aus
    GIMSK = (1<<PCIE);  // Interrupts für diese Pins erlauben
    sei();              // Interrupts global erlauben

    PORTB |= 0b00000010; // PB1 auf High schalten

    while (1)
    {
        PINB;
    }
}
```

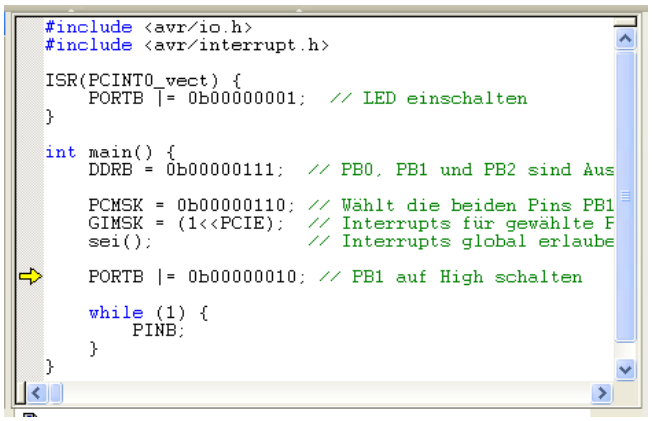
Der Interrupt-Vektor für Port Pin-Change heist PCINT0. Die Interrupt Funktion heißt daher „ISR(PCINT0_vect)“. Wenn der Interrupt auftritt, soll sie die LED an PB0 einschalten.

Im Hauptprogramm werden zunächst die drei Ports PB0, PB1 und PB2 als Ausgang konfiguriert. Dann wird die Auslösung von Interrupts für die beiden Pins PB1 und PB3 aktiviert.

Als nächstes setzt das Programm den Ausgang PB1 auf High. Dadurch wird von diesem Pin ein Interrupt ausgelöst. Das Hauptprogramm wird unterbrochen und die Interrupt-Funktion wird ausgeführt. Danach wird das Hauptprogramm fortgesetzt.

In diesem Fall liest das Hauptprogramm einfach immer wieder von PINB ohne den Wert jedoch weiter zu verarbeiten. Ich habe das gemacht, weil der Simulator sich mit leeren Schwer tut. So lässt sich das Programm besser im Simulator ausprobieren. Genau das tun wir jetzt:

Drücke F7, um das Programm zu kompilieren. Klicke dann auf den Menüpunkt Debug/Start Debugging um die Simulation zu starten:



Drücke dann F11 so oft, bis der gelbe Pfeil auf die oben dargestellte Zeile zeigt. Das Interrupt-System ist nun bereit, ausgelöst zu werden. Wenn du jetzt noch einmal F11 drückst, wechselt der Pin PB1 von Low nach High.

Wenn du jetzt noch ein paar mal F11 drückst, wirst du sehen, dass das Hauptprogramm einmal durch die Interrupt-Funktion unterbrochen wird.

Wenn du ganz genau hinschaust, fällt dir auch auf, dass der Signalwechsel von PB1 nicht sofort zum Interrupt führt, sondern erst 2 Takte verzögert. Das ist so in Ordnung, denn dies geht auch aus dem Diagramm im Kapitel „Pin Change Interrupt Timing“ des Datenblattes hervor.

6.11.2 Zugriff auf Variablen durch Interrupt-Routinen

Wenn du Variablen sowohl innerhalb als auch außerhalb von Interrupt-Funktionen veränderst, kann es zu einem Konflikt kommen:

```
uint16_t zaehler=0;

ISR(...)
{
    zaehler=zaehler+1;
}

int main(void)
{
    …
    zaehler=zaehler+100;
    …
}
```

Wenn das Hauptprogramm ausgeführt wird, kann folgendes unerwünschte Szenario passieren:

- Angenommen, die Variable zaehler hat gerade den Wert 400.
- Die Variable „zaehler“ wird in ein Prozessor-Register geladen, weil arithmetische Operationen nicht direkt im RAM ausführbar sind.
- Dann wird das Register um 100 erhöht, also auf 500.
    - Dann wird das Hauptprogramm durch einen Interrupt unterbrochen.
    - Die Interrupt-Routine erhöht den Wert der Variable um 1, also auf 401.
    - Das Hauptprogramm wird fortgesetzt.
- Das Hauptprogramm schreibt den Wert des Registers zurück in die Variable.
- Die Variable hat jetzt den Wert 500, richtig wäre aber 501 gewesen.

Auch wenn du Ausdrücke wie „zaehler+=100“ verwendest, tritt der gleiche Fehler gelegentlich auf. Du kannst sie Situation verbessern, indem du dem Prozessor an der kritischen Stelle verbietest, das Programm zu unterbrechen:

```
uint16_t zaehler;

ISR(...)
{
    zaehler=zaehler+1;
}

int main(void)
{
    …
    cli();
    zaehler=zaehler+100;
    sei();
    …
}
```

In dem so geänderten Hauptprogramm kann die Zeile „zaehler=zaehler+100“ nicht mehr unterbrochen werden.

Es gibt noch ein anderes recht häufiges Konflikt-Szenario:

```
uint8_t weitermachen=1;

ISR(...)
{
    weitermachen=0;
}

int main(void)
{
    …
    while (weitermachen)
    {
        // tu etwas
    }
    …
}
```

Bei dem obigen Programm geht es darum, dass das Hauptprogramm immer wieder etwas machen soll, solange die Variable „weitermachen“ den Wert 1 hat. Irgendein Ereignis (z.B. ein elektrisches Signal von außen) löst den Interrupt aus, und dann wird die Variable auf 0 gesetzt. Die while Schleife soll dann beendet werden.

So wie das Programm oben geschrieben ist, wird es nicht wie erwartet funktionieren, denn der Optimierer verändert den Programmcode ungefähr so:

```
uint8_t weitermachen=1;

ISR(...)
{
    weitermachen=0;
}

int main(void)
{
    …
    R2 = weitermachen;
    while (R2)
    {
        // tu etwas
    }
    …
}
```

Der Optimierer kopiert die Variable in ein Prozessor-Register, weil der wiederholte Zugriff auf Register schneller geht, als der wiederholte Zugriff auf eine Variable, die im RAM liegt. Das hat sich der Compiler ganz nett gedacht, aber in diesem Fall führt es dazu, dass die while Schleife niemals endet. Denn die Schleife prüft nun immer wieder den Wert von R2, während beim Interrupt aber nur der Wert von „weitermachen“ geändert wird.

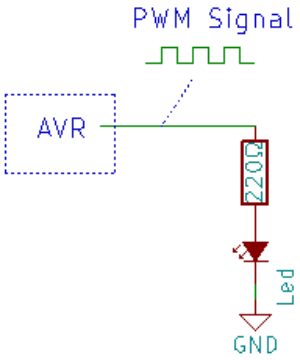
Dieses Problem kannst du lösen, indem du die Variable als „volatile“ deklarierst:

```
volatile uint8_t weitermachen=1;
```

Durch das Schlüsselwort „volatile“ teilst du dem Compiler mit, dass er den Zugriff auf diese Variable nicht optimieren darf. Damit verhält sich das Programm wie erwartet. Generell ist es keine schlechte Idee, alle Variablen als „volatile“ zu deklarieren, die von Interrupt-Funktionen verändert werden und außerhalb der Interrupt-Funktionen gelesen werden.

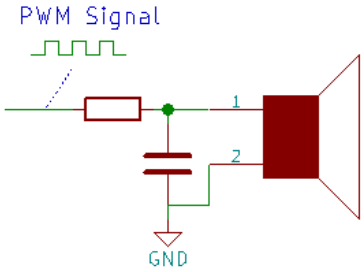
6.12 Timer

Timer dienen dazu, Ereignisse zu zählen, Zeiten zu messen, oder PWM Signale zu erzeugen. PWM Signale sind regelmäßige elektrische Impulse, deren Pulsbreiten variiert werden. PWM Signale kann man z.B. dazu verwenden, die Helligkeit einer Lampe oder die Leistung eines Motors zu steuern.



Wenn die Impulse schmal sind, leuchtet die LED nur schwach. Wenn die Impulse breiter sind, leuchtet die LED heller. Man muss die Impulsfrequenz nur hoch genug wählen (z.B. 400 Hz), dass das menschliche Auge kein Flackern sehen kann.

CD-Player benutzen PWM, um das digitale Tonsignal in eine analoge Spannung für die Lautsprecher umzuwandeln. Ein zusätzlicher Filter (z.B. aus Widerstand und Kondensator) glättet das Ausgangssignal, so dass eine analoge Spannung entsteht, deren Höhe durch Modulation der Pulsbreiten verändert wird.



Im Folgenden beschreibe ich die Fähigkeiten des Timers vom ATtiny13. Die Timer der größeren AVR Mikrocontroller haben weitere Fähigkeiten, die du dem jeweiligen Datenblatt entnehmen kannst. Du kannst durch Programmierung festlegen, welches Signal den Timer takten soll:

- Impulse am Eingang TCNT0, wahlweise bei Wechsel von Low nach High oder umgekehrt.
- Prozessor-Takt, optional geteilt durch 8, 64, 256 oder 1024.

Der Timer vom ATtiny13 ist 8-Bit groß, demzufolge kann er von 0-255 Zählen. Darüber hinaus hat der Timer zwei Vergleicher, die den aktuellen Zählwert ständig mit zwei programmierbaren Werten vergleichen. Die relevanten Register sind:

- TCNT0 enthält den Zähler
- OCR0A ist der erste Vergleichswert
- OCR0B ist der zweite Vergleichswert
- TCCR0A und TCCR0B konfiguriert den Timer
- TIMSK0 konfiguriert Interrupts
- TIFR0 zeigt , ob Ereignisse aufgetreten sind (Überlauf, Vergleich)

Die Möglichen Betriebsarten des Timers sind folgende:

6.12.1 Normaler Modus

Im normalen Modus zählt der Timer die Taktimpulse von 0-255. Danach fängt er wieder bei Null an, diesen Moment nennt man „Überlauf“. Beim Überlauf kann der Timer einen Interrupt auslösen. Die beiden Vergleicher können verwendet werden, um Interrupts beim Erreichen bestimmter Zahlenwerte auszulösen.

Dieser Modus ist gut geeignet, um das Hauptprogramm nach einer bestimmten Zeit oder Anzahl von externen Takt-Impulsen zu unterbrechen.

6.12.2 Timer nach Vergleich zurücksetzen (CTC)

Abweichend zum normalen Modus wird in diesem Modus der Zähler auf Null zurück gesetzt, wenn der Zählwert dem Vergleichswert OCR0A entspricht. OCR0B wird in dieser Betriebsart nicht verwendet.

Der Ausgang OC0A (=PB0) kann so eingerichtet werden, dass sein Pegel immer wechselt, wenn der Vergleichswert erreicht wurde. Somit entsteht am Ausgang OC0A ein Phasen-richtiges Rechtecksignal mit programmierbarer Frequenz (kein PWM Signal).

Dieser Modus ist z.B. geeignet, um mit einem Lautsprecher Töne zu erzeugen.

6.12.3 Schnelle Pulsweiten-Modulation (Fast PWM)

Der Zähler zählt fortwährend von 0 bis 255 und beginnt dann wieder bei 0. Beide Vergleicher erzeugen an ihren Ausgängen PWM Signale mit variabler Pulsbreite, die vom Vergleichswert abhängt. Dieser Modus wird üblicherweise verwendet, um die Leistung eines Motors oder die Helligkeit einer Lampe zu steuern.

6.12.4 Phasenrichtige Pulsweiten-Modulation (Phase Correct PWM)

Der Zähler zählt immer abwechselnd von 0 bis 255 und wieder zurück. Wie im Fast-PWM Modus erzeugen beide Vergleicher an ihren Ausgängen PWM Signale mit variabler Pulsbreite, die vom Vergleichswert abhängt. Allerdings ist die Frequenz des Signals nur halb so hoch, wie im Fast PWM Modus.

6.12.5 Programmierung

Das folgende Programm erzeugt ein (fast) PWM Signal, mit dem es die Helligkeit der Leuchtdiode verändert. Wir verwenden wieder den kleinen Mikrocomputer, bei dem die Leuchtdiode an PB0 angeschlossen ist.

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

int main(void)
{
    DDRB = 0b00000001;
    TCCR0A = (1<<COM0A1)+(1<<WGM01)+(1<<WGM00);
    TCCR0B = (1<<CS01);
```

```
    while (1)
    {

        for (uint8_t i=0; i<255; i++)
        {
            OCR0A=i;
            _delay_ms(10);
        }

        for (uint8_t i=255; i>0; i--)
        {
            OCR0A=i;
            _delay_ms(10);
        }
    }
}
```

Als erstes wird der Anschluss PB0 als Ausgang konfiguriert, denn dort ist die LED angeschlossen:

```
DDRB = 0b00000001;
```

Dann wird der Timer konfiguriert:

```
TCCR0A = (1<<COM0A1)+(1<<WGM01)+(1<<WGM00);
TCCR0B = (1<<CS01);
```

Aus dem Datenblatt habe ich entnommen, welche Bits in diesen beiden Registern gesetzt werden müssen:

- Die WGM Bits stellen die Betriebsart „Fast PWM“ ein.
- Die COM Bits stellen den Vergleicher ein. Der Ausgang des Vergleichers geht auf High, wenn der Zähler überläuft. Er geht wieder auf Low, wenn der Zähler den Vergleichswert erreicht. Ein großer Vergleichswert führt also dazu, dass der Ausgang lange auf High bleibt.
- Die CS Bits stellen die Taktfrequenz des Timers ein. Ich habe mich dazu entscheiden, die Taktfrequenz des Prozessors durch 8 zu teilen. So komme ich auf eine Taktfrequenz von 1.200.000/8=150.000 Hertz. Da der Zähler immer von 0-255 Zählt, teile ich diese Frequenz nochmal durch 256, um zu ermitteln, mit welcher Frequenz die LED „flackert“. Das ergibt ungefähr 586 Hertz.

Die Leuchtdiode wird also ungefähr 586 mal pro Sekunde ein und wieder aus geschaltet. Die Einschaltdauer beeinflusst, wie hell man das Licht empfindet.

Im Hauptprogramm wird die Einschaltdauer fortwährend verändert. Zuerst beginnt das Programm mit dem Vergleichswert 0, der alle 10 Millisekunden schrittweise bis auf 255 erhöht wird. Dann wird der Wert in der zweiten for() Schleife schrittweise wieder auf 0 verringert.

Dementsprechend wird die LED also zuerst immer heller bis zum Maximum, und dann wieder dunkler bis auf Null. Dieser Vorgang wird durch die umschließende while() Schleife endlos oft wiederholt.

6.13 EEPROM

AVR Mikrocontroller enthalten einen nicht flüchtigen Daten-Speicher, den EEPROM. Der Name bedeutet „Electrical Erasable Programmable Read Only Memory“, ein sehr verwirrender Name, da er einen Widerspruch enthält.

Die Wesentliche Eigenschaft des EEPROM ist schnell erklärt: Er funktioniert wie ein Flash Speicher, nur mit dem Unterschied, dass jedes Byte einzeln gelöscht werden kann. Flash-Speicher können nur Blockweise gelöscht werden.

Der EEPROM ist dazu gedacht, Konfigurationen zu speichern. Beispielsweise verwendet dein Fernseh-Empfänger sehr wahrscheinlich ein EEPROM, um die Einstellung der TV-Kanäle zu speichern.

Laut Datenblatt kann jede Speicherzelle des EEPROM mindstens 100.000 mal verändert werden, bevor mit ersten Fehlfunktionen zu rechnen ist. Der EEPROM des ATtiny 13 ist nur 64 Bytes klein.

Der EEPROM wird indirekt durch drei Register angesprochen:

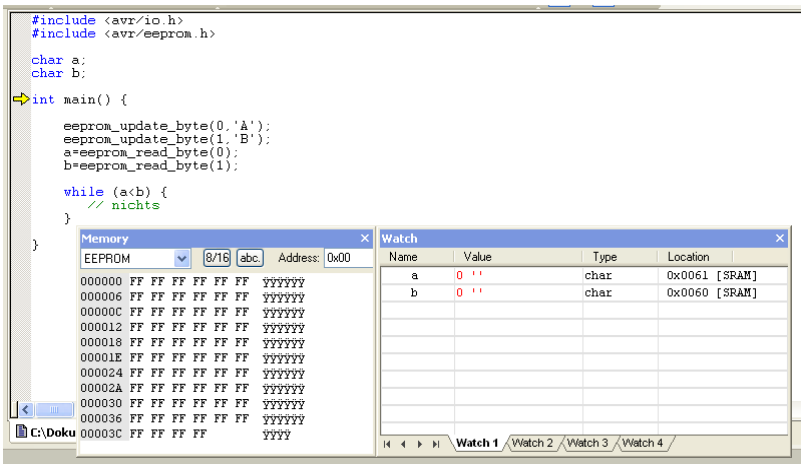
- In das Register **EEARL** schreibst du die Adresse (die Nummer) der Speicherzelle, die du lesen oder beschreiben willst.
- Über das Register **EEDR** kannst du anschließend den Wert aus der adressierten Speicherzelle lesen oder hinein schreiben.
- Darüber hinaus gibt es noch das Register **EECR**, mit dem der Zugriff auf den EEPROM Speicher gesteuert wird.

Sowohl das Lesen als auch das Schreiben in den EEPROM erfordert Prozeduren, die im Datenblatt detailliert beschrieben sind. Wir dürfen uns jedoch darüber freuen, dass diese Prozeduren bereits in der AVR C-Library enthalten sind.

6.13.1 Programmierung

Die Programmierung übst du am Besten mit dem Simulator, denn dann kannst du direkt auf dem Bildschirm sehen, ob der Zugriff wie erwartet funktioniert.

Tippe das folgende Programm ab:

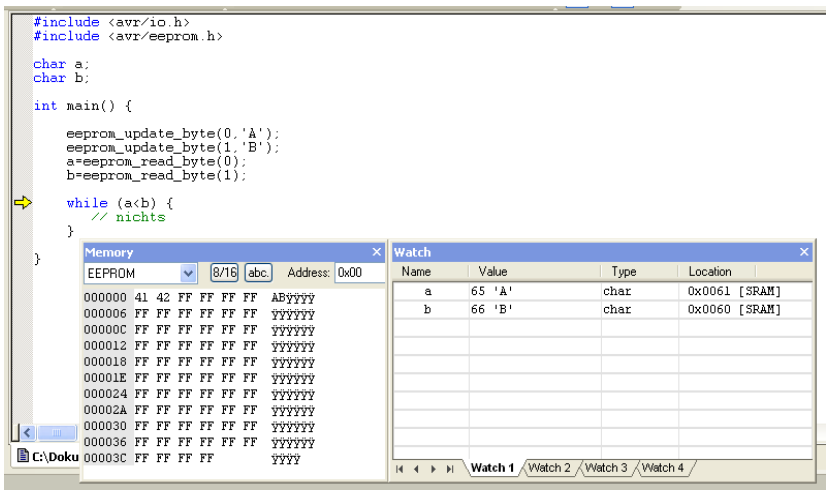


Dann drückst du F7 zum Compilieren, startest den Debugger durch den Menüpunkt Debug/Start Debugging. Öffne das Watch-Fenster mit der Tastenkombination Alt-1 und füge dort die Variablen a und b ein. Öffne das Memory Fenster mit der Tastenkombination Alt-4 und stelle es so ein, dass es den EEPROM anzeigt. Dein Bildschirm müsste dann ungefähr dem obigen Bildschirmfoto entsprechen.

Ausgangs-Situation ist ein jungfräulicher EEPROM Speicher, in dem alle Bits gesetzt sind. So sieht ein frisch gelöschter EEPROM Speicher aus.

Gehe jetzt mit der Taste F11 soweit durch das Programm, bis du die while() Schleife erreicht hast und beobachte dabei die Veränderungen in den beiden Fenstern „Memory“ und „Watch“. Dabei wird der gelbe Pfeil ab und zu in die allerletzte Zeile des Programm springen, nämlich immer dann, wenn der Simulator Programmcode aus der AVR C-Library ausführt.

Das Programm schreibt zunächst die beiden Buchstaben 'A' und 'B' in den EEPROM hinein, und dann liest es den Speicher wieder aus.



Die Funktion **eeprom_update_byte()** mach folgendes:

- Zuerst wartet sie, bis der EEPROM bereit ist. Nach jedem Schreibzugriff ist der EEPROM für ungefähr 3,4ms beschäftigt.
- Dann prüft sie anhand eines Lese-Zugriff, ob der neue Wert, der geschrieben werden soll, anders ist, als der alte Wert.
- Wenn der neue Wert anders ist, dann wird die Speicherzelle des EEPROM aktualisiert.

Im Gegensatz zur write-Funktion ist die Update-Funktion vorteilhafter, weil sie unnötige Schreibzugriffe überspringt, was angesichts der begrenzten Haltbarkeit des Speichers Sinn macht.

Die Library enthält entsprechende Funktionen auch für größere Datentypen, nämlich

- **eeprom_update_word()** für 16-Bit Variablen
- **eeprom_update_dword()** für 32-Bit Variablen
- **eeprom_update_float()** für float Variablen
- **eeprom_update_block()** für beliebig viele Bytes

Am Ende des Programms habe ich eine while() Schleife eingebaut damit die Variablen a und b benutzt werden und vom Compiler nicht weg optimiert werden.

Wenn du dir die Meldungen des Compilers anschaust, wirst duzwei Warnungen entdecken:

warning: passign argument 1 of … makes pointer from integer without cast

Die Warnung erscheint, weil die eeprom-Funktionen als ersten Parameter einen Zeiger erwarten, wir haben aber einen Integer-Wert (0 und 1) übergeben. Wir können die Warnung des Compilers so unterdrücken:

eeprom_update_byte(**(uint8_t*)** 0 ,'A');

Dadurch sagen wir dem Compiler: Die folgende 0 ist ein Zeiger auf einen 8-Bit Integer. Dann ist der Compiler zufrieden und meckert nicht mehr.

6.13.2 EEMEM Variablen

Eine andere elegantere Methode ist, EEMEM-Variablen zu deklarieren:

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

uint8_t erste_Zelle  EEMEM =0;
uint8_t zweite_Zelle EEMEM =0;

char a;
char b;
```

```
int main(void) {

    eeprom_update_byte(&erste_Zelle,'A');
    eeprom_update_byte(&zweite_Zelle,'B');
    a=eeprom_read_byte(&erste_Zelle);
    b=eeprom_read_byte(&zweite_Zelle);

    while (a<b)
    {
       // nichts
    }
}
```

Zuerst reservierst duzwei Bytes vom EEPROM Speicher und gibst diesen beiden Speicherzellen Namen und initiale Anfangswerte:

```
uint8_t erste_Zelle  EEMEM =0;
uint8_t zweite_Zelle EEMEM =0;
```

Auf diese Variablen darfst du nach wie vor niemals direkt zugreifen. Du musst die Zugriffe immer noch mit Hilfe der eeprom-Funktionen machen:

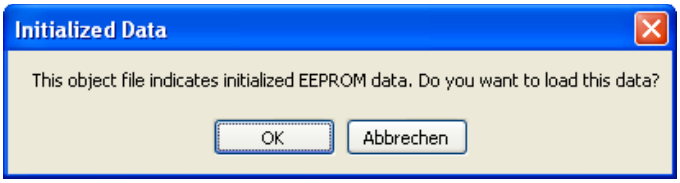
```
eeprom_update_byte(&erste_Zelle,'A');
eeprom_update_byte(&zweite_Zelle,'B');
```

Dabei übergibst du den Funktionen die Adresse der EEMEM Variablen, indem du das &-Zeichen davor schreibst.

Wenn du dieses Programm compilierst, erzeugt der Compiler zwei Ausgabe-Dateien:

- **test.hex** enthält das Programm, das du in den Flash-Speicher überträgst.
- **test.eep** enthält die initialen Werte, die du (ebenfalls mit dem ISP-Programmer) in den EEPROM-Speicher überträgst.

Starte den Debugger und staune über diese Meldung:



Der Debugger hat erkannt, das initiale Werte für den EEPROM vorliegen und möchte nun wissen, ob er diese auch in das simulierte EEPROM übertragen soll. Wenn du jetzt auf Abbrechen klickst, wird das simulierte EEPROM lauter 0xFF oder die Werte der vorherigen Simulation enthalten. Wenn du jedoch die Initialisierung erlaubst, dann werden die ersten beiden Speicherzellen des EEPROM mit 0x00 initialisiert, so wie es im Quelltext des Programmes geschrieben steht.

Probiere beide Möglichkeiten aus.

Übungsfrage: Warum ist es wohl erlaubt, die Datentypen **uint8_t** und **char** zu vermischen?

6.14 Energieverwaltung

AVR Mikrocontroller haben einige Funktionen, mit denen der Stromverbrauch gesenkt werden kann. Am offensichtlichsten ist natürlich die Reduktion der Taktfrequenz, das hatten wir schon. Aber es gibt noch mehr Möglichkeiten.

6.14.1 Sleep

Du kannst den Mikrocontroller einschlafen lassen. Dadurch wird der Prozessor-Kern angehalten. Er bleibt stehen, das Programm wird nicht weiter ausgeführt. Der Prozessor-Kern wacht erst wieder auf, wenn er einen Reset oder ein Interrupt-Signal empfängt.

Durch die folgenden beiden Programmzeilen bringst du den Prozessor zum einschlafen:

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

set_sleep_mode(<mode>);
sleep_mode();
```

Wenn der Prozessor wieder aufwacht, führt er zuerst die entsprechende Interrupt-Funktion aus und setzt dann das Hauptprogramm mit dem nächsten Befehl fort, der hinter „sleep_mode()“ steht.

Für den ATtiny13 sind folgende Sleep-Modi definiert:

- **0 = Idle**
  Der Prozessor-Kern wird angehalten, aber alle anderen Module (Timer, ADC, Watchdog) arbeiten weiterhin. Die Stromaufnahme reduziert sich etwa auf ein Viertel. Diese Module können einen Interrupt auslösen, der die CPU wieder aufwachen lässt.
- **1 = ADC Noise Reduction Mode**
  Dieser Modus ist dazu gedacht, die Genauigkeit des ADC zu verbessern. Es werden alle Teile des Mikrocontrollers angehalten, außer der ADC und der Watchdog. Wenn der ADC seine Messung beendet hat, kann er den Prozessor durch einen Interrupt wieder aufwecken.
- **2 = Power-Down**
  In diesem Modus wird gesamte Mikrocontroller angehalten. Die Stromaufnahme reduziert sich auf annähernd Null. Lediglich der Watchdog oder ein externes Signal kann den Power-Down Modus beenden.

6.14.2 Power Management

Einige Teilfunktionen des AVR kann man deaktivieren, um ein kleines bisschen Strom zu sparen. Dazu stehen beim ATTtiny13 diese Funktionen zur Verfügung:

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

power_adc_enable();
power_adc_disable();

power_timer0_enable();
power_timer0_disable();

power_all_enable();
power_all_disable();
```

Die enable() schaltet die entsprechende Funktion an, disable() schaltet die Funktion aus und spart somit Strom.

Schau in die Dokumentation der Library oder in die Datei power.h, um herauszufinden, welche Power-Management Funktionen für die größeren AVR Modelle zur Verfügung stehen.

6.14.3 Eingänge

Alle Eingänge verbrauchen zusätzlichen Strom, wenn sie nicht auf einem ordentlichen digitalem Pegel liegen (also weder High noch Low). Das gilt ganz auch für Eingänge, die gar nicht benutzt werden.

Um die Stromaufnahme zu reduzieren, bieten sich folgende Möglichkeiten an:

- Alle analogen Eingänge können normalerweise gleichzeitig auch digital abgefragt werden. Diesen digitalen Teil kannst du abschalten, indem du im Register DIDR0 entsprechende Bits setzt.
- Bei allen unbenutzten digitalen Eingängen kannst du den internen Pull-Up Widerstand einschalten, indem du im PORT-Register die entsprechenden Bits setzt.

6.15 Watchdog

Der Watchdog (zu deutsch: Wachhund) ist ein Hilfsmittel, um Programm-Abstürze zu erkennen. Er ist ein Timer, der in gewissen maximalen Zeitabständen auf null zurück gesetzt werden muss. Wenn das Programm dies versäumt (weil es abgestürzt ist), dann löst der Watchdog einen Reset aus.

Als Zeitfenster kann man wahlweise 16 ms, 32 ms, 64 ms, 125 ms, 250 ms, 500 ms, 1 s, 2 s, 4 s oder 8 s einstellen. Innerhalb dieser Zeit muss das Hauptprogramm sich regelmäßig melden, damit der Wachhund nicht auslöst.

6.15.1 Programmierung

Das folgende Beispielprogramm lässt die LED zuerst 5 Sekunden lang schnell flackern und dann 5 Sekunden lang langsamer flackern.

Der Watchdog wird auf 2 Sekunden eingestellt. Beim schnellen Flackern setzt das Programm den Watchdog regelmäßig zurück. Beim langsamen Flackern wird der Watchdog absichtlich nicht zurück gesetzt, um seine Wirkung zu demonstrieren.

Der Watchdog wird die zweite Flacker-Schleife daher nach zwei Sekunden durch einen Reset unterbrechen, was man an den Lichtzeichen der LED auch sehen kann: Sie blinkt zuerst 5 Sekunden schnell, dann 2 Sekunden langsam, dann kommt de Reset. Das Programm beginnt somit wieder von vorne.

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


int main(void)
{
    DDRB=1;
    wdt_enable(WDTO_2S); // 2 Sekunden

    // LED schnell flackern lassen
    for (int i=0; i<100; i++)
    {
        PORTB^=1;
        _delay_ms(50);
        wdt_reset();
    }

    // LED langsam flackern lassen
    for (int i=0; i<100; i++)
    {
        PORTB^=1;
        _delay_ms(100);
    }
}
```

Dabei gibt es noch eine Sache zu berücksichtigen: Wenn der Watchdog einen Reste auslöst, bleibt er weiterhin aktiviert. Allerdings wird das Zeitfenster durch den Reset auf 16ms reduziert, weil der Reset alle Register auf ihre Standardwerte zurück stellt.

Das Programm muss also bereits innerhalb der ersten 16ms nach dem Start den Watchdog auf die gewünschte Zeit einstellen sonst wird es nach dem ersten Reset durch den Watchdog immer wieder frühzeitig unterbrochen. Ein falsches Beispiel:

```
int main(void)
{
    DDRB=1;

    _delay_ms(500);
    wdt_enable(WDTO_2S); // 2 Sekunden

    // LED flackern lassen
    …
}
```

So wird das Programm nicht funktionieren. Nach dem ersten Reset durch den Watchdog wird das Programm neu gestartet und der Watchdog wartet nun maximal 16ms auf sein Signal. Doch wegen der delay() Funktion passiert das zu spät. Das Programm wird daher vom Watchdog erneut unterbrochen, während die delay() Funktion ausgeführt wird. Richtig wäre es so:

```
int main(void)
{
    DDRB=1;

    wdt_enable(WDTO_2S); // 2 Sekunden
    _delay_ms(500);

    // LED flackern lassen
    …
}
```

Bei dieser geänderten Reihenfolge wird gleich nach dem Programmstart (auch nach einem Watchdog-Reset) das Zeitfenster auf zwei Sekunden erhöht. Die anschließende Wartepause von 500 ms liegt somit im erlaubten Rahmen.

7 Nachwort

Wenn alles so gelaufen ist, wie ich mir das als Autor vorgestellt habe, dann hast du nun einiges über AVR Mikrocontroller gelernt. Du kannst nun für dich entscheiden, ob du eigene Schaltungen mit Mikrocontrollern entwickeln willst. Und du bist nun soweit vorbereitet, mit Fachliteratur und den Datenblättern des Herstellers klar zu kommen.

Wenn du weiter machen möchtest, dann denke dir praktische Anwendungsfälle aus. Suche im Internet nach geeigneten Bauteilen, studiere deren Datenblätter und experimentiere mit den Teilen herum. Ab und zu wird sicher ein Tolles Gerät dabei heraus kommen.

Sicher wirst du bald größere AVR Mikrocontroller verwenden wollen. Ich empfehle diese:

- 8 Pins: ATtiny 25, 45, 85
- 14 Pins: ATtiny 24, 44, 84
- 20 Pins: ATtiny 261, 461, 861
- 28 Pins: ATmega 48, 88, 168, 328
- 40 Pins: ATmega 164, 324, 644, 1284

Diese 5 Reihen gehören zu den klassischen AVR Mikrocontrollern der zweiten Generation, mit ISP Schnittstelle zum Programmieren und DebugWire zum Debuggen. Unter Hobbyelektronikern ist der **ATmega328P-PU** am weitesten verbreitet.

Im Band 2 dieser Buch-Reihe werde ich dir viele weitere Bauteile vorstellen, die ich bisher in meinen Entwicklungen verwendet habe. Und im Band 3 findest du spannende Experimente zu nachbauen.

Außerdem könnten dir die folgenden Internet Seiten gefallen:

- http://www.mikrocontroller.net
  Wissens-Sammlung und Diskussionsforum für Mikrocontroller
- http://www.rn-wissen.de
  Wissens-Sammlung für Roboter
- http://www.reichelt.de und http://www.tme.eu
  Versandhandel für Privatkunden

Viel Spaß bei deinem Hobby!

8 Anhänge

8.1 Musterlösungen zu Aufgaben

8.1.1 Lösungen zu: Grundlagen

1. Wovon hängt die Stromstärke ab?
   b) Von der Anzahl der bewegten Elektronen im Kabel
2. Warum überwindet ein Blitz die eigentlich nicht leitfähige Luft?
   a) Weil er eine hohe Spannung hat
3. Wenn jemand Strom „verbraucht“, verbraucht er dann die Elektronen?
   b) Nein, Elektronen sind immer da. Es kommt darauf an, ob sie sich bewegen.
4. Würde eine Leuchtdiode an einem einzelnen Akku funktionieren?
   b) Nein, ein einzelner Akku hat zu wenig Spannung
5. Warum verheizt man Strom in Widerständen?
   c) Weil sonst zu viel Strom fließen würde. Die Wärme ist ein notwendiges Übel.
6. Welchen Wert muss ein Widerstand haben, um eine LED an 12 Volt mit 10 mA zu betreiben?
   a) ungefähr 1000 Ohm
7. Wie viel Strom fließt durch einen 1000 Ohm Widerstand an 9 Volt?
   b) 9 mA
8. Wie viel Energie speichert ein 10 µF Kondensator?
   a) Es reicht gerade mal aus, um eine LED blitzen zu lassen
9. Wenn ein Akku 2000 mAh Kapazität hat, wie lange kann man damit ein Gerät betreiben, das 1000 mA benötigt?
   b) Zwei Stunden
10. Warum genügt ein Elektrolyt-Kondensator nicht, um die Versorgungsspannung eines Mikrocomputers zu stabilisieren?
    c) Weil er zu träge ist
11. Wie heißt bei Mikrochips die Gehäuseform, die wir (für den Einsatz auf Steckbrett und Lochraster-Platine) verwenden werden?
    a) PDIP und c) DIL

8.1.2 Lösungen zu: Der erste Mikrocomputer

1. Wie viele Pins hat ein ISP Stecker?
   b) sechs. Anmerkung: Früher waren es einmal 10 Pins.
2. Warum darfst du Mikrochips niemals falsch herum einstecken?
   a) Weil dann ein Kurzschluss entsteht, der nicht nur den Chip, sondern auch den ISP-Programmer und den USB-Port des Computers zerstören kann.
3. Was ist ein Mikrocontroller?
   c) Ein Mikroprozessor mit Speicher, der speziell für kleine Steuerungs-Aufgaben gedacht ist.
4. Wozu dient der rote Streifen auf Flachkabeln?
   b) Er kennzeichnet die erste Ader, die mit Pin1 des Steckers verbunden ist.
5. Ist es wichtig, dass gelötete Platinen „sauber“ aussehen?
   a) Ja, unsauber verarbeitete Platinen enthalten Fehler. Eventuell sogar Kurzschlüsse, die zu Schäden führen.

6. Was bedeutet es, wenn in einem Schaltplan eine Linie mit einem umgedrehten „T“ endet?
   a) Alle so gekennzeichneten Punkte sind miteinander verbunden und werden an den Minus-Pol der Stromversorgung angeschlossen.
7. Warum platziert man grundsätzlich neben jeden Mikrochip einen 100 nF Kondensator?
   b) Sie stabilisieren die Versorgungsspannung. Ohne diese Kondensatoren muss man mit sporadischen Fehlfunktionen rechnen.
8. Wie ist Pin 1 eines Mikrochips gekennzeichnet?
   c) Man zählt gegen den Uhrzeiger-Sinn. Oben ist irgendwie markiert.
9. Wenn man fertig gelötet hat, kann man dann den Strom einschalten?
   a) Nein, zuerst ist eine Sichtkontrolle angemessen. Danach sollte man zumindest die Leitungen der Spannungsversorgung durch messen, um unsichtbare Kurzschlüsse zu erkennen.
10. Was überträgt man in den Mikrocontroller?
    b) Den Byte-Code des Programms.
11. Worin unterscheiden sich Simulator und Debugger?
    a) Der Simulator läuft nur auf dem Computer. Er hat keine Verbindung zur externen Hardware. Der Debugger lässt das Programm auf der externen Hardware laufen und überwacht sie.
12. Mit welcher Taktfrequenz läuft dein erster Mikrocomputer?
    c) 1,2 Megahertz
13. Was darf man nicht vergessen, wenn man ein Programm in einen AVR Mikrocontroller überträgt?
    b) Man muss vorher den Programmspeicher löschen.

8.1.3 Lösungen zu: Programmieren in C

Generell gibt es in der Programmierung immer viele Wege, die zum Ziel führen. Sei daher nicht enttäuscht, wenn deine Lösung ganz anders aussieht, als meine. Letztendlich kommt es auf das Ergebnis an.

1. **Verändere das Programm so, dass es den Kammerton A (=440 Hz) abspielt.**

Berechne zuerst die Zeit für eine Schwingung bei 440 Hz: 1/440 = 0.0022727 Sekunden

Davon verteilst du jeweils die Hälfte auf die beiden delay() Funktionen, also jeweils 0.001136 Sekunden. Die Funktion erwartet den Wert allerdings in der Einheit Mikrosekunden also 1136.

Du brauchst die Werte aber gar nicht selbst berechnen, sondern kannst das dem Compiler überlassen. Dessen Optimierung wird die Formel automatisch durch einen einfachen Wert ersetzen. Die eleganteste Lösung sieht daher so aus:

```
int main(void)
{
    DDRB=24;
    while(1)
    {
        PORTB=8;
        _delay_us(1000000/440/2);  // oder 1136
        PORTB=16;
        _delay_us(1000000/440/2);  // oder 1136
    }
}
```

2. **Verändere das Programm so, dass es einen Intervall-Ton mit 2000 Hz abspielt, so wie ein Wecker.**

```
int main(void)
{
    DDRB=24;
    while(1)
    {
        for (int i=0; i<1000; i++)
        {
            PORTB=8;
            _delay_us(1000000/2000/2);  // oder 250
            PORTB=16;
            _delay_us(1000000/2000/2);  // oder 250
        }
        _delay_ms(500);
    }
}
```

3. **Finde heraus, bei welchen Frequenzen der Schallwandler am lautesten klingt und programmiere einen möglichst aufdringlichen Alarm-Ton.**

```
int main(void)
{
    DDRB=24;
    while(1)
    {
        for (int i=0; i<100; i++)
        {
            PORTB=8;
            _delay_us(1000000/3000/2);
            PORTB=16;
            _delay_us(1000000/3000/2);
        }
        _delay_ms(20);
        for (int i=0; i<100; i++)
        {
            PORTB=8;
            _delay_us(1000000/3500/2);
            PORTB=16;
            _delay_us(1000000/3500/2);
        }
        _delay_ms(20);
        for (int i=0; i<100; i++)
        {
            PORTB=8;
            _delay_us(1000000/4000/2);
            PORTB=16;
            _delay_us(1000000/4000/2);
        }
        _delay_ms(20);
        for (int i=0; i<100; i++)
        {
            PORTB=8;
            _delay_us(1000000/3500/2);
            PORTB=16;
            _delay_us(1000000/3500/2);
        }
        _delay_ms(20);
    }
}
```

4. Verändere das Programm so, dass es die Tonleiter C-D-E-F-G-A-H-C spielt.

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

void ton(char tonlage)
{
    for (int i=0; i<500; i++)
    {
        PORTB=8;

        switch (tonlage)
        {
            case 'C':
                _delay_us(1000000/523/2);
                break;
            case 'D':
                _delay_us(1000000/587/2);
                break;
            case 'E':
                _delay_us(1000000/659/2);
                break;
            case 'F':
                _delay_us(1000000/698/2);
                break;
            case 'G':
                _delay_us(1000000/784/2);
                break;
            case 'A':
                _delay_us(1000000/880/2);
                break;
            case 'H':
                _delay_us(1000000/988/2);
                break;
            case 'c':
                _delay_us(1000000/1047/2);
                break;
        }

        PORTB=16;

        switch (tonlage)
        {
            case 'C':
                _delay_us(1000000/523/2);
                break;
            case 'D':
                _delay_us(1000000/587/2);
                break;
            case 'E':
                _delay_us(1000000/659/2);
                break;
            case 'F':
                _delay_us(1000000/698/2);
                break;
            case 'G':
                _delay_us(1000000/784/2);
                break;
            case 'A':
                _delay_us(1000000/880/2);
                break;
```

```
            case 'H':
                _delay_us(1000000/988/2);
                break;
            case 'c':
                _delay_us(1000000/1047/2);
                break;
        }
    }

    _delay_ms(100);
}

int main(void)
{
    DDRB=24;

    while(1)
    {
        ton('C');
        ton('D');
        ton('E');
        ton('F');
        ton('G');
        ton('A');
        ton('H');
        ton('c');
    }
}
```

Für das hohe C habe ich einen kleinen Buchstaben verwendet, um ihn von dem tiefen C unterscheiden zu können.

8.1.4 Lösungen zu: Eingänge Abfragen

1. **Wozu war nochmal der Kondensator gut?**
   Er stabilisiert die Spannungsversorgung. Durch Schaltvorgänge würden sonst Störimpulse in der Stromversorgung entstehen, die Fehlfunktionen hervor ruft.
2. **Gestalte den C-Quelltext durch Verwendung von #define übersichtlicher.**

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

#define check_Taster PINB & 0b00000010
#define led_An       PORTB |= 0b00000001
#define led_Aus      PORTB &= 0b11111110

int main(void)
{
    DDRB  = 0b00000001;
    PORTB = 0b00000010;
    while (1)
    {
        if (check_Taster)
        {
            led_An;
        }
        else {
            led_Aus;
        }
    }
}
```

3. **Erkläre in eigenen Worten, warum man Bitweise Verknüpfungen benötigt, wenn man den Taster abfragt und die LED ansteuert.**
   Für diese Aktionen müssen einzelne Bits von Port B abgefragt bzw. verändert werden. Doch die Programmiersprache C kennt keine Befehle für bitweise Operationen. Durch bitweise Verknüpfungen filtern wir aus einem ganzen Byte das gewünschte Bit heraus. Der Compiler erzeugt daraus glücklicherweise optimierten Byte-Code, indem er letztendlich doch auf Bitweise Befehle der Assembler Sprache zurück greift.

8.2 Verfügbare AVR Mikrocontroller

Dieses Kapitel listet gängige AVR Mikrocontroller auf, die mit PDIP Gehäuse verkauft werden (Stand 2016)

Attiny 13A

PCINT5, ADC0, dW, /RESET, PB5	1	8	VCC
PCINT3, CLKI, ADC3, PB3	2	7	PB2, SCK, T0, ADC1, PCINT2
PCINT4, ADC2, PB4	3	6	PB1, MISO, INT0, AIN1, PCINT1, OC0B
GND	4	5	PB0, MOSI, AIN0, PCINT0, OC0A

Attiny 25, 45, 85

PCINT5, ADC0, dW, /RESET, PB5	1	8	VCC
/OC1B, PCINT3, XTAL1, CLKI, ADC3, PB3	2	7	PB2, SCK, USCK, SCL, ADC1, T0, INT0, PCINT2
OC01B, XTAL2, CLKO, PCINT4, ADC2, PB4	3	6	PB1, MISO, DO, AIN1, OC0B, OC1A, PCINT1
GND	4	5	PB0, MOSI, DI, SDA, AIN0, OC0A, OC1A, PCINT0, AREF

Attiny 24, 44, 84

VCC	1	14	GND
PCINT8, XTAL1, PB0	2	13	PA0, ADC0, AREF, PCINT0
PCINT9, XTAL2, PB1	3	12	PA1, ADC1, AIN0, PCINT1
PCINT11, /RESET, dW, PB3	4	11	PA2, ADC2, AIN1, PCINT2
PCNT10, INT0, OC0A, CKOUT PB2	5	10	PA3, ADC3, T0, PCINT3
PCINT7, ICP, OC0B, ADC7, PA7	6	9	PA4, ADC4, USCK, SCL, T1, PCINT4
PCINT6, OC1A, SDA, MOSI, ADC6, PA6	7	8	PA5, ADC5, DO, MISO, OC1B, PCINT5

Attiny 26, 261, 461, 861 (geklammerte Pins nicht beim Attiny 26)

(PCINT8) MOSI, DI, SDA, /OC1A, PB0	1	20	PA0, ADC0 (PCINT0, DI, SDA)
(PCINT9) MISO, D0, OC1A, PB1	2	19	PA1, ADC1 (PCINT1, DO)
(PCINT10) SCK, SCL, /OC1B, PB2	3	18	PA2, ADC2 (PCINT2, INT1, USCK, SC)
(PCINT11) OC1B, PB3	4	17	PA3, AREF (PCINT3)
VCC	5	16	GND
GND	6	15	VCC
(PCINT12, /OC1D, CLKI) ADC7, XTAL1, PB4	7	14	PA4, ADC3 (PCINT4, ICP0)
(PCINT13, OC1D, CLKO) ADC8, XTAL2, PB5	8	13	PA5, ADC4 (PCINT5, AIN2)
(PCINT14) ADC9, INT0, T0, PB6	9	12	PA6, ADC5, AIN0 (PCINT6)
(PCINT15) ADC10, /RESET, PB7	10	11	PA7, ADC6, AIN1 (PCINT7)

Attiny 2313A, 4313

/RESETt, dW, PA2	1	20	VCC
RxD, PD0	2	19	PB7, USCK, SCL, PCINT7
TxD, PD1	3	18	PB6, MISO, D0, PCINT6
XTAL2, PA1	4	17	PB5, MOSI, DI, SDA, PCINT5
XTAL1, PA0	5	16	PB4, OC1B, PCINT4
CKOUT, XCK, INT0, PD2	6	15	PB3, OC1A, PCINT3
INT1, PD3	7	14	PB2, OC0A, PCINT2
T0, PD4	8	13	PB1, AIN1, PCINT1
OC0B, T1, PD5	9	12	PB0, AIN0, PCINT0
GND	10	11	PD6, ICP

Atmega 8, 48, 88, 168, 328 (geklammerte Pins nicht beim Atmega 8)

(PCINT14) /RESET, PC6	1	28	PC5, ADC5, SCL (PCINT13)
(PCINT16) RxD, PD0	2	27	PC4, ADC4, SDA (PCINT12)
(PCINT17) TxD, PD1	3	26	PC3, ADC3 (PCINT11)
(PCINT18) INT0, PD2	4	25	PC2, ADC2 (PCINT10)
(PCINT19, OC2B) INT1, PD3	5	24	PC1, ADC1 (PCINT9)
(PCINT20) XCK, T0, PD4	6	23	PC0, ADC0 (PCINT8)
VCC	7	22	GND
GND	8	21	AREF
(PCINT16) XTAL1, TOSC1, PB6	9	20	VCC
(PCINT17) XTAL2, TOSC2, PB7	10	19	PB5, SCK (PCINT5)
(PCINT21, OC0B) T1, PD5	11	18	PB4, MISO (PCINT4)
(PCINT22, OC0A) AIN0, PD6	12	17	PB3, MOSI, OC2 (PCINT3, OC2A)
(PCINT23) AIN1, PD7	13	16	PB2, /SS, OC1B (PCINT2)
(PCINT0, CLKO) ICP1, PB0	14	15	PB1, OC1A (PCINT1)

Atmega 8515

OC0, T0, PB0	1	40	VCC
T1, PB1	2	39	PA0, AD0
AIN0, PB2	3	38	PA1, AD1
AIN1, PB3	4	37	PA2, AD2
/SS, PB4	5	36	PA3, AD3
MOSI, PB5	6	35	PA4, AD4
MISO, PB6	7	34	PA5, AD5
SCK, PB7	8	33	PA6, AD6
/RESET	9	32	PA7, AD7
RxD, PD0	10	31	PE0, ICP, INT2
TxD, PD1	11	30	PE1, ALE
INT0, PD2	12	29	PE2, OC1B
INT1, PD3	13	28	PC7, A15
XCK, PD4	14	27	PC6, A14
OC1A, PD5	15	26	PC5, A13
/WR, PD6	16	25	PC4, A12
/RD, PD7	17	24	PC3, A11
XTAL2	18	23	PC2, A10
XTAL1	19	22	PC1, A9
GND	20	21	PC0, A8

Atmega 16, 32, 8535 (geklammerte Pins nicht beim Atmega 8535)

XCK, T0, PB0	1	40	PA0, ADC0
T1, PB1	2	39	PA1, ADC1
INT2, AIN0, PB2	3	38	PA2, ADC2
OC0, AIN1, PB3	4	37	PA3, ADC3
/SS, PB4	5	36	PA4, ADC4
MOSI, PB5	6	35	PA5, ADC5
MISO, PB6	7	34	PA6, ADC6
SCK, PB7	8	33	PA7, ADC7
/RESET	9	32	AREF
VCC	10	31	GND
GND	11	30	AVCC
XTAL2	12	29	PC7, TOSC2
XTAL1	13	28	PC6, TOSC1
RxD, PD0	14	27	PC5, (TDI)
TxD, PD1	15	26	PC4, (TDO)
INT0, PD2	16	25	PC3, (TMS)
INT1, PD3	17	24	PC2, (TCK)
OC1B, PD4	18	23	PC1, SDA
OC1A, PD5	19	22	PC0, SCL
ICP1, PD6	20	21	PD7, OC2

Atmega 164, 324, 644, 1284

PCINT8, XCK, T0, PB0	1	40	PA0, ADC0, PCINT0
PCINT9, CLK0, T1, PB1	2	39	PA1, ADC1, PCINT1
PCINT10, INT2, AIN0, PB2	3	38	PA2, ADC2, PCINT2
PCINT11, OC0A, AIN1, PB3	4	37	PA3, ADC3, PCINT3
PCINT12, OC0B, /SS, PB4	5	36	PA4, ADC4, PCINT4
PCINT13, MOSI, PB5	6	35	PA5, ADC5, PCINT5
PCINT14, MISO, PB6	7	34	PA6, ADC6, PCINT6
PCINT15, SCK, PB7	8	33	PA7, ADC7, PCINT7
/RESET	9	32	AREF
VCC	10	31	GND
GND	11	30	AVCC
XTAL2	12	29	PC7, TOSC2, PCINT23
XTAL1	13	28	PC6, TOSC1, PCINT22
PCINT24, RxD0, PD0	14	27	PC5, TDI, PCINT21
PCINT25, TxD0, PD1	15	26	PC4, TDO, PCINT20
PCINT26 INT0, PD2	16	25	PC3, TMS, PCINT19
PCINT27, INT1, PD3	17	24	PC2, TCK, PCINT18
PCINT28, OC1B, PD4	18	23	PC1, SDA, PCINT17
PCINT29, OC1A, PD5	19	22	PC0, SCL, PCINT16
PCINT30, OC2B, ICP, PD6	20	21	PD7, OC2A, PCINT31

Modell	Pins	Flash (Kbytes)	EEPROM (Bytes)	RAM (Bytes)	Standard Version	L Version	V Version	16-bit Timer	8-bit Timer	PWM	RTC	Serielle Schnittstellen	Analoge Eingänge	Bemerkungen
Attiny13A	8	1	64	64	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	--	1	2	--	--	4	Kein Quarz
Attiny25	8	2	128	128	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	--	2	4	--	USI	4	
Attiny45	8	4	256	256	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	--	2	4	--	USI	4	
Attiny85	8	8	512	512	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	--	2	4	--	USI	4	
Attiny24	14	2	128	128	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	1	4	--	USI	8	
Attiny44	14	4	256	256	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	1	4	--	USI	8	
Attiny84	14	8	512	512	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	1	4	--	USI	8	
Attiny26	20	2	128	128	16Mhz @ 4.5-5.5V	8Mhz @ 2.7-5.5V	--	--	2	2	--	USI	11	Programm kann den Flash-Speicher nicht beschreiben.
Attiny261	20	2	128	128	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	3	--	USI	11	
Attiny461	20	4	256	256	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	3	--	USI	11	
Attiny861	20	8	512	512	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	3	--	USI	11	
Attiny2313A	20	2	128	128	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	1	4	--	USI, USART	--	
Attiny4313	20	4	256	256	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	1	4	--	USI,USART	--	
Atmega8	28	8	512	1024	16Mhz @ 4.5-5.5V	8Mhz @ 2.7-5.5V	--	1	2	3	Ja	SPI, USART, TWI	8	
Atmega48	28	4	256	256	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	6	Ja	SPI, USART, TWI	8	
Atmega88	28	8	256	512	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	6	Ja	SPI, USART, TWI	8	
Atmega168	28	16	512	1024	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	6	Ja	SPI, USART, TWI	8	
Atmega328	28	32	1024	2048	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	6	Ja	SPI, USART, TWI		
Atmega8515	40	8	512	512	16Mhz @ 4.5-5.5V	8Mhz @ 2.7-5.5V	--	1	1	3	--	SPI, USART	--	Unterstützt externes RAM
Atmega8535	40	8	512	512	16Mhz @ 4.5-5.5V	8Mhz @ 2.7-5.5V	--	1	2	4	--	SPI, USART, TWI	8	
Atmega16	40	16	512	1024	16Mhz @ 4.5-5.5V	8Mhz @ 2.7-5.5V	--	1	2	4	Ja	SPI, USART, TWI	8	
Atmega32	40	32	1024	2048	16Mhz @ 4.5-5.5V	8Mhz @ 2.7-5.5V	--	1	2	4	Ja	SPI, USART, TWI	8	
Atmega164	40	16	512	1024	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	6	Ja	SPI, USART, TWI	8	
Atmega324	40	32	1024	2048	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	6	Ja	SPI, USART, TWI	8	
Atmega644	40	64	2048	4096	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	6	Ja	SPI, USART, TWI	8	
Atmega1284	40	128	4096	8192	10Mhz @ 2.7-5.5V 20Mhz @ 4.5-5.5V	--	4Mhz @ 1.8-5.5V 10Mhz @ 2.7-5.5V	1	2	6	Ja	SPI, USART, TWI	8	

8.3 Material-Liste

Das sind alle Teile, die in diesem Buch Band 1 verwendet werden.

1 Digital-Multimeter, das einfachste Modell genügt
1 ISP Programmieradapter mit 6poligem Stecker.
1 Steckbrett (mit mindestens 400 Kontakten)
1 Batteriehalter mit 3 Zellen Größe AA oder AAA
1 m Mehradriges Datenkabel zum Zerlegen. Je mehr bunte Adern drin sind, umso besser.
1 Rolle verzinnter oder versilberter Kupferdraht ca. 0,5 mm Durchmesser.
1 Leuchtdiode rot mit 3mm oder 5mm Durchmesser.
1 Widerstand 27 Ohm ¼ Watt
2 Widerstände 220 Ohm ¼ Watt
1 Widerstand 820 Ohm ¼ Watt
2 Dioden 1N4001
1 Elektrolyt Kondensator 10 µF für mindestens 10 V
2 Glühlämpchen für 4-5 Volt mit maximal 1 Watt.
1 Lochraster-Platine mit Lötpunkten (keine Streifen)
1 IC Sockel DIL 8
1 Stiftleiste 2x3 Pins (oder länger, kann man abschneiden)
1 Kondensator 100 nF
2 Kurzhubtaster 6x6 mm
1 Mikrocontroller ATtiny13-PU oder ATtiny13A-PU (alternativ ATtiny25, 45 oder 85)
1 Piezo Schallwandler/Signalgeber (ohne integrierten Tongenerator)

Werkzeug: *(Kann man oft preisgünstig als Set kaufen)*

1 Kleiner Seitenschneider
1 Kleine Spitzzange
1 Lötkolben 15-30 Watt, oder Lötstation.
1 Entlötpumpe
50 g Lötzinn mit 1mm Durchmesser.