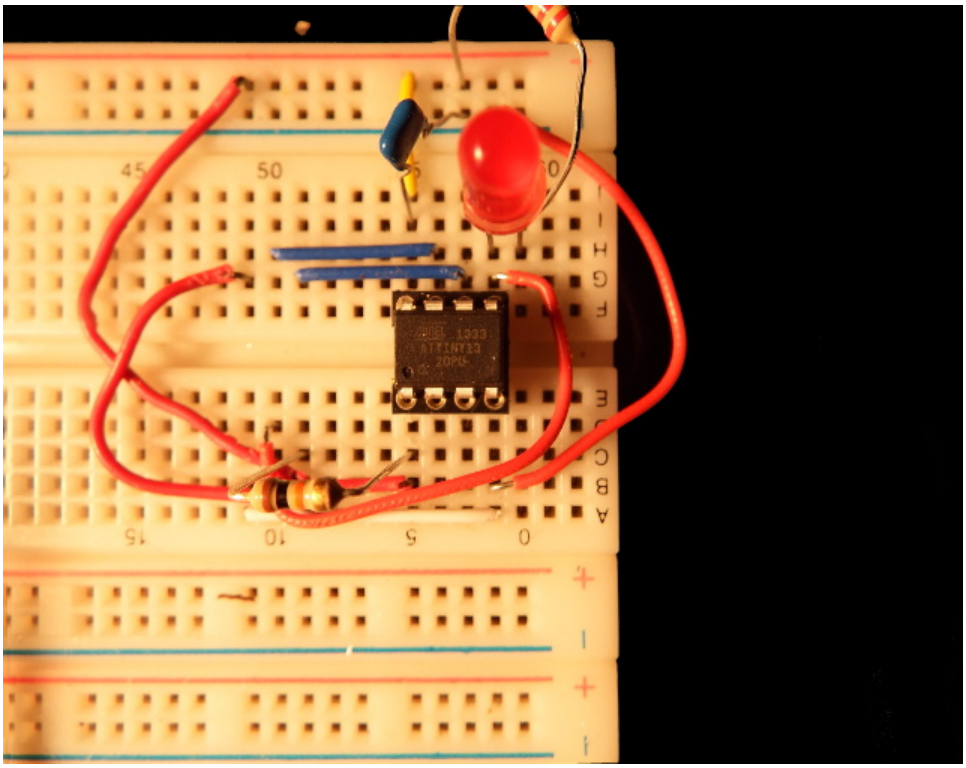
Mikroprozessor für Beginner

anhand von praktischen Beispielen mit ATMEL-Prozessoren in Assembler



von
Gerhard Schmidt, Kastanienallee 20, 64289 Darmstadt
Version 2, Januar 2018
Aktualisiert und mit avr_sim-Simulationen versehen

http://www.gsc-elektronic.net

Vorbemerkungen

Zum Konzept

Der folgende Text geht von der Beobachtung aus, daß Lernen schneller und besser geht, wenn man vor ein ganzheitliches Problem gestellt ist und dieses zur Lösung die Aneignung neuen Wissens erfordert. In diesem Fall sind Wissen über Elektronik, Mikroprozessoren und Programmieren in Assembler eng verwoben. Die drei Bereiche sind eng aufeinander bezogen und bilden ein Ganzes.

Das Programmieren wird hier nicht auf "Vorrat" erlernt, sondern in engem Zusammenspiel mit der Elektronik, der Hardware der Prozessoren und anhand von geeigneten praktischen Experimenten, bei denen der praktische Erfolg sofort kontrollierbar ist. Wer möchte, kann die Hardware zusammenbauen, den mitgelieferten Quelltext (Anhang 6: Links zu den asm-Quellcode-Dateien) assemblieren und in den Chip brennen. Wer nur das macht, wird allerdings nichts über den Prozessor und über Assembler lernen. Und wird auch nicht in der Lage sein, im Quelltext herumzuwühlen und was Eigenes daraus zu entwickeln.

Wer das systematische Erlernen bevorzugt, kann sich unter dieser Adresse ein Tutorial reinziehen oder dort auch bei Bedarf zusätzlich nachschlagen. Wer diesen Text als Nachschlagewerk verwenden möchte, wird Anhang 7: Themenverzeichnis als hilfreich empfinden, weil er schnell zur gesuchten Stelle im Text führt, in dem das betreffende Thema behandelt wird.

Lernkonzept und Assembler

Das Konzept, Hardware und Programmieren gemeinsam zu lernen, geht vernünftig nur in Assemblersprache. Wer wissen will, wie der Chip wirklich arbeitet und was geht und warum anderes nicht geht, sollte sich mit Assembler befassen. Es ist nun mal die einzige Sprache, die der Prozessor versteht und selber spricht (nach Übersetzung in Nullen und Einsen, natürlich, genannt Assemblieren). Wer hingegen mit Hochsprachen anfängt, befasst sich überwiegend mit den Eigenheiten dieser Sprache ("Wie geht hexadezimal in C?") und mit den von anderen in Assembler geschriebenen Bibliotheken, lernt aber leider rein gar nix über den Prozessor selbst. Schon die einfachsten Hardware-Aufgaben bleiben dann unlösbar und es bleibt nur die Suche im Internet, ob es vielleicht schon ein anderer gelöst hat. Falsch herum gelernt: gar nicht erst mit den Basics von Hard- und Software des Prozessors angefangen und, selbst daran natürlich völlig unschuldig, auf ewig Gefangener der eigenen Unkenntnis geblieben.

Lektionen – und was hier im Einzelnen an Beispielen gelernt werden soll Die 14 Beispiele wurden so ausgewählt, dass möglichst viel von der intern verfügbaren Hardware des Prozessors verwendet wird, so ihr Nutzen erkennbar wird und ihre Software-Ansteuerung und -kontrolle zielgerichtet demonstriert wird, einfache externe Beschaltung zum Einsatz kommt, damit sich der Verhau an Elektronik auf dem Breadboard noch in überschaubaren Grenzen hält, viele grundlegende Konzepte für die Assemblerprogrammierung erkennbar werden (lineare Programmierung von exakten Zeitschleifen, Interruptprogrammierung, Fließdiagramme zur Ablaufplanung, etc.), alle Techniken zur mathematischen Verarbeitung von Daten einmal vorkommen (Addition, Subtraktion, Multiplikation, Division, Tabellen) und das möglichst in ihrer 8-, 16-, 24-, 32- und 40-Bit-Manier (bis es langweilig wird, weil es immer dasselbe ist), viele der ca. 130 verschiedenen Prozessor-Instruktionen (Assembler-Befehle) zum praktischen Einsatz kommen und ihre optimale Nutzung erkannt werden kann (Quote = 35%, da ist noch viel Luft nach oben).

Wer also die eine oder andere ausgewählte Anwendung so gar nicht praktisch gebrauchen kann, kann vielleicht trotzdem aus dem Beispiel lernen, wie man ein Problem löst und kann die demonstrierten Prinzipien auf die eigene Anwendung übertragen und an die jeweils eigenen Bedürfnisse anpassen.

Nr.	Beschreibung, Link	Interne Hardware	Externe Hardware	Programmierkünste
1	Programmier-Interface Programmier-Interface: PC spricht mit dem Microcontroller	ISP-Interface der AVRs, Fuses	ISP6-Schnittstelle zum Programmiergerät	(Noch keine)
2	Eine LED einschalten Eine LED am Controller ein- und ausschalten	I/O-Port, Portausgang und -richtung	LEDs, Ansteuerung	Quellcode schreiben, assemblieren, Listings und Hexdateien, Programmausführung, unendliche Schleifen
3	Eine LED blinken lassen Schnelles Blinken mit der LED, Sekundenblinker	I/O-Port	LED-Ansteuerung	Verschachtelte Schleifen, Zählschleifen, Ausführungszeiten von Instruktionen, Nullflagge, zeitgenaue Zählschleifen
4	Eine LED blinkt mit dem Timer Der Timer steuert eine LED im Takt und schaltet ganz alleine an und aus	8-Bit-Timer, Vergleicher, OC-Pin-Steuerung	Impulsansteuerung von Ausgängen	Timer-Konfiguration, Starten und Anhalten des Timers, Timer-Modi Normal und CTC, Schlafmodus
5	LED mit Pulsweitenmodulation Helligkeitssteuerung der LED über Pulsweiten an OC-Ausgabepins	8-Bit-Timer, Pulsweitenmodus	Pulsweitensignale	Mathematische Ausdrücke in Assembler
6	LED mit Timer-Interrupts Eine LED mit Interrupts des Timers ansteuern	Interruptkontrolle, Timer-Interrupts	Steuerung von Ausgängen mit Interrupts	Interruptkonzept, Vektortabelle, Interruptprogrammierung, Timer-Interrupts, Interrupt-Service-Routinen
7	Eine LED mit Timer und Taster Taste startet einen Ablauf	INT0- und Timer-Int-Steuerung	I/O-Ansteuerung	Fließdiagramme für Ablaufplanungen, INT0- und Timer-CTC-Interrupts verheiraten
8	Eine LED mit Helligkeitssteuerung Timer, Tasterbedienung, Potentiometer-Helligkeitssteuerung und Interrupts	Analog-Digital-Wandler	Externe Spannungen messen	Noch mehr Fließdiagramme, Tastenprellunterdrückung, OC0A- und OC0B-Programmierung
9	Töne mit dem Timer Der Taster macht Töne und das Potentiometer macht die Tonhöhe	ADC, Timer, INT0	Lautsprecheranschluss	Tabellen, Notentabelle, Multiplikation 8-mal-8-Bit
10	Eine LCD-Anzeige am Tiny24 Initiierung von LCDs, Ausgabe von Zeichen auf einer LCD, exakte Verzögerungsschleifen, Eigendesign von LCD-Zeichen	I/O-Port-Richtungssteuerung	LCD-Anschluss und -steuerung	Exakte Verzögerungsschleifen für LCD, Initiierung von LCDs
11	Einschaltzähler mit dem EEPROM Die Anzahl Startvorgänge des Controllers im EEPROM zählen	EEPROM	RESET-Taster	Umwandlung von 8- und 16-Bit-Binärzahlen in ASCII-Zeichenfolgen und Anzeige auf der LCD
12	Infrarot-Sender -Empfänger Analyse von IR-Signalfolgen, Senden von IR-Signalfolgen, selbstlernender Fernsteuerempfänger	Zeitmessung mit Timer und INT0	Anschluss von IR-Empfangsmodulen und IR-Sendedioden	Anzeige von Binärzahlen im Hexadezimalformat, Messen und Verarbeiten von Signaldauern, .if-Direktive zur Versionsanpassung in Assembler
13	Ein Frequenz- und Induktivitätsmessgerät Frequenzmessung mit externem Interrupt und dem Analogvergleicher, Induktivität mit LC-Oszillator bestimmen	Analogvergleicher, PCINT	Analogvergleicher und LC-Oszillator anschließen	24/32/40-Bit-Binär-zu-Dezimal-Umwandlung, 24-Bit-Multiplikation, 40-durch-40-Bit-Division
14	Spannung, Strom, Temperatur messen Spannungen, Ströme und Temperaturen messen und anzeigen	ADC-Betriebsmodi, Differenzverstärker, eingebaute Thermometer	Spannungsteiler, Stromshunts	Intensive Multiplikationen und Pseudo-Fließkommazahlen-Operationen zur Anzeigeumwandlung
-	Fazit	-	-	-

Hardware

Benötigt werden für die Experimente neben dieser Beschreibung einige Bauelemente, die im Elektronikhandel beschafft werden können. Diese sind Lektion für Lektion in einer Bauteilebeschreibung vorgestellt. Im Anhang 5: Bauteileliste sind alle verwendeten Bauteile zusammengestellt.

Für die Experimente ab Lektion 11 (ATtiny24 mit 4*20-LCD) wurde eine kleine Experimentalplatine entwickelt (siehe tn24_lcd), so dass das Verdrahten des ATtiny24 und der LCD auf dem Breadboard entfällt. Stattdessen ist die Experimentierplatine mit einem sechsadrigen Flachbandkabel mit dem Breadboard zu verbinden, der Aufbau der Peripherie erfolgt dann wie beschrieben auf dem Breadboard. Alle hier veröffentlichte Software läuft ohne Änderungen auch auf der Experimentalplatine (bitte dazu die LCD-Include-Routine Lcd4Busy.inc verwenden und nicht die mit dem Experimentierboard veröffentlichte Version tn24_lcd_busy.inc!).

Copyright-Hinweis

Hilfsmittel bei der Erstellung dieses Dokumentes

Dieses Dokument entspricht recht genau dem Text und den Inhalten auf der Webseite

http://www.gsc-elektronic.net/mikroelektronik/index.html

Abweichungen im Layout erfolgten absichtlich. Insbesondere wurden die mehr als 300 verwendeten Abbildungen und Grafiken verkleinert und gelayoutet (der zu 99% gut funktionierende Umgang mit hunderten Bilddateien im OpenOffice ist gewöhnungsbedürftig, aber insgesamt gut gelöst).

Das Dokument wurde mit OpenOffice Writer durch Import der Webdateien erstellt (vielen Dank an die OpenOffice-Entwickler für den sehr gut funktionierenden Import von HTML- und sonstigen Dateitypen - vorzüglich). Die meisten grafischen Darstellungen wie Schaltbilder, erläuternde Grafiken und Fließdiagramme wurden mit OpenOffice Draw gezeichnet. Bildschirmabzüge der Simulationen mit avr_sim wurden mit PaintNet erstellt und bearbeitet. Die Bilder von Bauteilen und dem Aufbau wurden mit einer Fuji SL1000 mit Makroeinstellung aufgenommen und mit Gimp nachbearbeitet.

Update 2018

Im Dezember 2017/Januar 2018 wurde eine grundlegende Überarbeitung vorgenommen. Dabei wurde Folgendes gemacht:

- Alle mir sinnvoll erscheinenden Quellcodes wurden in den Simulator avr_sim gefüttert und einzelne Vorgänge (Port- und Timereinstellungen, Ausführungszeiten, usw.) mit Bildern anschaulich dargestellt. Ich hoffe, das hilft dem Einen oder Anderen, der den Quellcode alleine nicht verstanden hat sich ein anschauliches Bild von den Abläufen in der Prozessorhardware zu machen. Diese Simulationen sind jeweils hinter dem Quellcode zu finden und reich bebildert.
- In den .asm-Quellcodes (und auch in den Auflistungen im HTML-Code und im PDF-Gesamtdokument) wurden zusätzliche Erläuterungen eingearbeitet. Am Anfang wird kurz der Programmablauf erläutert, bei allen Interrupt-Service-Routinen wurden Hinweise zu den Bedingungen, die zu dem Interrupt führen, und zum Ablauf der Routine eingearbeitet. Manipulierte Flaggen und ihr Sinn werden ebenfalls erwähnt.
- Zu allen wichtigen Routinen im Quellcode, die ausserhalb von Interrupts ausgeführt werden, werden Hinweise auf auslösende Flaggen und eine Kurzbeschreibung des Ablaufs der Routine hinzugefügt.
- Einige kleinere Fehler und Unebenheiten wurden repariert.

Feedback

Wer Fehler melden möchte, Anregungen oder Hinweise oder einfach nur Lob und Tadel loswerden will kann diese über info@ und die auf der Titelseite genannte Webadresse gerne an mich senden. Ich freue mich über jede qualifizierte Äußerung.

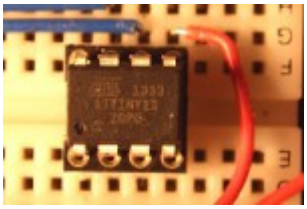
Besonderen Dank möchte ich Carsten Henckell sagen. Er hat bislang (Januar 2018) als Einziger eine detaillierte Stellungnahme zu dem Dokument abgegeben. Die in seiner Stellungnahme enthaltenen sehr wertvollen Hinweise habe ich in diesem Dokument berücksichtigt. Vielen Dank.

Änderungen am Text

Datum	Kapitel	Seiten	Änderung
03.03.20	8	73	Hinweis auf korrekte Reihenfolge des Lesens des AD-Wandlers
	10	130	Hinweis auf universelle LCD-Include-Routine
27.02.19	Instrukti-onsliste	231 - 235	Parametereinschränkungen bei MOVW, BRBS, IN, OUT, BSET, BCLR, BLD, BST hinzugefügt
16.07.18	Anhang 3	237	Direktive .IFNDEF hinzugefügt
07.06.18	12	165	Fehlerhafter Text in Kommentarzeile *„; cbi TCCR0A,COM0A1 ; + 2 = 5 Takte*“
24.05.18	Anhang 2	234	Einschränkungen bei der RCALL-Instruktion eingefügt.
	10	121	Im Quellcode von 10_Lcd_Display_1.asm wurde der R/W-Pin hinzugefügt, damit die Software auch auf einem separaten LCD-Modul funktioniert.

Gliederung der Lektionen

Inhaltsverzeichnis



Lektion 1: ISP-Programmier-Interface

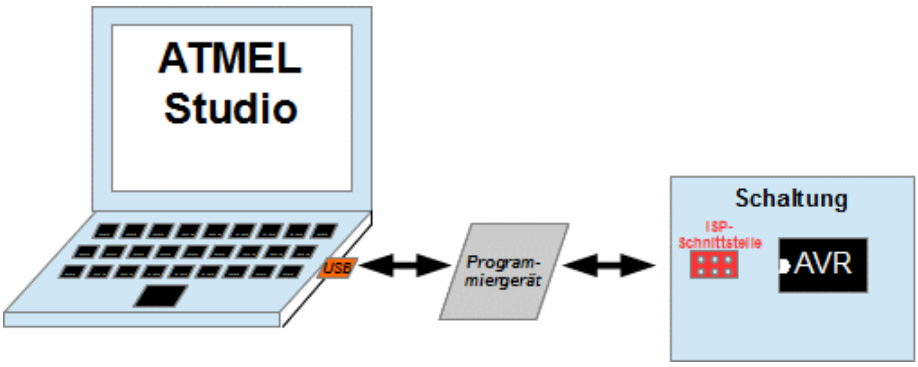
1.0 Übersicht

1.1 Einführung

Um Mikroprozessoren in den Griff zu bekommen, braucht man eine Steuerungsmöglichkeit. Mit Hilfe dieser Steuerung kann man

- Programmcode in den Prozessor schleußen, oder
- diesen auch wieder löschen,
- die Taktquelle für die Taktung des Prozessors auswählen,
- das EEPROM im Prozessor mit Inhalten befüllen,
- Inhalte des Programmspeichers und des EEPROMs auslesen, und
- den Chip gegen das Auslesen von Inhalten schützen.

Ohne diese Steuerungsmöglichkeit geht gar nix und der Prozessor schläft nur vor sich hin und tut gar nix.



In diesem Kurs greifen wir zum Programmieren direkt auf den Prozessor in der fertig aufgebauten Schaltung zu. Das bezeichnet man als "In-System Programming", kurz ISP. Das hat immense Vorteile: Die fertige Schaltung funktioniert wie programmiert, es werden keine weiteren und überflüssigen Zusatzschaltungen benötigt (wie beim Arduino oder beim STK500). Nicht nur sind komplizierte Verkabelungen unnötig, auch der Strombedarf der fertigen Schaltung wird nicht unnötig erhöht. Die ISP-Schnittstelle funktioniert bei allen AVR in gleicher Weise. Wenn also für eine Anwendung weitere Pins benötigt, nimmt man einfach den nächstgrößeren AVR-Typ, schreibt das Programm ein wenig um und programmiert diesen über die ISP-Schnittstelle in der fertig aufgebauten Schaltung. Baut man das Programmierinterface, eine sechspolige Steckverbindung, in die fertige Schaltung ein, kann das Programm nachträglich verändert werden. Weder ist dazu das Ausbauen des Prozessors noch sind irgendwelche Lötarbeiten notwendig.

In diesem Kapitel lernen wir, den Prozessor in der Schaltung anzusprechen und seine Typkennung auszulesen. Die anderen Möglichkeiten zur Chipsteuerung werden kurz aufgezeigt.

Top	Home	Einführung	Hardware	Bedienung

1.2 Hardware

1.2.1 Das Programmiergerät

Als Programmiergerät kommen viele in Frage. Ich verwende den AVR-ISPmkII von ATMEL, der am wenigsten Ärger macht und vom AVR Studio ideal unterstützt wird. Da es das aber nicht mehr zu kaufen gibt, muss man sich ein anderes ausdenken. Es funktionieren prinzipiell alle Adapter, die eine sechspolige ISP-Schnittstelle zur Verfügung stellen, die vom Studio und vom Betriebssystem unterstützt werden (Achtung! Nicht alle käuflichen Adapter funktionieren unter jedem Windows!).

Die meisten Adapter sind solche, die so tun als ob sie ein STK500 Board wären, z. B. der Diamex-AVR oder Usbasp. Da das STK500 mit dem ATMEL®-Studio über eine serielle RS232-Schnittstelle kommunizierten, installieren die Treiber für solche Programmiergeräte eine serielle USB-zu-RS232-Schnittstelle. Da das Studio aber nur die Schnittstellenadressen COM1 bis COM4 abfragt, muss man darauf achten, dass die Schnittstelle in diesem Adressbereich liegt.

Die Beispiele sind alle mit dem AVR-ISPmkII erstellt. Seit das kaputt ist, programmiere ich mit dem Diamex-AVR. Allerdings arbeite ich noch mit Windows®7. Wie sich neuere Windows-Versionen verhalten, kann ich leider nicht testen.

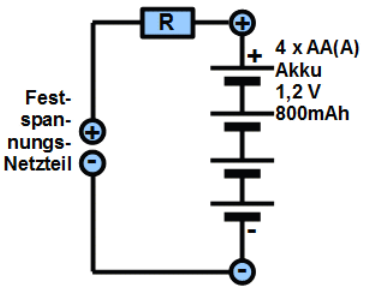
1.2.2 Stromversorgung

AVRs benötigen zum Funktionieren eine Betriebsspannung. Dazu reichen beim ATtiny13A und beim ATtiny13V 1,8 V, beim ATtiny13 2,7 V aus. Maximal darf bei allen Typen 5,5 V an die positive Stromversorgung VCC, darüber gehen sie kaputt.

Wir arbeiten hier mit vier AAA-NiMH-Akkuzellen zu je 1,2 V, macht 4,8 V. Die können entweder in einem Batteriehalter zusammengeschaltet werden oder als Akkupack fertig verschweist sein. Aus dieser Quelle kann auch das Programmiergerät mit versorgt werden (über die GND/VTG-Anschlüsse am ISP6), wenn das keine eigene Versorgung aus der USB-Buchse mitbringt.

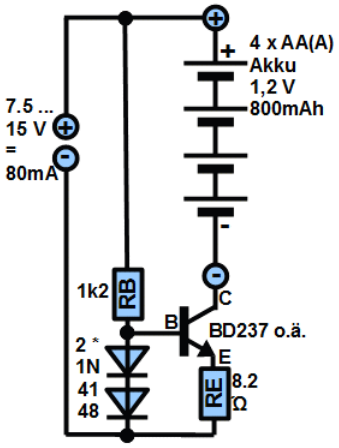
Wer Akkupacks nicht mag, kann jede andere beliebige Gleichspannungsquelle mit 4,5 bis 5,5 Volt verwenden. Sie kann auch weniger haben, bei den Experimenten ab Lektion 10 kommt aber eine LCD zum Einsatz, und die gibt es nur für 3,3 und 5 V Betriebsspannung.

Einfaches Ladegerät



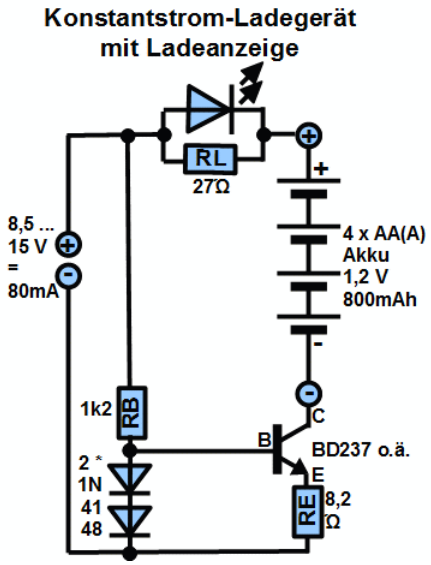
Fest-spannng	Widerstand R	Leistung
V	Ω	W
7,5	27	0,25
9	47	0,5
12	82	1

Konstantstrom-Ladegerät



Wer solche Akkupacks nur deswegen nicht mag, weil er dafür kein passendes Ladegerät nicht hat: dem kann abgeholfen werden. Das Schaltbild zeigt im linken Teil das einfachste Ladegerät: ein Festspannungsnetzteil mit einem Widerstand.

Rechts die gehobene Variante mit einer Konstantstromquelle für 80 mA Ladestrom. Das heisst ein völlig leerer Akkupack mit 800 mAh Kapazität sollte über Nacht für 10 Stunden lang an die Lademimik. Nicht länger, sonst geht er kaputt.

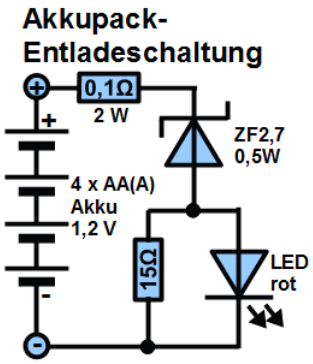


Kapa-zität	Lade-strom	RB	RE	RL
mAh	mA	Ω	Ω	Ω
1.000	100	680	6,8	22
1.200	120	560	5,6	18
1.500	150	470	4,7	15
1.800	180	390	3,9	12
2.200	220	270	3,3	10

Wer vier Einzelakkus in ein gemeinsames Gehäuse stecken will oder wessen Akkupack andere Kapazitäten aufweist, kriegt hier noch ein Ladegerät mit Hinweisen auf die Dimensionierung. Das hat auch noch eine Ladeanzeige, die an den laufenden Ladevorgang erinnert. Auch hier sollte nach 10 Stunden Schluss sein, um den Akku nicht zu überladen.

Auf jeden Fall sollte das Laden getrennt vom Prozessor erfolgen, da die Spannung eines vollgeladenen Akkupacks beim weiteren Laden locker die maximal 5,5 V Betriebsspannung des Prozessors überschreiten kann.

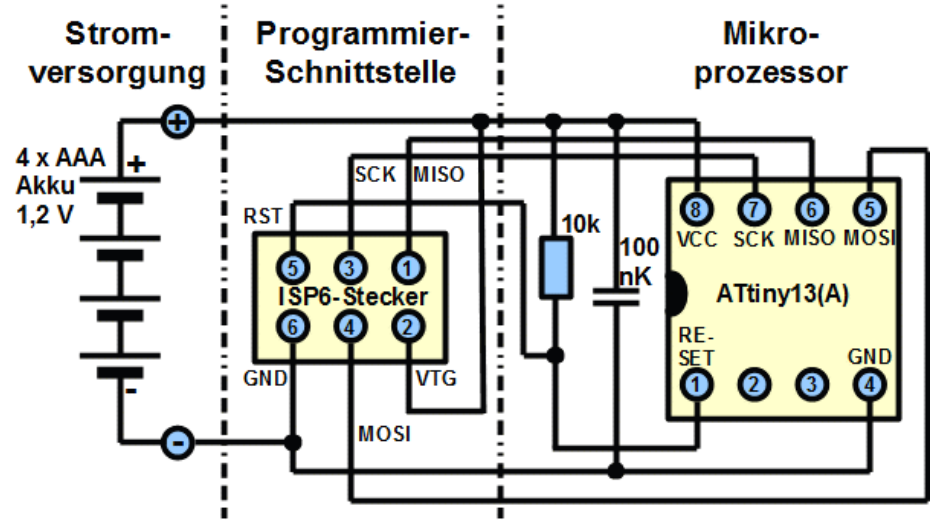
Der Transistor braucht bei 12 V ab ca. 150 mA, bei 15 V ab ca. 100 mA einen kleinen Kühlkörper.



Wer seinen Akkupack vor dem Laden kontrolliert entladen möchte, kann diese Schaltung benutzen. Sie entlädt den Akkupack mit 150 mA. Sinkt seine Spannung unter 4,7 V, geht die LED aus.

1.2.3 Die Programmier-Hardware

Die nötige Hardware für den Zugriff auf den AVR ist hier gezeigt. Die Stromversorgung der Schaltung erfolgt aus einem Viererpack AAA-Akkus mit je 1,2 Volt. Aus dieser Quelle wird auch das Programmiergerät AVR-ISPmkII mit versorgt (über die GND/ VTG-Anschlüsse am ISP6). Über die drei Anschlüsse MOSI, MISO und SCK erfolgt die Kommunikation des Programmiergeräts mit dem AVR-Chip. Der AVR-Chip geht in den Programmiermodus, wenn der RESET-Pin 1 auf niedrigem Potential liegt. Dieser Pin wird durch den Widerstand von 10k auf Plus gezogen und vom Programmiergerät über die Leitung RST auf Null gezogen. Ist die Programmierung beendet, lässt das Programmiergerät diese Leitung wieder los, der 10k-Widerstand zieht RESET auf Plus



und das im Professor abgelegte Programm legt los.

Zwischen der positiven und negativen Betriebsspannung liegt noch ein Keramikkondensator mit 100 nF. Dieser blockt Spannungsschwankungen ab, die von den Schaltvorgängen im Prozessor herrühren können. Der Kondensator ist so nahe als möglich an die Betriebsspannungs-Pins 4 (GND) und 8 (VCC) zu montieren.

Die Pins des Prozessors sind mit denjenigen Signalnamen bezeichnet, die hier praktisch zum Einsatz kommen. In anderen Anwendungen haben viele der Pins andere Funktionen bzw. wechseln ihre Funktion nur im Programmiermodus.

Mit dieser einfachen Hardware können wir schon mit dem AVR kommunizieren und in seine Innereien eingreifen.

1.2.4 Bauteile

Die folgenden Teile beschreiben die Bauteile.

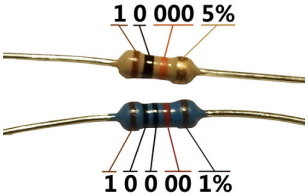
1.2.4.1 Der Akkupack

Das hier ist der verwendete Akkupack. Verwendbar ist alles, was zwischen vier und fünf Volt liefert. Wer ein Netzteil hat, das 5 V bei mindestens 100 mA liefert, braucht den Akkupack nicht.

1.2.4.2 Der ISP6-Stecker

Der ISP-Stecker ist eine zweireihige dreipolige Stiftleiste. Für fliegenden Aufbau und gedruckte Platinen reicht der Einbau einer solchen Stiftleiste aus. Bei Aufbau auf dem Breadboard muss ein Adapter gebastelt werden. Dieser ISP-Steckverbinder muss selbst gebaut werden. Dazu wird ein 3 * 4 Augen großes Stück einer Lochrasterplatine abgesägt und auf der Lötseite eine 2 * 3-polige Stiftleiste mittig aufgelötet (Stiftleiste nur bis Lochtiefe einstecken und mit Lötauge verlöten). Danach zwei 3-polige einreihige Stiftleisten auf der Bestückungsseite einlöten und mit den benachbarten Lötaugen der zweireihigen Stiftleiste verbinden.

1.2.4.3 Der 10k-Widerstand



Das hier sind zwei Bauformen von Widerständen mit 10k. Oben ist ein Kohleschicht-Widerstand, unten ein Metallfilm-Widerstand abgebildet. Die Kodierung des Widerstandswerts ist angegeben.

1.2.4.4 Der 100 nF-Keramikkondensator

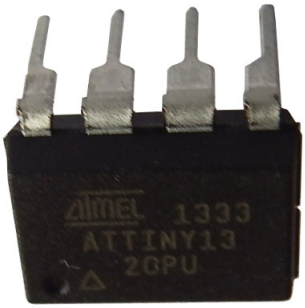
Das hier ist ein Keramik-Abblockkondensator mit 100 nF Kapazität.

1.2.4.5 Die 8-polige IC-Fassung

In diese Fassung kommt der 8-polige Mikroprozessor.

1.2.4.6 Der ATtiny13

Das hier ist der Mikroprofessor. Er hat in seinem Innern ganz viel Hardware, die wir noch kennenlernen werden, angeschlossen an acht äußere Beine, die etwas eigenwillig durchnummeriert sind (linkes Bild). Immerhin ist Pin 1 mehrfach gekennzeichnet. Zum Einen ist an der Eins ein Dreieck aufgedruckt. Zum anderen ist eine Kuhle eingeprägt. Zum Dritten besitzt der Chip mittig eine kleine Einbuchtung an der Seite, auf der die Eins liegt. Mit diesen Merkmalen ist es möglich, den Chip richtig herum in die Fassung zu stecken. Er passt natürlich nicht in die zugehörige Fassung. Das liegt daran, dass seine Beine nicht ganz rechtwinklig nach unten ragen sondern etwas nach außen gerichtet sind. Also biegen wir die vier Beine jeder Seite vorsichtig etwas einwärts, indem wir die jeweils vier Beine jeder Seite auf einer festen Oberfläche etwas eindrücken bis sie rechtwinklig vom Plastikgehäuse abstehen. Warum das produktionstechnisch nötig ist, erschließt sich nicht ganz. Wenn man das nicht macht, besteht die Gefahr, dass beim Eindrücken des Chips in die Fassung Beine ganz arg verbogen werden. Dann besteht Lebensgefahr für den Professor und man hat einen Heidenspaß damit, die verbogenen Beine wieder zu begradigen, ohne dass diese abbrechen und damit die €1,85 in die Tonne getreten werden können.

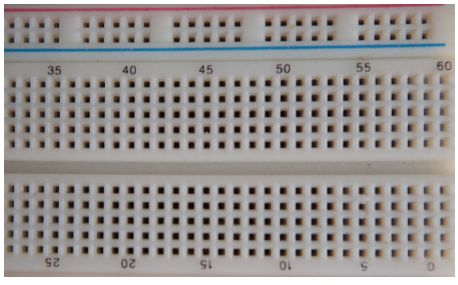


Es ist übrigens egal, ob wir den ATtiny13 oder den ATtiny13A verwenden. Der A ist neuer und billiger als der ohne A, hat einen etwas geringeren Strombedarf und ist ansonsten vom Programmieren her völlig identisch.

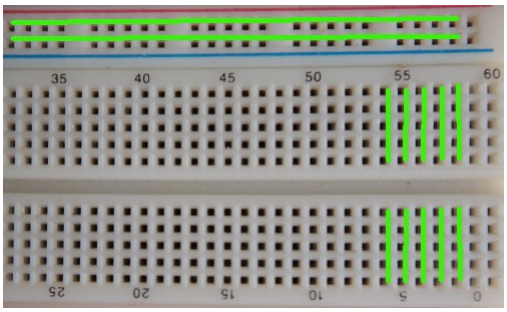
Top	Home	Einführung	Hardware	Bedienung

1.3 Der Aufbau

1.3.1 Das Breadboard



Das hier ist ein Ausschnitt aus dem Breadboard. Breadboard bedeutet "Brütbrett", also ein Experimentier- oder Entwicklersystem.
Mit dem Board kann man Schaltungen und Programme aus-



probieren, bevor diese in eine gedruckte Schaltung und in die "Großproduktion" gehen.

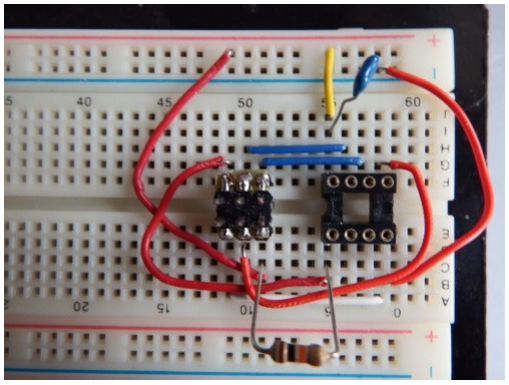
Die grünen Linien rechts zeigen, welche Löcher mit welchen intern verbunden sind (im unteren Bereich sind nur fünf Verbindungen gezeigt). Die oberen beiden Reihen sind für die Stromversorgung, die unteren Spalten für die Bauteile. Daraus wird jetzt auch klar, weshalb der ISP6-Adapter so eigenartig aufgebaut werden muss: die Trennung in der Mitte ist nun mal zwei Punkte auseinander, eine doppelreihige Stiftleiste würde kurzgeschlossen.

1.3.2 Der Aufbau

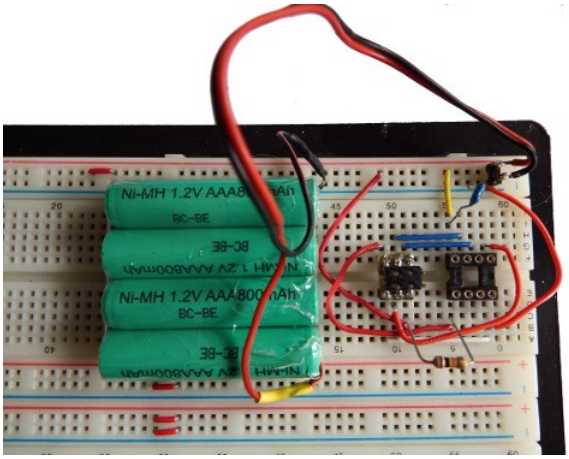
Der Aufbau geht in folgender Reihenfolge vonstatten:

1. ISP6-Adapter und IC-Fassung platzieren.
2. Stromversorgungsleitungen und Direktverbindungen verdrahten.
3. Alle anderen Bauteile (Widerstand, Kondensator) platzieren und verdrahten.
4. Alle Verbindungen zwischen ISP6-Adapter und IC-Fassung "durchklingeln", das heißt mit dem Durchgangsprüfer oder einem Ohmmeter auf Übereinstimmung mit dem Schaltbild überprüfen.
5. Stromversorgung (Akkupack) mit den Stromversorgungsleitungen des Breadboards verbinden.
6. Mit einem Voltmeter an Pin 4 (GND, Minus) und 8 (VCC, Plus) sowie am ISP6-Stecker an den Pins 6 (GND, Minus) und 2 (VTG, Plus) überprüfen, ob die Betriebsspannung dort in Höhe und Polarität richtig anliegt.
7. Den ATtiny13 in die Fassung drücken und das Programmiergerät an den ISP6-Adapter anschließen. Die Kontrolllampe am Programmiergerät sollte jetzt grün leuchten.

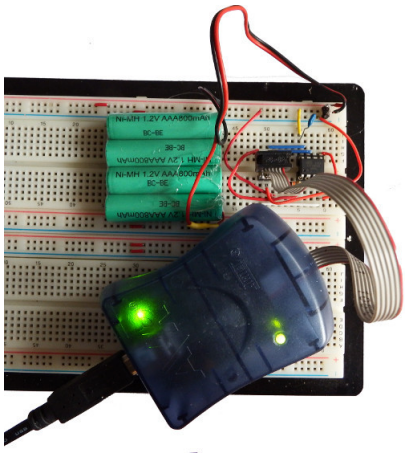
Das Auslassen einzelner Schritte bei dieser Aufbaureihenfolge (besonders der Schritte 4 und 6) wird garantiert mit Fehlermeldungen quittiert. Und für die Fehlersuche sind genau diese beiden Schritte auszuführen, also warum nicht gleich so. Die folgenden Bilder zeigen die Stadien des Aufbaus.



So sieht die Bestückung mit allen Bauteilen aus. Der Akku und der ATtiny sind noch nicht angeschlossen, um die Prüfung der Verbindungen ungestört durchführen zu können.



Der Akkupack setzt das Ganze unter Spannung, noch ohne den Tiny. Jetzt mit dem Voltmeter auf richtige Polaritäten prüfen, um den Tiny nicht den Polaritätstod erleiden zu lassen.



Hier ist jetzt alles komplett und der Programmier-Adapter angeschlossen. Der signalisiert nun mit einem grünen Licht, falls er ein solches hat, dass er korrekt aus der USB-Schnittstelle und mit einem weiteren Lämpchen, dass er korrekt aus der Zielschaltung versorgt wird. Rote Lampen würden sagen, dass wir irgendetwas falsch gebaut haben.

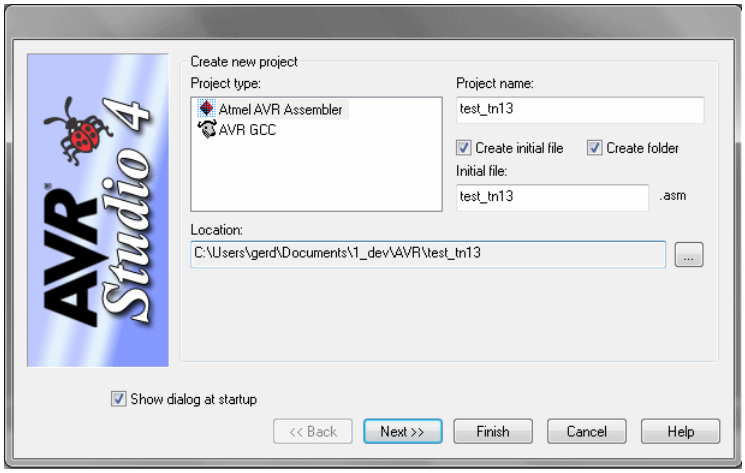
Damit kann es jetzt losgehen mit den Experimenten und dem Zugriff auf die Innereien des Herrn Professors.

1.4 Bedienung

Um an die Innereien des Professors heranzukommen, brauchen wir Software. Die kriegen wir von ATMEL kostenlos, wenn wir dort im AVR-8Bit-Teil nach dem Studio suchen. Allerdings nur für Windoof.

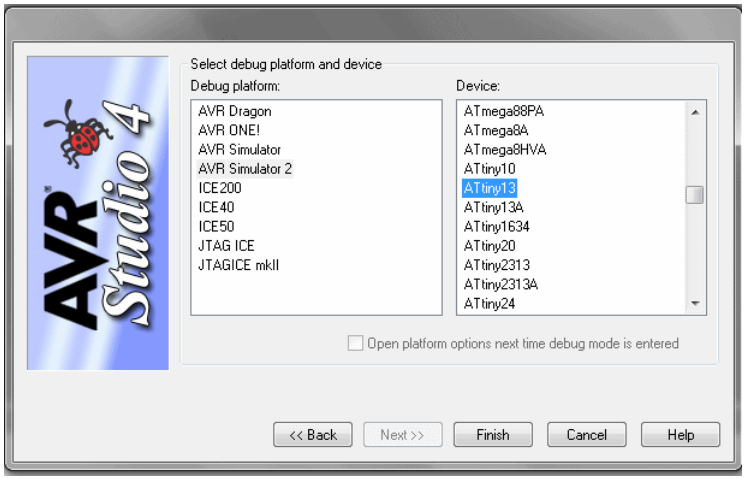
Die folgenden Beispiele sind mit dem ATMEL Studio in der Version 4 erstellt. Das ist eine ältere Version der Studio-Software, die es bei ATMEL auch noch zum Download gibt. Die hat zwar noch erhebliche Fehler, die bis zum Absturz der Software führen können oder Fehler im Timing von RJMP-Anweisungen.

Die Version 4 ist aber immer noch besser als die späteren Versionen 5, 6 und 7. Die arbeiten grundsätzlich genauso, sehen nur ein wenig anders aus und brauchen unangenehm viel Zeit und Speicher, bis sie das tun, was sie sollen, weil sie völlig überladen sind (Elefanten-Software).



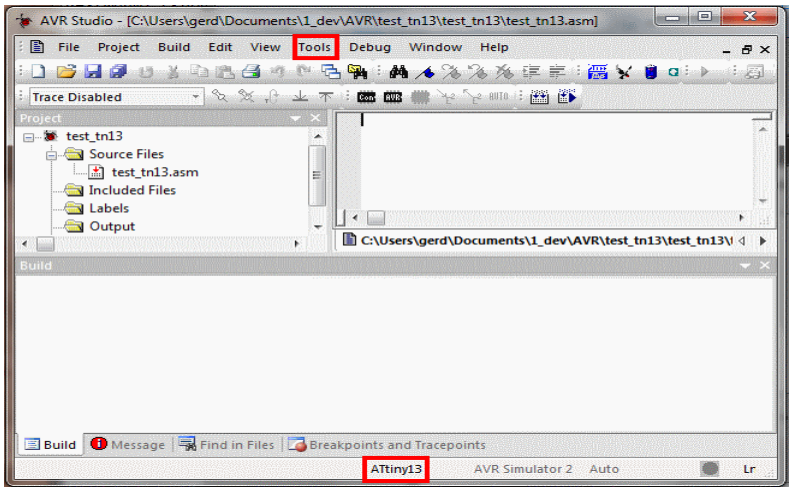
Wichtig: gleich zu Beginn "Assembler" auswählen.

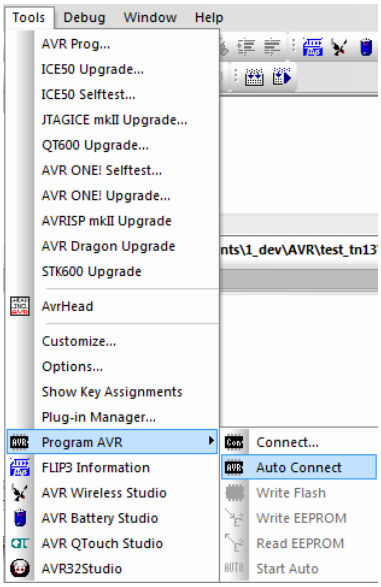
So sieht das Studio nach dem Start aus: es fordert uns auf, ein neues Projekt anzulegen. Dazu wählen wir ein Assemblerprojekt, geben dem Projekt einen aussagekräftigen Namen, lassen das Studio eine entsprechende Quelldatei gleichen Namens in einem von uns eingestellten Ordner des Dateisystems anlegen und gehen auf Next.

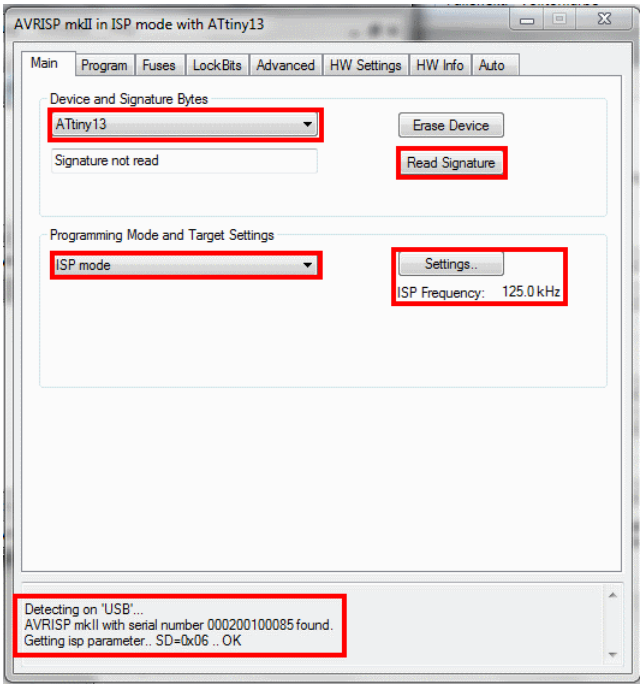


Wir wählen als Plattform den Simulator und als Device den ATtiny13 aus. Für unsere Erkundungen der Innereien ist das Tools-Menue entscheidend.

Nachdem wir das Programmiergerät der USB-Buchse nähergebracht haben, wählen wir im Tools-Menue "Program AVR" und "Auto Connect" in der Hoffnung, es möge erraten, was wir tun möchten.

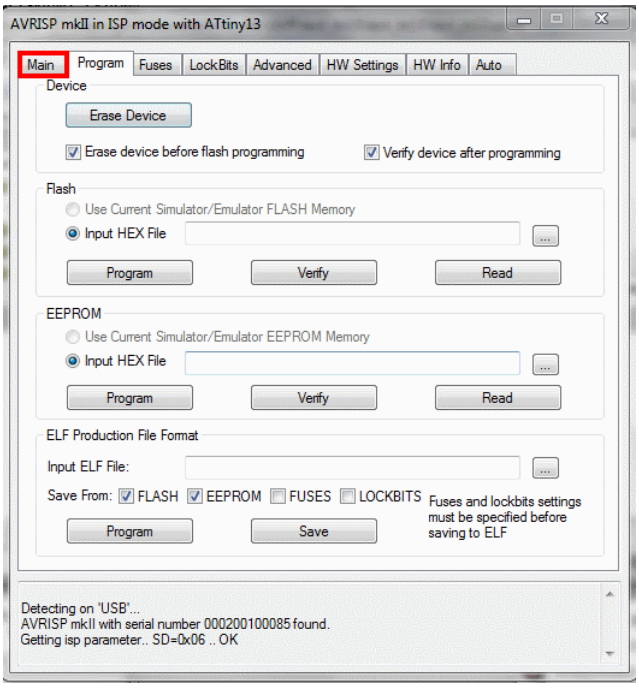


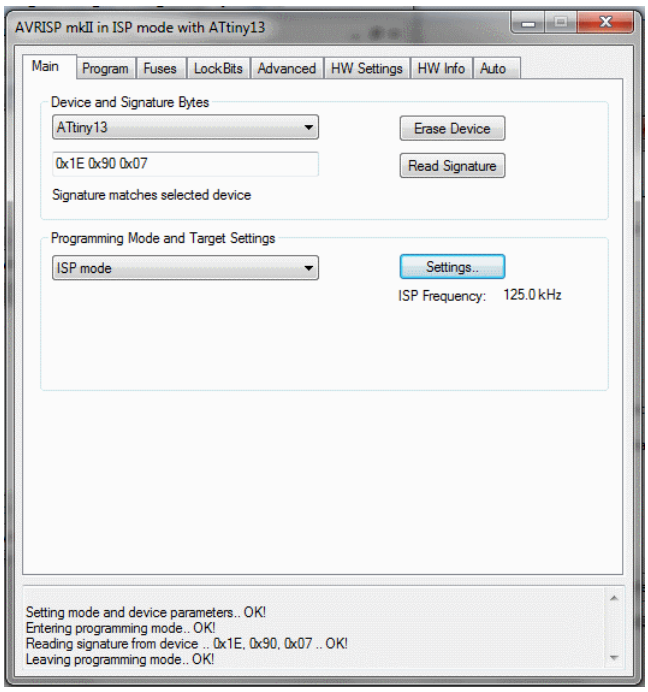




Hat er die USB-Schnittstelle erkannt, öffnet sich das Toolsfenster. In diesem Fenster schalten wir auf den Tab "Main" um.

Hier gibt es nun einiges einzustellen. Im Feld "Device and Signature Bytes" wählen wir schon mal den ATtiny13 aus, drücken aber noch nicht den Button "Read Signature". Zuerst lesen wir im unteren Feld, was uns die Software mitteilt.

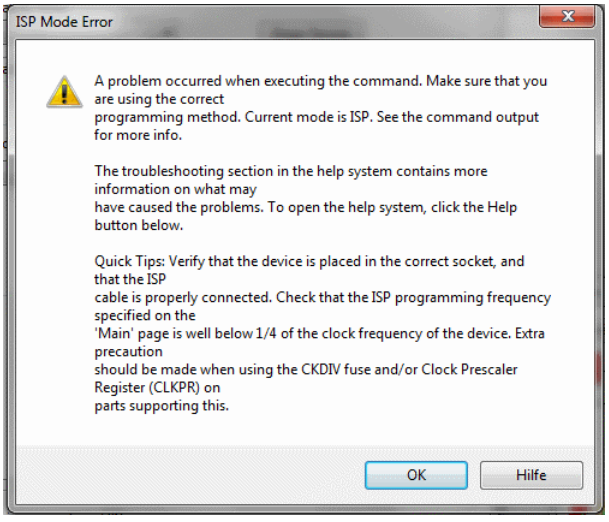




Dann vergewissern wir uns, dass in den "Programming Mode and Target Settings" der ISP-Mode und dass die ISP-Frequenz unter
300 kHz liegt (wenn nicht: Button „Settings" betätigen und die Frequenz neu einstellen. Erst dann können wir die Signatur lesen.

Die ISP-Frequenz wurde hier erfolgreich eingestellt.

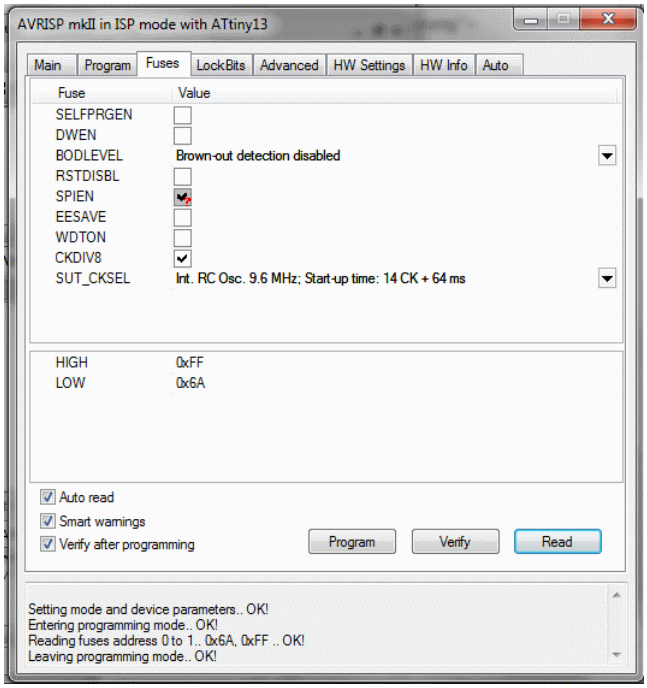
Das ist der Bericht des Programmiergeräts. Es hat die Signatur gelesen, mit dem Sollwert des ATtiny13 verglichen und festgestellt, dass sie korrekt ist. Alles ist prima, unser Gerät funktioniert.

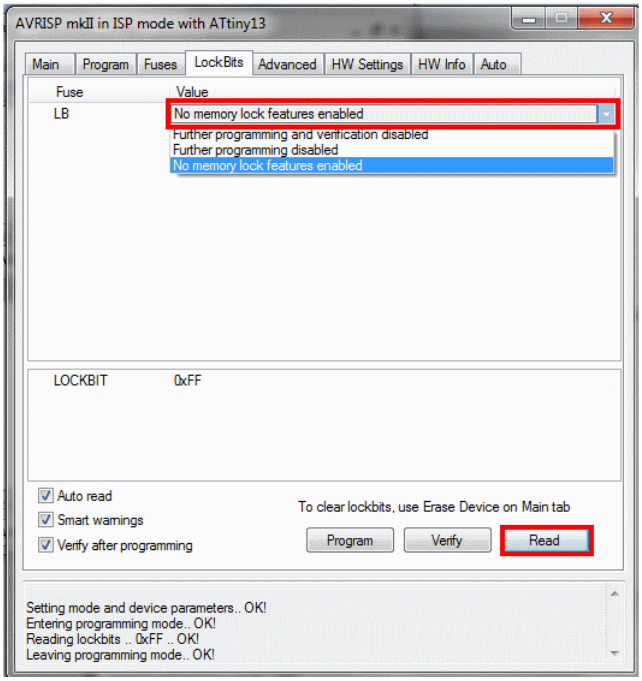


Wäre dem nicht so, dann käme jetzt diese oder eine ähnliche Fehlermeldung. Dann ist Fehlersuche angesagt. Also brav Spannungen messen und Schaltung durchklingeln, wie oben beschrieben.

Um noch einen Blick auf die Fuses des Herrn Professors zu werfen, wird der Tab "Fuses" betätigt. Hier sollte erst mal gar nix geändert werden, weil unkoordinierte Verstellungen schnell zu einem unbrauchbaren und nicht mehr ansprechbaren Chip führen können. Wer hier schon verstehen möchte, was die einzelnen Fuses so bewirken, wirft einen oder mehrere Blicke in das Device Databook des ATtiny13 (zu kriegen unter dieser Adresse, 176 Seiten).

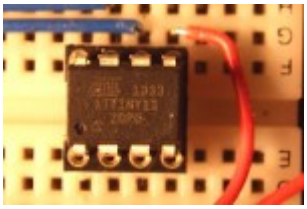
Das hier versteckt sich unter dem Tab "Lockbits": die Schalter zum Abstellen des Auslesens. Sie können nur überschrieben werden, wenn man den gesamten Speicher löscht (mit dem Button im Tab "Main"). Perfekter Softwareschutz.





Soweit die Innereien des Herrn Professors. Mehr kommt später.

Top	Home	Einführung	Hardware	Bedienung



Lektion 2: Eine LED anschalten

Das womit jede Einführung anfängt: es werde Licht. Und es ward: nach ein paar Zeilen Quellcode, Assemblieren und programmieren.

2.0 Übersicht

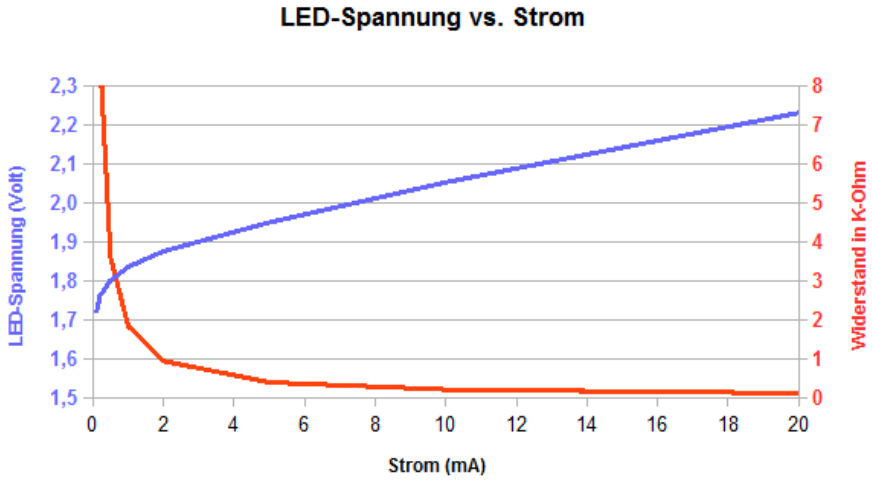
- 2.1 Einführung
- 2.2 Hardware
- 2.3 Programmierung
- 2.4 Simulation der Abläufe im Mikrocontroller

2.1 Einführung

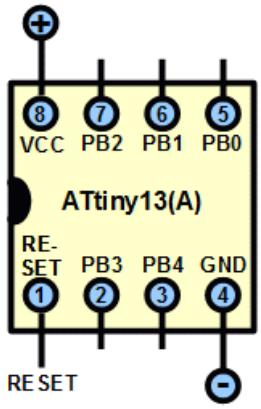
Das erste gigantische Mikroprozessor-Programm schaltet eine Leuchtdiode an. Dabei lernen wir wie ein Programm aussieht, wie es assembliert wird, was dabei herauskommt und wie es in den Mikroprofessor übertragen wird. Wir lernen viel über Leuchtdioden und über I/O-Pins und ihre Manipulation.

2.1.1 Leuchtdioden

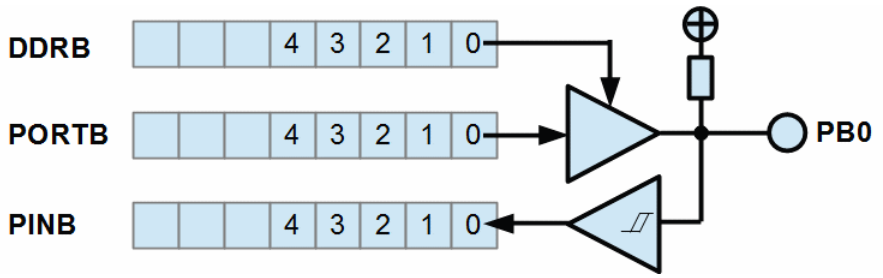
Leuchtdioden senden Licht aus, wenn Strom durch sie hindurch fließt. So weit, so gut. In einem gewissen Bereich gilt: je mehr desto mehr. Der Bereich ist aber sehr eng begrenzt. Zuviel Strom macht dann nicht mehr Licht, sondern nur mehr Wärme. Im Bereich niedriger Ströme von wenigen Milliampere (mA) ist die Lichtzunahme am höchsten, bei hohen praktisch nicht mehr merklich. Auch ist unser Auge und Sehvermögen alles andere als linear, so dass wir "viel" und "viel mehr" zunehmend weniger unterscheiden können. Elektrisch betrachtet ist eine Leuchtdiode ein etwas krummes Bauelement, weil ihr Widerstand mit steigendem Strom abnimmt und fast auf Null sinkt. Das ist der Grund dafür, dass Leuchtdioden immer irgendwie strombegrenzt betrieben werden müssen, weil sie selbst dem Strom nur wenig Widerstand entgegensetzen. Das kann ein Vorwiderstand sein oder ein Stromregler auf Halbleiterbasis. Die Größe von Vorwiderständen kann mit dem Ohm'schen Gesetz R = U / I ermittelt werden. Am Vorwiderstand bleibt die Differenz zwischen der Durchlassspannung der Leuchtdiode und der Betriebsspannung hängen, also z.B. 5,0 - 2,1 V = 2,9 V. Für einen Strom von 10 mA ergibt sich ein Vorwiderstand von R = 2,9 / 0,010 = 290 Ohm.



2.1.2 Prozessorpins als Ein- und Ausgänge



Der Prozessor ATtiny13 hat 8 Pins, von denen 5 als Portpins geschaltet werden können. Sie werden mit PB0 bis PB4 bezeichnet. Jeder dieser Portpins wird über ein Bit in zwei verschiedenen Prozessorspeicherstellen manipuliert, die als DDRB (Datenrichtungsregister) und als PORTB (Ausgangsregister) bezeichnet werden. Der logische Zustand des Portpins kann im PINB (Eingangsregister) gelesen werden.



Jeder Portpin kann in vier Modi geschaltet werden (jeweils für PB0):

1. Ausgangstreiber aus (DDB0 = 0), Pull-Up-Widerstand aus (PORTB0 = 0). In diesem Modus dient der Pin als hochohmiger Eingang, der Zustand am Portpin kann per Software aus dem Register PINB eingelesen (PINB0) und ausgewertet werden.
2. Treiber aus (DDB0 = 0), Pull-Up-Widerstand an (PORTB0 = 1). Dabei zieht ein interner Widerstand von etwa 50k den Pin auf Plus und kann mittels eines Schalters oder Tasters extern auf Null gezogen werden. Beim Einlesen von PINB0 wird bei offenem Schalter Eins erhalten, bei geschlossenem Taster Null.
3. Treiber an (DDB0 = 1), Ausgang auf Null (PORTB0 = 0). Der Pin liegt auf sehr niedriger Spannung in der Nähe der negativen Betriebsspannung. Der Pin kann mit etwa 50 mA belastet werden, wobei die Ausgangsspannung etwas ansteigt.
4. Treiber an (DDB0 = 1), Ausgang auf Eins (PORTB0 = 1). Der Pin liegt auf hoher Spannung in der Nähe der positiven Betriebsspannung. Der Pin kann mit bis zu 30 mA belastet werden, dabei sinkt die Ausgangsspannung ab.

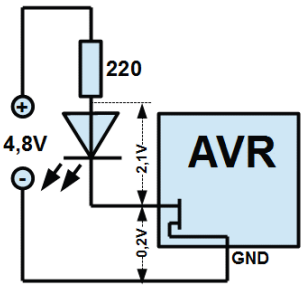
2.1.3 Prozessorpins als Ausgang

Um eine Leuchtdiode mit dem Prozessor einzuschalten resultieren daraus grundsätzlich zwei Möglichkeiten:

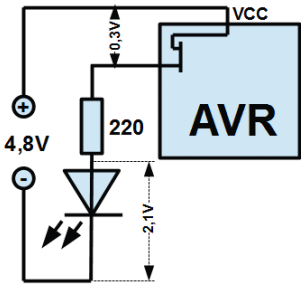
1. Ein Pin des Prozessors wird als Ausgang geschaltet und liefert Strom durch den Vorwiderstand und die Leuchtdiode, die mit dem Minuspol verbunden sind, oder
2. ein Pin des Prozessors wird als Ausgang geschaltet und zieht Strom durch den Vorwiderstand und die Leuchtdiode, die mit dem Pluspol verbunden sind.

Beide Fälle unterscheiden sich: um die Leuchtdiode im ersten Fall zum Leuchten zu bringen, muss der Ausgangspin auf Eins liegen (auf der Betriebsspannung). Im zweiten Fall muss der Ausgangspin auf Null (negative Betriebsspannung) liegen, damit die LED leuchtet.

Hier sind beide Möglichkeiten mit den jeweiligen Spannungsverhältnissen dargestellt. Links die



Sink-Schaltung



Source-Schaltung

Schaltung 2, bei der der Treibertransistor im Prozessor den LED-Strom auf GND zieht ("Sink"), rechts liefert der Treibertransistor den Strom für die LED ("Source"). Die Ausgangstreiber des Prozessors haben in der Sourceschaltung einen etwas grösseren Spannungsverlust im Vergleich zur Sinkschaltung. Bei 10 mA und fast 5 V Betriebsspannung macht das noch nicht so arg viel Unterschied, bei grösserem Strom und bei niedrigeren Be-

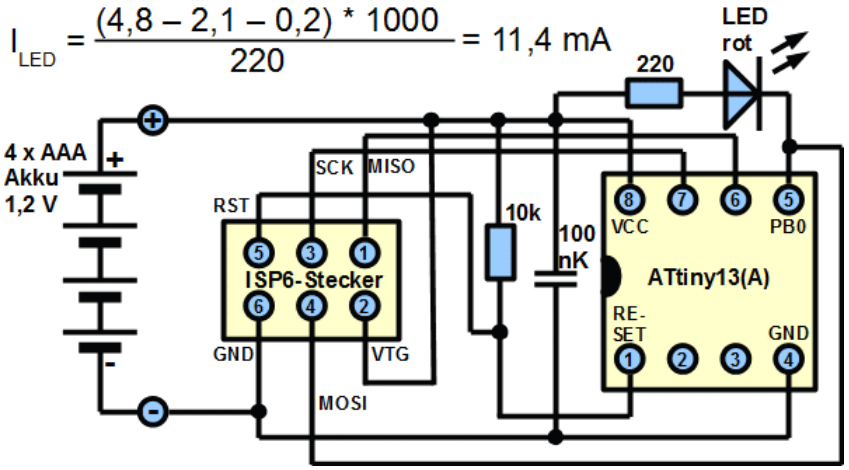
triebsspannungen von z. B. 2,7 V ist der Unterschied gravierend. Als Auslegungsregel sollten Ausgangspins grundsätzlich in Sinkschaltung betrieben werden. Ist das nicht möglich, sollten die Datenblätter bezüglich der realisierbaren Ströme konsultiert werden.

Alle Ausgänge sind übrigens dauerhaft kurzschlusssicher. Liegen allerdings bei mehreren I/O-Pins gleichzeitig Kurzschlüsse vor, kann die maximale Wärmelast des Chips überschritten werden. Sind größere Ströme oberhalb 30 mA zu treiben, sollte ein Transistortreiber zwischengeschaltet werden.

2.2 Hardware

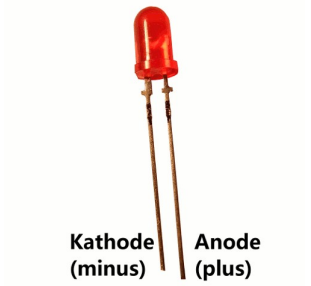
2.2.1 Schaltbild

Um die LED einzuschalten ist hier eine LED an den Portpin PB0 angeschlossen, deren Anode (Pluspol) über einen Vorwiderstand von 220 Ohm mit dem Pluspol verbunden ist. Die LED wird also im Sink-Modus betrieben: ist der Portpin Null, leuchtet die LED. Ist der Portpin Eins, ist sie ausgeschaltet. Links oben ist noch die Formel zur Berechnung des LED-Stroms angegeben. Die 4,8 V sind unsere Akku-Betriebsspannung, die 2,1 V die Durchlassspannung der LED bei ca. 10 mA LED-Strom und die 0,2 V die Treiberspannung bei ca. 10 mA Sinkstrom. Größere Ströme bringen nichts, der Helligkeitsunterschied ist praktisch nicht zu sehen. Nur bei anderen LED-Typen macht ein höherer Strom Sinn. Der Rest der Schaltung bleibt der gleiche, den wir schon zum Zugriff auf den Chip verwendet hatten.



2.2.2 Bauteile

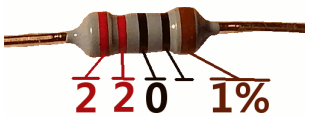
2.2.2.1 Die LED



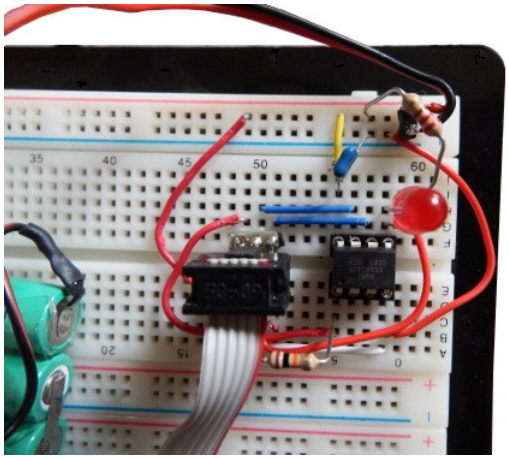
Das hier ist eine rote Standard-LED mit 5 mm. Der längere Anschlussdraht ist die Anode, der Pluspol. Wird sie falsch herum angeschlossen, fließt kein Strom. Wie alle Halbleiterdioden leiten sie in verkehrter Richtung aber dennoch, allerdings liegt diese sogenannte Zenerspannung bei roten LED bei 16 V.

2.2.2.2 Der 220 Ohm-Widerstand

Das hier sind zwei Bauformen des Widerstands von 220 Ohm. Oben ein Kohleschicht-Widerstand mit 5% Toleranz, unten Metallfilm mit 1% Toleranz.



2.2.3 Der Aufbau



Der LED-Aufbau erfolgt so, dass die Kathode mit PB0 (Pin 5) des ATtiny13 verbunden wird, die Anode kommt an die benachbarte leere Spalte. Von dort wird der 220 Ohm-Widerstand mit der Plus-Schiene verbunden. Das war es. Auch mit Strom leuchtet da jetzt mal rein gar nix. Das liegt daran, weil ein nativer ATtiny13 bei jedem Programmstart alle seine Portpins abschaltet. Um diese Portpins einzuschalten, muss erst Programmcode ausgeführt werden.

2.3 Programmierung

2.3.1 Programmspeicher

Um dem ATtiny13 Leben einzuhauchen, muss er programmiert werden. In diesem Fall mit den beiden Hexadezimalworten 9AB8 und 98C0 am Beginn seines Programmspeichers. Die entsprechen binär den Werten 1001.1010.1011.1000 und 1001.1000.1100.0000 an den Adressen 0000 und 0001 seines Flash-Speichers. Beim Einkauf (und nach dem Erase mit dem Programmiergerät) stehen im gesamten Programmspeicher nur lauter Einsen herum, der Prozessor tut damit rein gar nichts. Das gleiche würde übrigens passieren, wenn der Flashspeicher mit lauter Nullen gefüllt wäre. Damit lernen wir die beiden ersten Befehle des Prozessors kennen: 0000.0000.0000.0000 und 1111.1111.1111.1111. Sie heißen "Tu nichts" oder englisch "No operation", kurz NOP. Der Programmspeicher des ATtiny13 hat 1.024 Bytes, in die 512 Befehlsworte zu je 16 Bits passen. Das hört sich nicht nach viel an, aber in Assembler ist das eine ganze Menge. Unser kompliziertestes Programm wird einige zehn Befehlsworte haben, und ich bin auch mit komplizierten Abläufen noch nie an diese Grenze herangekommen. Meine Schrittmotorsteuerung hatte 141 Befehlsworte. Wenn also Arduino-Freaks über so wenig Programmspeicher die Nase rümpfen, liegt das nur an ihrem total uneffektiven Speicherfresser-C-Stil.

2.3.2 Quellcode

Da man sich Binärcodes wie oben 9AB8 und 98C0 nur sehr schwer merken kann, hat man dafür menschengerechtere Kürzel erfunden, woraus und mit deren Hilfe dieses binäre Kauderwelsch auf einfache Weise erzeugt werden kann. In diesem Fall schreibt der Programmierer die beiden Zeilen

```
sbi DDRB,DDB0
cbi PORTB,PORTB0
```

in eine Textdatei. Aus diesen beiden Zeilen macht der Assembler, ein Übersetzungsprogramm, die beiden Befehlsworte 9AB8 und 98C0. Das Kauderwelsch der beiden Zeilen ist nicht so arg viel verständlicher als die Binär- oder Hexadezimalworte. Es erschließt sich erst, wenn man die ausfürliche Version der beiden Zeilen dazu schreibt:

- Setze Bit im I/O-Register [set bit I/O, SBI] der Datenrichtung von Port B [DDRB] des Portbits DDB0 [DDB0] auf Eins
- Lösche Bit im I/O-Register [clear bit I/O, CBI] im Ausgangsport des Ports B [PORTB] das Portbit PORTB0 [PORTB0] durch Setzen auf Null

Das hört sich schon etwas verständlicher an. Mit diesem Hintergrundwissen kann man sich nun

SBI und CBI vielleicht besser merken. Dem Übersetzungsprogramm, dem Assembler, ist es übrigens egal, ob Instruktionen oder Parameter in Groß- oder Kleinbuchstaben geschrieben sind.

Grundätzlich entspricht jede Zeile im Quellcode (z.B. SBI DDRB,DDB0) genau einer ganz bestimmten Operation des Prozessors, einem einzigen Befehlswort. Nur Operationen, die der Prozessor tatsächlich physisch beherrscht, haben eine Entsprechung in einem Assemblerkürzel. Das ist unter den Programmiersprachen einzigartig, denn alle anderen Sprachen verwenden Kürzel, die eine Vielzahl von einzelnen Prozessoroperationen nach sich ziehen. In Assembler kann man sich sicher sein, dass jede Codezeile sich in genau eine Operation des Prozessors verwandelt. Die Assemblerkürzel sind nur Repräsentationen des Prozessorkönnens. Sie werden als „Mnemonics" (Gedächtnisstützen) bezeichnet.

Das ist auch der Grund dafür, warum sich Assemblerinstruktionen in vielen unterschiedlichen Dialekten nicht groß darin unterscheiden, was der Prozessor tatsächlich macht. Jeder Prozessor kann zwei Zahlen addieren, in unterschiedlichen Dialekten wird das manchmal nur unterschiedlich mit Gedächtnisstützen belegt. Wer also AVR-Assembler beherrscht, muss sich bei PIC-Assembler nur geänderte Mnemonics und andere Fähigkeiten des Prozessors draufschaffen. Das Prinzip, ein Mnemonic steht für einen bestimmten Ablauf im Prozessor, bleibt das Gleiche.

2.3.3 Assemblieren

In der Codezeile "sbi DDRB,DDB0" steht "sbi" für eine Prozessorinstruktion, "DDRB" und "DDB0" sind hingegen Parameter für diese Instruktion. "DDRB", das Datenrichtungsregister von Port B, ist im Portbereich des Prozessors angesiedelt. Das Handbuch zum Prozessor sagt in seiner Portliste dazu:

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x18	PORTB	–	–	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
0x17	DDRB	–	–	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x16	PINB	–	–	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0

DDRB liegt also im Portbereich an der Adresse 0x17, das im Quellcode verwendete Wort DDRB übersetzt sich daher in 0x17, DDB0 in 0x00. DDRB und DDB0 sind Symbole für Zahlen, die man sich nicht merken will. Und deren Merken auch sinnlos wäre, denn bei einem anderen AVR-Typ könnte dieser Port an einer anderen Adresse liegen. Die beiden Ports DDRB und PORTB sind im Handbuch folgendermaßen beschrieben:

DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
	–	–	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

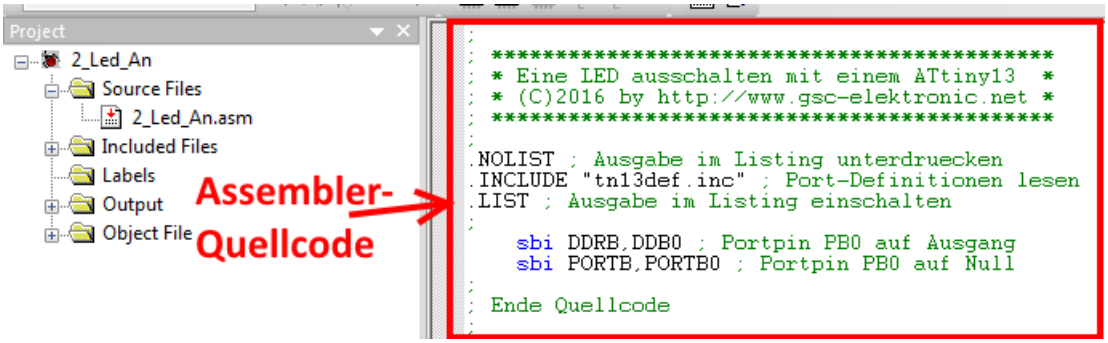
PORTB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
	–	–	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

R/W bedeutet, diese Bits im Port lassen sich sowohl Lesen (R) als auch Schreiben (W). Initial Value bedeutet, dieses Bit wird bei einem Reset des Prozessors auf diesen Wert gesetzt.

2.3.4 Quellcode schreiben

Um Assemblerprogramme zu schreiben, braucht man einen einfachen Texteditor, der einfach ASCII-Zeichen in eine Datei schreibt, z.B. Notepad (in Windoof) oder KWrite (in Linux-KDE) und nennt diese "irgendwie.asm". Wer es etwas komfortabler haben will, schreibt den Quellcode im Editorfenster des Studios. Hier legen wir ein neues Projekt mit Namen "2_Led_An" an und schreiben im Editorfenster:



Im Quellcode (hier als asm-Datei in Assemblerformat zum Download) sind neben vielen Kommentaren (Kommentare beginnen immer mit Semikolon) noch die drei Zeilen

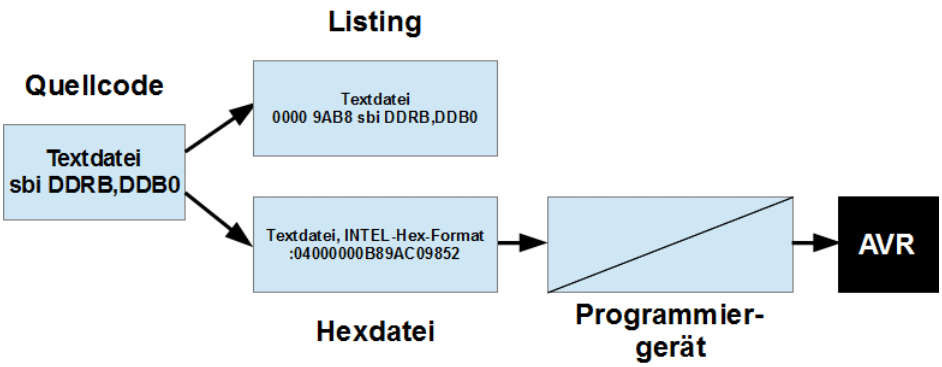
```
.NOLIST ; Ausgabe im Listing unterdruecken
.INCLUDE „tn13def.inc“ ; Port-Definitionen lesen
.LIST ; Ausgabe im Listing einschalten
```

eingebaut. Die Datei tn13def.inc enthält alle Definitionen des Prozessortyps, dort stehen die Adressen von DDRB und PORTB und sind die Bits DDB0 und PORTB0 angegeben. Die Datei findet sich im Programmpaket des Studios und das Studio findet sie. Damit das Listing nicht mit diesen ellenlangen Definitionen vollgeschrieben wird, wird seine Ausgabe mit der Assemblerdirektive .NOLIST abgeschaltet, danach mit .LIST wieder eingeschaltet.

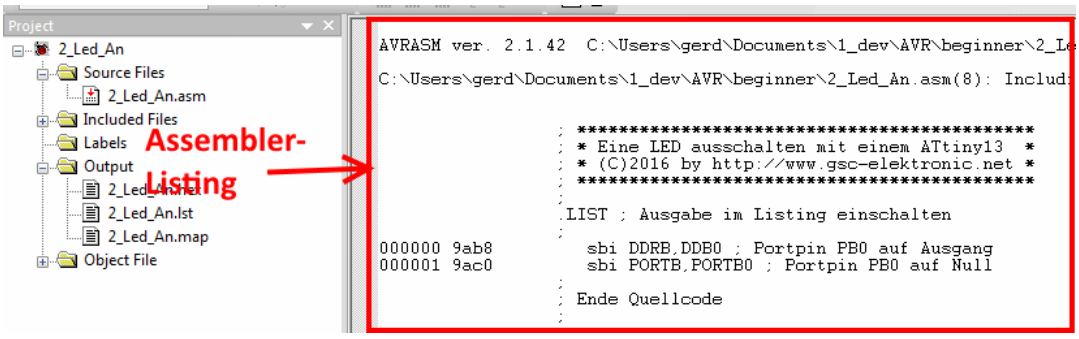
Zur Erleichterung der Erkennung der Art von Einträgen färbt der Studio-Editor diese in unterschiedlichen Farben ein.

2.3.5 Assemblieren

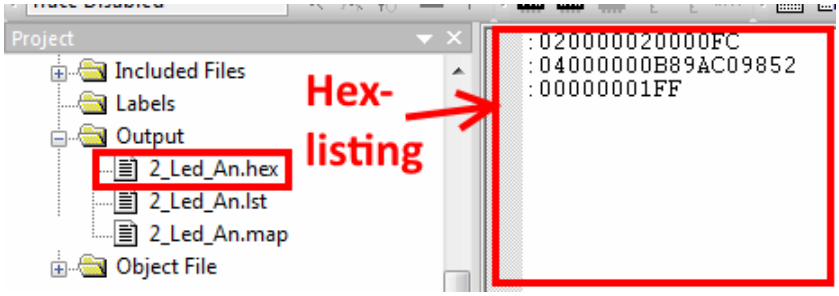
Das hier ist der weitere Weg des Quellcodes im überblick. Mit dem Assemblieren werden aus dem Quellcode ein Listing und eine Datei mit Hexcode erzeugt. Der Hexcode kann dann an eine Schreibsoftware weitergeleitet und mittels eines Programmiergeräts in den Flashspeicher des AVR übertragen werden.



Assemblieren geht mit einem externen Assembler oder mit dem im Studio eingebauten Assembler. Als externen Assembler empfehle ich meinen eigenen, gavrasm. Studio-Intern wird einfach der Menuepunkt "Build" angestoßen und das Übersetzungswerk geht ab. Es endet mit folgenden Ergebnissen:

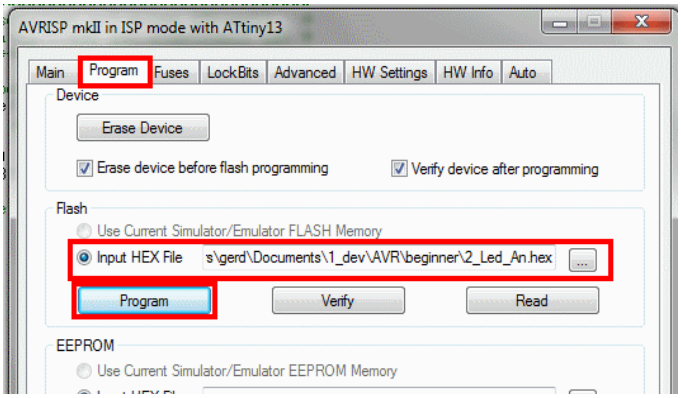


Hier legt der Assembler alle seine Ergebnisse, Beobachtungen und Meldungen ab. Hier kann man sich den erzeugten Hex-Code des gesamten Programmes anschauen. Das muss man aber nur dann, wenn der Assembler Fehler beim Assemblieren meldet.

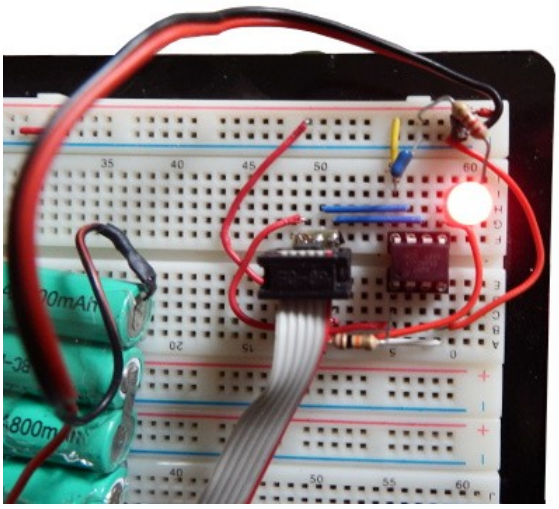


Dies ist der erzeugte Hexcode in menschenlesbarer Form, im Intel-Hex-Format. Dieser enthält nur den nackten Maschinencode, zusammen mit Adressangaben und Prüfbytes. Um diesen Code müssen wir uns nicht näher kümmern.

2.3.6 In den Flashspeicher brennen



Um das binäre Programm in den Chip zu übertragen, starten wir im Studio wieder die Tools und dort "Autoconnect". Im Tabulator "Main" versichern wir uns, dass der Prozessor ansprechbar ist, indem wir die Typenbytes auslesen. Dann steuern wir den Tabulator "Program" an. Dort wählen wir mit dem kleinen Quadrat hinter "Input Hex File" die vom Assembler erzeugte Datei aus und starten mit dem "Program"-Button die Übertragung in den Chip.



Nach einigem Gezappel der LED, die die Programmierimpulse anzeigt, bleibt die LED dauernd an. Unsere zwei Programmzeilen haben ganze Arbeit geleistet und machen, was sie sollen.

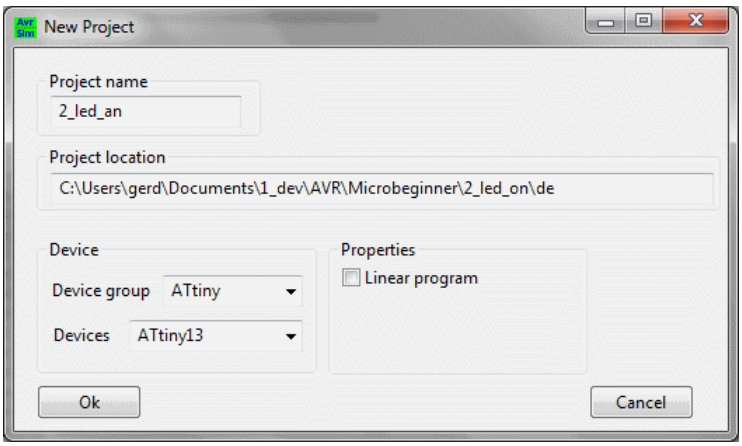
Top	Home	Einführung	Hardware	Programmierung

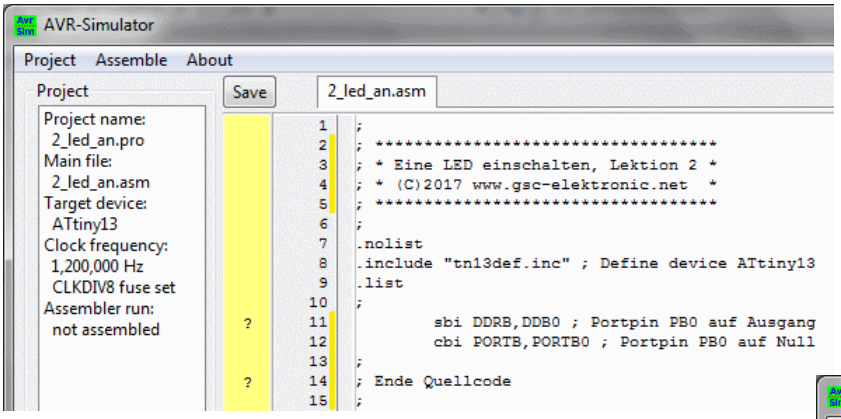
2.4 Simulation der Abläufe im Mikrocontroller

Mit dem Simulator avr_sim können die Abläufe im Mikrocontroller sichtbar gemacht werden. Ein Simulator „tut so, als ob" er ein Mikrocontroller wäre, ist aber nur ein möglichst getreues Abbild der Innereien des Controllers auf dem PC. Er liest die Befehle ein und führt sie aus „als ob".

Um avr_sim mit dem hier beschriebenen Programm zu laden, schreiben wir den Quellcode in eine neue Datei.

Dazu starten wir avr_sim und wählen im Menue unter „Project“ und „New“ die Erzeugung einer neuen Datei aus. Nach der Eingabe eines Projektnamens und der Auswahl des Projektordners (in die Editierzeile „Project location“ klicken und einen Ordner auswählen), dann unter „Device“ ATtiny und ATtiny13 wählen und unter „Properties“ den Haken bei Interrupts wegnehmen und mit OK bestätigen.



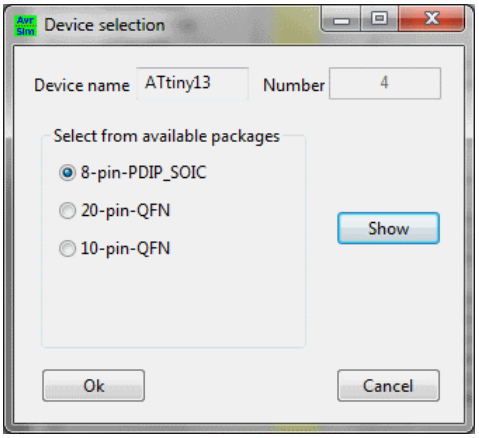


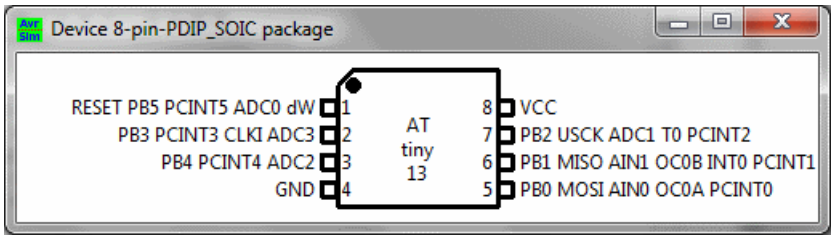
Im Editorfenster löschen wir den gesamten Text und tippen unseren ersten Quellcode in Assemblersprache ein.

Mit dem Knopf „Save“ speichern wir den neuen Text auf der Festplatte ab.

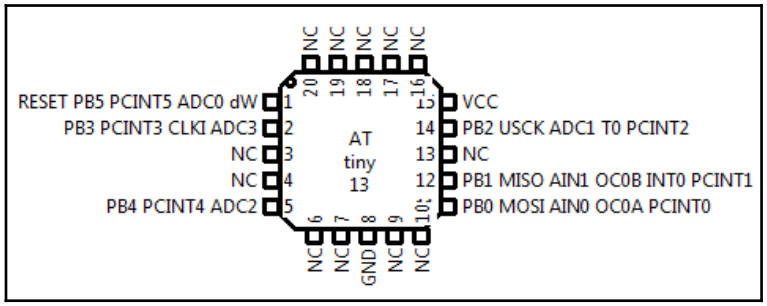
Dann erscheint ein etwas eigenartiges Fenster und will von uns wissen, in welcher Verpackung wir den ATtiny13 verwenden wollen. Angeboten sind 8-pin PDIP/SOIC, 20-pin-QFN und 10-pin-QFN. Mit dem Knopf „Show“ können wir uns die Verpackungen angucken.

Beim ATtiny13 ist es ganz egal, was wir auswählen. Es gibt aber AVR-Typen, die in unterschiedlichen Packungen unterschiedlich viele I/O-Pins haben, deswegen das Fenster hier.



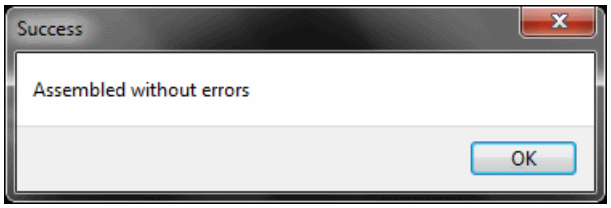


Das ist die PDIP/SOIC-Packung (die wir auch auswählen).



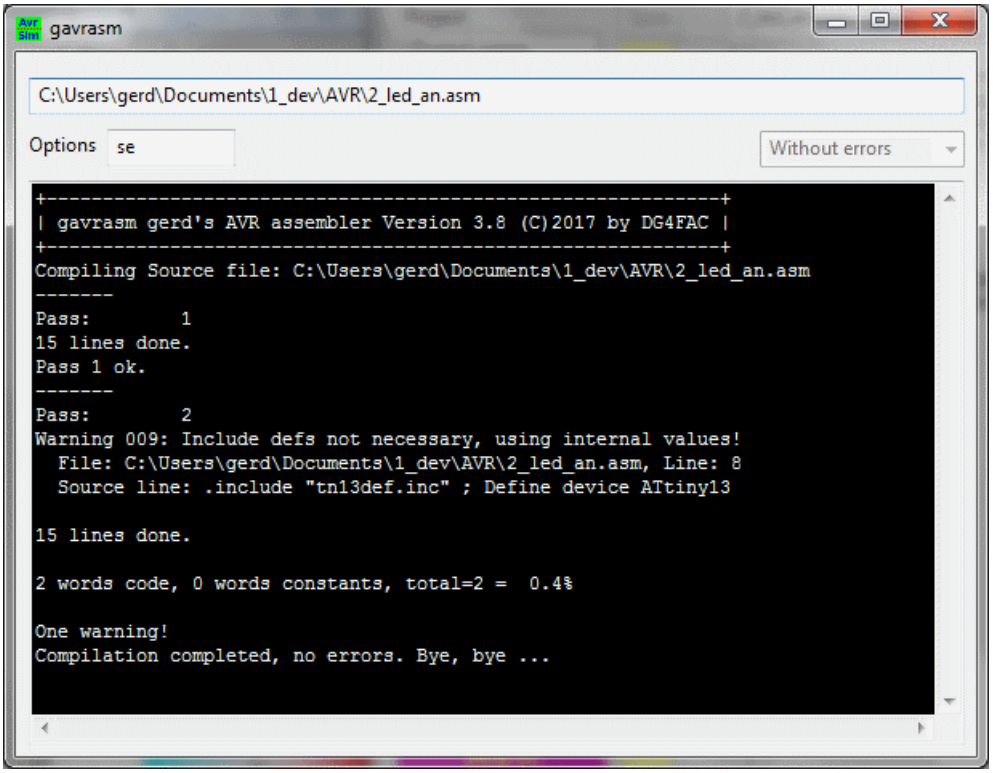
Und das wäre die entsprechende 20-pin-QFN-Packung mit ganz vielen gar nicht beschalteten Pins (NC = not connected).

Nun müssen wir „assemblieren“, damit aus dem Text Code gemacht wird, den der Controller versteht. Dazu drücken wir den Menueeintrag „Assemble“. Das öffnet zwei neue Fenster.



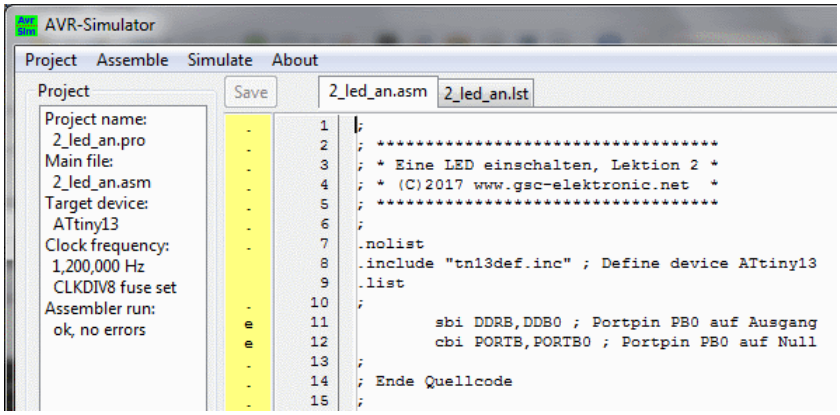
Das Erste teilt uns mit, dass unser Übersetzungsversuch geglückt ist. Erscheint hier was mit „errors“, dann haben wir uns vertippt und müssen den Fehler suchen und beheben.

Das zweite Fenster ist der Assembler mit seinen Meldungen.

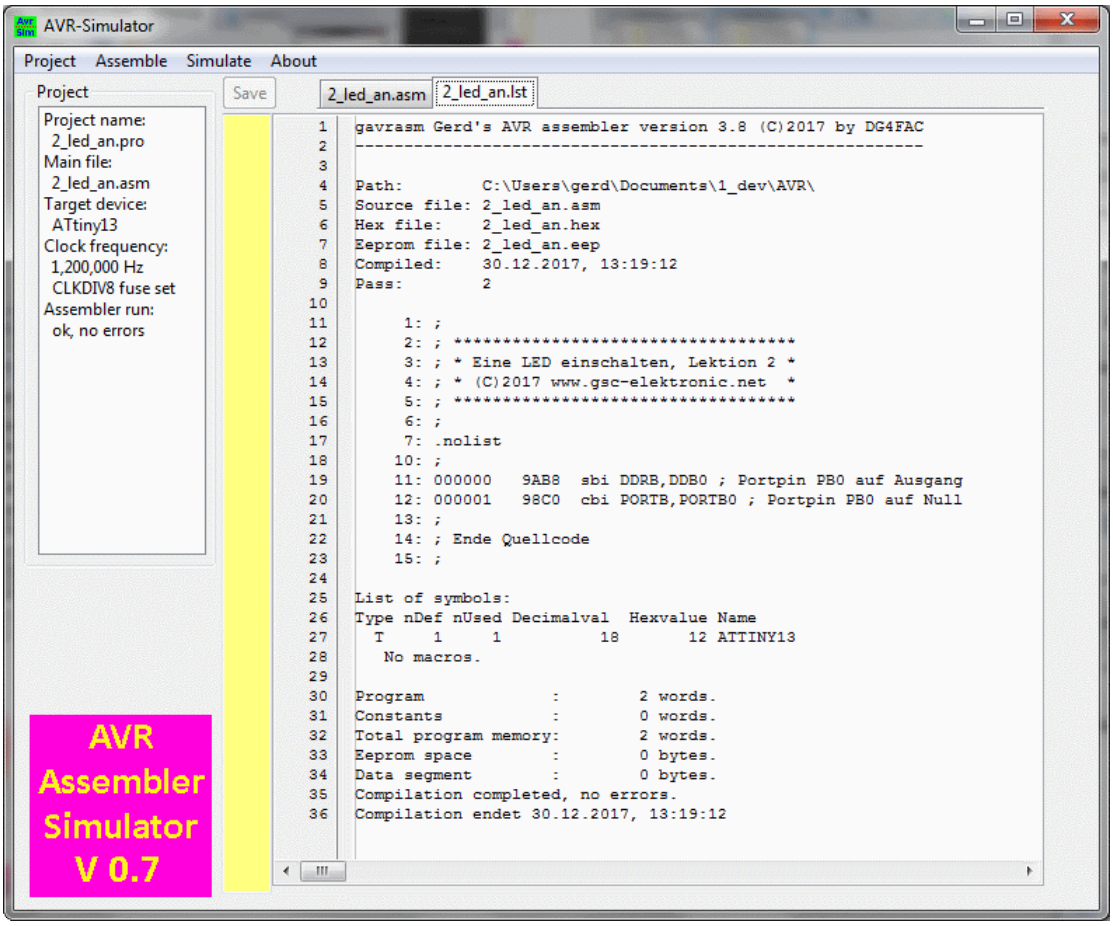


Die ausgegebene Warnung für Zeile 8 können wir getrost ignorieren, sie ist nur was für Spezialisten, die die eingebauten Definitionen nicht mögen könnten.

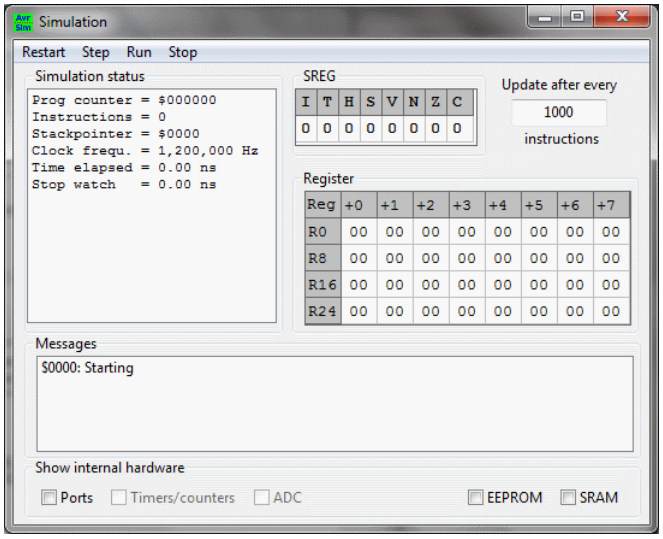
Wichtig ist nur die Meldung am Ende: „Compilation completed, no errors.“. Sie zeigt erfolgreiches Assemblieren des eingefütterten Quellcodes an.



Die Anzeige hat nun im gelben Feld Punkte bei den nicht ausführbaren Zeilen und zwei kleine „e“ bei den beiden Zeilen, die ausführbaren Code erzeugen („executable“). Außerdem ist ein weiterer Reiter im Tabulator sichtbar, der das Listing des Assembliervorganges enthält. Das sieht so aus:

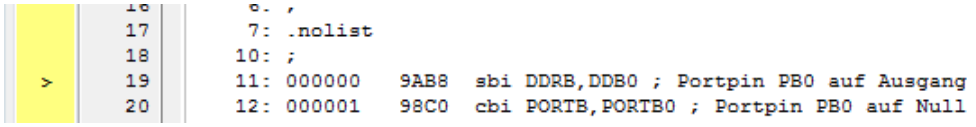


Die beiden ausführbaren Codezeilen haben nun Adressen (000000 und 000001) und ausführbaren Code (9AB8 und 9BC0), beides in hexadezimalem Zahlenformat. Was wir nicht sehen ist die neu erzeugte Hexdatei, die den Programmcode enthält und hinten „.hex“ heißt.

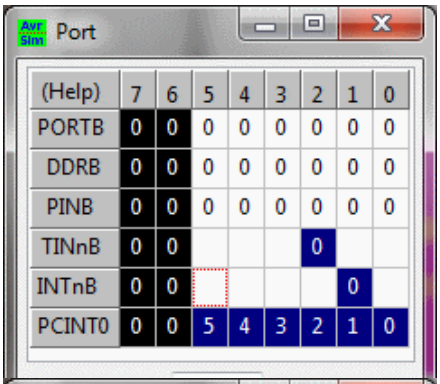


Zum Simulieren des Programmes betätigen wir den Menueeintrag „Simulate“. Das lädt die Hex-Datei in den Simulator und fördert das linke Fenster zutage.

Hier sind das Statusregister des Prozessors (in binär) und seine 32 Register zu sehen (in hexadezimal). Außerdem der Simulationsstatus.



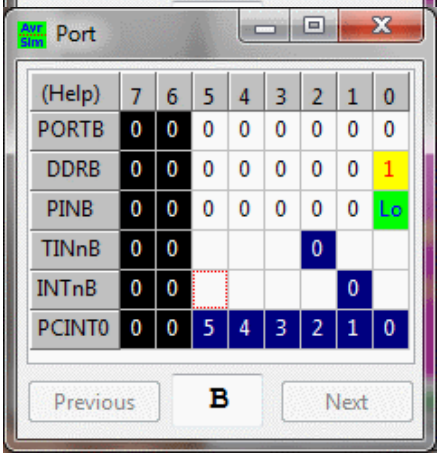
Fast kaum merklich zeigt jetzt ein kleines „>“ im gelben Feld des Editorfensters auf die erste ausführbare Codezeile im Listing. Hier geht also die Simulation los, bei Adresse 000000.



Klicken wir im Simulationsfenster unter „Show internal hardware" das „Ports" Auswahlfeld an, erscheint eine Darstellung von Port B. Der Reset hat alle PORTB- und DDRB-Bits auf Null gestellt, beim Lesen von Port PINB kämen auch überall Nullen heraus.

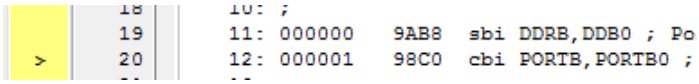
Die restlichen Bestandteile des Ports B werden später verwendet und können einstweilen ignoriert werden.

Mit dem Menueeintrag „Step" im Simulationsfenster simulieren wir die Ausführung der ersten Instruktion im Prozessor, also „sbi DDRB,DDB0".



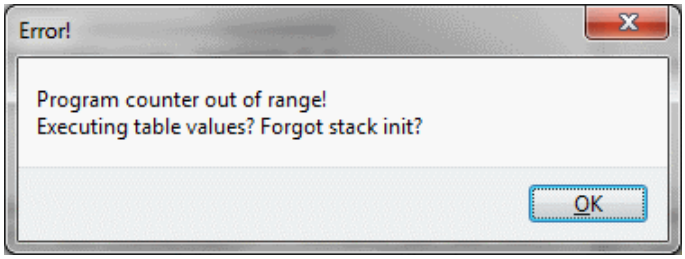
Der Effekt der Instruktion zeigt sich im Ports-Fenster: Bit 0 im DDRB-Portregister in Port B ist „1" geworden. Das Portbit PB0 wird jetzt vom Zustand des PORTB-Registers im selben Bit 0 angetrieben und daher auf Null gesetzt. Beim Lesen des Eingabe-Portregisters PINB käme bei diesem Bit eine Null heraus („Lo" = Low = 0).

Da der I/O-Portpin jetzt low ist und die Leuchtdiode und der Widerstand an den Pluspol angeschlossen sind, geht die LED jetzt an.



Der Programmzeiger zeigt jetzt auf den nächsten ausführbaren Befehl.

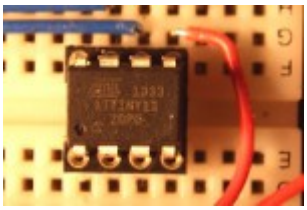
Der nächste Schritt (mit „Step" im Menue) wird das Portbit PORTB0 auf Null setzen, was es aber schon ist. Es passiert also rein gar nichts am Port. Nun passiert aber folgendes: es gibt eine Fehlermeldung vom Simulator!



Das rührt daher, weil der Programmzähler nach der Ausführung der zweiten Instruktion jetzt auf Adresse 000003 steht, unser Programm aber gerade mal nur zwei Worte hat. Die Fehlermeldung ist daher in Ordnung. Sie passiert immer dann, wenn unsere Programmausführung in undefinierte Bereiche des Programmspeichers abschweift, also wenn etwas an unserem Quellcode mächtig schief gegangen ist.

Soweit die ersten Schritte der Simulation. Simulation hilft aber nicht nur beim Verständnis dessen, was in dem Prozessor gerade vor sich geht. Mit der Simulation kann man auch Ausführungszeiten messen oder die Ausführung von Teilen des Codes überprüfen (z.B. wenn der Controller partout nicht das macht, was man sich gedacht hat, dass er gerade tun solle). Wir werden die Simulation in den weiteren Lektionen ausgiebig verwenden, um dem geneigten Leser ihre segensreiche Verwendung als wichtiges Tool nahezubringen.

Top	Home	Einführung	Hardware	Programmierung



Lektion 3: Eine LED blinkt

Mit dieser Lektion kommt Leben in die Bude: die LED blinkt. Zuerst hektisch, dann gemütlich.

3.0 Übersicht

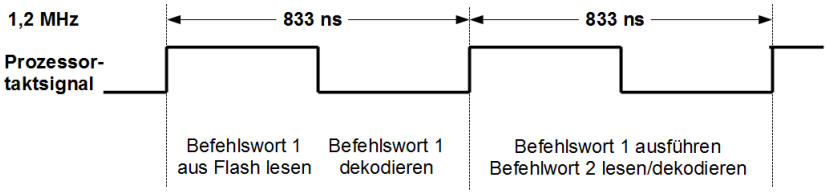
3.1 Einführung

Zum Blinken einer LED muss diese rhythmisch ein- und ausgeschaltet werden. Die Grundlagen zum Ein- und Ausschalten hatten wir schon in der vorherigen Lektion: cbi PORTB,PORTB0 schaltet sie ein, sbi PORTB,PORTB0 aus. Das war es. Dummerweise braucht ein mit 1,2 MHz betriebener Prozessor dafür nur 4 / 1.200.000 Sekunden = 0,000.003.33 Sekunden, was für das menschliche Auge etwas zu schnell ist. Die Möglichkeiten, den Prozessor gezielt mit etwas anderem zu beschäftigen, werden hier aufgezeigt.

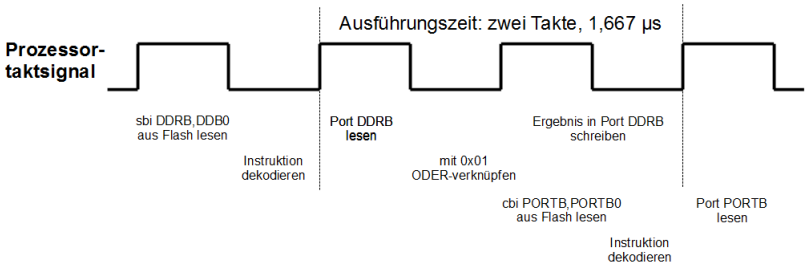
3.1.1 Ausführung von Instruktionen durch den Prozessor

So bearbeitet ein AVR programmierte Befehlsworte:

1. Das nächste Befehlswort wird aus dem Flashspeicher gelesen.
2. Das Befehlswort wird dekodiert, das heißt in Schrittfolgen zerlegt.
3. Das Befehlswort wird ausgeführt. Währenddessen wird bereits das nächste Befehswort gelesen und dekodiert ("Pre-Fetch").



Eigentlich dauert die Bearbeitung eines Befehlswortes daher zwei Taktzyklen. Da aber während der Ausführung schon das nächste Befehlswort geholt und dekodiert wird, braucht die Instruktion effektiv nur einen Taktzyklus. Das funktioniert meistens, aber dann nicht, wenn die Instruktion mit einem Sprung zu einer anderen Adresse im Flashspeicher verbunden ist. In diesem Fall ist das schon gelesene und dekodierte Wort wertlos, da es nicht zur Ausführung kommt. Daher dauern alle Instruktionen, bei denen gesprungen wird, mindestens zwei Taktzyklen. Wegen des Pre-Fetch-Mechanismus sind AVR fast doppelt so schnell wie sie das ohne diesen wären. Vergleicht man verschiedene Prozessortypen bezüglich ihrer Ausführungsgeschwindigkeit, reicht es nicht aus, die MHz Takt zu vergleichen. Zu vergleichen ist, wieviel Taktzyklen je Instruktion anfallen und ob die Ausführung durch Pre-Fetch beschleunigt wird. So kann aus den gleichen MHz Takt das bis zu vierfache an Geschwindigkeit herauskommen. Fast alle Instruktionen des AVR-Prozessors benötigen einen Taktzyklus. Es gibt jedoch auch Instruktionen, die derer zwei benötigen.



Die beiden, die wir in der vorherigen Lektion verwendet haben, gehören zu dieser eher seltenen Spezies. Das kommt in diesem Fall daher, dass für das Setzen eines Bits erst der gesamte Port eingelesen, dann das erste Bit

gesetzt und der gesamte Port dann wieder geschrieben wird. Die Ausführungszeit beträgt mit Pre-Fetch zwei Taktzyklen.

3.1.2 Ausführungszeiten von Instruktionen

Die Ausführungszeiten aller Instruktionen stehen im Device-Handbuch in der Tabelle "Instruction Set Summary" in der Spalte "Clocks".

21. Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V	1

Wann immer es auf Ausführungszeiten und Timing ankommt, wie im Falle des Sekundenblinkers, ist das die erste Anlaufstelle. Auch in Anhang 2: Instruktionsliste AVR-Assembler kann man nachschlagen.

Top	Home	Einführung	Hardware	Schnellblinker	Sekundenblinker

3.2 Hardware, Bauteile und Aufbau

Für die Blinker kommt die gleiche Hardware zum Einsatz, wie wir sie schon in Lektion 2 aufgebaut haben.

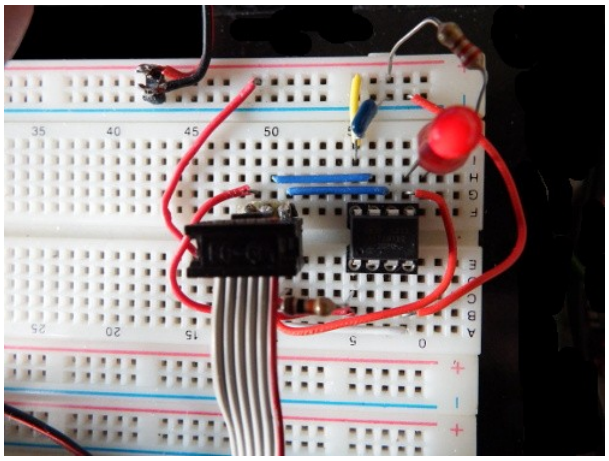
3.3 Der Schnellblinker

3.3.1 Der einfachste Schnellblinker

Wie eingangs beschrieben, bräuchten wir nur die Codezeilen

```
sbi DDRB,DDB0 ; PB0 ist Ausgang, Treiber einschalten cbi PORTB,PORTB0 ; LED einschalten
sbi PORTB,PORTB0 ; LED ausschalten
```

zum Ein- und Ausschalten der LED verwenden. Das Ergebnis wäre eine ganz schwach leuchtende LED.

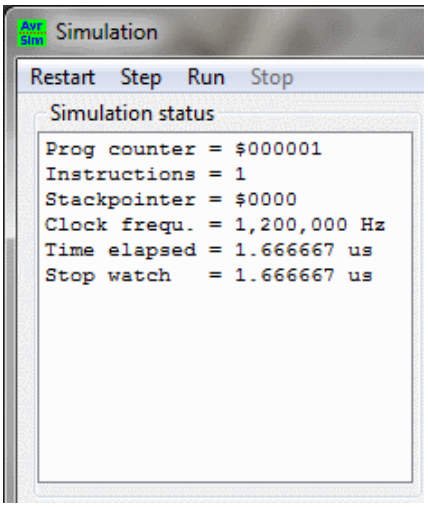


Die Ursache für diese Schwäche ist, dass die Diode nur für zwei Taktzyklen eingeschaltet ist, dann aber 511 Takte bei ausgeschalteter LED folgen. Der Prozessor arbeitet nämlich seinen ganzen Flashspeicher mit NOPs ab, bis er wieder von vorne beginnt.

Abstellen können wir dies, indem wir eine Sprunginstruktion einfügen, die nach dem Abschalten direkt wieder zum Anschalten verzweigt. Das geht so:

```
    sbi DDRB,DDB0 ; PB0 ist Ausgang, Treiber einschalten
Schleife:
    cbi PORTB,PORTB0 ; LED einschalten
    sbi PORTB,PORTB0 ; LED ausschalten
    rjmp Schleife ; zum Einschalten springen
```

Schleife: nennt man ein Label, es ist die Sprungsadresse für den Rücksprung. Der Rücksprung ist die Instruktion „RJMP", in Deutsch „Relativer Sprung".



Die Instruction Set Summary im Device Data Book bringt folgende Ausführungszeiten zutage:

```
    sbi DDRB,DDB0 ; PB0 ist Ausgang, Treiber einschalten
Schleife:
    cbi PORTB,PORTB0 ; LED einschalten, 2 Takte
    sbi PORTB,PORTB0 ; LED ausschalten, 2 Takte
    rjmp Schleife ; zum Einschalten springen, 2 Takte
```

Das bedeutet, die LED ist für zwei Takte angeschaltet und für vier Takte ausgeschaltet. Das bedeutet 33% Helligkeit:

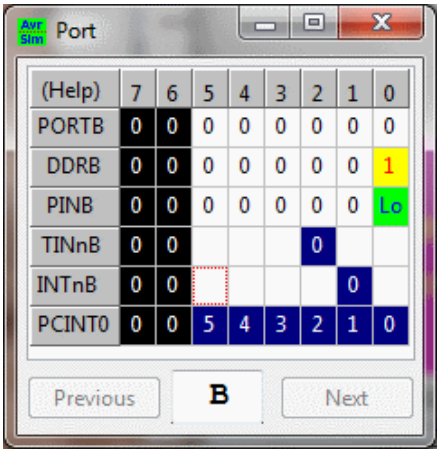
2 / (2 + 4) oder 1 / 3.

Für 50% Helligkeit wäre folgendes nötig (zum Quellcode im asm-Format):

```
    sbi DDRB,DDB0 ; PB0 ist Ausgang, Treiber einschalten
Schleife: cbi PORTB,PORTB0 ; LED einschalten, 2 Takte
    nop ; Nichts tun, 1 Takt
    nop ; Nichts tun, 1 Takt
    sbi PORTB,PORTB0 ; LED ausschalten, 2 Takte
    rjmp Schleife ; zum Einschalten springen, 2 Takte
```

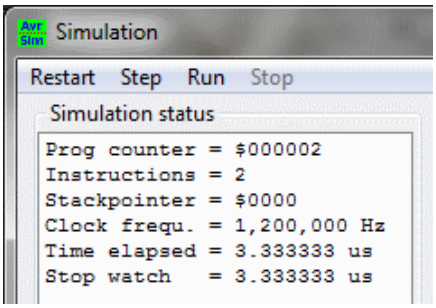
Jetzt ist die LED für vier Takte an und für vier Takte aus. Macht genau 50% duty cycle. Allerdings beträgt die Blinkfrequenz nunmehr immer noch 1.200.000 / 8 = 150.000 Hz. Also etwas zu viel für das menschliche Auge.

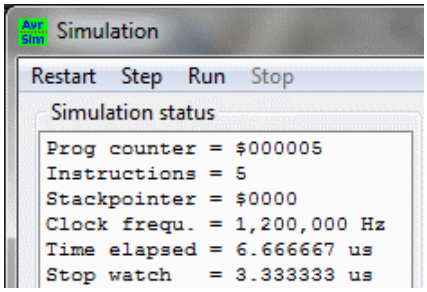
Bei der Simulation mit avr_sim können die Abläufe sicht- und messbar gemacht werden. Die erste Instruktion, *sbi DDRB,DDB0*, wurde ausgeführt. Sie dauert bei 1,2 MHz Takt genau 1,667 µs, wie es das Datenbuch von ATMEL ausweist.



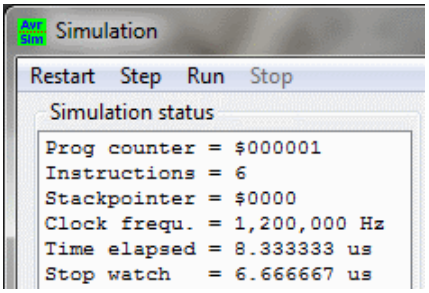
Bit 0 im Richtungsport des Prozessors ist danach gesetzt, der Pin ist nun Ausgang.

Die nächste Instruktion, *cbi PORTB, PORTB0,* wurde ausgeführt. Sie dauert ebensolange wie die Erste.





Und so lange dauert es von der *cbi PORTB,PORTB0*- bis zur *sbi PORTB,PORTB0*-Instruktion, mit den beiden NOP zwischendurch.



Und so lange dauert ein ganzer Durchlauf von Schleife bis Schleife: genau doppelt so lang. Die erzeugte Rechteckspannung am Ausgang PB0 ist also symmetrisch,

die LED zu 50% an und aus.

3.3.2 Verzögerter Schnellblinker, 8-Bit

Um die Blinkfrequenz zu senken, muss noch mehr Verzögerung in den Code. Die folgende Sequenz ist eine typische Verzögerungsschleife:

```
.equ cZaehler = 250 ; definiere eine Konstante
     ldi R16, cZaehler ; lade Register mit Konstante Schleife:
     dec R16 ; zaehle Register abwaerts brne Schleife ; verzweige, wenn Null-Flagge nicht gesetzt
```

Hier wird ein Register (R16) verwendet, um abwärts zu zählen. Jeder AVR hat 32 solcher Register, von denen sich aber nur R16 bis R31 in einer einzigen Instruktion mit einem Festwert beladen lassen. Register sind frei verfügbare Speicherstellen mit je 8 Bits. Sie können daher Werte zwischen 0 und 255 aufnehmen. Die Konstante cZaehler legt fest, wie oft die Schleife durchlaufen wird. „LDI" bedeutet Load Immediate, „DEC" Decrease oder vermindern, „BRNE" Branch if Not Equal (verzweige wenn nicht Null).

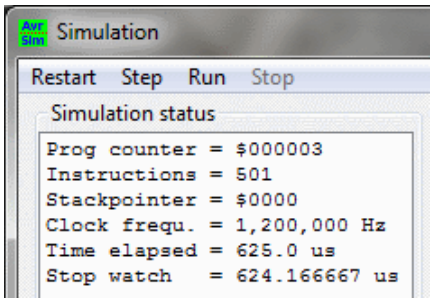
Wiederum fördert das Instruction Set Summary folgende Taktzyklen zutage:

```
.equ cZaehler = 250 ; (kein Takt)
     ldi R16, cZaehler ; 1 Takt
Schleife:
     dec R16 ; 1 Takt
     brne Schleife ; 2 Takte beim Verzweigen, 1 Takt ohne Verzweigung
```

Die einzelnen Instruktionen nehmen folgende Takte in Anspruch:

```
.equ cZaehler = 250 ; (erzeugt keinen Code)
     ldi R16, cZaehler ; 1 Takt
Schleife:
     dec R16 ; cZaehler Takte
     brne Schleife ; 2 * (cZaehler – 1) Takte plus 1 Takt
```

Der gesamte Durchlauf nimmt also 1 + cZaehler + 2 * cZaehler – 2 + 1 Takte in Anspruch. Das ergibt 3 * cZaehler oder 750 Takte. Zeitlich dauert die Schleife 750 / 1.200.000 oder 625 µs.



Auch das wieder im avr_sim-Simulator. Mit

```
Loop:
     ldi R16,250
Loop2:
     dec R16
     brne Loop2
     rjmp Loop
```

ergeben sich diese Ausführungszeiten. Der Durchlauf dauert also wie voherberechnet 625 µs, der Schleifendurchlauf ohne *ldi R16,250* ein Takt weniger. Eine gut vorherberechenbare Sache also.

3.3.3 Verzögerter Schnellblinker, 16 Bit

Das ist immer noch wesentlich zu kurz. Versuchen wir es also mit 16 Bit. Wir benötigen dafür 16-Bit-Register, der AVR hat davon vier Stück. Das sind die Registerpaare R25:R24, R27:R26, R29:R28 und R31:R30. Sie lassen sich wie Einzelregister behandeln, manche Instruktionen wirken sich aber jeweils auf das ganze Paar aus. Die 16-Bit-Schleife sieht so aus:

```
.equ cZaehler = 50000 ; 1 bis 65535 (erzeugt keinen Code) ldi R25, HIGH(cZaehler) ; 1 Takt
     ldi R24,LOW(cZaehler) ; 1 Takt
Schleife:
     sbiw R24,1 ; abwaerts, 2 Takte, 2 * cZaehler
     brne Schleife ; 2 * (cZaehler – 1) Takte plus 1 Takt
```

Das „SBIW" bewirkt, dass vom Registerpaar R25:R24 immer eins abgezogen wird, und zwar wortweise (16-Bit). Kommt beim Abziehen Null heraus, wird die Z-Flagge im Statusregister gesetzt, ansonsten ist sie Null. Die Instruktion „BRNE" wertet diese Flagge aus, um zur Schleife

zurückzukehren, solange diese noch nicht Null ist.

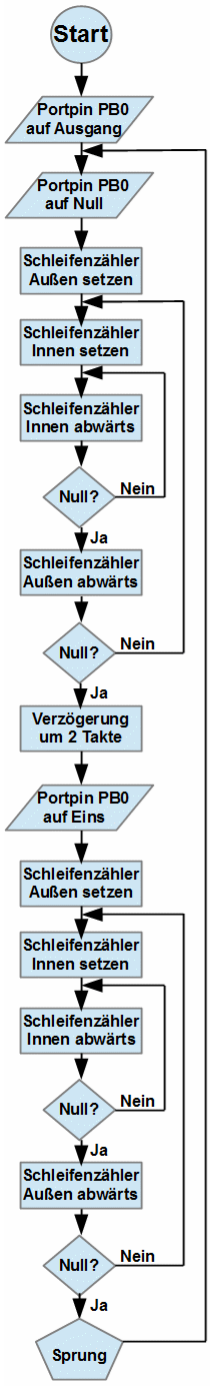
Nun nimmt der Durchlauf 1 + 1 + 2 * cZaehler + 2 * (cZaehler - 1) + 1 = 4 * cZaehler + 1 Takte ein. Bei 50.000 Durchläufen sind das 200.001 Takte oder 0,167 Sekunden. Das kommt der Sekunde schon näher, aber noch nicht ganz.

3.4 Sekundenblinker

Die Sekunde kriegen wir daher nur mit einer Kombination einer 8-Bit- mit einer 16-Bit-Verzögerungsschleife hin. Und das geht so (zum Quellcode im asm-Format):

```
; ***********************************************
; * Eine LED blinken lassen mit einem Attiny13  *
; * (C)2016 by http://www.gsc-elektronic.net    *
; ***********************************************
;
.NOLIST ; Ausgabe im Listing unterdruecken
.INCLUDE „tn13def.inc“ ; Port-Definitionen lesen
.LIST ; Ausgabe im Listing einschalten
;
; Register definieren
;
.def ZaehlerA = R16 ; Aeusserer 8-Bit-Zaehler
.def ZaehlerIL = R24 ; Innerer 16-Bit-Zaehler, LSB
.def ZaehlerIH = R25 ; Innerer 16-Bit-Zaehler, MSB
;
; Konstanten definieren
;
.equ cInnen = 2458 ; Zaehler Innere Schleife
.equ cAussen = 61 ; Zaehler Aeussere Schleife
;
; Programmstart
;
    sbi DDRB,DDB0 ; Portpin PB0 auf Ausgang
;
; Programmschleife
;
Schleife:
    cbi PORTB,PORTB0 ; Portpin PB0 auf Null, Led an, 2 Takte
; Verzoegerungsschleifen
    ldi ZaehlerA,cAussen ; Aeusserer 8-Bit-Zaehler setzen, 1 Takt Schleife1:
    ldi ZaehlerIH,HIGH(cInnen) ; Innerer 16-Bit-Zaehler setzen, 1 Takt
    ldi ZaehlerIL,LOW(cInnen) ; 1 Takt
Schleife1i:
    sbiw ZaehlerIL,1 ; Inneren 16-Bit-Zaehler abwaerts zaehlen, 2 Takte
    brne Schleife1i ; wenn noch nicht Null: zurueck, 2 Takte bei Sprung, 1 Takt ohne Sprung
    dec ZaehlerA ; aeusseren 8-Bit-Zaehler abwaerts, 1 Takt
    brne Schleife1 ; aeussere 8-Bit-Schleife wiederholen, 2 Takte bei Sprung, 1 Takt ohne Sprung
    nop ; Verzoegerung, 1 Takt
    nop ; Verzoegerung, 1 Takt
    sbi PORTB,PORTB0 ; Portpin PB0 auf Eins, Led aus, 2 Takte
; Verzoegerungsschleifen
    ldi ZaehlerA,cAussen ; aeusserer 8-Bit-Zaehler setzen, 1 Takt Schleife2:
    ldi ZaehlerIH,HIGH(cInnen) ; Innerer 16-Bit-Zaehler setzen, 1 Takt
    ldi ZaehlerIL,LOW(cInnen) ; 1 Takt
Schleife2i:
    sbiw ZaehlerIL,1 ; Inneren 16-Bit-Zaehler abwaerts zaehlen, 2 Takte
    brne Schleife2i ; wenn noch nicht Null: zurueck, 2 Takte bei Sprung, 1 Takt ohne Sprung
    dec ZaehlerA ; aeusseren 8-Bit-Zaehler abwaerts, 1 Takt
    brne Schleife2 ; aeussere 8-Bit-Schleife wiederholen, 2 Takte bei Sprung, 1 Takt ohne Sprung
; Zyklusende
    rjmp Schleife ; von vorne beginnen, 2 Takte
;
; Ende Quellcode
;
```

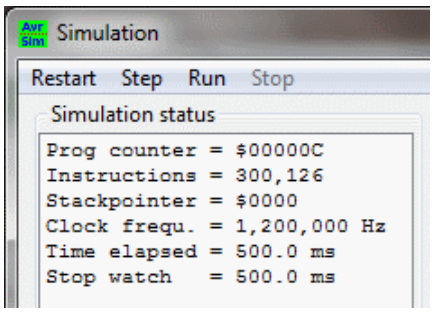
Das ist der Ablauf im Detail. Innere 16-Bit- und äussere 8-Bit-Schleife sind zweifach identisch vorhanden.



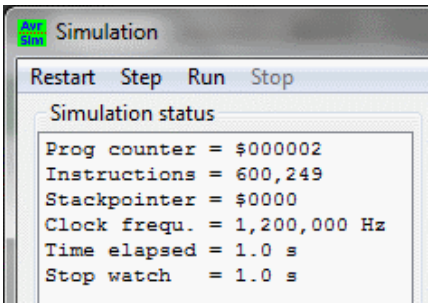
Instruktionen	Takte	Anzahl Ausführungen pro Schleifen-Durchlauf	Gesamttakte
SBI DDRB,DDB0	2	(außerhalb Schleife)	
CBI PORTB,PORTB0	2	1	2
LDI rAussen,cAussen	1	1	1
LDI rInnenH,HIGH(cInnen) LDI rInnenL,LOW(cInnen)	1 1	cAussen cAussen	2*cAussen
SBIW rInnenL,1	2	cAussen * cInnen	2*cAussen*cInnen
BRNE Innen1i	2 (Sprung) 1 (Nicht-Sprung)	cAussen * (cInnen - 1) cAussen	2*cAussen*cInnen -cAussen
DEC rAussen	1	cAussen	cAussen
BRNE Innen1	2 (Sprung) 1 (Nicht-Sprung)	cAussen – 1 1	2*cAussen-1
NOP NOP	1 1	1 1	2
SBI PORTB,PORTB0	2	1	2
LDI rAussen,cAussen	1	1	1
LDI rInnenH,HIGH(cInnen) LDI rInnenL,LOW(cInnen)	1 1	cAussen cAussen	2*cAussen
SBIW rInnenL,1	2	cAussen * cInnen	2*cAussen*cInnen
BRNE Innen2i	2 (Sprung) 1 (Nicht-Sprung)	cAussen * (cInnen - 1) cAussen	2*cAussen*cInnen -cAussen
DEC rAussen	1	cAussen	cAussen
BRNE Innen2	2 (Sprung) 1 (Nicht-Sprung)	cAussen – 1 1	2*cAussen-1
RJMP Schleife	2	1	2

Gesamttakte = 8 * (cAussen*cInnen + cAussen + 1)

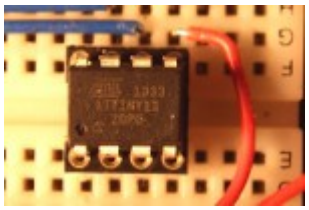
Die sich ergebende Formel für die Anzahl Takte ist relativ einfach, die Ermittlung der beiden Konstanten cAussen und cInnen für ein optimales Ergebnis ist weniger trivial. Die im Quellcode verwendeten Konstanten sind einzigartige Lösungen für diese Aufgabe.



Der Simulator avr_sim sagt auch, dass nach der halben und der ganzen Sekunde alles korrekt ist.



Top	Home	Einführung	Hardware	Schnellblinker	Sekundenblinker



Lektion 4: Eine LED blinkt mit dem Timer

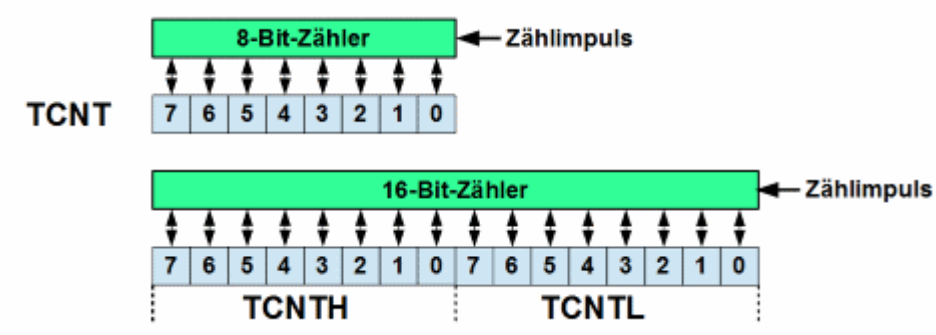
Mit dieser Lektion beenden wir die elendig langweiligen Schleifen und lassen die interne Timer-Hardware die langwierige Timing-Aufgabe vollautomatisch und exakt verrichten.

4.0 Übersicht

4.1 Einführung in die Timer-Hardware

Der im Prozessor eingebaute Zähler (Timer/Counter, TC) ist die am häufigsten benutzte Hardwarekomponente. Der Timer wird auch bei späteren Lektionen zur Anwendung kommen, deshalb hier schon mal ein paar Anwendungen, um die LED auf verschiedene Arten blinken zu lassen. Unterschiedliche AVR-Typen haben eine unterschiedliche Anzahl Zähler eingebaut. Diese sind mit Null beginnend durchnummeriert. Bei der Benennung von Timer-Ports beziehen sich 0, 1 oder 2 immer auf die Zählernummer. Der ATtiny13 hat nur einen Zähler, dennoch ist bei allen Portbenennungen eine Null eingefügt.

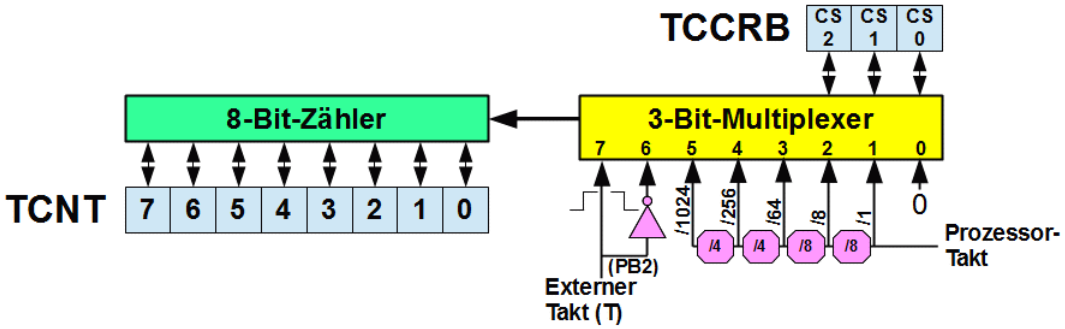
4.1.1 Timer



Timer in den AVR sind 8 oder 16 Bit breite Zähleinrichtungen. Sie zählen in der Regel aufwärts, bei 8 Bit Breite von 0 bis 255, bei 16 Bit Breite von 0 bis 65.535 und starten danach einfach wieder bei Null. Ihr Zählerstand ist im Port TCNT (8 Bit) bzw. in den beiden Ports TCNTH und TCNTL (16 Bit) zugänglich. Wird der Zählerport beschrieben, wird der Zähler auf diesen Stand gesetzt.

4.1.2 Timer-Impulsquellen

Die Möglichkeiten, mit denen der Zähler mit Impulsen versorgt werden kann, sind vielfältig. Mit dem Kontrollport B des 8-Bit-Timers, TCCRB, wird die Taktquelle des Timers ausgewählt. Multiplexer bedeutet „adressgesteuerter Umschalter". Drei Bits im Kontrollregister wählen damit die Signalquelle aus.



Acht verschiedene Quellen stehen zur Auswahl:

#	Bin	Modus	Taktung
0	000	Aus	Kein Signal, Zählen abgeschaltet
1	001	Timer	Prozessortakt
2	010	Timer	Prozessortakt durch 8
3	011	Timer	Prozessortakt durch 64
4	100	Timer	Prozessortakt durch 512
5	101	Timer	Prozessortakt durch 1.024
6	110	Counter	externer Anschluss T, fallende Flanken
7	111	Counter	externer Anschluss T, steigende Flanken

Mit den Zeilen

```
ldi R16,1 ; R16 auf 1
out TCCR0B,R16 ; in Kontrollport B
```

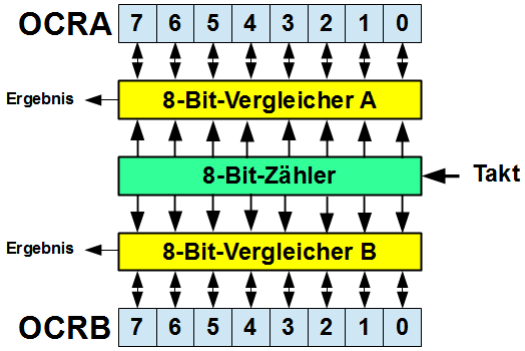
lässt sich der Timer mit dem Prozessortakt starten, mit

```
clr R16 ; R16 auf Null
out TCCR0B,R16 ; in Kontrollport B
```

wird der Timer wieder gestoppt. CLR setzt ein Register auf Null. Mit der Instruktion "OUT" werden alle acht Bits des angegebenen Portregisters mit dem Inhalt des angegeben Registers beschrieben.

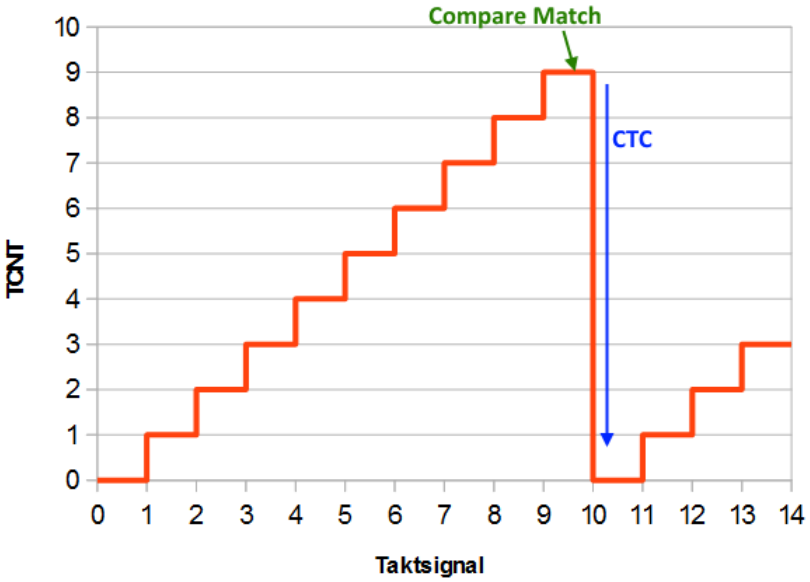
4.1.3 Timer und Compare-Match

Die ganze Timerei wäre nicht viel wert, wenn wir nun ständig den Timer auf seinen Zählerstand hin überwachen müssten, um einen Ablauf mit bestimmter Dauer zu erreichen. Für diese Aufgabe wurden Vergleicher eingebaut.



In der Regel haben alle Timer zwei Vergleicher, A und B. Mit dem Eintreten des Compare Match Ereignisses können automatische Vorgänge ausgelöst werden oder der Prozessor kann unterbrochen werden. Weiteres zum automatischen Unterbrechen siehe spätere Lektionen.

4.1.4 Timer im CTC-Modus



Die naheliegendste Anwendung der Vergleicher ist es, den Zähler bei Erreichen des Compare-Wertes auf Null zu setzen und von vorn beginnen zu lassen. Dies wird als CTC bezeichnet ("Clear timer on compare"). Man beachte, dass der Compare-Match nicht bei sofortiger Gleichheit rücksetzt, sondern erst mit Eintreten des nächsten Zählimpulses. Das garantiert eine exakt definierte Länge von Impulsen unabhängig von der physischen Vergleichsgeschwindigkeit. Im vorliegenden Fall wird das Vergleichsregister auf Neun eingestellt, womit 10 Zählimpulse Dauer zum CTC führen (Impulse 0 bis 9 = 10 Impulse). CTC ist bei allen Timern mit dem Compare-Register A möglich. Compare-Register B kann für andere Zwecke verwendet werden. Manche AVRs verfügen auch noch über ein weiteres Register, mit dem ein weiterer CTC-Mode möglich ist und die Vergleicher A und B für andere Aufgaben frei sind.

4.1.5 Ausgänge mit dem Timer manipulieren

Die zweithäufigste Verwendung von Compare-Match ist das Setzen oder Löschen von Portpins. Damit macht sich der Match außerhalb bemerkbar und es können elektrische Impulse nahezu beliebiger Dauer an Ausgangspins erzeugt werden. Für den ATtiny13 ist PB0 für den Vergleicher A festgelegt (Ausgang OC0A), für B PB1 (OC0B). Zur Verwendung genügt es, den entsprechenden Portpin als Ausgang zu definieren (seinen Ausgangstreiber einzuschalten, z.B. mit sbi DDRB,DDB0) und mit zwei Kontrollbits das Verhalten dieses Pins festzulegen. Vier Modi stehen zur Verfügung:

COM0A1 COM0B1	COM0AO COM0B0	Funktion OC0A/OC0B
0	0	Portpin wird nicht beeinflusst
0	1	Portpin wechselt Polarität bei Compare Match (Torkeln, Toggle)
1	0	Portpin bei Compare Match auf Null setzen
1	1	Portpin bei Compare Match auf Eins setzen

Zur Beachtung: das gilt nur für den Fall, dass der Zähler im normalen Zählmodus betrieben wird. Für andere Modi passiert mit diesen Bits was anderes. Damit stehen für die Blinkerei ganz neue Möglichkeiten zur Verfügung.

Top	Home	Einführung	Hardware	Standard	Takt	CTC	128kHz

4.2 Hardware, Bauteile und Aufbau

Für die Blinker kommt die gleiche Hardware zum Einsatz, wie wir sie schon in Lektion 2 aufgebaut haben.

4.3 Timer mit Standardeinstellungen

4.3.1 Funktionsweise

Die einfachste Art, den Blinker zu bauen, ist den Timer mit dem größtmöglichen Vorteiler bis 255 zählen zu lassen und bei einem Compare-Match den Ausgang torkeln zu lassen. Damit stellt sich eine Blinkfrequenz von

1.200.000 / 1.024 / 256 / 2 = 2,29 Hz

ein. Das ist schneller als es soll und wäre für eine Uhr ziemlich daneben.

4.3.2 Programm

Hier ist das Programm (zum Quellcode im asm-Format). Es schaltet zuerst den Portpin PB0 als Ausgang, setzt dann das Compare A Register, versetzt den Ausgang OC0A in den Toggle-Modus und startet den Timer mit dem durch 1024 geteilten Takt.

```
;
; ************************************
; * Timer zum Blinken                *
; * (C)2016 by www.gsc-elektronic.net *
; ************************************
;
.NOLIST
.INCLUDE „tn13def.inc“
.LIST
;
; ---------- Register -----------------
.def rmp = R16 ; Vielzweckregister
;
; ---------- Timing -------------------
; Interner RC-Oszillator = 9.600.000 Hz
; Clock-Vorteiler CLKPR  =         8
; Interner Prozessortakt = 1.200.000 Hz
; TC0-Vorteiler          =     1.024
; TC0-Tick               =     1.171,875 Hz
; TC0-Durchlauf          =       256
; TC0-Durchlauffrequenz  =         4,578 Hz
; TC0-Toggle-Frequenz    =         2,289 Hz
;
; ---------- Start --------------------
; PB0 als Ausgang
    sbi DDRB,DDB0 ; PB0 als Ausgang
; setze Compare Match
    ldi rmp,0xFF ; Match bei 255
    out OCR0A,rmp ; in Compare Match A
; toggle PB0 bei Compare Match
    ldi rmp,1<<COM0A0 ; Toggle Mode
    out TCCR0A,rmp ; in Kontrollregister A
; Timer starten
    ldi rmp,(1<<CS02)|(1<<CS00) ; Prescaler 1024
    out TCCR0B,rmp ; in Kontrollregister B
Schleife:
    rjmp Schleife
```

Nachdem die Hardware gestartet ist, wird der Prozessor nicht mehr benötigt und geht in eine unendliche Schleife (Schleife: rjmp Schleife).

Bit	7	6	5	4	3	2	1	0	
	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Eine Besonderheit ist noch zu klären, die Formulierung „ldi rmp, 1<<COM0A0“. Damit wird das Portbit COM0A0 im Port TCCR0A auf Eins gesetzt. COM0A0 ist das Bit 6 in diesem Port. Eigentlich wäre zu schreiben: „ldi rmp,0b01000000“ (0b startet eine Binärzahl). Spätestens eine Stunde nachdem wir das geschrieben haben, will uns nicht mehr einfallen, was „01000000“ nun eigentlich heißt (dass es sich um das Bit COM0A0 handelt). Deshalb die Formulierung „ldi rmp,1<<COM0A0“. Sie bedeutet

- Nimm eine Eins, also 0b00000001,
- schiebe diese COM0A0-mal nach links (<<),
- weil COM0A0 in der Datei "tn13def.inc" mit ".equ COM0A0 = 6" definiert ist, wird sechs mal nach links geschoben (<< bedeutet Linksschieben) und dabei von rechts eine binäre Null hineingeschoben,
- also wird aus 0b00000001 schließlich 0b01000000.

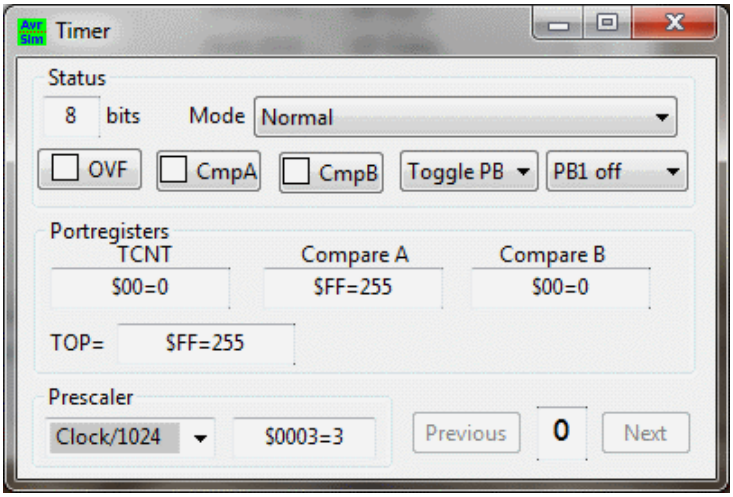
Es ist wichtig zu verstehen, dass die Schieberei hierbei nur vom Assembler beim Übersetzen des Programmcodes vollzogen wird. << hat mit der Schiebeinstruktion des Prozessors nichts zu tun. Die gibt es nämlich auch, sie heißt "LSL" (Logical Shift Left) und braucht als Parameter ein Register, das nach Links geschoben werden soll. In der Zeile "ldi rmp,(1<<CS02)|(1<<CS00)" werden gleichzeitig zwei Bits auf Eins gesetzt. Weil CS02 in der def.inc mit 2 definiert ist, CS00 aber mit Null, kommt bei den beiden Klammerausdrücken einerseits 0b00000100 (eine 1 zweimal links geschoben) und andererseits 0b00000001 (eine 1 keinmal links geschoben) heraus. Das Zeichen | sagt dem Assembler, die beiden Ergebnisse binär mit ODER zu verknüpfen, woraus dann 0b00000101 wird. Auch hier hat das nichts mit dem Prozessor zu tun, dessen ODER-Verknüpfung zweier Register heißt "OR". Das | macht allein der Assembler beim Übersetzen in Maschinencode.

Bit	7	6	5	4	3	2	1	0
TCCR0B	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Die Zeichen << und | sind mathematische Operatoren des Assemblers, wovon es noch viele weitere gibt. Wie z. B. +, -, * und /. Alles nur Mitteilungen an den Assembler, der Prozessor kriegt davon nichts mit.

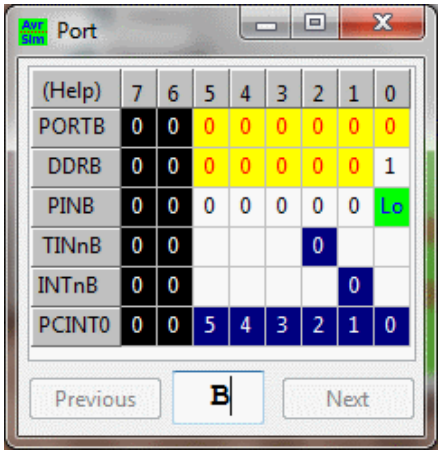
4.3.3 Simulation des Timerbetriebs in diesem Modus

Die nachfolgende Simulation verwendet den Simulator avr_sim, um den Timer in diesem Modus nachzuahmen.



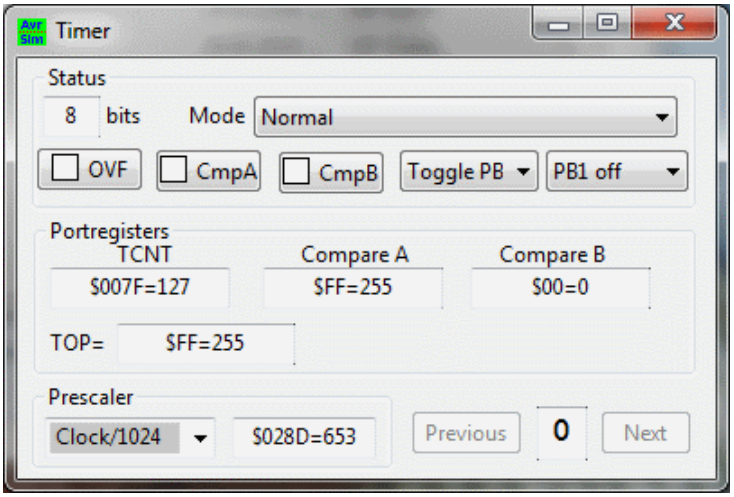
Nach dem Initiieren ist der Timer TC0 in diesem Zustand:

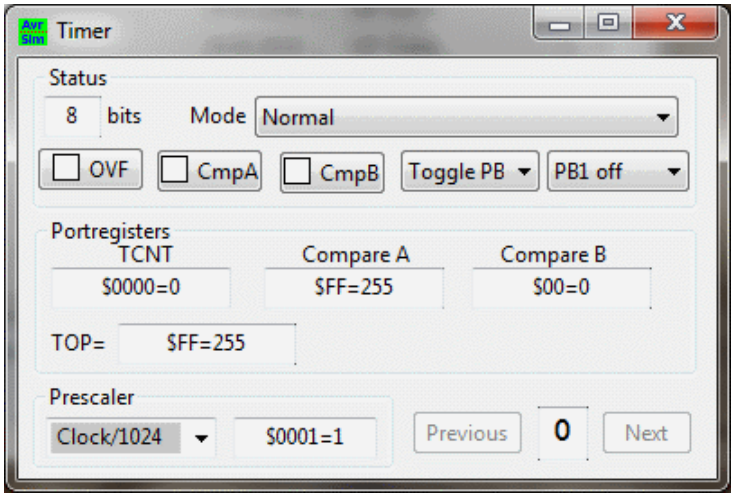
- Der Timer-Modus ist „Normales Aufwärtszählen“ („normal counting“).
- Der Ausgangspin PB0 torkelt („toggles“) wenn der Timer den im Vergleichsregister A eingestellten Wert erreicht und der nächste Zählimpuls eintritt.
- Vergleichswert A („Compare A“) und der Höchstwert („TOP“) sind auf 255 gesetzt.
- Der Vorteiler (prescaler) teilt die Taktfrequenz des Prozessors durch 1.024, der Timer läuft also mit 1.171,875 Hz Taktfrequenz.



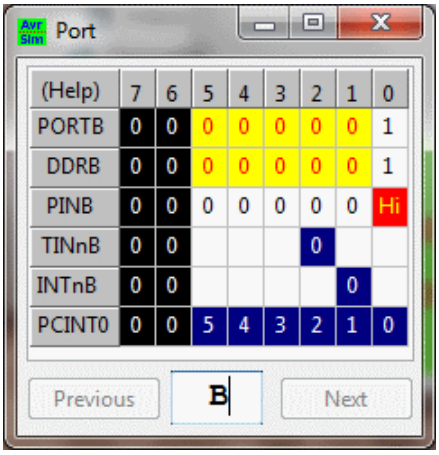
Nach der Initiierung ist das Richtungsbit DDB0 des Ports B Eins. Das bedeutet, dass der Ausgabepin PB0 dem Zustand des Portregisters PORTB0 folgt, auf niedriges Potential geht und die LED, die mit der Anode an der positiven Betriebsspannung liegt, eingeschaltet ist.

Nun zählt der Timer. Jeder Einzelschritt erhöht den Vorteilerwert. Erreicht der Vorteiler den Wert 1.024 wird das Portregister TCNT um Eins erhöht und der Vorteiler startet bei Null neu.

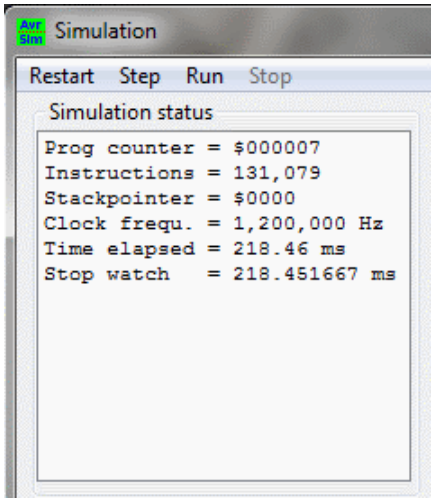




Nach 256 * 1.024 = 262.144 Taktzyklen erreicht der Timer den Vergleichswert im Portregister „Compare A" und, wenn der nächste Taktimpuls eintritt, tritt ein Compare Match ein. Da der Timer damit auch seinen Höchstwert („TOP") überschreitet, beginnt das Zählregister TCNT dann wieder bei Null (Zählerüberlauf).



Das Überschreiten des Compare-Match-A-Wertes lässt auch das Portbit PORTB0 torkeln, der Ausgangspin PB0 wird jetzt high und die LED wird ausgeschaltet. Diese Umschaltung erfolgt durch den Timer automatisch, dazu ist keinerlei weitere Prozessoraktivität nötig.



Der Simulator hat zu diesem Zeitpunkt 131.079 Instruktionen ausgeführt, die meisten davon relative Sprünge *RJMP*, die zu ihrer Ausführung zwei Taktzyklen brauchen. 218,46 ms sind seit dem Start vergangen. Die Stoppuhr („Stop watch"), die nach der Initiierung zurückgesetzt worden ist, zeigt 218,452 ms an, was mit unseren vorberechneten 4,58 Timerzyklen pro Sekunde und einer Blinkfrequenz von 2,29 Hz übereinstimmt.

4.3.4 Vor- und Nachteile

Der Vorteil dieser Lösung im Vergleich zur Verzögerungsschleifen-Lösung in Lektion 3 ist, dass der Prozessor nicht mit Unsinn wie dem Abzählen von Schleifen beschäftigt wird. Der Zählvorgang läuft im Timer ab, der Schaltvorgang am Portpin wird vom Vergleicher erledigt. Der Prozessor könnte sich um andere Aufgaben kümmern, ohne dass die Zählerei und Schalterei irgendwie gestört würde.

Der Nachteil dieser Lösung ist, dass es mit der Sekundenblinkerei nicht hinhaut, das Blinken ist doppelt schneller. Ursache ist, dass das Teilen des Prozessortakts durch 1.024 und durch 256 immer krumme Werte liefert (es sei denn, der Prozessor liefe mit 2,4576 MHz Takt, wofür wir einen externen Taktgeber bräuchten) und weil der Takt 1,2 MHz zu schnell ist und die Vorteilerei zu groß für eine Sekunde ist. Daher müssen andere Lösungen her, bei denen der Prozessortakt entscheidend verlangsamt wird.

4.4 Prozessortakt verlangsamen

4.4.1 Prozessortakt-Vorteiler CLKPR manipulieren

Alle Tiny- und Mega-Typen des AVR neueren Datums haben eine Einrichtung, die das Teilen des Taktes interner und externer Oszillatoren ermöglichen, bevor diese auf den Prozessor und alle andere interne Hardware losgelassen werden. Als Teilerfaktoren stehen Zweierpotenzen von 1 bis 256 zur Verfügung. Bei 9,6 MHz internem RC-Oszillatortakt entspricht das 9,6 MHz bis 37,5 kHz Prozessortakt.

Table 6-8. Clock Prescaler Select

CLKPS3	CLKPS2	CLKPS1	CLKPS0	Clock Division Factor
0	0	0	0	1
0	0	0	1	2
0	0	1	0	4
0	0	1	1	8
0	1	0	0	16
0	1	0	1	32
0	1	1	0	64
0	1	1	1	128
1	0	0	0	256

Ohne es zu wissen, haben wir diesen Vorteiler bereits praktisch verwendet. Er ist nämlich werksseitig auf das Teilen durch Acht eingestellt. Das können wir durch Löschen der CKDIV8-Fuse im Prozessor aufheben (siehe Lektion 1), wobei der Prozessor dann mit 9,6 MHz Takt läuft (Achtung! Bei niedrigen Betriebsspannungen schafft der langsamere Typ ATtiny13V diesen Takt nicht! Die Fuse kann dann nur mit erhöhter Betriebsspannung wieder gesetzt werden!). Um nicht nur 1 oder 8 als Teilerfaktoren verwenden zu können, muss auf den Port CLKPR zugegriffen werden. Damit das nicht aus Versehen passiert, gibt es eine bestimmte Prozedur, um das zu erledigen. Zuerst muss das Bit CLKPCE im Port CLKPR auf Eins gesetzt werden (alle anderen Bits MÜSSEN beim Schreiben Null sein). Sofort danach werden die CLKPS-Bits gesetzt (mit CLKPCE auf Null). Kurze Zeit später ist das Schreibzugriffsfenster wieder zu.

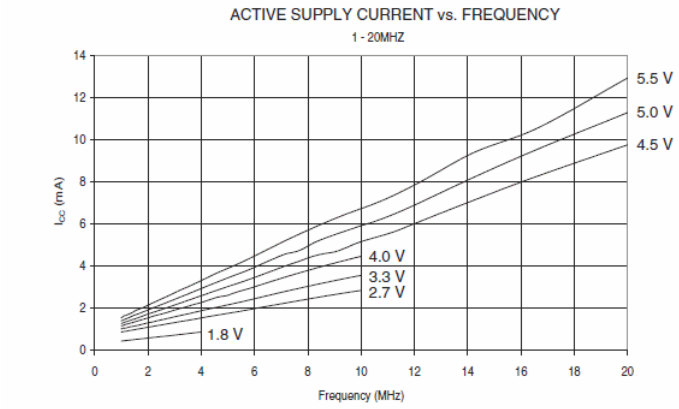
Bit	7	6	5	4	3	2	1	0	
	CLKPCE	–	–	–	CLKPS3	CLKPS2	CLKPS1	CLKPS0	CLKPR
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	See Bit Description				

Das Setzen von 32 als Vorteiler erledigt man mit folgendem Programmcode:

```
ldi R16,1<<CLKPCE ; Program Enable Bit auf Eins
out CLKPR,R16 ; in Port CLKPR schreiben
ldi R16,(1<<CLKPS2)|(1<<CLKPS0) ; Teiler = 32
out CLKPR,R16 ; in Port CLKPR schreiben
```

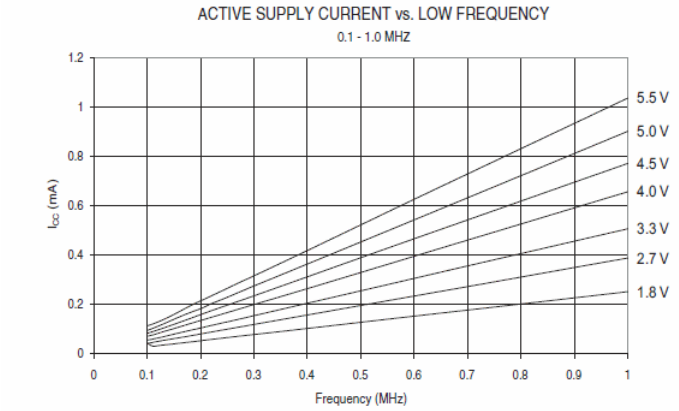
Nun läuft der Prozessor mit 9,6 Mhz / 32 = 300 kHz Takt. Aber Obacht! Beim Programmieren mittels ISP muss jetzt auch der Takt niedriger (< 75 kHz) angesetzt werden, sonst meldet der Programmieradapter einen Fehler.

Figure 19-2. Active Supply Current vs. Frequency (1 - 20 MHz)



Auf diese Weise wird der Prozessor zur lahmen Ente gemacht. Das macht man aber nur, wenn es denn nötig ist, wenn man z. B. Strom sparen will. Der Strombedarf ist nämlich etwa linear mit der Taktfrequenz, wie die nebenstehenden Auszüge aus dem Device Databook zeigen.

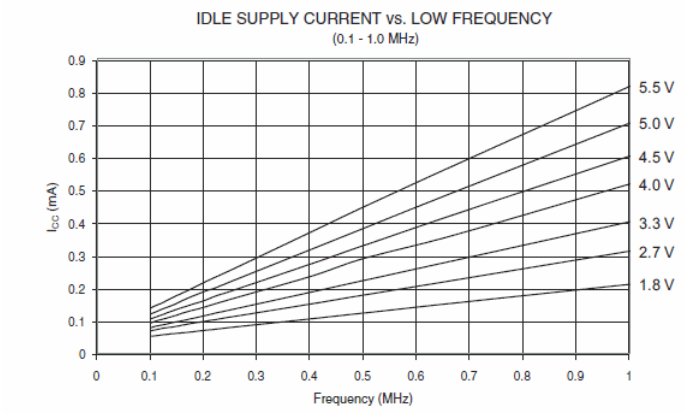
Figure 19-1. Active Supply Current vs. Frequency (0.1 - 1.0 MHz)



Mit der Herabsetzung der Taktfrequenz auf 300 kHz hat sich der Strombedarf des Prozessors von 1 bis 2 mA auf ganze 0,3 mA erniedrigt. Das tut einer Batterie oder einem Akku ganz gut und verfünffacht seine Haltbarkeit (als wenn es sonst nichts anderes anzutreiben gäbe).

4.4.2 Prozessor schlafen legen

Figure 19-7. Idle Supply Current vs. Frequency (0.1 - 1.0 MHz)



Eine weitere wirksame Methode zum Stromsparen ist es, den Prozessor nicht in "Schleife: rjmp Schleife" zu schicken, wenn er nicht gebraucht wird, sondern ihn ganz schlafen zu legen. Dann stellt er das Lesen aus dem Flashspeicher, das Dekodieren und Ausführen von Instruktionen ganz ein. Das nennt sich Idle und bedeutet so viel wie Leerlauf.

Auch dieser Zustand muss erst eingeschaltet werden. Er lässt sich einschalten, indem das Sleep-Enable-Bit im Port MCUCR (Micro controler universal control register) gesetzt wird und die SLEEP-Instruktion ausgeführt wird. Also etwa so:

```
ldi R16,1<<SE ; Schlafen ermöglichen
out MCUCR,R16 ; in Kontrollregister
sleep ; schlafen legen
```

In unserem Fall weckt nichts mehr den Prozessor auf, nur das Programmiergerät mit einem Reset. Es gibt aber auch andere Weckmechanismen, die kommen aber erst später zum Einsatz.

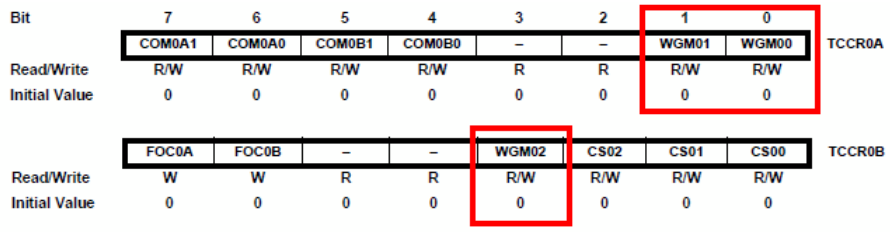
4.5 Timer im CTC-Modus

Table 11-8. Waveform Generation Mode Bit Description

Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on(1)(2)
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM (Phase Correct)	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	TOP	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM (Phase Correct)	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	TOP	TOP

Notes: 1. MAX = 0xFF
2. BOTTOM = 0x00

Der Vergleicher A im Timer kann noch für eine weitere Betriebsart des Timers verwendet werden, nämlich den CTC-Modus. In unserem Fall kann man diese Betriebsart verwenden, um krumme, d. h. nicht-zweierpotenzige, aber ganzzahlige Teilerverhältnisse zu erreichen. Damit kommen wir näher an die Sekunde heran als im vorherigen Beispiel. Das hier sind alle Betriebsmodi des Timers. Der CTC-Modus ist einstellbar, indem das WGM1-Bit auf 1 gesetzt wird. Der höchste Wert des Timers (TOP) ist die im Port OCRA gespeicherte Zahl. Danach setzt der Timer wieder zurück.



Die drei WGM-Bits WGM02, WGM01 und WGM00 des Timers 0 sind - unlogischerweise - in den zwei Ports TCCR0A und TCCR0B verteilt. Uns ficht das nicht an, da wir nur WGM01 auf Eins setzen müssen. Da dazu auch noch das Bit COM0A0 im TCCR0A zu setzen ist, kommt wieder das (1<<WGM01) | (1<<COM0A0) zum Einsatz.

Top	Home	Einführung	Hardware	Standard	Takt	CTC	128kHz

4.5.1 Programm

Das riesige Programm ist hier gelistet (zum Quellcode im asm-Format). Zuerst wird die Taktfrequenz des Prozessors verstellt, dann das Portbit PB0 als Ausgang geschaltet, dann der Timer programmiert und zuletzt der Prozessor schlafen gelegt. Für den Fall, dass er aufwachen sollte, ist noch das erneute Schlafenlegen programmiert.

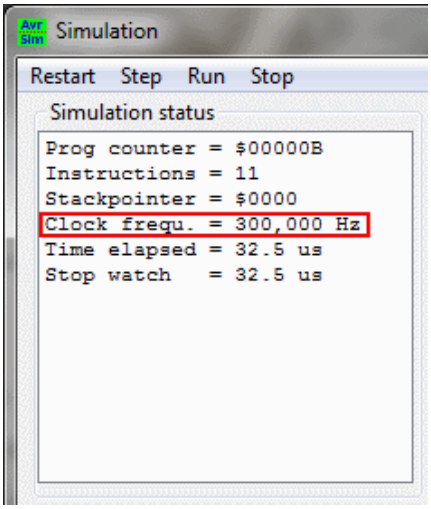
```
;
; ************************************
; * LED-Blinker mit Timer            *
; * (C)2016 by www.gsc-elektronic.net *
; ************************************
;
.NOLIST
.INCLUDE „tn13def.inc“
.NOLIST
;
; ------ Register ---------------------
.def rmp = R16 ; Vielzweckregister
;
; ------ Timing -----------------------
; Interner RC-Oszillator = 9.600.000 Hz
; Vorteiler CLKPR        =        32
; Taktfrequenz Prozessor =   300.000 Hz
; Vorteiler TC0          =     1.024
; Timer-Tick TC0         =       292,97 Hz
; Timer-CTC-Teiler       =       146
; Taktfrequenz CTC-OC0A  =         2,006 Hz
; Frequenz CTC-Toggle    =         1,003 Hz
;
; ------ Konstanten -------------------
.equ fRC = 9600000 ; Prozessortakt
.equ pClk = 32 ; Clock-Prescaler
.equ pTC0 = 1024 ; TC0-Prescaler
.equ pCtc = fRC / pClk / pTC0 / 2 ; Teiler
.equ cCtc = pCtc - 1 ; CTC-Wert
;
; ------ Programmstart ----------------
Start:
; Die Taktfrequenz des Prozessors
; herabsetzen
    ldi rmp,1<<CLKPCE ; CLKPR-Schreiben
       ; ermoeglichen
    out CLKPR,rmp ; an CLKPR-Port
    ldi rmp,(1<<CLKPS2)|(1<<CLKPS0) ; Teiler = 32
    out CLKPR,rmp ; an CLKPR-Port6
    ; PB0 als Ausgang
    sbi DDRB,DDB0 ; Datenrichtung Ausgang
    ; Den Timer programmieren und starten
    ldi rmp,cCtc ; Den Vergleicherwert …
    out OCR0A,rmp ; ... in das Compare-
```

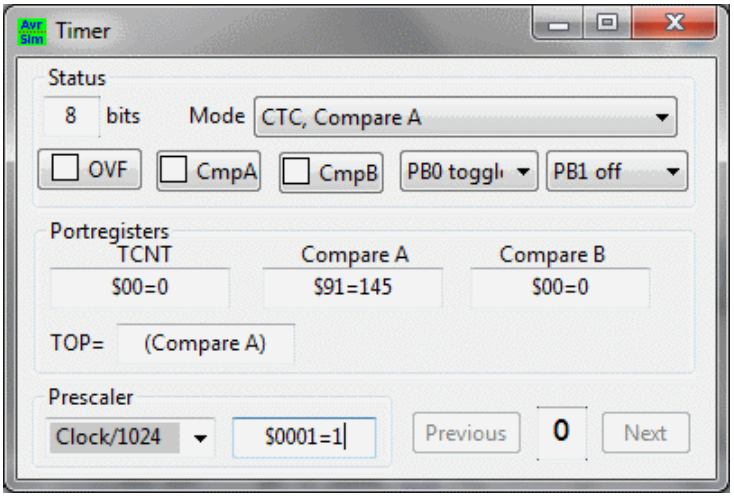
```
; Register A schreiben
ldi rmp,(1<<COM0A0) | (1<<WGM01) ; Toggle
; OC0A, CTC-Mode
out TCCR0A,rmp ; in Kontrollregister A
ldi rmp,(1<<CS02) | (1<<CS00) ; Vorteiler
          ; 1024
out TCCR0B,rmp ; in Kontrollregister B
; Schlafen ermoeglichen
     ldi rmp,1<<SE ; Schlafmodus Idle
     out MCUCR,rmp ; in Kontrollregister
Aufwachen:
     sleep ; Prozessor schlafen legen
     nop ; nach dem Aufwachen
     rjmp Aufwachen ; wieder schlafen legen
;
```

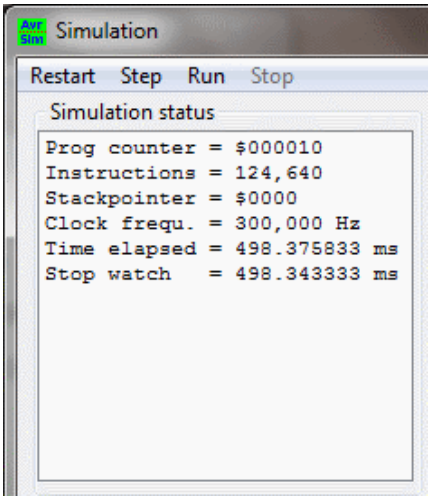
Nicht vergessen: der Prozessor arbeitet ab jetzt mit 300 kHz Takt, die ISP-Frequenz muss für das erneute Ansprechen mit dem Programmiergerät herabgesetzt werden.

4.5.2 Simulieren des Timerbetriebs im CTC-Modus

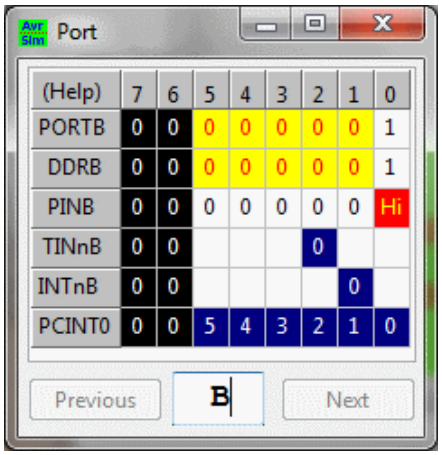
Wieder simulieren wir den Timer mit avr_sim. Das ist der Zustand nach der Initiierung. Die Taktfrequenz des Prozessors wurde durch das Schreiben in das CLKPR-Portregister auf 300 kHz herabgesetzt. Der Timer ist im CTC-Modus, mit dem Vergleichsregister Compare A als TOP-Wert. Der Pin PB0 torkelt bei Überschreiten des TOP-Werts.

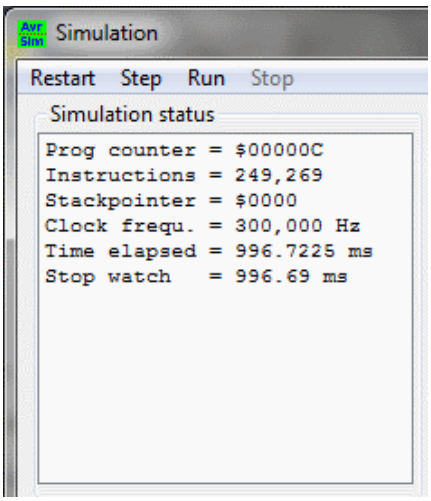






Das ist der Zustand wenn das erste Compare-Match-Ereignis erreicht ist: 498 ms sind vorbei und das Portbit PB0 ist gesetzt, was die LED ausschaltet.





Das zweite Compare-Match-Ereignis wird nach 996.69 ms erreicht. Das ist sehr nah an der Sekunde, aber nicht genau eine Sekunde.

4.5.3 Vor- und Nachteile

Der Strombedarf der Schaltung ist jetzt stark herabgesetzt, durch Takterniedrigung auf 300 kHz und Schlafmodus. Weniger geht nur noch, wenn wir die Betriebsspannung von 4,8 auf z. B. 3,3 V herabsetzen.

Nachteilig ist, dass wir noch immer nicht genau die eine Sekunde getroffen haben. Die Abweichung ist zwar viel geringer als die Abweichung des internen RC-Generators vom Soll-Wert (+/-10%), aber für eine Uhr selbst dann nicht zu gebrauchen, wenn wir einen externen Quarzoszillator anschließen würden.

4.6 Timer im 128kHz-Modus

Der ATtiny13 verfügt noch über einen weiteren internen RC-Oszillator, der per Fuse als Taktquelle ausgewählt werden kann. Er hat eine Frequenz von 128 kHz, die per CLKPR weiter geteilt werden kann, bis herunter auf 500 Hz. Tun Sie das nicht, es sei denn, sie haben ein Programmiergerät mit weniger als 125 Hz Taktfrequenz. Das AVRISPmkII kann 50,1 Hz, wäre also für diese Aufgabe noch geeignet.

4.6.1 Funktionsweise

Mit 128 kHz Taktfrequenz des Prozessors und des Timers muss für eine CTC-Toggle-Frequenz insgesamt durch 64.000 geteilt werden. 1.024 als Timer-Vorteiler führt bereits zu krummen 62,5 Hz. Mit 512 als Vorteiler muss der CTC auf 250 eingestellt werden, damit genau 2,00000 Hz Toggle-Frequenz herauskommen. Das ist mit dem 8-Bit-Timer im ATtiny13 realisierbar, der ja bis 255 zählen kann.

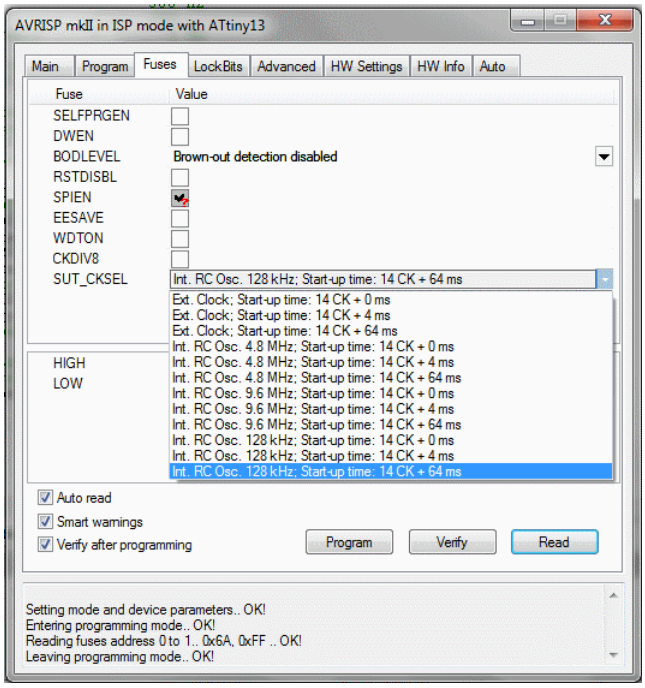
4.6.2 Programm

Das Programm (zum Quellcode im asm-Format) ähnelt dem vorherigen, nur der Timer-Vorteiler und der CTC-Wert ist anders. Bevor wir das programmieren, müssen wir die 32 im CLKPR aus dem vorherigen Experiment wieder loswerden. Dazu schreiben wir im vorherigen Program die Zeile "ldi rmp,(1<<CLKPS2)|(1<<CLKPS0)" in "ldi rmp,(1<<CLKPS1)|(1<<CLKPS0)" um, assemblieren und programmieren das Ergebnis in den Chip. Nun ist der CLKPR wieder "normal" und es kann das Folgende programmiert werden. Wir können dies aber auch lassen und stattdessen die CKDIV8-Fuse vorübergehend setzen (siehe unten).

```
;
; ***************************************
; * Blink mit Timer, RC-Osz. mit 128 kHz *
; * (C)2016 by www.gsc-elektronic.net    *
; ***************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ------- Register -----------------------
.def rmp = R16 ; Vielzweckregister
;
; ------- Timing -------------------------
; Interner RC-Oszillator = 128.000 Hz
; Clock-Vorteiler CLKPR  =       1
; Interne Taktfrequenz   = 128.000 Hz
; TC0-Vorteiler          =     256
; TC0-Timertick          =     500 Hz
; TC0-CTC-Teiler         =     250
; TC0-Togglefrequenz     =       1 Hz
;
; --------- Konstanten -------------------
.equ cCtc = 250-1
```

```
;
; --------- Programm ---------------------
  ; PB0 als Ausgang
  sbi DDRB,DDB0 ; Portbit als Ausgang
  ; Timer Compare-Match A auf 250
  ldi rmp,cCtc ; Match-A-Wert
  out OCR0A,rmp ; in Match-Register A
  ; Timer als CTC und Toggle Ausgang A
  ldi rmp,(1<<COM0A0)|(1<<WGM01) ; Toggle
       ; und CTC
  out TCCR0A,rmp ; in Kontrollregister A
  ldi rmp,1<<CS02 ; Vorteiler auf 256
  out TCCR0B,rmp ; in Kontrollregister B
  ; Schlafen ermoeglichen
  ldi rmp,1<<SE
  out MCUCR,rmp
  ; Schlafschleife
Schleife:
  sleep ; schlafen legen
  nop ; nach Aufwachen
  rjmp Schleife
;
```

Programmiert man das in den Tiny, blinkt es hektisch, da der Takt noch bei 1,2 MHz liegt. Jetzt müssen wir den Takt auf den internen 128 kHz-Oszillator einzustellen.

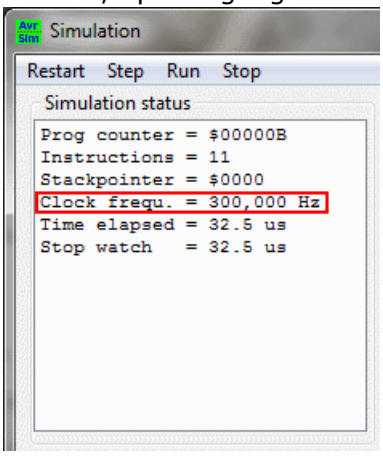


Dazu wird in der Fuses-Abteilung der Programmiertools der 128 kHz-Oszillator ausgewählt und der Haken bei der CKDIV8-Fuse entfernt. Wenn Du den Haken nicht gleichzeitig mit der Oszillatorumstellung verstellst, ist das Unglück schon passiert: Der Prozessor läuft mit 16 kHz Takt und ist nur noch ansprechbar, wenn die ISP-Frequenz unter 5 kHz verstellt werden kann.

Das Ganze geht mit dem Button "Program" an das Programmiergerät und hat hoffentlich das taktvolle Ein-Sekunden-Blinken der LED zur Folge.

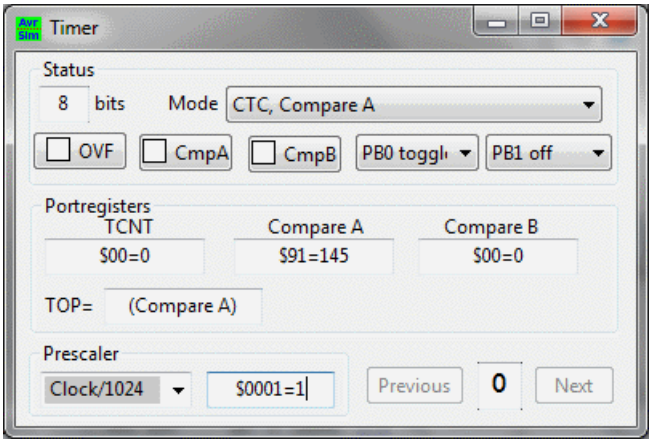
4.6.3 Simulation des Timerbetriebs bei 128 kHz Takt

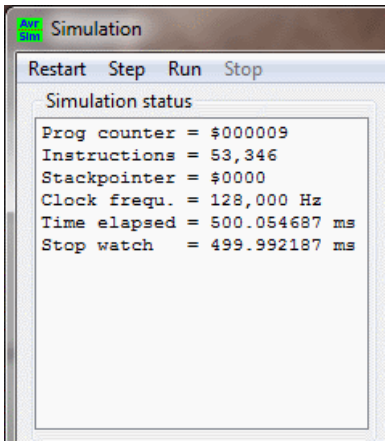
Simulation mit avr_sim ergibt Folgendes. Nach dem Init mit einer Taktfrequenz von 128 kHz sind 62,5 µs vergangen.



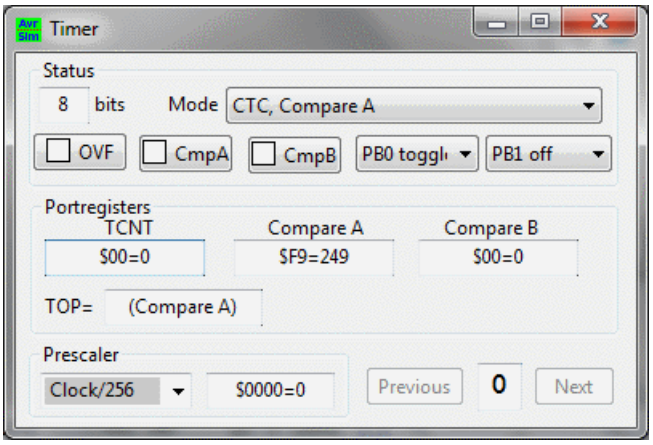
Der Timer ist im CTC-Modus mit dem Compare-Register A als TOP-Wert.

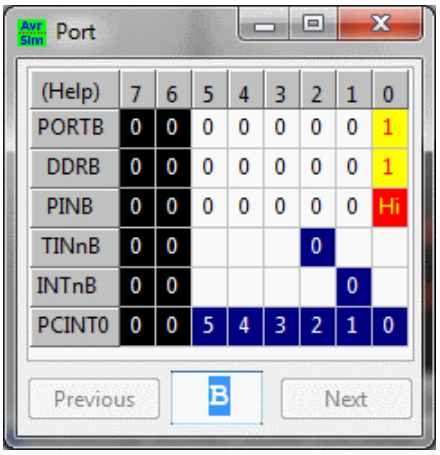
Nachdem der erste Compare Match auftrat sieht die Simulation so aus:



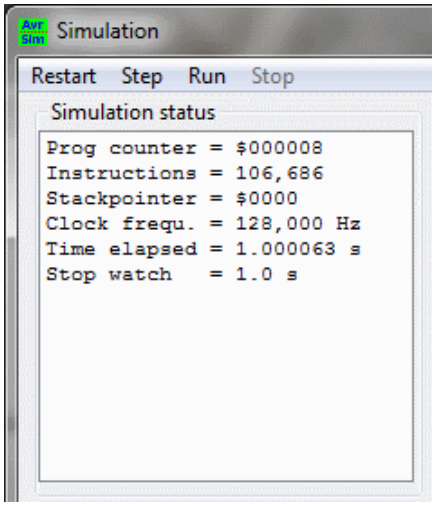


Die halbe Sekunde ist ziemlich genau eingehalten.



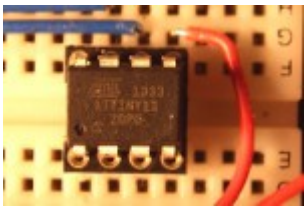


Alles ist korrekt auch beim zweiten Compare Match.



4.6.4 Vor- und Nachteile

Jetzt stimmt das Teilerverhältnis zwar theoretisch ganz genau, über die Genauigkeit des 128 kHz-Oszillators schweigt sich ATMEL in seinen Datenblättern aber völlig aus. Um einen externen Quarzoszillator kommen wir also bei einer Uhr nicht herum. Immerhin hat die ganze komplizierte Timing-Mimik nun einen Stromverbrauch, der bei einem Fünfzigstel der LED liegt.



Lektion 5: Eine LED über eine PWM einstellen

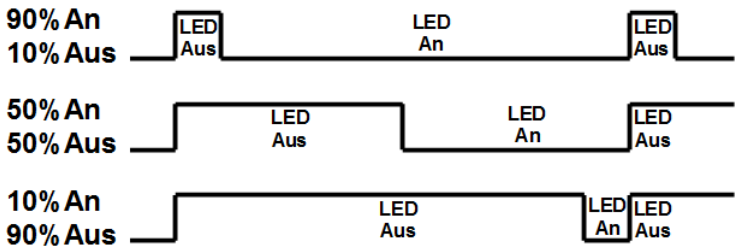
Bis jetzt war Blinken mit der LED angesagt. Jetzt geht das Geblinke weiter, aber mit einer sehr hohen Frequenz. Dafür verändern wir jetzt die LED-Helligkeit von ganz schwach bis ganz hell. Und zwar echt linear und nicht mit einem nichtlinearen Strom.

5.0 Übersicht

- 5.1 Einführung in den PWM-Modus des Timers
- 5.2 Hardware, Bauteile und Aufbau
- 5.3 Fast PWM Modus
- 5.4 Timer im phasenkorrekten PWM-Modus

5.1 Einführung in den PWM-Modus des Timers

5.1.1 8-Bit-PWM



PWM heißt Puls-Weiten-Modulation. Dazu setzt der Timer beim Neustart einen Ausgang (OCA, OCB) und wechselt bei Erreichen der Vergleicherwerte (OCRA, OCRB) dessen Polarität. Mit Erreichen des höchsten Wertes (255) und mit dessen Überschreitung wird wieder der Ausgangswert am Pin hergestellt.

Je später im Zählbereich der Vergleich eintritt, desto länger wird die erste Phase und desto kürzer die zweite Phase mit umgekehrter Polarität. Dieses Verhalten kann dazu dienen, mit dem Ausgangssignal Antriebsleistungen von Motoren oder eben auch LED-Helligkeiten zu regeln.

Table 11-6. Compare Output Mode, Fast PWM Mode[(1)]

COM01	COM00	Description
0	0	Normal port operation, OC0B disconnected.
0	1	Reserved
1	0	Clear OC0B on Compare Match, set OC0B at TOP
1	1	Set OC0B on Compare Match, clear OC0B at TOP

Note: 1. A special case occurs when OCR0B equals TOP and COM0B1 is set. In this case, the Compare Match is ignored, but the set or clear is done at TOP. See "Fast PWM Mode" on page 64 for more details.

Die Bits COM0A1 und COM0A0 im Port TCCR0A legen fest, wie sich der Ausgang OC0A verhält. Mit 0b10 wird der Ausgang zu Beginn auf Eins gesetzt, mit 0b11 auf Null. Analog kontrollieren COM0B1 und COM0B0 den Ausgang OC0B.

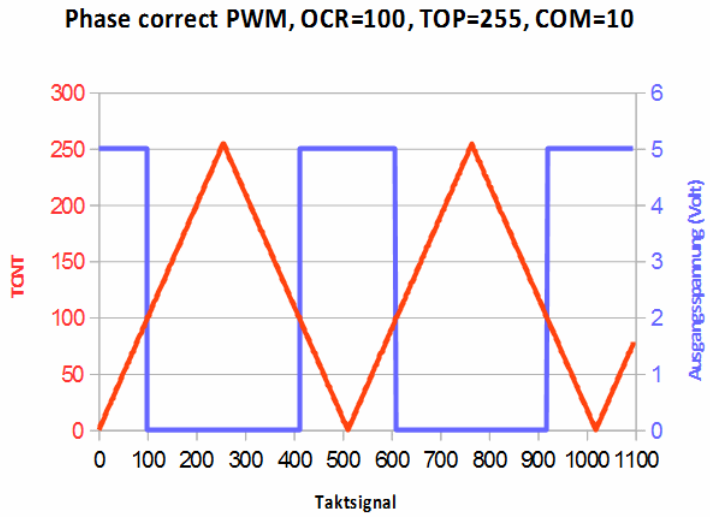
Dies hier zeigt die Impulsfolge am Ausgang OC0A, die sich bei Fast-PWM mit TOP=255 bei einem Vergleicherwert von 100 ergibt.

Bei Fast PWM kann durch die Wahl des Timer-Vorteilers die PWM-Frequenz beeinflusst werden. Bei den folgenden Taktfrequenzen ergeben sich bei den verschiedenen Vorteilern die folgenden PWM-Frequenzen.

Takt	P=1	P=8	P=64	P=256	P=1.024
9,6 MHz	37,5 kHz	4,69 kHz	586 Hz	146,5 Hz	36,6 Hz
1,2 MHz	4,69 kHz	586 Hz	73,2 Hz	18,3 Hz	4,6 Hz
128 kHz	500 Hz	62,5 Hz	7,8 Hz	1,95 Hz	0,49 Hz

Bei 1,2 MHz liegen alle PWM-Frequenzen im hörbaren Bereich (bei Motorsteuerungen wegen der Nebengeräusche wichtig). Ein Takt von 128 kHz kommt für Motor-PWM kaum in Frage.

5.1.2 Phasenkorrekte 8-Bit-PWM



Das ist der Signalverlauf bei einer phasenkorrekten PWM. Hier zählt der Zähler auf- und abwärts, jeweils bei Erreichen des Vergleichswerts erfolgt das Umschalten des Ausgangssignals. Das bedingt eine Halbierung der PWM-Frequenz.

Die phasenkorrekte PWM-Version kommt dann zum Einsatz, wenn sich der Vergleichswert sehr häufig und in großen Beträgen verändert. Dann zittert das Signal deutlich weniger als im Fast-Modus.

5.1.3 PWM mit unterschiedlicher Auflösung

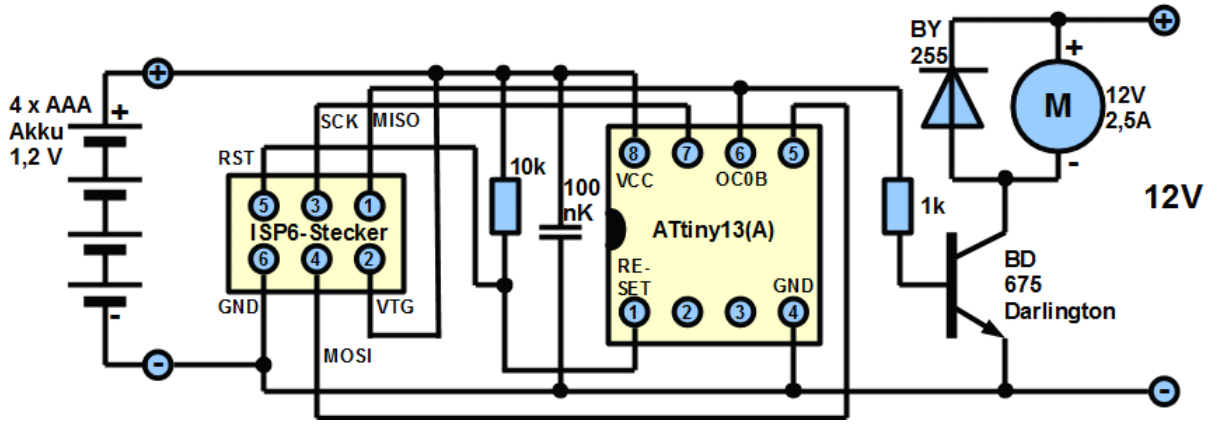
Mit einer Kombination des PWM-Modus mit der CTC-Steuerung des Timers lassen sich niedriger auflösende PWM einstellen. So ist z. B. für eine Motorsteuerung eine PWM mit acht Bit Auflösung (256 Stufen, 0,39% Genauigkeit) meistens überkandidelt. Um die PWM-Frequenz zu steigern, könnte eine verkürzte Zykluszeit gewählt werden. Setzt der Timer bei 16 zurück, erhält man eine 4-Bit-PWM, die sich in 16 Stufen verstellen lässt (6,25% Genauigkeit).

Da bei einem 16-Bit-Timer der Vergleicher A für die CTC verwendet wird, kann nur der Vergleicher B für PWM verwendet werden. Hat der AVR auch einen 16-Bit-Timer, dann ist PWM wegen der extrem langen Zyklen (65.536 Takte, 0,0015% Genauigkeit) nur sinnvoll, wenn dessen Auflösung auf die Anwendung sinnvoll verringert wird. Bei 16-bittigen Timern ist daher ein weiterer Vergleicher verfügbar (ICR), mit dem der CTC-Modus gesteuert werden kann, so dass die beiden Kanäle A und B für PWM-Signalerzeugung frei sind. Der ATtiny13 hat keinen 16-Bit-Timer.

5.2 Hardware, Bauteile und Aufbau

Für die PWM-Regelung kommt die gleiche Hardware zum Einsatz, wie wir sie schon in Lektion 2 aufgebaut haben.

Eine PWM-Anwendung für eine Motorsteuerung ist im Folgenden abgebildet. Hier treibt der OC0B-Ausgang einen NPN-Transistor, der den 12V-Motor per PWM-Signal antreibt.



5.3 Fast PWM Modus

Mit den bisher dargestellten Funktionsweisen, Eigenschaften und Methoden lässt sich die PWM-Ansteuerung der LED leicht programmieren. Als Zyklusfrequenz wählen wir 1,2 MHz (Vorteiler = 1) geteilt durch 256 = 4,7 kHz. Das ist sowohl für das menschliche Auge als auch für handelsübliche Digitalkameras schnell genug.

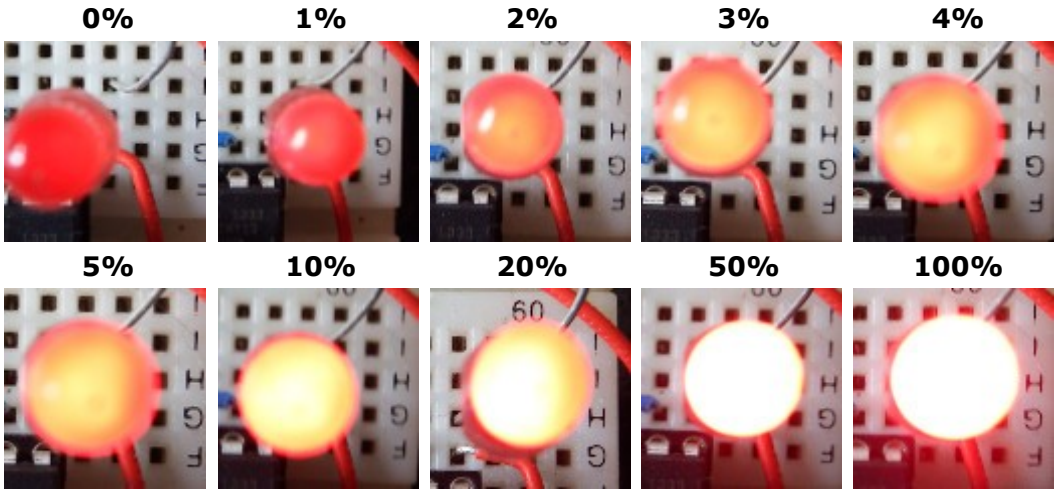
5.3.1 Programm

Indem wir die Polarität der PWM umkehren, heben wir die Umkehr durch die LED (0 = aus, 1 = ein) wieder auf (zum Quellcode im asm-Format).

```
;
; ****************************************
; * PWM-Ansteuerung einer LED im Fast-Modus *
; * (C)2016 by www.gsc-elektronic.net       *
; ****************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; -------- Register -------------------------
.def rmp = R16
;
; -------- Programm -------------------------
  ; Ausgang OC0A initiieren
  sbi DDRB,DDB0 ; OC0A als Ausgang
  ; PWM-Wert in Timer 0
  ldi rmp,20 * 256 / 100 ; 20% Helligkeit
  out OCR0A,rmp ; in Vergleichsregister A
  ; Timer 0 in Fast PWM Modus, Output A zu
        ; Beginn Low
  ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)|
(1<<WGM00)
  out TCCR0A,rmp ; in Kontrollregister A
  ; Timer 0 mit Vorteiler 1 starten
  ldi rmp,1<<CS00 ; Vorteiler auf 1
  out TCCR0B,rmp ; in Kontrollregister B
  ; Schlafen ermoeglichen
  ldi rmp,1<<SE ; Sleep enable
  out MCUCR,rmp ; in Universal-Kontrollregister
Schleife:
  sleep ; schlafen legen
  nop ; Aufwachen
  rjmp Schleife ; wieder schlafen legen
;
; Ende Quellcode
;
```

5.3.2 Ergebnis

Das geht recht einfach. Und das hier ist das Ergebnis:



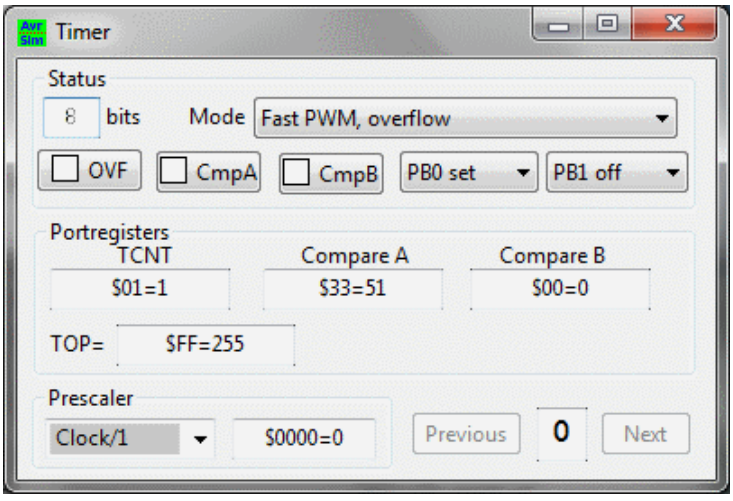
Die Unterschiede in der Helligkeit sind klar erkennbar.

Entscheidend ist, dass sich die PWM in Wirklichkeit nie auf Null drehen lässt: immer wird der erste Taktimpuls aktiviert, auch wenn das Vergleichsregister auf Null steht. Wer also tatsächlich ausschalten will, entkoppelt OC0A vom Timer und setzt den Portpin auf Null oder Eins.

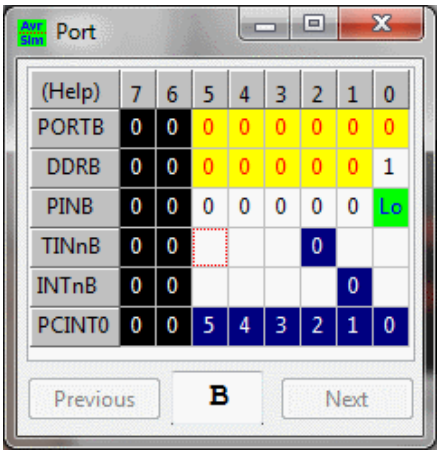
Top	Home	Einführung	Hardware	Standard	CTC	128kHz

5.3.3 Simulation des PWM-Modus

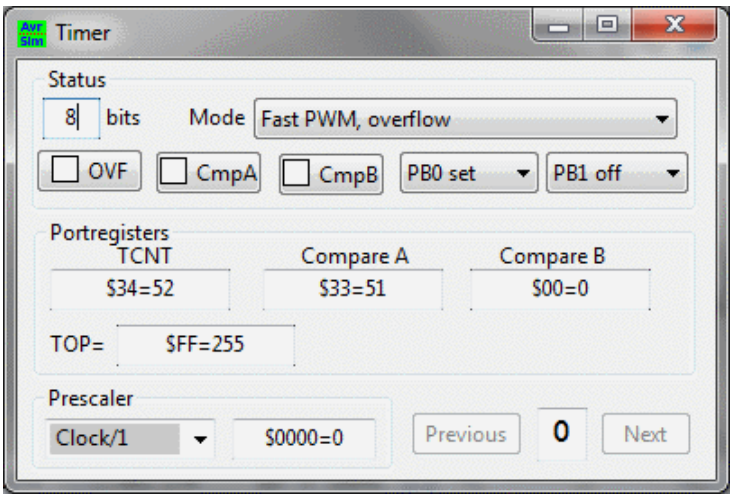
avr_sim kann dazu verwendet werden, um das Timing im PWM-Mode zu überprüfen.



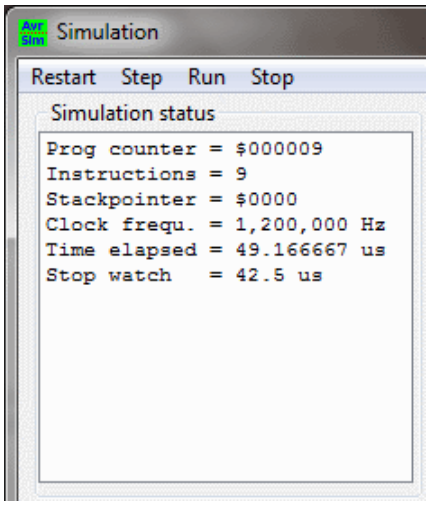
Nach 6,67 µs ist der Timer im Fast-PWM-Modus. Der Vergleichswert in A ist 51, Ausgangspin PB0 wird beim Vergleichswert auf Eins (LED aus) und zu Beginn des PWM-Zyklusses auf Null (LED an) gesetzt.

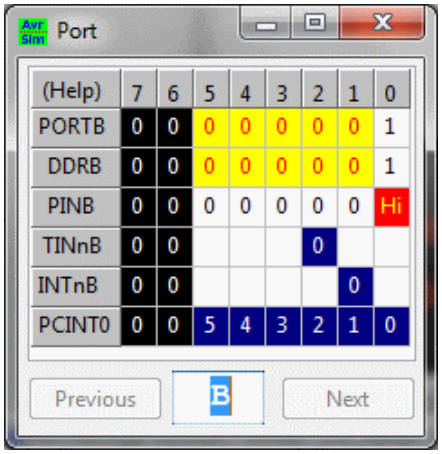


Der Port B nach der Initiierung, Pin PB0 ist als Ausgang gesetzt. Das bedeutet, dass die LED über die ersten (51 + 1) Zyklen an ist und bis 256 (Null) aus, was 20,3% Einschaltdauer entspricht.

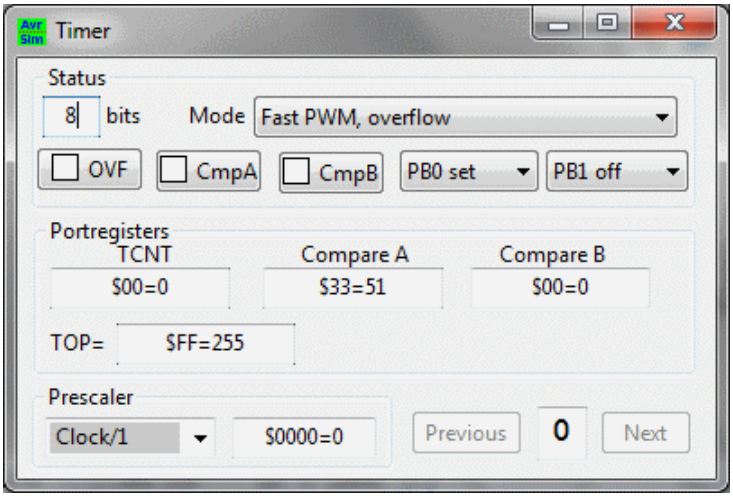


Das ist der Zähler, wenn der erste Match aufgetreten ist. Die bis dahin verbratenen 42,5 µs sind korrekt.

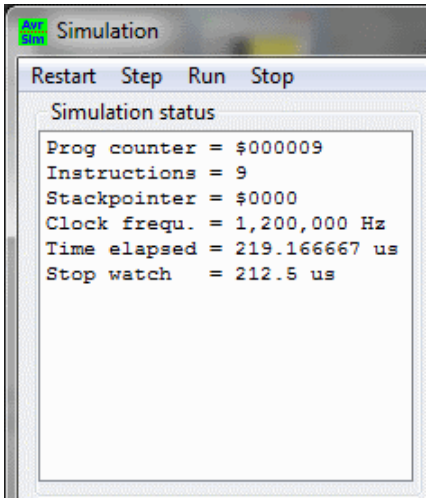




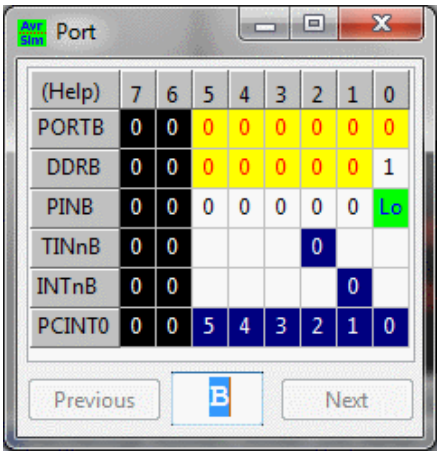
Das erste Vergleichsereignis setzt PORTB0 und schaltet damit die LED aus.



Der erste Überlauf des Zählers von 255 auf Null erfolgt nach 212,5 µs, was einer PWM-Frequenz von 1.200 / 256 = 4,7 kHz entspricht.

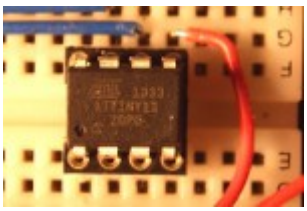


Der Überlauf hat das Portbit PORTB0 zurückgesetzt, die LED ist wieder an und der PWM-Zyklus beginnt erneut.



5.4 Timer im phasenkorrekten PWM-Modus

Den phasenkorrekten PWM-Modus kriegen wir, indem wir das WGM01-Bit im obigen Quellcode auf Null setzen (aus der zu setzenden Bitliste im Quellcode entfernen). Die PWM zählt jetzt auf- und dann abwärts und arbeitet daher mit der halben Frequenz, 2,34 kHz. Immer noch schnell genug für diese Anwendung, aber ein Motor dürfte jetzt deutlich hörbar summen.



Lektion 6: Eine LED blinkt mit dem Interrupt

Mit dieser Lektion treten wir schon in die Interrupt-Programmierung ein und beenden die lineare Programmierung. Nur sehr einfache Aufgaben lassen sich ohne Interrupts erledigen. Am häufigsten sind mehrere Aufgaben (scheinbar) gleichzeitig zu erledigen, was nur mit Interrupts vernünftig geht. Und genau das soll hier als grundlegende Programmiertechnik erlernt werden.

6.0 Übersicht

6.1 Einführung in die Interrupt-Programmierung

Der Vorteil der Interrupt-Programmierung ist, dass mehrere zeitkritische Vorgänge gleichzeitig ablaufen und trotzdem alle eintretenden Ereignisse korrekt ablaufen und bearbeitet werden. Spätestens dies ist das Ende aller Warte- und Verzögerungsschleifen.

Interrupts sind automatisierte Unterbrechungen des Programms. Läuft beispielsweise der Timer über (von FF auf 00), können wir mittels eines Bits im Timer veranlassen, dass bei diesem Ereignis ein Interrupt ausgelöst wird. Ist das Bit gesetzt, dann bewirkt jeder Überlauf

- das Setzen eines Bits in einem festgelegten Portregister (eine Interruptbedingung ist eingetreten),
- dass der Prozessor die gegenwärtige Ausführungsadresse in einen Speicher ablegt,
- die Ausführung an einer bestimmten, eindeutig festgelegten Adresse im Flashspeicher fortsetzt, und
- das gesetzte Bit im Portregister wieder löscht (die Interrupt-Ausführung ist erfolgt).

Was jetzt weiter passiert, ist Sache der selbstgeschriebenen Interrupt-Service-Routine. Dort wird erledigt, was beim Eintreten des Interrupts zu erledigen ist. Nach Erledigung kehrt der Programmablauf wieder dahin zurück, wo er zum Zeitpunkt der Unterbrechung war.

Bei diesem Konzept müssen zwei Bedingungen sichergestellt werden. Erstens darf während der Ausführung der Interrupt-Service-Routine nicht irgend ein anderer Interrupt dessen Ausführung unterbrechen (verschachtelter Interrupt). Daraus resultiert zweitens, dass bei gleichzeitig eintretenden Unterbrechungen eine Reihenfolge festgelegt werden muss, wer zuerst mit der Bearbeitung dran ist (Priorität).

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Die erste Bedingung wird durch ein Flaggenbit erfüllt, das im Statusregister SREG, einem Port, lokalisiert ist. Es heißt "I" und liegt im Bit 7 des Ports. Mit einer Null in diesem Bit wird kein Interrupt ausgeführt, welche Hardware auch immer nach Unterbrechung ruft. Es muss daher nach dem Initiieren der Hardware auf Eins gesetzt werden, damit überhaupt ein Interrupt ernstgenommen wird. Das Eins-Setzen dieses Bits geht mit der Instruktion "SEI" (SEt I), sein Löschen mit "CLI" (CLear I). Genau dieses Bit wird automatisch auf Null gesetzt, sobald der entsprechende Interrupt in Bearbeitung geht (und der Programmablauf verzweigt). Ist die Service-Routine beendet, muss sie dieses Bit wieder auf Eins setzen, um weitere anstehende oder neu eintretende Unterbrechungen wieder zuzulassen. Ohne dieses Setzen würden gar keine weiteren Unterbrechungen mehr zugelassen, die einsamen Rufer nach Unterbrechung würden ver-

hungern.

Damit wird auch klar, weshalb im obigen Ablauf jeder Unterbrecher ein Bit setzen muss, wenn der Fall eintritt: es kann durchaus sein, dass er erst später dran ist, wenn andere Interrupts noch in Bearbeitung sind. Und so lange signalisiert dieses Bit eben Unterbrechungsbedarf. Damit ist auch die Priorisierung von Interrupts klar: sie ist notwendig um zu bestimmen, wer als erstes dran ist, wenn mehrere dieser Bits gleichzeitig Unterbrechungsbedarf signalisieren. Die Priorität ist fest (siehe unten) und nicht beeinflussbar. Je höher der Interrupt in der Vektorenliste steht (siehe weiter unten), desto höher die Priorität.

Weil jede Interrupt-Service-Routine den weiteren Fortgang der Bearbeitung erst mal blockiert, gilt als wichtigste Regel, diese Service-Routine so kurz und knapp wie möglich zu halten. Keine Zeit hier, um langwierige Operationen oder Verzögerungsschleifen unterzubringen. Optimalerweise beschränkt sich diese Routine auf 10 bis 15 Instruktionen. Langwierige Bearbeitungen werden nach außerhalb dieser Routine zu verlegen sein. Das bedeutet, dass die Interrupt-Service-Routine mit "außerhalb" über Flaggen kommunizieren muss. Mehr dazu später.

6.1.1 Stapelspeicher

Um die Bearbeitung von Interrupts zu ermöglichen, braucht es einen Speicher, wo die Unterbrechungsadresse zeitweise abgelegt werden kann. Dazu verwendet der AVR den sogenannten Stack (Stapel) im statischen RAM. Der ATtiny13 hat 64 Bytes SRAM, reichlich um die zwei Byte lange Adresse abzulegen. Der Stapel wird immer am Ende des Speichers angelegt, indem der Stapelzeiger SPL (Stack Pointer Low) auf die Adresse RAMEND eingestellt wird. Ablegen auf den Stapel erfolgt immer so, dass die Stapeladresse niedriger wird, Entnahme vom Stapel erhöht die Adresse entsprechend wieder.

Ein einmal angelegter Stack kann im Programm auch zum zeitweisen Ablegen von Registerinhalten verwendet werden. Mit "PUSH R0" wird der Inhalt des Registers R0 auf den Stapel abgelegt, mit "POP R0" wieder gelesen.

Der Stapel kann ferner dazu verwendet werden, um zeitweilig zu einer anderen Programmadresse zu verzeigen und später wieder zurückzukehren. Mit "RCALL Label" wird die aktuelle Ausführungsadresse auf den Stapel gelegt und dann an die Adresse "Label"verzweigt. Von dort wird mit der Instruktion „RET" wieder an die aufrufende Stelle zurückgekehrt.

Dasselbe passiert bei einem Interrupt. Nur dass hier die Adressablage auf dem Stapel und die Verzweigung vollautomatisch geschieht. Anstelle "RET" wird das angesprungene Programm mit der Instruktion "RETI" beendet, die auch gleich noch das I-Bit im Statusregister auf Eins setzt und die Interruptbearbeitung wieder einschaltet.

6.1.2 Interruptvektoren

Wo springt der Prozessor nun bei einem Interrupt hin? An die ersten Adressen im Flashspeicher. Dort befinden sich die sogenannten Interruptvektoren. Beim ATtiny13 sind das folgende:

#	Adrs.	Name	Beschreibung
0	0000	RESET	Reset, Anlegen der Betriebsspannung, Brown-Out, Watchdog-Reset
1	0001	INT0	Signalflanke am Eingang INT0
2	0002	PCINT0	Signalflanke an einem Eingang
3	0003	TIM0_OVF	Timer 0 Überlauf
4	0004	EE_RDY	Schreibzyklus beim EEPROM abgeschlossen
5	0005	ANA_COMP	Polaritätswechsel am Analogvergleicher
6	0006	TIM0_COMPA	Timer 0 Vergleicher A
7	0007	TIM0_COMPB	Timer 0 Vergleicher B
8	0008	WDT	Watchdog-Ereignis

#	Adrs.	Name	Beschreibung
9	0009	ADC	AD-Wandlung beendet

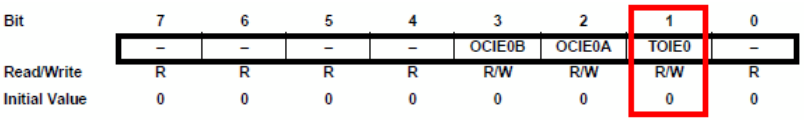
Die ersten 10 Programmworte in jedem interruptgesteuerten Programm sehen daher so aus:

```
.CSEG ; Assemblieren in den Flashspeicher (Code Segment)
.ORG 0 ; Adresse auf Null (Reset- und Interruptvektoren beginnen bei Null)
  rjmp Start ; Reset Vektor, Sprung zur Initiierung
  reti ; INT0-Int, nicht aktiv
  reti ; PCINT-Int, nicht aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  reti ; ADC-Int, nicht aktiv
;
; Programmstart beim Reset
;
Start:
  ; [Hier beginnt das Programm]
```

Das "RETI" bei allen nicht verwendeten Vektoren stellt sicher, dass bei einem versehentlich aktivierten Interrupt ein geordneter Rückweg ausgeführt wird.

Hier hat sich bei vielen Programmierern eine Unart entwickelt. Sie setzen die gewünschte Adresse eines verwendeten Vektors mit der Assembler-Direktive „.ORG Vektoradresse“. Daraus macht der Assembler in Verbindung mit dem Programmiergerät dann eine Vektortabelle, die mit lauter NOP-Instruktionen (mit unprogrammierten FFFF) durchsetzt ist. Für die hintersten Vektoren gibt es gar keine Entsprechung, da steht dann schon weiterer Programmcode oder irgendeine andere Interrupt-Service-Routine. Hat man dann einen Interrupt zugelassen, aber vergessen, den Interrupt mit einem Vektor zu versehen, dann wird statt Nichtausführung des fehlenden Vektors (dort stünde bei ordentlichem Programmieren noch ein RETI herum) nach einigen NOP's der nächste programmierte Vektor oder eben das, was auf die Vektortabelle folgt, ausgeführt. Fehlt diesem Code ein RETI, werden fürderhin einfach alle Interrupts blockiert. Das gibt lustige Folgefehler, die zu finden und zu beseitigen einen ziemlichen Aufwand verursachen kann. So kann man sich selbst lustige Beine stellen und das Leben erschweren, ohne groß nachzudenken.

6.1.3 Der Timerüberlauf-Interrupt



Um diesen Interrupt zu nutzen, reicht es aus, das Overflow-Interrupt-Enable-Flag TOIE0 im Timer Interrupt Mask Register TIMSK0 zu setzen, z. B. mit "ldi R16 1<<TOIE0" und "out TIMSK0 R16".

Damit der Interrupt angenommen wird, muss natürlich das I-Flag im Statusregister auf 1 gesetzt werden. Damit etwas passiert, muss natürlich noch eine Interrupt-Service-Routine her. Dazu muss in der Interruptvektortabelle in der Zeile mit dem Overflow-Int ein "rjmp ovflw_isr" hin und die Interrupt-Service-Routine ovflw_isr ist zu schreiben. Die kann z. B. zwischen die Vektortabelle und die Start-Routine. So eine typische Routine ist hier dargestellt:

```
ovflw_isr:
  in R15,SREG ; Statusregister in Register sichern
  dec R17 ; ein Register abwärts zählen
  brne ovflw_isr1 ; springe wenn noch nicht Null
  sbr R18,0b00000001 ; setze Bit 0 im Register R18
  ldi R17,10 ; zaehle ab 10 abwärts
ovflw_isr1:
  out SREG,R15 ; Statusregister wiederherstellen
  reti ; Rückkehr vom Interrupt, I-Flagge setzen
```

SBR setzt alle Bits in einem Register, die in der 8-Bit-Maske Eins sind. Die Instruktion und auch ihr Hexcode ist diesselbe wie ORI Register,Maske.

Typisch ist, dass zu Beginn das Statusregister gerettet werden muss. Da viele Instruktionen der Service-Routine die dortigen Flaggen beeinflussen (z. B. die Null-Flagge Z oder die Überlaufflagge C) können, muss dessen Originalzustand zu Beginn gesichert und am Ende wieder hergestellt werden. Schließlich kann die Routine jederzeit zuschlagen, auch wenn woanders mit diesen Flaggen gerade gearbeitet wird. Das gibt schwer zu findende Fehler, weil sie nur selten eintreten und immer an anderen Stellen des Codes auftreten. Einhaltung dieser Grundregel „Sichern/Wiederherstellen von SREG in ISRs" ist daher reiner Selbstschutz.

6.1.4 Der Compare-Match-Interrupt

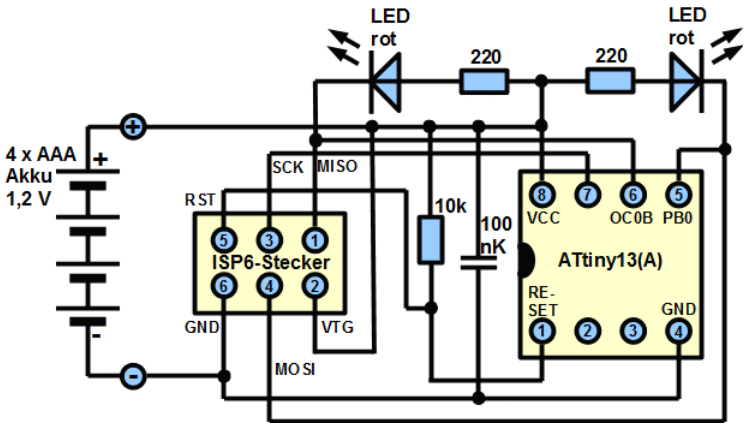
Setzt man eine oder beide Compare-Match-Flaggen OCIE0A oder OCIE0B, dann wird im Anschluss an jede Übereinstimmung mit dem Vergleichswert ein Interrupt ausgelöst. Bei einem Timer im CTC-Modus mit Compare-A tritt nach jedem Rücksetzen des Timers ein Interrupt auf, der zum Vektor TIM0_COMPA führt. Entsprechendes gilt für den Vergleicher B. Selbiges gilt für den PWM-Mode.

6.1.5 Interrupts und Schlafmodus

Im Schlafmodus Idle, den wir bereits verwendet haben, wecken alle Interrupts den Prozessor wieder auf. Damit dieser wieder schlafen gelegt werden kann, muss er in einer Schleife wieder die Instruktion "SLEEP" kriegen. Vor dem Schlafenlegen kann noch geprüft werden, ob eine Interrupt-Service-Routine noch eine Flagge hinterlassen hat, irgendetwas zu tun. Dann wird das eben jetzt erledigt, die Flagge wieder gelöscht und dann erst schlafen gelegt.

Top	Home	Einführung	Hardware	Overflow-Int	CTC-Int

6.2 Hardware, Bauteile und Aufbau



Für die Interrupt-Experimente kommt die gleiche Hardware zum Einsatz, wie wir sie schon in Lektion 2 aufgebaut haben. Jetzt kommt aber eine zweite LED mit Vorwiderstand am Portausgang OC0B (an Pin 6) hinzu. Diese Bauteile kennen wir schon aus Lektion 2.

6.3 Timer mit Overflow-Interrupt

6.3.1 Aufgabenstellung

Die folgende Aufgabenstellung ist zu lösen:

- Die LED blinkt im Sekundentakt.
- Bei jeder fünften Sekunde setzt das Blinken aus, die LED bleibt für eine Sekunde dunkel.
- Als weiteres Gimmick soll am Portpin PB1/OC0B eine weitere LED ihre Helligkeit sanft ändern.

6.3.2 Lösungsschritte

Es ist klar, dass sich diese Aufgabe nicht mit den bisherigen Blinkmechanismen lösen lässt, weil wir für diesen fünften Impuls keine geeigneten Mittel haben und das gleichzeitige Wechseln der Helligkeit der zweiten LED ein arg komplexes Geflecht erfordern würde. Es muss ein Mechanismus her, diesen Aussetzer und die Helligkeitssteuerung zu realisieren. Die Lösung ist ein interruptgesteuerter Zählmechanismus.

So ein Programm sieht grundlegend anders aus als ein Linearprogramm, wie wir es bislang verwendet haben. Weil fast alle Programme Interrupts benutzen, ist es wichtig, diese völlig andere Struktur zu verstehen.

Wir verwenden dazu den Timer-Overflow-Interrupt. Um damit den Sekundentakt hinzukriegen, muss die LED mit zwei Hertz (0,5 Sekunden an + 0,5 Sekunden aus) angesteuert werden. Von 1.200.000 Hz geteilt durch 256 (Timer-Overflow-Takte) = 4.687,5 Hz herab müssten wir einen Interruptzähler bis 4.687,5 / 2 = 2.344 (aufgerundet) zählen lassen. Da das grösser als 255 ist, brauchen wir dafür einen 16-Bit-Zähler.

Weil das Ausblenden der fünften Sekunde etwas aufwändigere Operationen bedingt, verlegen wir diesen Teil nach außerhalb der Interrupt-Service-Routine. Das wäre hier zwar nicht nötig, weil wir nur einen einzigen Interrupt bedienen müssen und bis zum nächsten Interrupt 256 Takte hin sind. Aber wir machen das jetzt hier mal um das Prinzip zu lernen, das bei komplexeren Abläufen ohnehin erforderlich ist.

6.3.3 Programm

Das hier ist eine der denkbaren Lösungen (zum Quellcode im asm-Format).

```
;
; ************************************
; * Timer mit Overflow-Interrupt     *
; * (C)2016 by www.gsc-elektronik.net *
; ************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ---------- Programmablauf -------------
;
; Das Programm erledigt zwei Vorgaenge parallel:
;   - Eine LED am PB0 blinkt ungefaehr im Sekun-
;     dentakt. Der fuenfte Impuls fehlt.
;   - Eine zweite LED am PB1 erhoeht ihre
;     Helligkeit. Nach Erreichen ihrer hoechsten
;     Helligkeit beginnt der Zyklus wieder von
;     vorne.
; Beide Vorgaenge werden durch den 8-Bit-Zaehler
;   TC0 gesteuert. Er laeuft mit einem Vorteiler
;   von 1 und loest nach 256 Impulsen den Over-
;   flow-Interrupt aus (alle 256 / 1.200.000 =
;   213 us). Ein 16-Bit-Zaehler zaehlt von 2.344
;   abwaerts auf Null und alle 0,5 Sekunden wird
;   die erste LED umgeschaltet. Beim fuenften
;   Impuls unterbleibt die Umschaltung auf LED
;   An.
;   Die zweite LED wird im Fast-PWM-Modus des
;   Timers am OC0B-Ausgang mit einem pulsgesteu-
;   erten Signal angesteuert: zu Beginn jedes
;   Timer-Zyklusses wird die LED ausgeschaltet
;   und mit Erreichen des Vergleichswertes B
;   bis zum Ende des Zyklus eingeschaltet. Der
;   Vergleichswert wird alle 213 us erniedrigt.
;
; ---------- Register -------------------
; frei: R0 .. R14
.def rSreg = R15 ; SREG-Zwischenspeicher
.def rmp = R16 ; Vielzweckregister
.def rimp = R17 ; Vielzweckregister Interrupts
.def rFlag = R18 ; Flaggenregister
  .equ bPol = 0 ; Flagge Polaritaetswechsel
.def rZaehl= R19 ; Blinkzaehler
.def rPwm = R20 ; PWM-Zaehler
; frei: R20 .. R23
.def rCntL = R24 ; Zaehler, LSB
.def rCntH = R25 ; dto., MSB
;frei: R26 .. R31
;
; ---------- Ports, Portbits ------------
.equ pLedOut = PORTB ; LED-Ausgabeport
.equ bLedOut = PORTB0
.equ pLedDdr = DDRB ; LED-Richtungsport
.equ bLedDdr = DDB0
.equ pLedIn  = PINB ; LED-Eingabeport
.equ bLedIn  = PINB0
;
.equ bPwmOut = PORTB1 ; Ausgabeport PWM
.equ bPwmDdr = DDB1 ; Richtungsport PWM
;
; ---------- Timing ---------------------
.equ cClock  = 1200000 ; Takt
.equ cPolwechsel = 2 ; Taktfrequenz Led An/Aus
.equ cPresc  = 1 ; Vorteiler Timer
.equ cCount  = cClock / 256 / cPresc / cPolwechsel
+ 1
.equ cBlink  = 5 ; Blinkzaehler
;
; ---------- Reset- und Interrupt-Vektoren -----
.CSEG ; Programmcode
.ORG 0 ; Reset- und Vektorenadresse
  rjmp Start ; Reset Vektor, Sprung zur Initi-
    ; ierung
  reti ; INT0-Int, nicht aktiv
  reti ; PCINT-Int, nicht aktiv
  rjmp Tc0IsrO ; TIM0_OVF, aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
```

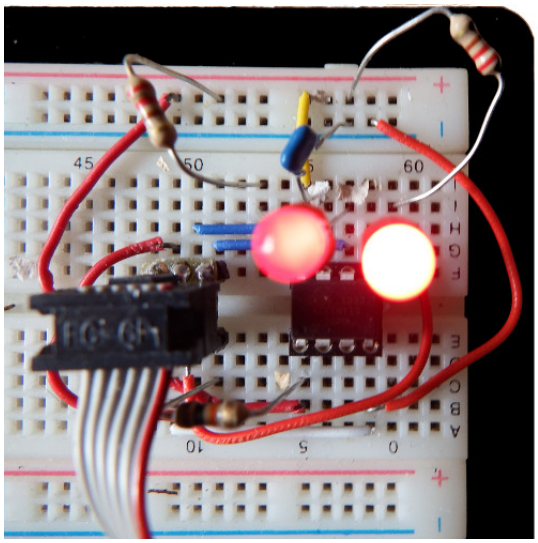
```
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  reti ; ADC-Int, nicht aktiv
;
; ---------- Interrupt-Service-Routinen --------
; Timer 0 Overflow:
;   Der Timer laeuft mit einem Vorteiler von 1
;   und ruft bei jedem Ueberlauf von 255 auf 0
;   diese Interrupt-Service-Routine auf (256 /
;   1.200.000 = 213 us, Frequenz = 1.200.000 /
;   256 = 4,6875 kHz).
;   In dieser wird der 16-Bit-Zaehler
;   rCntH:rCntL, der zu Beginn und beim Erreichen
;   von Null auf 1.200.000 / 256 / 2 = 2.343
;   gesetzt wird, um eins vermindert. Ist der
;   Zaehler gleich Null, wird die Flagge rPol im
;   Flaggenregister rFlag gesetzt und der Zaehler
;   neu gestartet.  Das ist alle 2.343 * 256 /
;   1.200.000 = 0,499984 s der Fall.
;   Ausserdem wird das LSB des Zaehlers daraufhin
;   ueberprueft, ob die unteren fuenf Bits Null
;   sind (ist alle 32 Timer-Interrupt-Takte =
;   213 us * 32 = 68,3 ms der Fall). Wenn das
;   der Fall ist, wird der Vergleichswert B
;   (rPwm) um Eins vermindert und die Helligkeit
;   der LED nimmt ab. War das 256 mal der Fall (=
;   68,3 ms * 256 = 17,5 s) startet der Hellig-
;   keitswert wieder mit voller Helligkeit.
Tc0IsrO: ; Timer 0 Overflow ISR
  in rSreg,SREG ; Statusregister sichern
  sbiw rCntL,1 ; Zaehler um Eins verringern
  brne Tc0IsrO1 ; springe wenn nicht Null
  sbr rFlag,1<<bPol ; Flagge Polaritaetswechsel
      ; setzen
  ldi rCntH,HIGH(cCount) ; Zaehler neu
      ; starten
  ldi rCntL,LOW(cCount)
Tc0IsrO1:
  mov rimp,rCntL ; Low Byte Zaehler
      ; kopieren
  andi rimp,0b00011111 ; die unteren fuenf
      ; Bit isolieren
  brne Tc0IsrO2 ; springe wenn nicht Null
  dec rPwm ; PWM-Wert abwaerts
  out OCR0B,rPwm ; in Vergleichsregister B
Tc0IsrO2:
  out SREG,rSreg ; Statusregister wieder
      ; herstellen
  reti ; Ende ISR, I-Bit setzen
;
; ---------- Initiieren und Programmschleife ----
Start:
  ; Stapel initiieren
  ldi rmp,LOW(RAMEND) ; Stapelzeiger auf
      ; SRAM-Ende
  out SPL,rmp ; Stapelzeiger setzen
  ; LED-Ausgang aktivieren
  ldi rmp,(1<<bLedDdr)|(1<<bPwmDdr) ; LED-
      ; Ausgaenge an, Treiber an
  out pLedDdr,rmp ; in Port-Richtungs-
      ; register
  ; Zaehler initiieren
  ldi rZaehl,cBlink ; Blinkzaehler
      ; initiieren
  ldi rCntH,HIGH(cCount) ; Zaehler auf
      ; Startwert
  ldi rCntL,LOW(cCount)
  ; Vergleichsregister initiieren
  ldi rmp,0xFF ; Compare A auf 255
  out OCR0A,rmp ; in Vergleichsregister A
  clr rPwm ; Startwert PWM
  out OCR0B,rPwm ; in Vergleichsregister B
  ; Zaehler 0 als starten mit Overflow-
      ; Interrupt
  ldi rmp,(1<<COM0B1)|(1<<WGM01)|(1<<WGM00)
      ; Fast PWM, COM0B
  out TCCR0A,rmp ; in Timer Kontroll-
      ; register A
  ldi rmp,1<<CS00 ; Vorteiler durch 1
  out TCCR0B,rmp ; in Timer Kontroll-
      ; register B
  ldi rmp,1<<TOIE0 ; Overflow-Interrupt
  out TIMSK0,rmp ; in Timer-Interrupt-Maske
  ; Schlafmodus
  ldi rmp,1<<SE ; Sleep-Enable, Mode Idle
  out MCUCR,rmp ; in Universal-Kontroll-
      ; register
  ; Interrupts ermoeglichen
  sei ; setze Interrupt-Flagge im
      ; Statusregister
; Programmschleife
Schleife:
  sleep ; schlafen legen
  nop ; nach Aufwachen
  sbrc rFlag,bPol ; ueberspringe naechste
      ; Instruktion wenn Flagge Null
  rcall Polaritaet ; Flagge bearbeiten:
      ; Polaritaetswechsel
  rjmp Schleife ; wieder schlafen legen
;
; --------- Polaritaet der LED wechseln --------
Polaritaet:
  cbr rFlag,1<<bPol ; Flagge loeschen
  sbis pLedOut,bLedOut ; ueberspringe
      ; naechste Instruktion wenn Led An
  rjmp Polaritaet3 ;  Led ist an, springe
      ; zu LED aus
  ; Led ist Aus
  cpi rZaehl,0 ; ist Blinkzaehler Null?
  breq Polaritaet1 ; ja, ist Null, starte
      ; Zaehler neu
  ; Led Aus, Zaehler nicht Null
  dec rZaehl ; nein, vermindere
      ; Blinkzaehler
  brne Polaritaet2 ; Led einschalten
  ret ; kein Polaritaetswechsel
Polaritaet1: ; Neustart Zaehler
  ldi rZaehl,cBlink ; starte Zaehler neu
  ret ; kein Polaritaetswechsel
Polaritaet2: ; Led einschalten
  cbi pLedOut,bLedOut ; Led an
  ret
Polaritaet3: ; Led ausschalten
  sbi pLedOut,bLedOut ; Led aus
  ret
;
; Ende Quellcode
;
```

Neu sind die Instruktionen

- IN Register,Port: liest alle acht Bits in einem Rutsch von einem Port in ein Register,
- SBR Register,Bitmaske: setzt alle Bits in einem Register, die in der Maske Eins sind, geht nur mit R16 und höher,
- CBR Register,Bitmaske: löscht alle Bits in einem Register, die in der Maske Eins sind, geht nur mit R16 und höher,
- ANDI Register,Maske: löscht alle Bits im Register, die in der Maske Null sind, geht nur mit R16 und höher,
- CPI Register,Konstante: vergleicht die Konstante (zieht sie ab, speichert aber das Ergebnis

nicht) mit dem Register und setzt die Flaggen im Statusregister, geht nur mit R16 und höher,
- BREQ Label: springt zur Adresse Label, wenn die Z-Flagge gesetzt ist,
- SBRC Register,Bit: überspringt die nächste Instruktion, wenn das Bit im Register Null ist, geht nur mit R16 und höher,
- SBRS Register,Bit: überspringt die nächste Instruktion, wenn das Bit im Register Eins ist, geht nur mit R16 und höher,
- SBIS Port,Bit: überspringt die nächste Instruktion, wenn das Bit im Port Eins ist, geht nur mit den Ports bis 31,
- SBIC Port,Bit: überspringt die nächste Instruktion, wenn das Bit im Port Null ist, geht nur mit den Ports bis 31.

6.3.4 Das Ergebnis



So leuchten zwei LEDs in unterschiedlichem Modus: die eine abrupt mit fehlendem fünftem Impuls, die andere mit sanft veränderter Helligkeit. Und das alles mit minimiertem Strombedarf, weil der Prozessor überwiegend im Schlafmodus herumschnarcht und nur bei Bedarf geweckt wird und danach in der Regel wieder schlafen gelegt wird.

6.3.5 Programmgliederung

Die erkennbare Gliederung des Programmes in

0. Registerdefinitionen, Portfestlegungen und Konstanten zu Beginn,
1. Reset- und Interruptvektoren,
2. Interrupt-Service-Routinen,
3. Hauptprogramm-Init mit Hardwarekonfiguration und Startwerten,
4. Schleifenbearbeitung, und
5. Bearbeitungsroutinen außerhalb von Interrupt-Service-Routinen

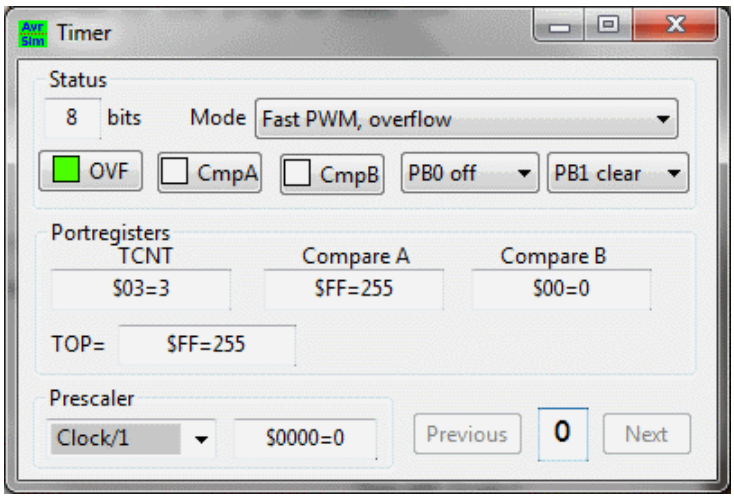
macht bei interruptgesteuerten Programmen immer Sinn, um den Überblick auch bei komplizierteren Abläufen zu behalten. Es hilft immens bei der Fehlersuche, weil man potenzielle Fehlerverursacher schneller findet.

6.3.6 Simulation der Vorgänge

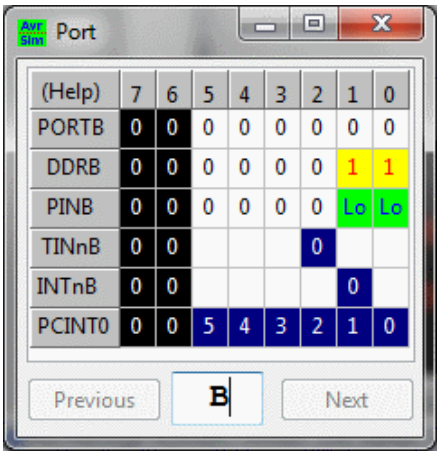
Die Simulation wird mit avr_sim unter den folgenden Rahmenbedingungen durchgeführt:

- Controllertakt: 1.2 MHz
- TC0-Vorteiler: 1
- TC0 TOP-Wert: 255
- Überlauf-Interrupt nach 256 / 1,200,000: 213 µs
- 16-Bit Abwärtszähler: 2.344
- Polaritätswechsel alle 2,344 * 213 µs: 500 ms
- Polaritätswechselfrequenz = 2 * 500 ms: 1 Sekunde

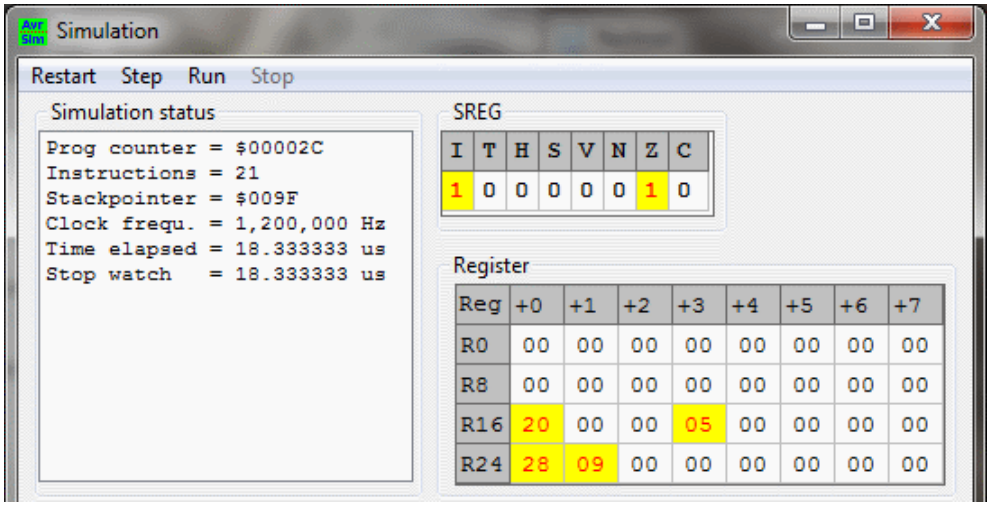
Dies hier sind die Einstellungen nach der Initiierung der Hardware:



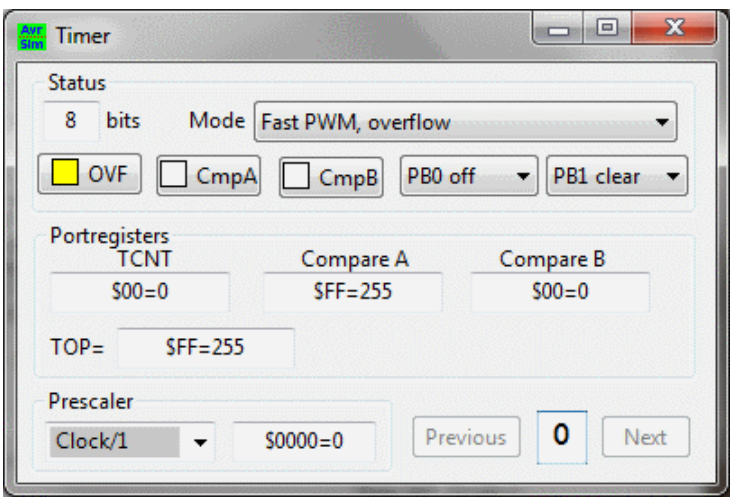
Der Timer TC0 ist im Fast-PWM-Modus, sein TOP-Wert und sein Vergleichswert A sind auf 255 gesetzt, der PWM-Vergleichswert im Compare-B-Portregister ist Null. Der Vorteiler ist auf 1 gesetzt und sein Überlauf-Interrupt-Enable-Bit ist gesetzt. Beim Erreichen des Compare-B-Wertes wird die LED am Ausgang PB1 auf Null geschaltet (PB1 clear), ist also an. Zu Beginn jedes PWM-Zyklusses ist PB0 gleich Eins, die LED also ausgeschaltet. Je größer also der Vergleichswert, desto schwächer die LED-Helligkeit (invertierende PWM).



Nach dem Init sind im Port B beide Richtungsbits gesetzt, um die beiden LEDs anzusteuern. Die Portbits PORTB0 und PORTB1 sind Null, beide LEDs sind an.

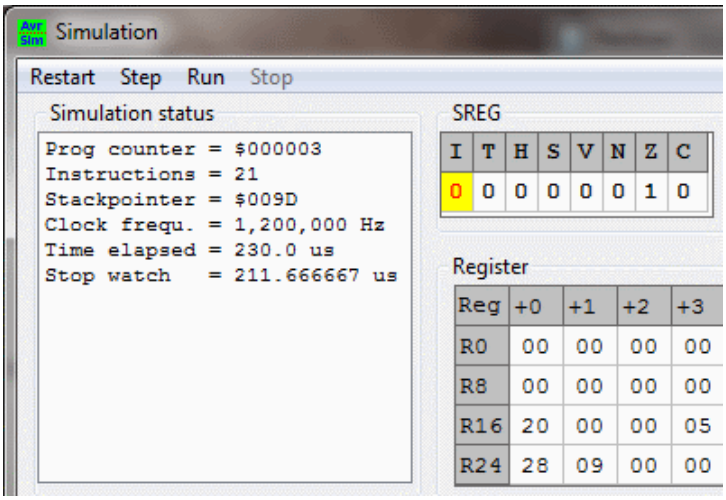


Die Initiierung dauerte 18,33 µs, hat den Stapelzeiger („stackpointer") auf das Ende des SRAMs im ATtiny13 gesetzt und den 16-Bit-Zähler R25:R24 auf 2.344 (=hexadezimal 0928) gestellt. Der LED-Abwärtszähler in R19 wurde auf 5 gesetzt und das Interrupt-Enable-Bit I im Statusregister SREG wurde gesetzt. Damit kann der interrupt-gesteuerte Zirkus so richtig beginnen.

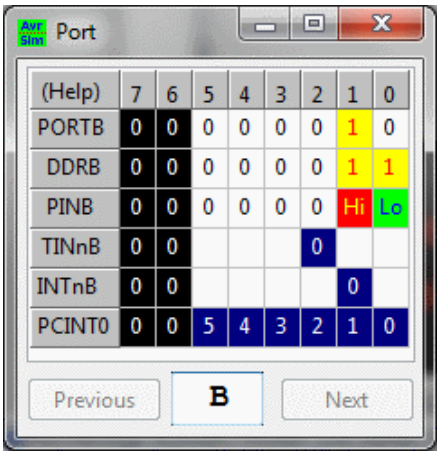


Der erste Überlauf des Timers ist aufgetreten, der Timer startet neu und hat seine Überlauf-Int-Flagge gesetzt. Mit der Ausführung der nächsten Instrukti-

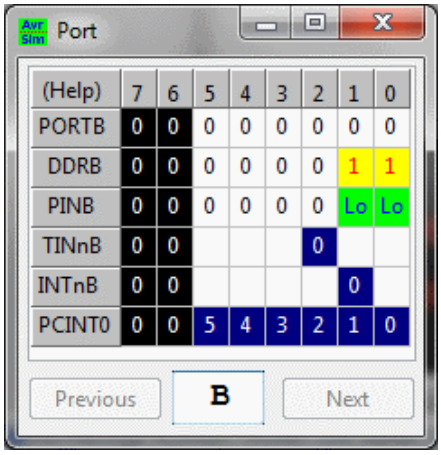
on führt der Controller den Interrupt aus und verzweigt zur Interrupt-Behandlungsroutine.



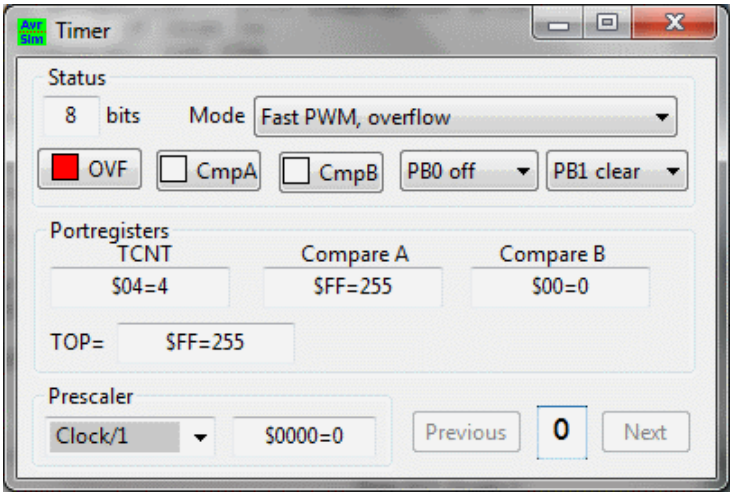
Der erste Überlauf-Int passiert nach 211,67 µs, zusammen mit den Interrupt-Enable-Instruktionen, dem Setzen der I-Flagge und der *SLEEP*-Instruktion 213 µs.



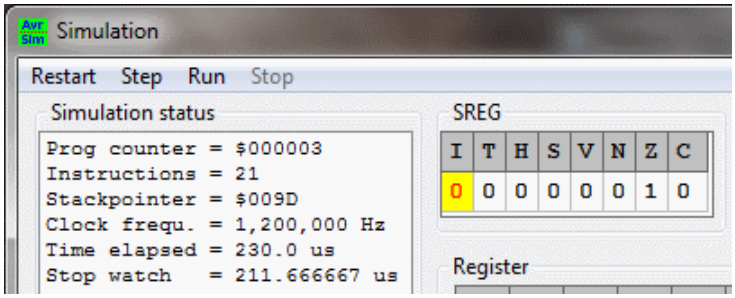
Beim Überlauf wurde des Portbit PB1 gesetzt, die LED also ausgeschaltet und der PWM-Zyklus neu begonnen.



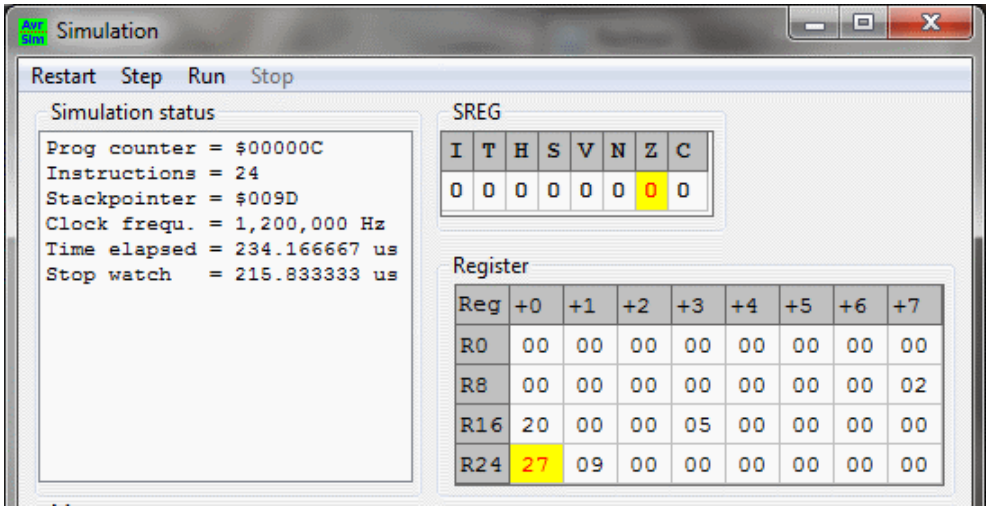
Mit der nächsten Instruktion wird schon der Compare Match B ausgeführt, weil sowohl TCNT als auch Compare B bei der letzten Instruktion Null waren. Die LED war aus für einen Zählzyklus und wird jetzt eingeschaltet. Bitte erinnern: solch ein OC-Ausgang kann nicht vollkommen auf Null gehen, ein einziger Zyklus ist die LED immer aus, auch wenn Compare B auf Null steht.



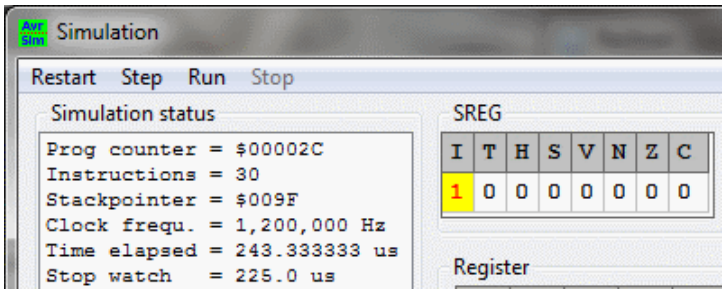
Vollkommen unabhängig von den Vorgängen am Portausgang wird jetzt der Timer-Interrupt ausgeführt. Der Controller hat die Adresse des Programmzählers (der auf dem *SLEEP* steht, auf dem Stapel abgelegt (zwei Bytes) und ist zur Vektoradresse des TC0-Überlaufes verzweigt.



Durch die Interruptausführung wurde der Stapelzeiger um zwei erniedrigt und das I-Bit im Statusregister SREG gelöscht.

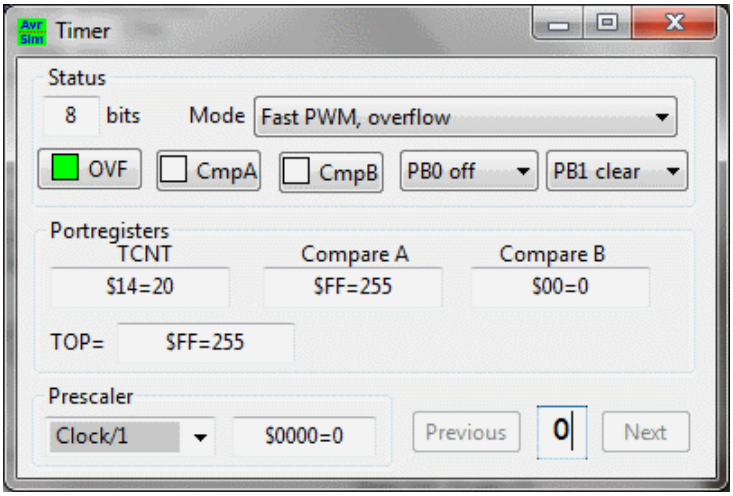


Innerhalb der Interrupt-Behandlungsroutine wurde der 16-Bit-Zähler in R25:R24 um Eins erniedrigt. Weil er noch nicht Null ist (siehe die nicht gesetzte Z-Flagge im SREG), wird auch die bPol-Flagge nicht gesetzt, der Polaritätswechsel bei der ersten LED an PB0 erfolgt daher noch nicht.



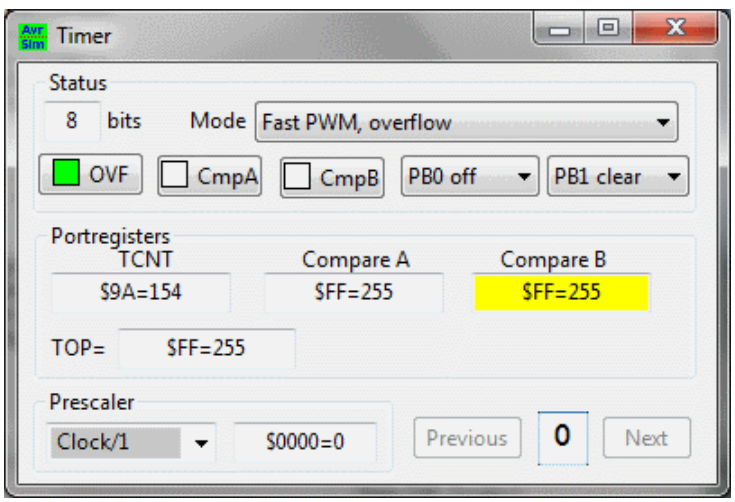
Mit der *RETI*-Instruktion wird der Interrupt beendet.

Das Interrupt-Request-Bit des Timers wurde schon bei der Ausführung des Interrupts automatisch zurückgesetzt, das Overflow-Enable-Bit des Timers bleibt aber weiter gesetzt.



Der Stapelzeiger hat wieder seinen alten Wert, denn *RETI* holt die beiden Bytes wieder vom Stapel und verzweigt wieder zur SLEEP-Instruktion, wo der Interrupt den Programmablauf unterbrochen hat. Die I-Flagge im SREG ist wieder gesetzt, auch das hat *RETI* erledigt.

Nun versinkt der Controller aber keineswegs wieder in den Schlaf, weil er vom Timer-Interrupt aufgeweckt wurde. Mit der nächsten Instruktion nach dem *SLEEP* wird der Programmablauf fortgesetzt und mit *sbrc rFlag,bPol* und *rcall Polaritaet* die Polaritätsflagge abgefragt. Die ist 2.343 mal nicht gesetzt und der Controller überspringt die *RCALL*-Instruktion und geht mit *rjmp Schleife* wieder in den Schlaf. Beim 2.344-sten Mal nicht: dann wird mit *RCALL* zum nachfolgenden Code verzweigt und ein Polaritätswechsel bei der LED an PB0 durchgeführt.

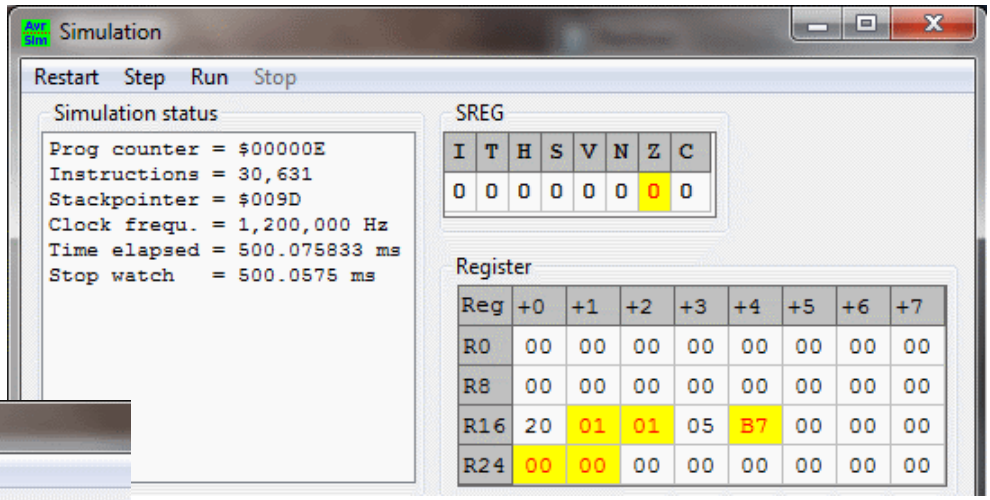


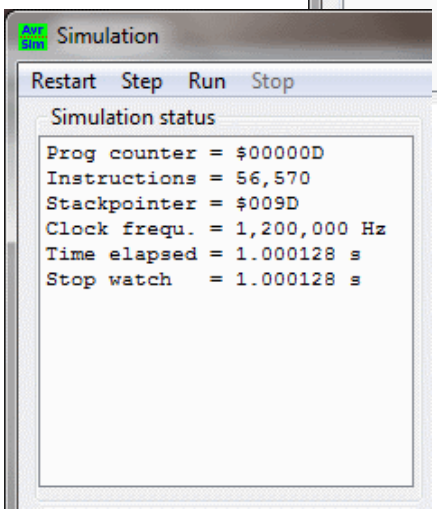
Eine besondere Bedingung ergibt sich, wenn der Compare-Wert für Compare B in der Interruptbehandlungs-Routine geändert wird. Der zum Einsatz kommende Compare-Wert B wird beim Fast-PWM-Modus erst dann in das Vergleichsregister übernommen, wenn der ablaufende PWM-Zyklus sein Ende er-

reicht hat. Bis das der Fall ist, kommt noch der alte Vergleichswert zum Einsatz (Gelbfärbung des Compare-B-Werts signalisiert das). Durch diesen Mechanismus wird sichergestellt, dass kein Compare-Ereignis verpasst wird, wenn der alte Wert über TCNT, der neue Wert aber unter TCNT liegt. Auch kann der Fall nicht eintreten, dass in einem Zyklus zwei Compare-Matches eintreten (wenn der alte Wert niedriger liegt als TCNT, der neue Wert aber über TCNT). Alle solche Katastrophen werden durch das zeitweise Zwischenspeichern des neuen Werts vermieden.

Das ist die Zeit wenn der erste Zähl-Zyklus in R25:R24 beendet ist:

- Der 16-Bit-Zähler in R25:R24 hat Null erreicht.
- Das Flaggenbit bPol in R18 ist gesetzt und wird das Ausführen der Polaritätswechsel-routine nach dem *SLEEP* auslösen,
- Ungefähr 500 ms sind abgelaufen.





Die Zählzeit für das Blinken der LED an PB0 ist ungefähr zutreffend. Es wäre genauer, wenn wir am immer gleichen Breakpoint anhalten würden und die Stopuhr definiert rücksetzen würden (mit der Maustaste in die Stopuhr-Zeile klicken).

Hier ist das Ende des zweiten PWM-Zyklusses erreicht. Etwa eine Sekunde ist seither vergangen bis die LED wieder eingeschaltet wird.

6.3.7 Vor- und Nachteile

Der Vorteil bei dieser Lösung ist die hohe Flexibiltät (leichte Modifizierbarkeit). So kann die Blinkfrequenz, der zu überspringende Blinkimpuls oder der LED-Port einfach durch Änderung von Konstanten im Kopf auf anderen Bedarf umgestellt werden.

Der Nachteil der Lösung ist, dass die Blinkdauer nur ungefähr stimmt. Das ist darauf zurückzuführen, dass das Teilen durch 256 nicht zu passenden Ganzzahlen führt. Diesen Nachteil behebt das nächste Lösung.

Top	Home	Einführung	Hardware	Overflow-Int	CTC-Int

6.4 Timer mit CTC-Interrupt

Damit es exakt zugeht, z. B. bei einer Uhr, ist eine exaktere Lösung zwingend. Die geht wie hier gezeigt.

6.4.1 CTC-Auswahl

Um exakte Teilerverhältnisse zu erhalten, muss die Kombination von Taktfrequenz, Vorteiler, CTC-Wert und Software-Zähler exakt den Zielwert ergeben. In der Tabelle sind für alle Taktfrequenzen des ATtiny13 und für alle Vorteiler diejenigen CTC-Teiler angegeben, die sich mit einem 8- oder 16-Bit-Zähler realisieren lassen, um 1 Hz zu erhalten. Die Verhältnisse sind farb-

lich gekennzeichnet.

Farben:	16-Bit-Zähler	8-Bit-Zähler	CTC=256	CTC=256/8-Bit-Zähler

Daraus ergeben sich folgende Ergebnisse (nur CTC-Werte > 100 sind aufgelistet).

Takt (Hz)	Prescaler	Ganzzahlige CTC-Teiler							
9.600.000	1	256 (37500)	250 (38400)	240 (40000)	200 (48000)	192 (50000)	160 (60000)	150 (64000)	
	8	250 (4800)	240 (5000)	200 (6000)	192 (6250)	160 (7500)	150 (8000)	128 (9375)	
	64	250 (600)	240 (625)	200 (750)	150 (1000)				
	256	250 (150)	150 (250)						
	1024								
4.800.000	1	256 (18750)	250 (19200)	240 (20000)	200 (24000)	192 (25000)	160 (30000)	150 (32000)	128 (37500)
	8	250 (2400)	240 (2500)	200 (3000)	192 (3125)	160 (3750)	150 (4000)		
	64	250 (300)	200 (375)	150 (500)					
	256	250 (75)	150 (125)						
	1024								
1.200.000	1	250 (4800)	240 (5000)	200 (6000)	192 (6250)	160 (7500)	150 (8000)	128 (9375)	
	8	250 (600)	240 (625)	200 (750)	150 (1000)				
	64	250 (75)	150 (125)						
	256								
	1024								
128.000	1	256 (500)	250 (512)	200 (640)	160 (800)	128 (1000)			
	8	250 (64)	200 (80)	160 (100)	128 (125)				
	64	250 (8)	200 (10)						
	256	250 (2)							
	1024								

Für 1,2 MHz Takt bietet sich eine CTC-Teilung durch 250 (125 für 2 Hz) bei einem Vorteiler von 64 an. Da die zweite LED ein PWM-Signal erzeugen muss, muss der Timer im PWM-Modus betrieben werden.

6.4.2 Programm

Für das Zählen reicht nun ein 8-Bit-Register (zum Quellcode im asm-Format).

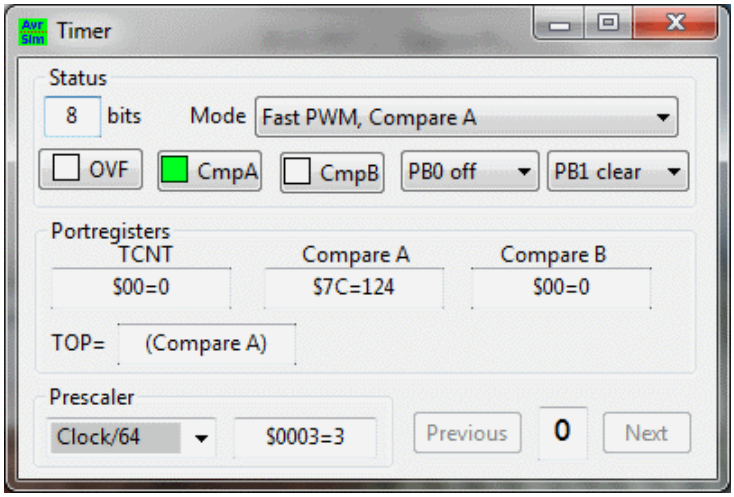
```
;                                                ;     nen RC-Generators mit +/- 10%).
; ************************************           ;   - Eine zweite LED am PB1 erhoeht ihre
; * Timer mit COMP-A-Interrupt          *       ;     Helligkeit. Nach Erreichen ihrer hoechsten
; * (C)2016 by www.gsc-elektronik.net   *       ;     Helligkeit beginnt der Zyklus wieder von
; ************************************           ;     vorne.
;                                                ; Beide Vorgaenge werden durch den 8-Bit-Zaehler
.NOLIST                                          ;   TC0 gesteuert. Er laeuft mit einem Vorteiler
.INCLUDE "tn13def.inc"                           ;   von 64 und setzt nach 125 Zaehlimpulsen wie-
.LIST                                            ;   der auf Null zurueck. Das dauert 64 * 125 /
;                                                ;   1.200.000 = 6,67 ms. Durch einen Software-
; ---------- Programmablauf -------------        ;   zaehler in einem Register wird dieser Takt
;                                                ;   bis auf 0,5 Sekunden verlaengert (Teilen
; Das Programm erledigt zwei Vorgaenge parallel: ;   durch 75), um die Sekunden-LED zu steuern.
;   - Eine LED am PB0 blinkt im Sekundentakt.    ;   Die zweite LED wird im Fast-PWM-Modus des
;     Der Sekundentakt wird ganz genau eingehal- ;   Timers am OC0B-Ausgang mit einem pulsgesteu-
;     ten (im Rahmen der Genauigkeit des inter-  ;   erten Signal angesteuert: zu Beginn jedes
```

```
;   Timer-Zyklusses wird die LED ausgeschaltet
;   und mit Erreichen des Vergleichswertes B
;   bis zum Ende des Zyklus eingeschaltet. Der
;   Vergleichswert wird alle 6,67 ms erniedrigt.
;
; ---------- Register ------------------
; frei: R0 .. R14
.def rSreg = R15 ; SREG-Zwischenspeicher
.def rmp = R16 ; Vielzweckregister
.def rimp = R17 ; Vielzweckregister bei Inter-
    ; rupts
.def rFlag = R18 ; Flaggenregister
    .equ bPol = 0 ; Flagge Polaritaetswechsel
.def rZaehl= R19 ; Blinkzaehler
.def rCtcInt = R20 ; CTC-Interrupt-Zaehler
.def rPwm = R21 ; PWM-Zaehler
; frei: R22 .. R31
;
; ---------- Ports, Portbits ------------
.equ pLedOut = PORTB ; LED-Ausgabeport
.equ bLedOut = PORTB0
.equ pLedDdr = DDRB ; LED-Richtungsport
.equ bLedDdr = DDB0
.equ pLedIn  = PINB ; LED-Eingabeport
.equ bLedIn  = PINB0
;
.equ bPwmOut = PORTB1 ; zweite LED im PwmMode
.equ bPwmDdr = DDB1
;
; ---------- Timing ---------------------
.equ cClock  = 1200000 ; Takt
.equ cPolwechsel = 2 ; Taktfrequenz Led An/Aus
.equ cPresc  = 64 ; Vorteiler Timer
.equ cCtcInt = 125 - 1 ; CTC-Int-Zaehler
.equ cCtcCnt = cClock / cPresc / cPolwechsel /
(cCtcInt +1)
.equ cBlink  = 5 ; Blinkzaehler
;
; ------- Reset- und Interrupt-Vektoren -------
.CSEG ; Programmcode
.ORG 0 ; Reset- und Vektorenadresse
  rjmp Start ; Reset Vektor, Sprung zur Initi-
     ; ierung
  reti ; INT0-Int, nicht aktiv
  reti ; PCINT-Int, nicht aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  rjmp Tc0IsrA ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  reti ; ADC-Int, nicht aktiv
;
; ---------- Interrupt-Service-Routinen --------
;
; TCO Interrupt Compare Match A
;   Der Timer laeuft mit einem Vorteiler von 64
;   bis zum Zaehlerstand von (124 + 1), setzt
;   sich dann wieder auf Null und loest diesen
;   Interrupt aus.
;   Bei diesem Interrupt wird der rPwm-Zaehler
;   erniedrigt (die Helligkeit der LED erhoeht)
;   und, wenn Null erreicht wird, auf seinen An-
;   fangswert gesetzt. rPwm kommt in das Ver-
;   gleichsregister B fuer die PWM an Pin PB1.
;   Dann wird der Abwaertszaehler rCtcInt ernie-
;   drigt. Ist dieser Null, wird die Flagge rPol
;   gesetzt und dieser wieder neu gestartet.
;
Tc0IsrA: ; Timer 0 Compare A Interrupt
  in rSreg,SREG ; Statusregister sichern
  dec rPwm ; PWM-Zaehler erniedrigen
  brne Tc0IsrA1 ; springen wenn nicht Null
  ldi rPwm,cCtcInt ; Neustart PWM-Zaehler
Tc0IsrA1:
  out OCR0B,rPwm ; PWM aktualisieren
  dec rCtcInt ; Zaehler um Eins verringern
  brne Tc0IsrA2 ; springe wenn nicht Null
  sbr rFlag,1<<bPol ; Flagge Polaritaetswechsel
setzen
  ldi rCtcInt,cCtcCnt ; Zaehler neu starten
Tc0IsrA2:
  out SREG,rSreg ; Statusregister wieder her-
    ; stellen
  reti ; Ende ISR, I-Bit setzen
;
; ---------- Initiieren und Programmschleife ----
Start:
  ; Stapel initiieren
  ldi rmp,LOW(RAMEND) ; Stapelzeiger auf
      ; SRAM-Ende
  out SPL,rmp ; Stapelzeiger setzen
  ; LED-Ausgang aktivieren
  ldi rmp,(1<<bPwmDdr)|(1<<bLedDdr) ; LED-
      ; Ausgaenge, Treiber an
  out pLedDdr,rmp ; in Richtungsregister
  ; Zaehler initiieren
  ldi rZaehl,cBlink ; Blinkzaehler
      ; initiieren
  ldi rCtcInt,cCtcCnt ; Zaehlerstartwert
  ldi rPwm,cCtcInt ; PWM initiieren
  ; Compare A-Wert fuer CTC setzen
  ldi rmp,cCtcInt ; CTC-Teiler durch 125
  out OCR0A,rmp ; in Vergleicher A
  ; Zaehler 0 normal starten mit Overflow-
      ; Interrupt
  ldi rmp,(1<<COM0B1)|(1<<WGM01)|(1<<WGM00)
       ; PWM am Port OC0B, Fast PWM
  out TCCR0A,rmp ; in Timer Kontroll-
      ; register A
  ldi rmp,(1<<WGM02)|(1<<CS01)|(1<<CS00)
      ; Fast PWM, Vorteiler durch 64
  out TCCR0B,rmp ; in Timer Kontroll-
      ; register B
  ldi rmp,1<<OCIE0A ; Compare A-Interrupt
  out TIMSK0,rmp ; in Timer-Interrupt-Maske
  ; Schlafmodus
  ldi rmp,1<<SE ; Sleep-Enable, Mode Idle
  out MCUCR,rmp ; in Universal-Kontroll-
      ; register
  ; Interrupts ermoeglichen
  sei ; setze Interrupt-Flagge im
      ; Statusregister
; Programmschleife
Schleife:
  sleep ; schlafen legen
  nop ; nach Aufwachen
  sbrc rFlag,bPol ; ueberspringe naechste
      ; Instruktion wenn Flagge Null
  rcall Polaritaet ; Flagge bearbeiten:
      ; Polaritaetswechsel
  rjmp Schleife ; wieder schlafen legen
;
; --------- Polaritaet der LED wechseln --------
Polaritaet:
  cbr rFlag,1<<bPol ; Flagge loeschen
  sbis pLedOut,bLedOut ; ueberspringe
      ; naechste Instruktion wenn Led An
  rjmp Polaritaet3 ;  Led ist an, springe
      ; zu LED aus
  ; Led ist Aus
  cpi rZaehl,0 ; ist Blinkzaehler Null?
  breq Polaritaet1 ; ja, ist Null, starte
      ; Zaehler neu
  ; Led Aus, Zaehler nicht Null
  dec rZaehl ; nein, vermindere Blinkzaehler
  brne Polaritaet2 ; Led einschalten
  ret ; kein Polaritaetswechsel
Polaritaet1: ; Neustart Zaehler
  ldi rZaehl,cBlink ; starte Zaehler neu
  ret ; kein Polaritaetswechsel
Polaritaet2: ; Led einschalten
  cbi pLedOut,bLedOut ; Led an
  ret
```

```
Polaritaet3: ; Led ausschalten                    ;
  sbi pLedOut,bLedOut ; Led aus                   ; Ende Quellcode
  ret                                             ;
```

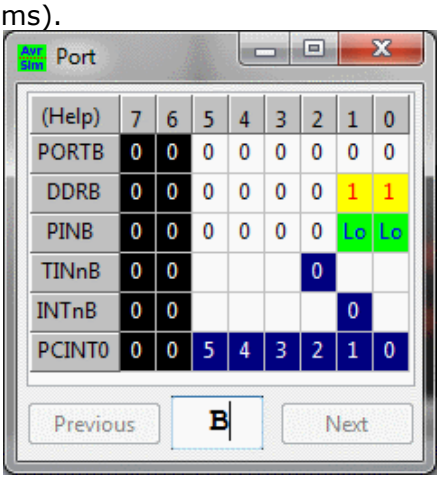
Natürlich sehen wir den Taktunterschied bei der LED gegenüber der vorherigen Lösung nicht, weil das menschliche Auge unempfindlich ist für solche minimalen Unterschiede. Wir wissen aber, dass das jetzt genau ist, weil wir es so programmiert haben.

6.4.3 Simulation



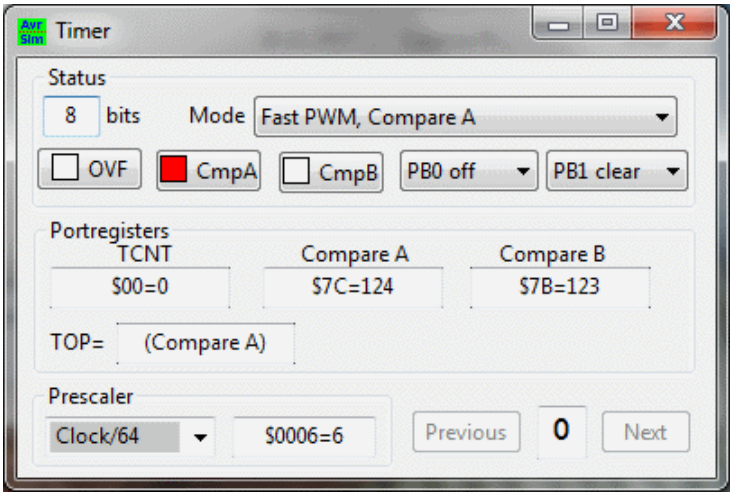
Das sind die Konfigurationen des Timers und des Ports nach der Initiierung.

Der Timer ist im Fast-PWM-Modus, mit einem Vorteiler von 64. Sein TOP-Wert wird vom Compare-Register A bestimmt (124, Rücksetzen bei (124 + 1), Dauer = 125 * 64 / 1.200.000 = 6,667 ms).

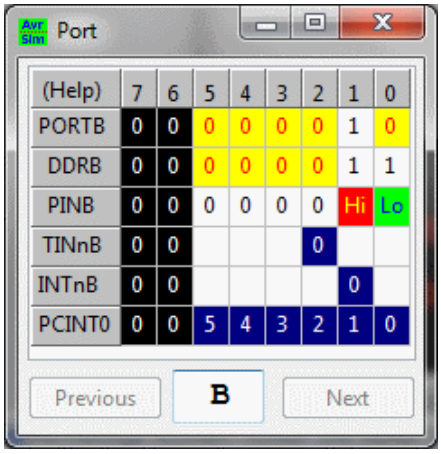


Der Port B treibt die LED-Ausgänge über die PORTB-Bits PORTB0 und PORTB1. PORTB1 wird wieder vom Timer gesteuert (Beginn PWM-Zyklus: LED aus, Compare Match B: LED an).

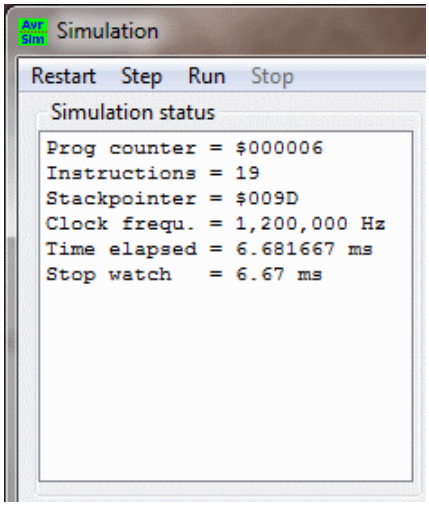
Beim Compare Match A wird der TC0 Compare Match Int ausgelöst.



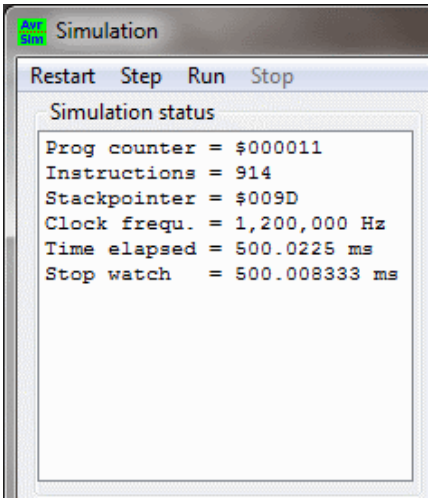
Der Timer hat den CTC-Wert 124 erreicht, mit dem nächsten Timer-Tick das Zählregister TCNT rückgesetzt und führt den Compare-Match-A-Interrupt aus. Der Compare-Match-B-Wert wurde auf 123 gesetzt, einen Puls vor dem Compare-Match A.



Der Portausgang PB1 ist auf Eins gesetzt, die LED ist aus.



Seit dem Eintritt in den Schlafmodus sind exakt 6,67 ms vergangen, die Zeit stimmt mit der Berechnung überein.

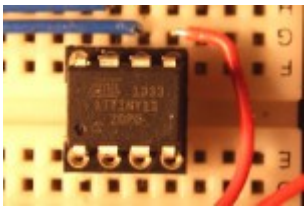


Das ist das Ende des ersten Zyklus: die Polaritätsflagge wird gesetzt und eine halbe Sekunde ist vergangen.

Exaktes timing mit dem Timer im CTC-Modus kann für fast beliebige Zeiten eingestellt werden. Dabei spielt es keine Rolle, was der Prozessor sonst noch tun mag und wie lange das dauert. In diesem Fall wurden zwei Aufgaben kombiniert (Schalten der Sekunden-LED, Helligkeitsregelung bei der zweiten LED), ohne dass sich beide Aufgaben irgendwie beissen.

Viele andere Aufgaben können in diesen Ablauf eingehängt werden. Und das alles mit nur einem einzigen Timer.

Top	Home	Einführung	Hardware	Overflow-Int	CTC-Int



Lektion 7: Eine LED blinkt mit dem Tasten-Interrupt

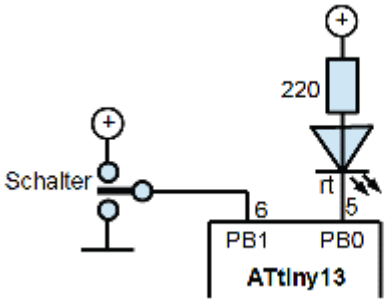
Mit dieser Lektion schließen wir eine Taste an den Prozessor an und verwenden den INT0-Interrupt, um nicht per Schleifen auf Drücke auf die Taste warten zu müssen.

7.0 Übersicht

- 7.1 Einführung in Tasten und die INT0-Programmierung
- 7.2 Aufgabenstellung
- 7.3 Hardware, Bauteile und Aufbau
- 7.4 Programm

7.1 Einführung in Tasten und die INT0-Programmierung

7.1.1 Tasten an Inputports



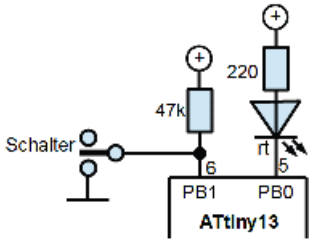
PB1	PB0	LED
0	1	Aus
1	0	An

Schaltet man bei einem Portpin den Ausgangstreiber mit DDRp/portbit=0 aus, dann kann man den Zustand des Pins im PORT PINp einlesen und abhängig von diesem Zustand Programmteile ausführen.

Um eine LED am Portpin PB0 einzuschalten, solange ein Schalter am Portpin PB1 einschaltet (mit der Betriebsspannung verbunden ist), wäre für die nebenstehende Schaltung das folgende Programm nötig:

```
  sbi DDRB,PB0 ; LED-Ausgangstreiber an PB0 einschalten
Schleife:
  sbis PINB,PINB1 ; ueberspringe Instruktion, wenn Inputpin Eins ist
  sbi PORTB,PORTB0 ; LED ausschalten
  sbic PINB,PINB1 ; ueberspringe Instruktion, wenn Inputpin Null ist
  cbi PORTB,PORTB0 ; LED einschalten
  rjmp Schleife
```

Was geschieht, wenn der Schalter gar nicht angeschlossen oder kaputt ist? Der Eingang PB1 ist dann offen. Er reagiert wegen seines extrem hohen Eingangswiderstandes auf alles, was sich in seiner Umgebung signalmäßig so abspielt, z. B. elektrische Felder des 230 V-Netzes, die statische Ladung von Fingern in der Nähe oder Pegelwechsel am Nachbarpin. Ergebnis ist Zappeln am Eingangspin und hektisches Flackern der LED.



Dagegen ist ein elektronisches Kraut gewachsen: der Pull-Up-Widerstand. Er wird gegen die positive Betriebsspannung geschaltet, setzt den Eingangswiderstand auf 47 kOhm herab und sorgt dafür, dass der Pin auf einem definierten Potential liegt. Der Strombedarf bei geschlossenem Schalter ist nicht allzu hoch (0,1 mA), auch wenn mit einem Mäuseklavier acht Eingänge gleichzeitig auf Null gezogen werden.

Da die Aufgabe von Pull-Up-Widerständen sehr oft vorkommt, gibt es die Möglichkeit, diese prozessorintern zuzuschalten. Sie werden eingeschaltet, indem das Datenrichtungsregister-Bit auf Null und das Ausgabeport-Bit auf Eins gesetzt wird. Z. B.

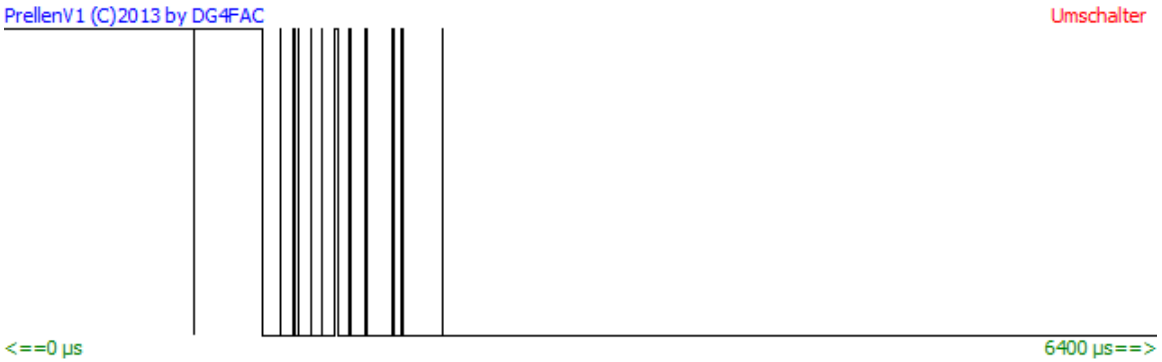
```
  cbi DDRB,DDB1 ; Richtung = Input
  sbi PORTB,PORTB1 ; Pull-Up einschalten
```

Warum Pull-Up und nicht Pull-Down dürfte historische Gründe haben (weil TTL-Eingänge von sich aus immer auf Eins oder High stehen).

Dasselbe gilt, wenn statt eines Schalters ein Taster an den Eingang angeschlossen ist. Dann ist der Pull-Up umso wichtiger, weil der im losgelassenen Zustand ja tatsächlich elektrisch in der Luft hängt.

7.1.2 Tasten und Schalter prellen

Eine unangenehme Eigenschaft ist, dass mechanische Tasten und Schalter immer prellen. Das heißt, sie schließen und öffnen nicht mit einem Mal sondern mehrmals. Das sorgt im Zeitbereich bis 10 oder 20 ms für einen echten Signalschwarm. Programmtechnisch bedeutet das, dass die Software auf solche Schwärme vorbereitet werden muss, damit sie von nachfolgenden Signalen nicht irritiert ist.



Das Prellen spielt erst bei der nächsten Lektion eine wichtige Rolle.

7.1.3 Der INT0-Interrupt

Bit	7	6	5	4	3	2	1	0	
	–	PUD	SE	SM1	SM0	–	ISC01	ISC00	MCUCR
Read/Write	R	R/W	R/W	R/W	R/W	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Der INT0-Interrupt überwacht den entsprechenden INT0-Eingang (beim ATtiny13 Pin6) auf auftretende Pegel und Pegelflanken und verzweigt zum INT0-Interrupt-Vektor, wenn der entsprechende Interrupttyp einschaltet ist. Die mit ISC01 und ISC00 auswählbaren Pegel und Flanken sind folgende:

ISC01	ISC00	Interruptauslösung
0	0	Niedriger Pegel löst Interrupt aus
0	1	Jeder Wechsel des Pegels löst Interrupt aus
1	0	Fallender Pegel löst Interrupt aus
1	1	Steigender Pegel löst Interrupt aus

Die erste Einstellung ist ziemlich fatal, weil der Defaultwert ist, dass Nullpegel Interrupts auslösen. Das bedeutet, dass versehentliches Aktivieren des Interrupts in diesem Modus zu einer Dauerblockade des Prozessors führen: der Interrupt schlägt sofort wieder zu, wenn der letzte fertig bearbeitet ist, solange der Eingang Null ist. Da der INT0-Interrupt auch noch die höchste Priorität hat, kommt auch kein anderer mehr durch. Damit kriegt man einen AVR zur Totalblockade mit nachhaltiger Arbeitsverweigerung.

Die ISC-Bits liegen im gleichen Port wie das SE-Bit zum Schlafen. Die Pegelauswahl erfolgt daher immer zusammen mit dem SE-Bit.

Bit	7	6	5	4	3	2	1	0	
	–	INT0	PCIE	–	–	–	–	–	GIMSK
Read/Write	R	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

Und so kriegt man den INT0 dazu, Interrupts anzufordern: Eins-setzen des INT0-Bits im Port GIMSK. Die Instruktion dafür ist:

```
  ldi R16,1<<INT0 ; INT0-Bit setzen
  out GIMSK,R16 ; und in General Interrupt Mask schreiben
```

Im Bereich der Interrupt-Vektoren steht der INT0-Interrupt an oberster Stelle hinter dem Reset-Vektor, mit höchster Priorität. Im ATtiny13:

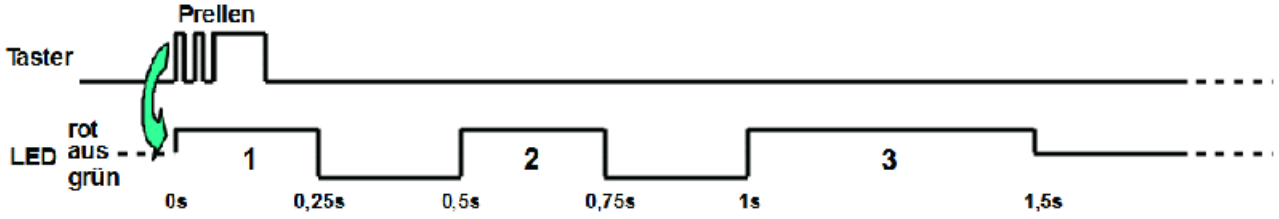
```
.CSEG ; Assemblieren in den Flashspeicher (Code Segment)
.ORG 0 ; Adresse auf Null (Reset- und Interruptvektoren beginnen bei Null)
  rjmp Start ; Reset Vektor, Sprung zur Initiierung
  rjmp Int0_Isr ; INT0-Int, aktiv
  reti ; PCINT-Int, nicht aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  reti ; ADC-Int, nicht aktiv
;
; Interrupt-Service-Routinen
;
Int0_Isr: ; INT0-ISR
  in R15,Sreg ; SREG retten
  [...] ; weitere Aktionen
  out SREG,R15 ; SREG wieder herstellen
  reti ; zurueckkehren
;
; Programmstart beim Reset
;
Start:
  ; [Hier beginnt das Programm]
```

Das war es schon und es kann an die Aufgabe gehen.

Top	Home	Einführung	Hardware	Programm

7.2 Aufgabenstellung

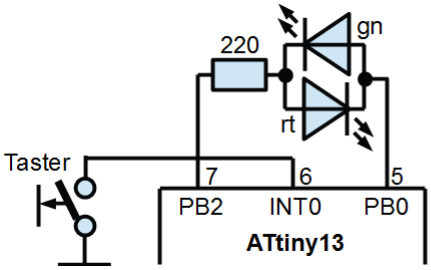
Das hier ist die komplexe Aufgabe:



Ein Tastersignal startet eine Signalfolge der LED, die zwei kurze und ein drittes längeres Signal mit dem dargestellten Timing umfasst. Die Lösung soll prell-unempfindlich sein.

7.3 Hardware, Bauteile und Aufbau

7.3.1 Hardware



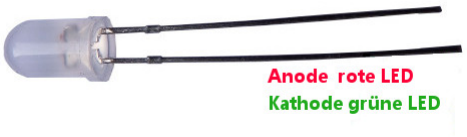
PB2	PB0	LED
0	0	Aus
0	1	grün
1	0	rot
1	1	Aus

Die Duo-LED ist an den Portbits PB2 (Anode rot) und PB0 (Anode grün) angeschlossen. Die Ausgangssignale an beiden Portpins zeigt die Tabelle. Die Taste ist mit einem Anschluss an den INT0-Eingang (PB1) angeschlossen, der zweite Anschluss liegt auf Masse (Minus).

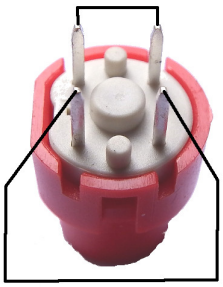
7.3.2 Bauteile

7.3.2.1 Duo-LED

Die Duo-LED enthält zwei LED, eine rote und eine grüne. Der längere Anschlussdraht ist die Anode der roten LED und die Kathode der grünen.



7.3.2.2 Taster

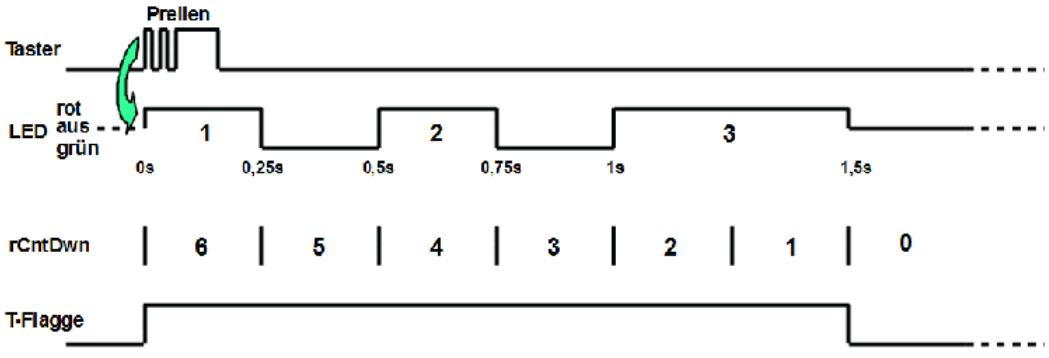


Das ist ein Taster. Jeweils zwei seiner vier Anschlussstifte sind verbunden, Drücken der Taste schließt die Anschlüsse.

Top	Home	Einführung	Hardware	Programm

7.4 Programm

7.4.1 Ablauf



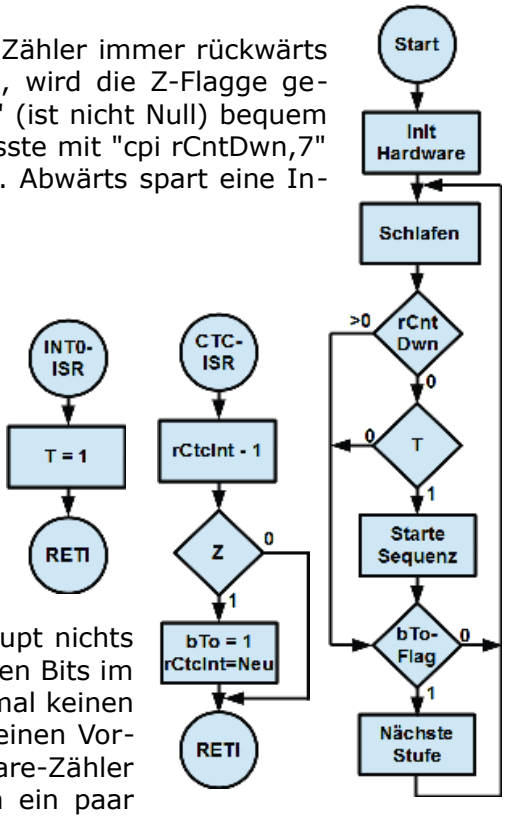
Um die Aufgabe zu lösen ist ein klarer Ablauf aufzuzeichnen. In diesem Fall ist er offenkundig so.

Um die Farben der LED zu den richtigen Zeiten zu wechseln, verwenden wir

einen Abwärtszähler rCntDwn. Der Zähler beginnt bei sechs und muss alle 0,25 Sekunden abwärts zählen. Bei den Zählerständen 5 und 3 muss auf Grün umgeschaltet werden. Ist der Zähler Null, dann endet der Zyklus, die LED werden abgeschaltet und die T-Flagge auf Null gesetzt.

Noch ein Wort darüber, warum Assemblerprogrammierer Zähler immer rückwärts laufen lassen: erreicht er beim Dekrementieren die Null, wird die Z-Flagge gesetzt. Und die lässt sich mit "BREQ" (ist Null) und "BRNE" (ist nicht Null) bequem zum Verzweigen benutzen. Liefe der Zähler aufwärts, müsste mit "cpi rCntDwn,7" und dann erst könnte mit "breq Label" verzweigt werden. Abwärts spart eine Instruktion.

7.4.2 Ablaufdiagramme



Die T-Flagge ist ein frei verfügbares Bit 6 im Statusregister (siehe Lektion 6). Es lässt sich mit "SET" auf Eins setzen und mit "CLT" löschen. Mit den bedingten Sprungbefehlen "BRTS" und "BRTC" lassen sich Verzweigungen realisieren, wenn das T-Bit geSetzt oder geCleared ist. Eine Besonderheit tritt auf, wenn die T-Flagge in einer Interrupt-Service-Routine geändert werden soll: mit dem Sichern und Wiederherstellen des Statusregisters würde das zwischendurch gesetzte T-Flag wieder überschrieben. In unserem Fall macht das nix, weil der INT0-Interrupt nichts weiter tun muss als nur die T-Flagge setzen. Da die anderen Bits im SREG in der Routine nicht verändert werden, gibt es diesmal keinen Konflikt. Um das Timing exakt hinzukriegen, wählen wir einen Vorteiler von 8, einen CTC-Teiler von 150 und einen Software-Zähler bis 250. Das ergibt genau 0,25 Sekunden. Da wir noch ein paar Takte Verwaltung zu tun haben, ist die Genauigkeit nur scheinbar, aber die Abweichungen sind hinnehmbar. Und besser und eleganter als elendig lange Verzögerungsschleifen allemal.

Das lange Diagramm ist der gesamte Ablauf ohne die beiden Interrupt-Service-Routinen. Diese beiden laufen separat ab und kommunizieren ihre erreichten Zustände (die Taste wurde gedrückt, der Timer hat 0,25 s erreicht) über zwei Flaggen (T, bTo) an das Hauptprogramm, das schlafend auf diese beiden Ereignisse wartet.

Das Programm startet mit den Reset- und Interruptvektoren. Im Hauptprogramm folgt das Initiieren der Portpins und das Ermöglichen des INT0-Interrupts. Danach tritt der schlafende Wartezustand ein. Tritt ein Tasteninterrupt ein, dann wird das nach dem Aufwachen dadurch bemerkt, indem das T-Flag gesetzt ist. Damit das Setzen der T-Flagge nicht erneut zu einem Neustart einer schon ablaufenden Blinksequenz führt, muss zuerst geprüft werden, ob eine solche Sequenz bereits gestartet ist und gerade abläuft. Das wird am Registerinhalt des Countdown-Zählers "rCntDwn" erkannt. Ist er nicht Null, wird die folgende Prüfung auf eine gesetzte T-Flagge übersprungen. Wenn nicht, wird T geprüft. Ist T gesetzt, wird nun eine Blinksequenz gestartet. Dazu wird beim Timer

- der CTC-Wert in das Vergleichsregister geschrieben,
- der CTC-Modus in das Kontrollregister A geschrieben,
- mit einem Vorteiler von 8 im Kontrollregister B gestartet, und
- der Vergleicher-A-Interrupt ermöglicht.

Die LED wird auf rot geschaltet und der Countdown-Zähler mit 6 gestartet.

Sind die vorgewählte Anzahl Timer-Interrupts eingetreten, setzt der Timer die bTo-Flagge. Ist bTo gesetzt, wird in der Hauptprogrammschleife die nächste Phase des Ablaufs bearbeitet. Hat der Ablauf rCntDwn Null erreicht, wird der Timer und die LED abgeschaltet. In diesem Zustand verharrt das Programm schlafend bis zum nächsten Tastendruck.

7.4.3 Das Programm

Das hier ist das Programm (zum Quellcode im asm-Format:

```
;
; ************************************
; * Taster mit INT0-Interrupt        *
; * (C)2016 by www.gsc-elektronic.net *
; ************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ------ Programmablauf ---------------
;
; Das Programm wartet auf negative Flanken
; am INT0-Eingang. Tritt eine solche ein,
; wird mittels der T-Flagge der folgende
; Ablauf gestartet:
; - Ein Abwaertzaehler wird auf 6 gesetzt.
; - Die LED wird auf Rot geschaltet.
; - Der Timer wird im CTC-Modus mit einem
;   Teiler von 150 im Compare-A-Register
;   gestartet.
; Beim Compare-A-Interrupt wird ein
; Softwarezaehler von 250 auf Null ab-
; waerts gezaehlt. Wird Null erreicht,
; wird der Zaehler mit 250 neu gestartet
; und die bTO-Flagge gesetzt.
; Ist die bTO-Flagge gesetzt, wird sie
; wieder rueckgesetzt und der Abwaerts-
; zaehler um Eins vermindert. Bei den
; folgenden Zaehlerstaenden werden diese
; Operationen vorgenommen:
; 5, 3: LED auf gruen
; 4, 2: LED auf rot
; 0: Ablauf beenden, Zaehler abschalten,
;    T-Flagge ruecksetzen
;
; --------- Register ------------------
; frei R0 .. R14
.def rSreg = R15 ; Sichern Statusregister
.def rmp = R16 ; Vielzweckregister
.def rimp = R17 ; Vielzweckregister Interrupts
.def rFlag = R18 ; Flaggenregister
  .equ bTo = 0 ; Timeout-Flagge Timer
.def rCntDwn = R19 ; Count Down
.def rCtcCnt = R20 ; CTC-Count-Down
; frei R21 .. R31
;
; --------- Ports ---------------------
.equ pOut = PORTB ; Output-Port
.equ pDir = DDRB  ; Direction-Port
.equ pIn  = PINB  ; Input-Port
.equ bARO = PORTB2 ; Anode LED rot Ausgabe
.equ bKRO = PORTB0 ; Kathode LED rot Ausgabe
.equ bPuO = PORTB1 ; Pull-Up Taster Ausgabe
.equ bARD = DDB2 ; Anode LED rot Richtung
.equ bKRD = DDB0 ; Kathode LED rot Richtung
;
; --------- Timing --------------------
; Clock       = 1200000 Hz
; Prescaler   = 8
; CTC-Teiler  = 150
; Counter     = 250
; ----------------------------
; Signaldauer = 0,250 Sekunden
;
; --------- Konstanten ----------------
.equ cCtcCmp = 149 ; CTC-Teiler - 1
.equ cCtcInt = 250 ; Int-Zaehler
;
; --------- Reset- und Int-Vektoren ---
.CSEG ; Assemblieren Flashspeicher (Code Segment)
.ORG 0 ; Adresse auf Null (Reset- und
; Interruptvektoren beginnen bei Null)
  rjmp Start ; Reset Vektor, Sprung zum Init
  rjmp Int0_Isr ; INT0-Int, aktiv
  reti ; PCINT-Int, nicht aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  rjmp Tc0CmpA ; TIM0_COMPA-Int, aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  reti ; ADC-Int, nicht aktiv
;
; Interrupt-Service-Routinen
;
; INT0 wird von fallenden Flanken am Tasten-
; eingang ausgeloest. Das T-Flag wird gesetzt.
;
Int0_Isr: ; INT0-ISR
  set ; setze T-Flagge
  reti ; zurueckkehren
;
; TC0-Compare-A wird bei Ueberschreitung des
; Compare-A-Wertes ausgeloest. Der Zaehler
; rCtcCnt wird abwaerts gezaehlt. Erreicht er
; Null, wird er neu gestartet und die bTO-Flagge
; gesetzt.
;
Tc0CmpA: ; TC0 Compare A ISR
  in rSreg,SREG ; sichere SREG
  dec rCtcCnt ; CTC-Zaehler abwaerts
  brne Tc0CmpA1 ; noch nicht Null
  sbr rFlag,1<<bTo ; Timeout-Flagge setzen
  ldi rCtcCnt,cCtcInt ; Neustart Zaehler
Tc0CmpA1:
  out SREG,rSreg ; SREG wieder herstellen
  reti
;
; Programmstart beim Reset
;
Start:
  ; Stapel-Init
  ldi rmp,LOW(RAMEND)
  out SPL,rmp;
  ; LED-Ausgaenge initiieren
  ldi rmp,(1<<bARD)|(1<<bKRD) ; Ausgaenge
  out pDir,rmp ; an Richtungsport
  ; Pull-Up-Widerstand einschalten
  sbi pOut,bPuO ; Pull-up an
  ; Anfangszustand setzen
  clr rCntDwn ; Countdown-Zaehler aus
  clt ; Busy-Flagge aus
  ; Schlafen ermoeglichen, Ext. Int
  ldi rmp,(1<<SE)|(1<<ISC01) ; Schlafen, Flanke
  out MCUCR,rmp
  ; INT0 ermoeglichen
  ldi rmp,1<<INT0 ; INT0-Interrupts an
  out GIMSK,rmp ; in General Interrupt Maske
  ; Interrupts einschalten
  sei ; Interrupt-Flagge setzen
Schleife:
  sleep
  nop
```

```
  tst rCntDwn ; Countdown-Zaehler = Null?
  brne Schleife1 ; nein, nicht starten
  brtc Schleife1 ; springe bei T = 0
  rcall Starten ; Sequenz starten
Schleife1:
  sbrc rFlag,bTo ; Springe, wenn CTC-Flagge aus
  rcall Countdown ; abwaerts bearbeiten
  rjmp Schleife
;
; Sequenz starten
; - Startwert Abwaertszaehler setzen
; - LED auf rot
; - bTO-Flagge loeschen
; - Timer 0 im CTC-Modus starten
;
Starten:
  ldi rCntDwn,6 ; Startwert erster Zyklus
  ldi rmp,(1<<bPuO)|(1<<bARO) ; rote LED, Pullup
  out pOut,rmp ; an Ausgangsport
  ldi rCtcCnt,cCtcInt ; Neustart CTC-Int-Zaehler
  cbr rFlag,1<<bTo ; Timeout-Flagge Null
  ldi rmp,cCtcCmp ; CTC-Teilerwert
  out OCR0A,rmp ; in Vergleichsregister A
  ldi rmp,1<<WGM01 ; TC0 als CTC mit Compare A
  out TCCR0A,rmp ; in Kontrollregister A
  ldi rmp,1<<CS01 ; Prescaler = 8
  out TCCR0B,rmp ; in Kontrollregister B
  ldi rmp,1<<OCIE0A ; Compare-A-Interrupt
  out TIMSK0,rmp ; in TC0-Int-Maske
  ret ; Fertig
;
; 250 ms vorbei, naechste Stufe
; - bTO-Flagge loeschen
; - rCntDwn vermindern
; - bei 0: LED ausschalten, Timer aus
; - bei 5 und 3: LED gruen schalten
; - bei 4 und 2: LED rot schalten
;
Countdown:
  cbr rFlag,1<<bTo ; Flagge ruecksetzen
  dec rCntDwn ; naechste Stufe abwaerts
  breq CountdownAus ; beendet, ausschalten
  cpi rCntDwn,5 ; Zyklus = 5?
  breq CountDownGreen ; Lampe auf Gruen
  cpi rCntDwn,3 ; Zyklus = 3?
  breq CountDownGreen
  ; LED auf rot schalten
  ldi rmp,(1<<bPuO)|(1<<bARO) ; Rot und Pullup
  out pOut,rmp ; an Ausgabeport
  ret
CountDownGreen: ; LED  auf Gruen
  ldi rmp,(1<<bPuO)|(1<<bKRO) ; Gruen und Pullup
  out pOut,rmp ; an Ausgabeport
  ret
CountdownAus: ; Beendet, alles ausschalten
  clr rmp ; Zaehler ausschalten
  out TCCR0B,rmp ; in Kontrollregister B
  out TIMSK0,rmp ; Timer-Int aus
  ldi rmp,1<<bPuO ; alles aus ausser Pullup
  out pOut,rmp ; LED aus
  clt ; T-Flagge aus
  ret
;
; Ende Quellcode
;
```

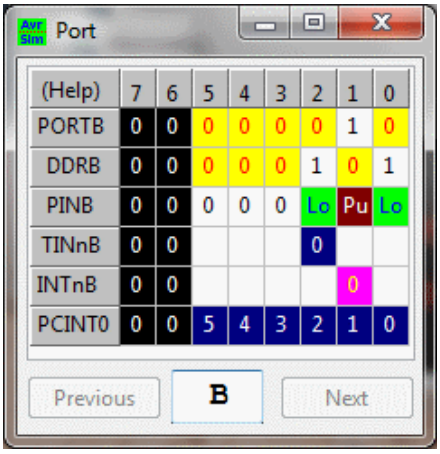
Zwei Instruktionen sind neu:

- CLR Register: setzt Registerinhalt auf Null und setzt die Z-Flagge,
- TST Register: prüft ob der Registerinhalt Null ist (dasselbe wie ein ODER mit sich selbst, "OR Register,Register").

7.4.4 Simulation der Vorgänge

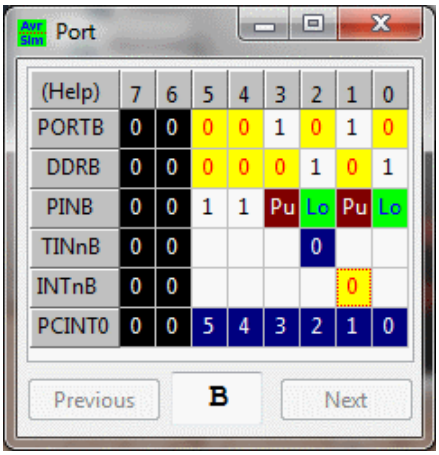
Um diese Abläufe zu simulieren füttern wir den Quellcode in avr_sim und steppen uns durch die einzelnen Sequenzen des Quellcodes.

Das ist der Zustand des Port B nach dem Init:

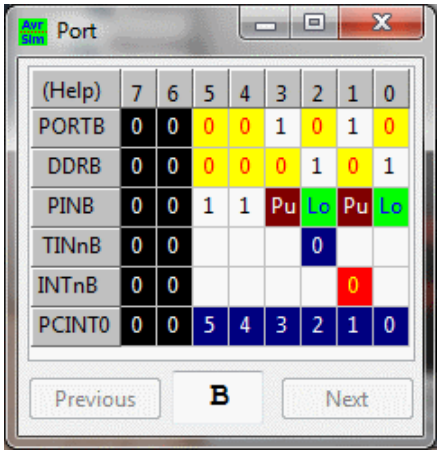


- Die zwei In-/Output-Pins (I/O-Pin) PB0 und PB2 sind als Ausgänge geschaltet, ihre Richtungsbits sind gesetzt ihr Treiber ist eingeschaltet und die Ausgänge sind Low. Die Duo-LED zwischen PB0 und PB2 ist daher abgeschaltet.
- An PB1, an dem der Taster angeschlossen ist, ist das Richtungsbit Null und der Portausgang PORTB1 Eins. Das schaltet den internen Pullup-Widerstand an. Beim Lesen des Portregisters PINB käme daher bei diesem Bit eine Eins heraus, solange die Taste nicht gedrückt wird.
- An PB1 ist auch der INT0-Eingang lokalisiert. Die Init-Software hat für fallende Flanken (das sind Übergänge von 1 auf 0) den INT0-Interrupt ermöglicht. Er wird ausgeführt, wenn der Taster geschlossen wird.

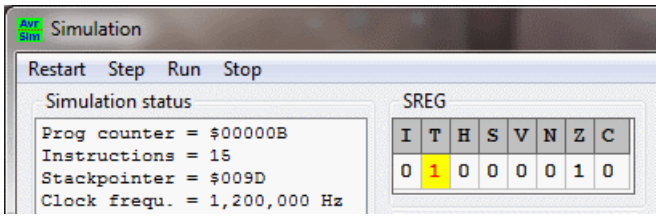
Nun geht der Herr Professor in den Idle-Schlafmodus und Port B wartet auf Signale an PB1.



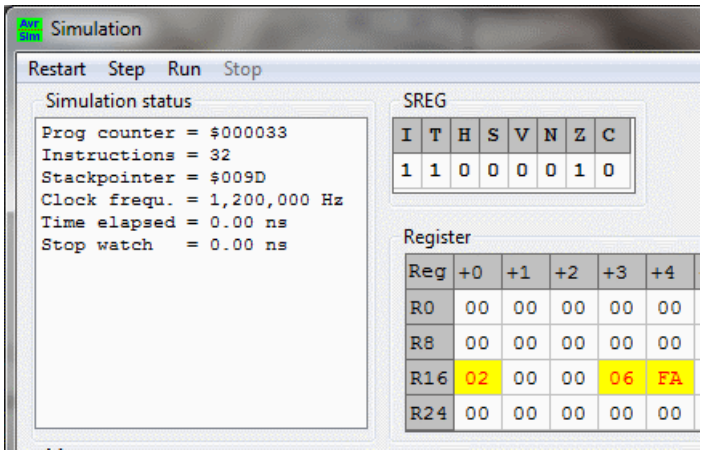
Durch Klicken auf das Portbit INT0 lösen wir eine INT0-Interruptanforderung aus. Sofern kein anderer Interrupt gerade bearbeitet wird, wird die Anforderung mit der nächsten Instruktion ausgeführt. Selbst wenn nun noch andere Interruptanforderungen vorlägen, käme INT0 trotzdem dran, denn er hat die höchste Priorität (er steht in der Vektorliste ganz oben).



Der Controller wacht damit auf, legt die derzeitige Ausführungsadresse auf dem Stapel ab und verzweigt zur INT0-Vektoradresse.

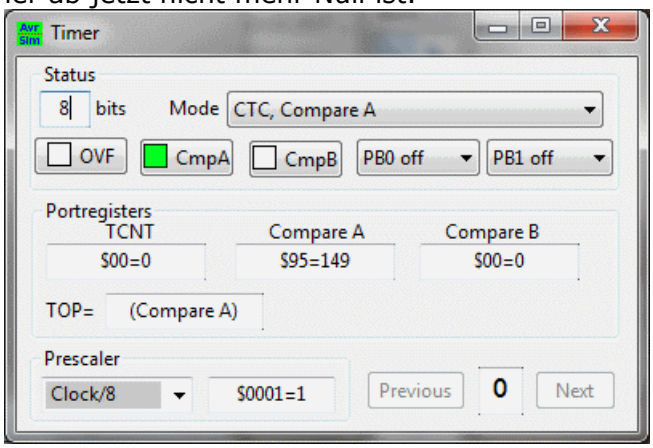


Die einzige Tätigkeit in der INT0-Serviceroutine ist das Setzen der T-Flagge im Statusregister SREG und startet damit den Ablauf.



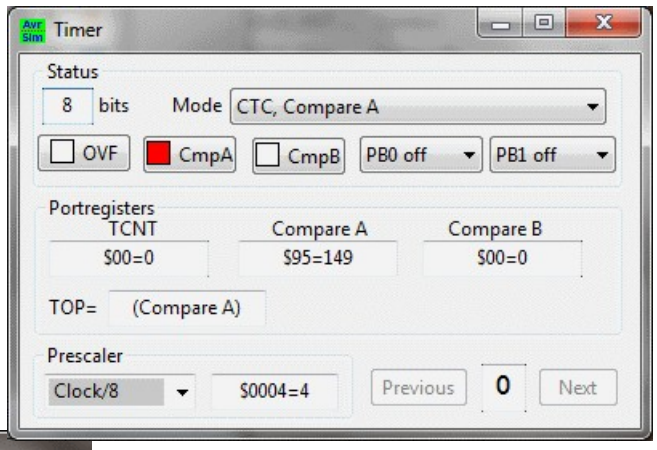
Nach dem Aufwachen bemerkt der Herr Professor, dass die T-Flagge gesetzt ist und er ruft mit *RCALL StartSeq* die Unterroutine auf, die den Ablauf startet. Durch den *RCALL* wird der gerade durch die Rückkehr vom INT0-Interrupt wieder erhöhte Stapelzeiger wieder um zwei Positionen niedriger, weil die Rückkehradresse auf dem Stapel abgelegt wird.

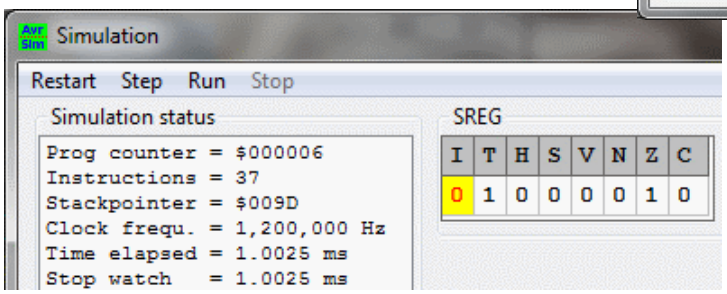
Die Unterroutine StartSeq hat zuerst geprüft, ob der Abwärtszähler auf Null steht. Wenn nicht, wird wieder schlafen gelegt. Ist er auf Null, wird der Abwärtszähler in R19 auf 6 gesetzt, um eine Sequenz zu beginnen. Der CTC-Zähler in R20 wurde auf 250 gesetzt, damit 250 CTC-Sequenzen ausgeführt werden. Man beachte, dass die T-Flagge gesetzt bleibt. Sie wird erst gelöscht, wenn die Zählsequenz auf Null läuft. Weitere INT0-Interrupts tun daher nichts, weil der Abwärtszähler ab jetzt nicht mehr Null ist.



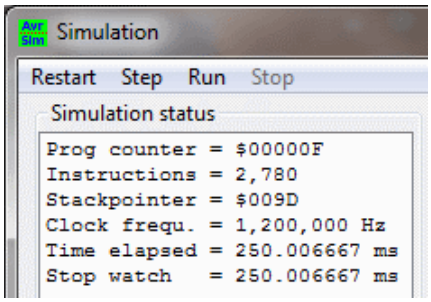
Die Routine StartSeq hat den Timer TC0 in den CTC-Modus gebracht. Sein TOP-Wert ist 149, daher startet der Timer nach 150 Zählimpulsen neu. Der Prozessortakt wird mit dem Prescaler durch 8 geteilt, daher dauert der CTC-Zyklus 8 * 150 / 1.200.000 = 1,00 ms.

Nachdem TC0 150 erreicht hat (und neu gestartet wurde) löst er einen Compare Match A aus und verzweigt zum CMP0A-Interrupt-Vektor.

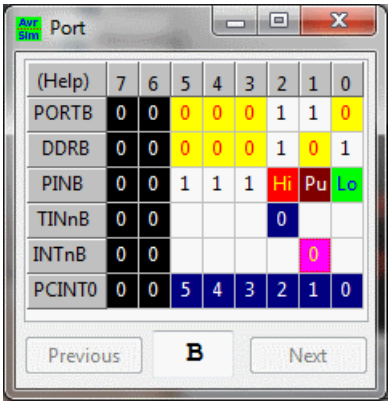




Wie vorherberechnet erfolgt der erste Compare Match Interrupt nach 1.0025 ms. Die "Überzeit" von 2,5 µs ist dadurch verursacht, dass die Rücksprungadresse auf dem Stapel abgelegt und das I-Flag im SREG gelöscht werden muss, bevor die Vektoradresse angesprungen werden konnte. Misst man von Sprung bis Sprung immer bei der gleichen Instruktion, verschwindet diese Extrazeit und die Millisekunde stimmt genau. Für das menschliche Auge ist aber der kleine Unterschied beim allerersten Zyklus ohnehin nicht sichtbar.

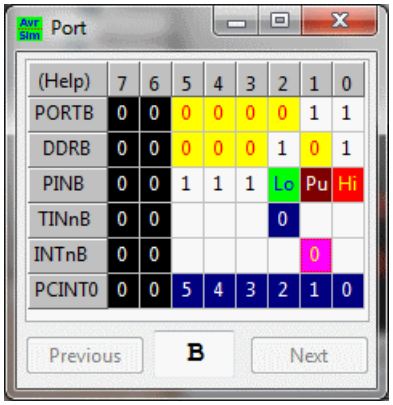


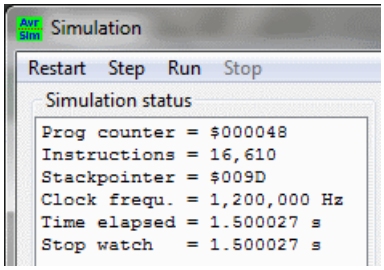
Hier sind 250 CTC-Zyklen abgelaufen. Die bTO-Timeout-Flagge wird nun gesetzt um das Time-Out zu signalisieren. Die abgelaufene Zeit seit dem Timer-Start is ziemlich genau.



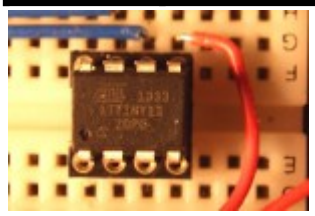
Nun, da die bTO-Flagge gesetzt ist, ist es Zeit die Farbe der Duo-LED von rot nach grün umzukehren. Hier wechseln die Bits PB0 und PB2 und die machen das schon.

Das wiederholt sich fünf mal, davon einmal ohne Farbwechsel.





Nach 1,5 Sekunden ist der ganze Zyklus vorbei, die T-Flagge wird gelöscht und es kann mit Tastendruck von vorne beginnen.



Lektion 8: Helligkeitsregelung mit AD-Wandler

Mit dieser Lektion wird der im Prozessor integrierte AD-Wandler in Betrieb gesetzt.

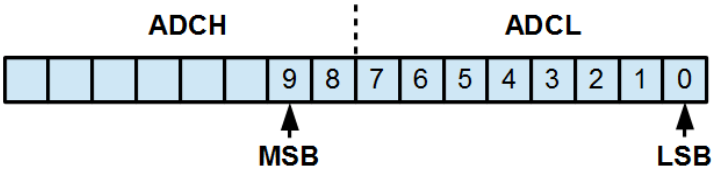
8.0 Übersicht

8.1 Einführung in die AD-Wandlung

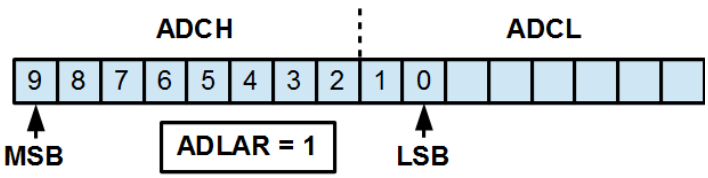
AD-Wandler machen folgendes:

- Sie speichern zu Beginn des Messzeitraums eine Spannung und entkoppeln sie dann von der Quelle ("Sample-and-Hold").
- Dann vergleicht er diese Spannung mit der Hälfte der Spannungsreferenz. Ist die Spannung kleiner als die Hälfte der Spannungsreferenz, ist das oberste Bit Null, andernfalls Eins.
- Der nächste Vergleichswert ist die Häfte der Spannungsreferenz plus ein Viertel der Spannungsreferenz (oberstes Bit = Eins) oder ein Viertel der Spannungsreferenz (oberstes Bit = Null). Ist die Spannung höher als diese Vergleichsspannung, ist das zweithöchste Bit Eins, andernfalls Null.
- So geht es weiter (mit einem Achtel, Sechzehntel, 32-stel, etc. der Spannungsreferenz), bis alle Bits festgestellt sind.

Der im ATtiny13 eingebaute AD-Wandler hat eine Auflösung von 10 Bits, kann also ein 1.024-stel oder ca. 1 Promille der verwendeten Referenzspannung messen. Das Ergebnis einer Messung folgt der Formel Ergebnis = (Messspannung * 1024) / Spannungsreferenz. Als Spannungsreferenz kann mit dem Bit REFS0 zwischen der Betriebsspannung (REFS0 = 0) und einer internen Spannungsreferenz von 1,1 V (REFS0 = 1) gewählt werden.



Das Ergebnis braucht den Port ADCL (niedrigere 8 Bit) und zwei Bits des Ports ADCH (die nicht benötigten 6 Bits ergeben beim Lesen immer Null).



Mit dem Bit ADLAR lässt sich dieses Verhalten ändern: mit ADLAR = 1 werden die 10 Bits linksbündig geschoben (LAR = Left Adjust Result). Im Port ADCH lassen sich dann die obersten 8 Bit des Ergebnisses lesen. Werden nur acht Bit benötigt kann auf das Lesen und Auswerten von ADCL verzichtet werden.

Bit	7	6	5	4	3	2	1	0	
	–	REFS0	ADLAR	–	–	–	MUX1	MUX0	ADMUX
Read/Write	R	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

In diesem Port befinden sich die Bits REFS0 und ADLAR. Die beiden Bits MUX1 und MUX0 wählen aus, von welchem Eingang das Messsignal kommt (ADC0 bis ADC3). Sollen mehrere Eingänge gemessen werden, muss der MUX-Port nacheinander gewählt und der Wandlungsprozess angestoßen werden.

Bit	7	6	5	4	3	2	1	0	
	–	–	ADC0D	ADC2D	ADC3D	ADC1D	AIN1D	AIN0D	DIDR0
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Der oder die Eingangspin(s), der/die als ADC-Eingang/-Eingänge verwendet wird/werden, wird/werden nicht als normaler Eingang verwendet. Um Strom zu sparen, können diese Eingangsstufen stillgelegt werden. Das geschieht, indem die entsprechenden Bits im Port DIDR0 Eins gesetzt werden. Man beachte die eigenwillige Nummerierung der Bits.

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Die gesamte restliche Steuerung des ADC erfolgt im Port ADCSRA. Darin bedeuten ADCPS2 bis 0 einen Vorteiler für das Taktsignal (000 = 2, 111 = 128), der die Dauer der Wandlungen steuert. Pro Wandlung werden 13 vorgeteilte Taktsignale benötigt. ADIE ermöglicht den Interrupt, wenn die Wandlung erfolgt ist. ADIF wird als Flagge gesetzt, wenn der Interrupt eintritt. ADATE ermöglicht den Autostart, wenn bestimmte Bedingungen eintreten (sonst muss der Start per Programm erfolgen). ADSC startet die Wandlung und bleibt solange Eins, bis die Wandlung abgeschlossen ist. ADEN schaltet den AD-Wandler ein.

Eine typische Sequenz zum Starten des ADC beim Kanal ADC3, rechtsbündig, mit der Betriebsspannung als Referenz, Interrupt und mit niedrigster Taktrate sieht dann so aus:

```
ldi rmp,(1<<MUX1)|(1<<MUX0) ; Kanal 3
out ADMUX,rmp ; in AD-Multiplexer
ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
out ADCSRA,rmp ; an ADC-Kontrollregister
```

Das waren die wichtigsten Verhältnisse, die zur Verwendung des ADC nötig sind.

Noch ein wichtiger Hinweis zum Auslesen des Ergebnisses: Der AD-Wandler blockiert beim Lesen des ADCL Neustarts bis das zugehörige ADCH ebenfalls gelesen ist. Auf diese Weise gehören ADCL und ADCH immer zum gleichen Datensatz. Liest man aber nur ADCL, dann startet der AD-Wandler keine weitere Wandlung mehr. Dasselbe passiert, wenn man zuerst ADCH und danach erst ADCL liest. Das führt dazu, dass der AD-Wandler nicht funktioniert und auch keine Interrupts mehr auslöst. Das kann einen zum Wahnsinn treiben. Im 8-Bit-ADLAR-Modus tritt das nicht auf, da dabei ja nur ADCH gelesen wird.

8.2 Einführung in die PCINT-Programmierung

In der vorherigen Lektion wurde die Taste am INT0-Eingang für Schaltaufgaben verwendet. Ist der Eingang INT0 nicht zu verwenden und blockiert oder muss eine weitere Taste überwacht werden, muss das mit dem PCINT-Interrupt erfolgen. Der PCINT funktioniert etwas anders als der INT0.

Bit	7	6	5	4	3	2	1	0	
	–	–	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Durch den PCINT sind alle Eingabepins überwachbar. Soll ein Kanal auf Pegelwechsel hin überwacht

werden, wird sein entsprechendes Bit in der PCINT-Maske PCMSK auf Eins gesetzt. Sind zwei oder mehr dieser Bits gesetzt, muss per Software festgestellt werden, an welchem der Pins ein Wechsel erfolgt ist. Im Gegensatz zum INT0 kann die zu überwachende Flankenart nicht vor-eingestellt werden, alle Wechsel lösen einen Int aus.

Bit	7	6	5	4	3	2	1	0	
	–	INT0	PCIE	–	–	–	–	–	GIMSK
Read/Write	R	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

Der entsprechende Interrupt muss im General Interrupt Mask Register durch Setzen von PCIE ermöglicht werden.

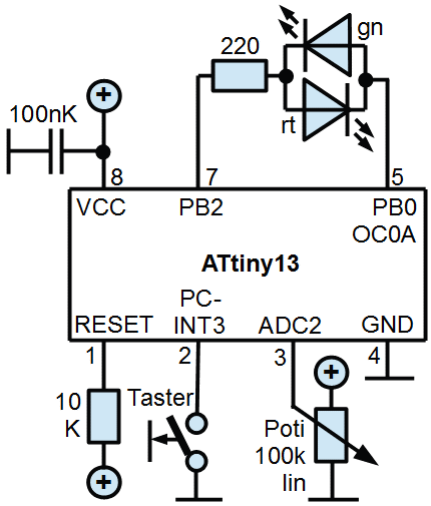
Das ist schon alles, was es braucht, um Überwachung jedes beliebigen Pins zu realisieren.

Top | Home | Einführung ADC | Einführung PCINT | Hardware | Hell1 | Hell2 | Hell3 | Hell4

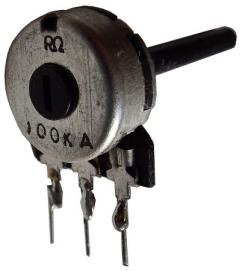
8.3 Hardware, Bauteile und Aufbau

8.3.1 Schaltung

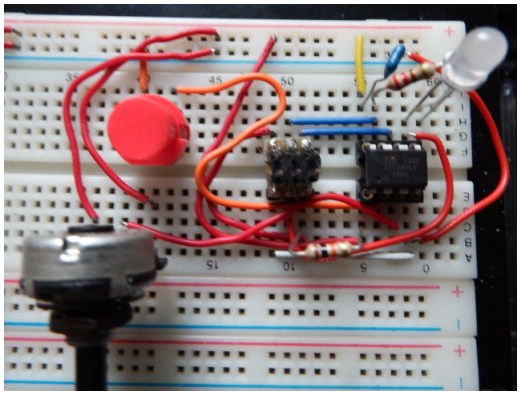
Die Duo-Led ist wie bei der voherigen Schaltung angeschlossen. Der Taster ist an Pin 2, den PCINT3 verlegt. Das Poti ist mit dem Schleifer an ADC2 an Pin 3 angeschlossen, er teilt die Betriebsspannung auf. Die ISP-Anschlüsse sind wie bisher angeschlossen, hier aber nicht eingezeichnet.



8.3.2 Bauteil Potentiometer



Das hier ist das Poti. Der mittlere Anschluss ist der Schleifer. Alle drei Anschlüsse sind etwas zu breit für die Breadboard-Löcher, deshalb kriegen sie kurze Anschlussdrähte angelötet.



8.3.3 Der Aufbau

Das hier ist der Aufbau. Die Platzierung der Bauteile ist unkritisch.

Top | Home | Einführung ADC | Einführung PCINT | Hardware | Hell1 | Hell2 | Hell3 | Hell4

8.4 Helligkeitsregelung

8.4.1 Aufgabe 1

In der ersten Aufgabe soll die Helligkeit der roten LED mit dem Poti einstellbar sein. Die Helligkeit soll steigen, wenn das Poti nach rechts gedreht wird (höhere Spannung).

8.4.2 Lösung

Es ist klar, dass dazu

- die LED vom Timer im Fast-PWM-Modus gesteuert werden muss,
- die Stellung des Poti regelmäßig gemessen werden muss,
- der Messwert, der zwischen 0 und 1023 liegen kann, auf den Vergleicher-Wertebereich von 0 bis 255 zurechtgestutzt werden muss, und
- der Vergleicherwert mit diesem umgerechneten Wert gesetzt werden muss.

Damit wir das Teilen des 10-Bit-Wertes, der vom ADC kommt, durch vier lernen, gehen wir erst mal nicht über das ADLAR-Bit des ADC.

Eigentlich ist diese Aufgabe auch linear lösbar, da wir nur

- den AD-Wandler starten,
- warten müssen, bis dieser fertig ist,
- den Wert teilen und in das Vergleichsregister schreiben müssen,

während der TC0-Timer im Fast-PWM-Mode das Ansteuern der LED übernimmt. Bei diesem Vorgehen wartet der Prozessor in einer Schleife 95% seiner Zeit auf die Fertigmeldung des ADC. Da das unintelligent, unästhetisch, Verschwendung von Strom und da bei den nachfolgenden Aufgaben sowieso der ADC-Interrupt vonnöten ist, machen wir das gleich im Interruptmodus.

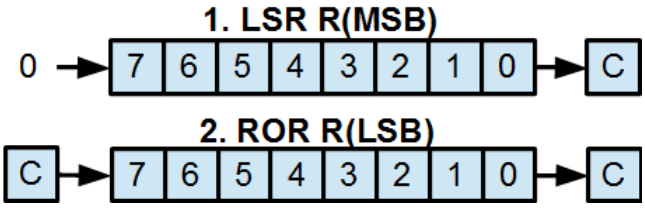
8.4.3 Programm 1

Das Programm gibt es hier (hier im Quellcode im asm-Format).

```
;
; ************************************
; * Helligkeitsregler mit ADC        *
; * (C)2016 by www.gsc-elektronic.net *
; ************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; --------- Programmablauf --------------
;
; Das Programm liest laufend AD-Wandler-
; werte ein, teilt sie durch vier und
; schreibt sie in das Vergleichsregister
; einer 8-Bit-PWM im Timer.
; Der Timer schaltet den Ausgang PB0
; beim Beginn des PWM-Zyklusses auf Null
; und bei Erreichen des Vergleichswerts
; auf Eins. Damit steigt die Helligkeit
; mit der Potentiometerstellung linear an.
;
; ---------- Register ---------------------
; frei: R0 .. R12
.def rAdcL = R13 ; LSB ADC-Ergebnis
.def rAdcH = R14 ; MSB dto.
.def rSreg = R15 ; Sicherungsregister
.def rmp   = R16 ; Vielzweckregister
.def rimp  = R17 ; Vielzweckregister Interrupts
;
; ---------- Ports ------------------------
.equ pOut = PORTB ; Ausgabeport
.equ pDir = DDRB  ; Richtungsport
.equ bRAO = PORTB2 ; Ausgabe Anode rote LED
.equ bRAD = DDB2   ; Richtung Anode rote LED
.equ bRKO = PORTB0 ; Ausgabe Kathode rote LED
.equ bRKD = DDB0   ; Richtung Kathode rote LED
;
; ---------- Timing -----------------------
; Prozessortakt = 1.200.000 Hz
; ADC-Vorteiler =       128
; ADC-Zyklen    =        13
; Messfrequenz  =       721 Hz
; TC0-Vorteiler =        64
; PWM-Stufen    =       256
; PWM-Frequenz  =        73 Hz
;
; ---------- Rest- und Interruptvektoren ---
.CSEG ; Assemblieren Flashspeicher (Code Segment)
.ORG 0 ; Adresse auf Null (Reset- und
; Interruptvektoren beginnen bei Null)
  rjmp Start ; Reset Vektor, Sprung zur
Initiierung
  reti ; INT0-Int, nicht aktiv
  reti ; PCINT-Int, nicht aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  rjmp AdcIsr ; ADC-Int, aktiv
;
; Interrupt-Service-Routinen
```

```
;
; ADC-Ready-Interrupt
; Wird vom ADC immer dann ausgeloest, wenn
; eine Wandlung erfolgt ist.
; Liest den Wandlerwert ein, teilt ihn
; durch vier und schreibt ihn in das
; Timer-Vergleichsregister A. Danach wird
; die naechste Wandlung angestossen.
;
AdcIsr: ; Interrupt Service Routine ADC
  in rSreg,SREG ; Sichern Statusregister
  ; ADC-Ergebnis lesen
  in rAdcL,ADCL ; Lese ADC-Ergebnis, LSB
  in rAdcH,ADCH ; dto., MSB
  ; ADC-Ergebnis durch 4 teilen
  lsr rAdcH ; unterstes Bit MSB in Carry
  ror rAdcL ; Carry in LSB schieben
  lsr rAdcH ; zweites Bit MSB in Carry
  ror rAdcL ; Carry in LSB schieben
  ; neuen PWM-Wert setzen
  out OCR0A,rAdcL ; in Compare Register A
  ; Neustart ADC (Teiler 128, Interrupt)
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rimp ; in ADC-Kontrollregister
  out SREG,rSreg ; Wiederherstellen Status
  reti
;
; ---------- Hauptprogramm-Init ------------
Start:
  ; Stapel anlegen
  ldi rmp,LOW(RAMEND) ; Stapelzeiger auf RAMEND
  out SPL,rmp ; in Stapelport
  ; LEDs konfigurieren und einschalten
  ldi rmp,(1<<bRAD)|(1<<bRKD) ; Portpins Ausgabe
  out pDir,rmp ; in Richtungsregister
  ldi rmp,(1<<bRAO)|(1<<bRKO) ; Ausgabe auf 1
  out pOut,rmp ; in Portregister
  ; Timer als 8-Bit-PWM
  ldi rmp,0x80 ; halbe Helligkeit
  out OCR0A,rmp ; in Compare Register A
  ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)|
(1<<WGM00)
  out TCCR0A,rmp ; in Kontrollregister A
  ldi rmp,(1<<CS01)|(1<<CS00) ; Vorteiler 64
  out TCCR0B,rmp ; in Kontrollregister B
  ; ADC-Wandler konfigurieren und starten
  ldi rmp,1<<MUX1 ; ADC-Signaleingang = ADC2
  out ADMUX,rmp ; in ADC-Multiplexer-Port
  ldi rmp,1<<ADC2D ; Disable Porttreiber-Hardware
  out DIDR0,rmp ; in Disable-Port
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; in ADC-Kontrollregister
  ; Schlafmodus und Interrupts
  ldi rmp,1<<SE ; Schlafen ermoeglichen
  out MCUCR,rmp ; in Kontrollregister
  sei ; Interrupts zulassen
; Programmschleife
Schlafen:
  sleep ; schlafen legen
  nop ; Nach Aufwecken durch Int
  rjmp Schlafen ; wieder schlafen legen
;
; Ende Quellcode
;
```

Neu ist hier LSR und ROR. Das müssen wir näher ansehen, es wird bei Manipulationen von 16-Bit-Zahlen öfter gebraucht.

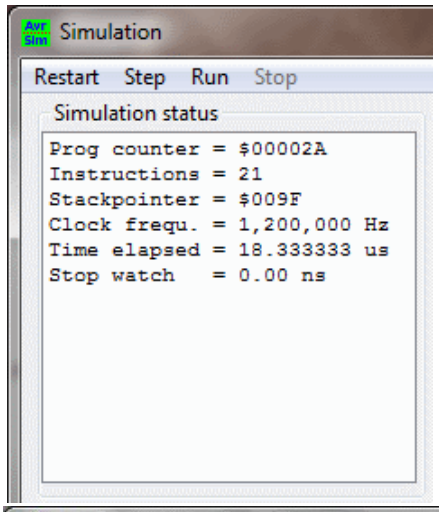


Aufgabe dabei ist es, die beiden niedrigsten Bits aus dem ADCH (MSB) von links her in das Register mit ADCL zu schieben. Das doppelte Rechtsschieben entspricht dem Teilen des Wertes in ADCH:ADCL durch vier, die beiden überschüssigen Bits aus ADCL fallen dabei rechts heraus. Da der ATtiny13-Prozessor kein 16-Bit-Schieben beherrscht, muss es selbst gestrickt werden. Dazu wird zunächst eine Null von links her in das MSB hineingeschoben, wobei gleichzeitig das rechts herausgeschobene Bit 0 in das Carry-Bit C des Statusregisters geschoben wird. Mit der Instruktion ROR (ROtate Right) wird der Inhalt des Carry-Bits in das LSB-Register von Links hereingeschoben. Macht man die Schritte 1 und 2 zweimal nacheinander, dann sind die beiden untersten MSB-Bits im durch vier geteilten unteren LSB angekommen.

Wieder sehen wir das gleiche Prinzip: jede Instruktion in Assembler löst genau einen Schritt des Prozessors aus. Die gesamte Operation hier braucht vier Schritte und vier einzelne Instruktionen. Das Ergebnis ist eine feinfühlige Helligkeitsregelung der roten LED mit dem Potentiometer.

Top	Home	Einführung ADC	Einführung PCINT	Hardware	Hell1	Hell2	Hell3	Hell4

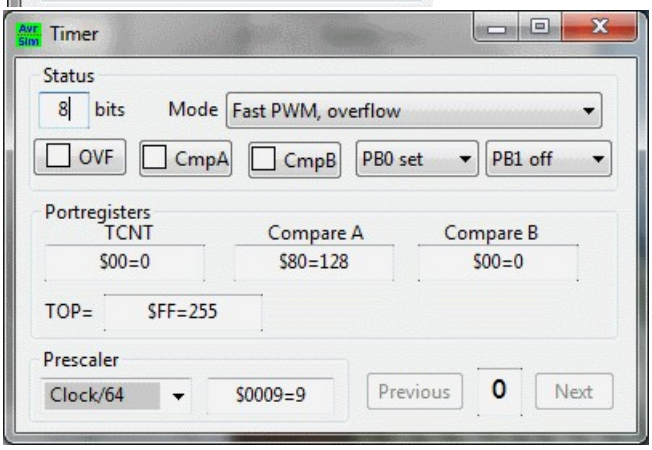
8.4.4 Simulation der Programmausführung



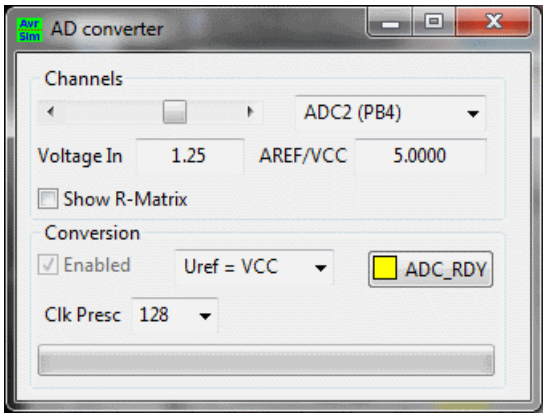
Die folgende Simulation verwendet wieder den Simulator [avr_sim](avr_sim).

Die Initiierungsphase des Programms ist durchlaufen, 18,3 µs sind vergangen.

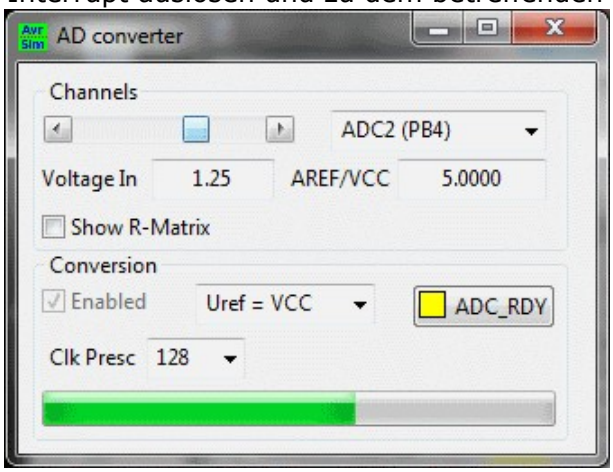
Der Ausgangsport wurde zum Betrieb der Duo-LED an PB2 und PB0 initiiert: die beiden Richtungsbits sind gesetzt, beide PORT-Bits sind Null und die LED ist ausgeschaltet.



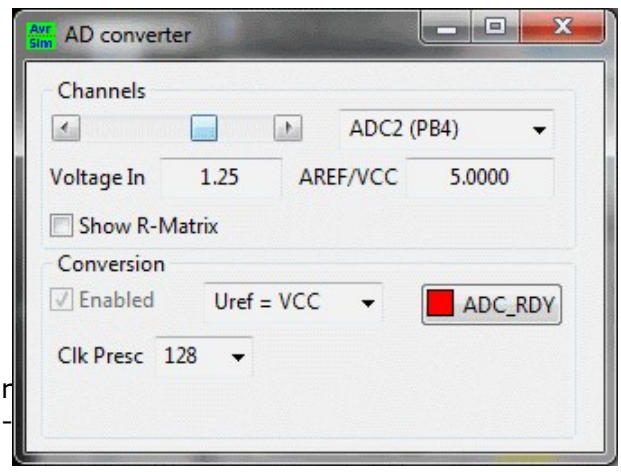
Der Timer TC0 ist in den Fast-PWM-Modus gebracht, mit Überlauf beim TOP-Wert 255 und Nullsetzen des I/O-Ports PB0 bei Compare Match A, das auf 128 (ungefähr halbe Intensität) eingestellt ist.



Der AD-Wandler-Kanal ADC2 ist ausgewählt. Er arbeitet mit der Betriebsspannung von 4,8 V als Referenzspannung, die Spannung an diesem Eingang wurde zur Simulation auf 1,25 V eingestellt. Die Wandlung wird vom Prozessortakt, geteilt durch 128, getaktet. Das gelb markierte Feld in der Interruptabteilung zeigt, dass der AD-Wandler am Ende des Umwandlungsprozesses den AD-Complete-Interrupt auslösen und zu dem betreffenden Interrupt-Vektor springen wird.



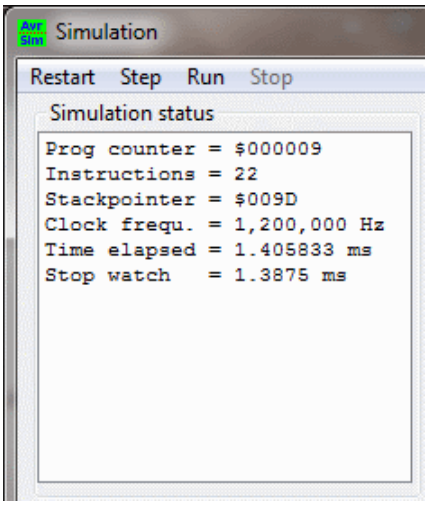
Die AD-Wandlung ist gestartet, der Umwandlungsfortschritt wird angezeigt.

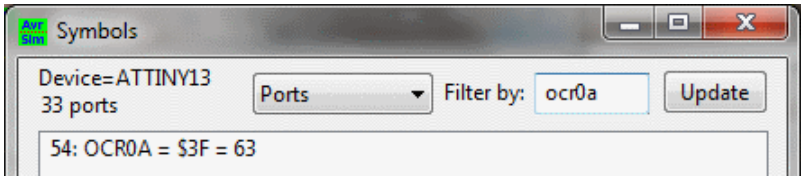


Nach kompletter Umwandlung hat der AD-Wandler die Interruptflagge gesetzt. Die Programmausfüh-

rung wird, falls kein anderer Interrupt gerade ausgeführt wird und falls kein anderer, höherwertigerer Interrupt ansteht, mit der nächsten Instruktion zum AD-Complete-Vektor verzweigen.

Bis zu diesem Interrupt sind seit dem Abschluss der Initphase 1,38 ms vergangen, in denen der Prozessor geschlafen hat. Da der ADC mit 1.200.000 / 128 = 9.375 Hz getaktet wird, also mit 0,107 ms pro Takt, und da die AD-Wandlung 13 Taktschritte erfordert sind die 1,38 ms korrekt.





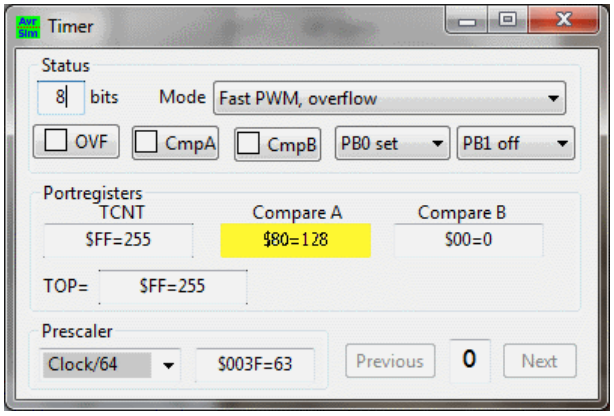
Das ADC-Ergebnis

$$1023 * 1.25 / 5.0 = 255$$

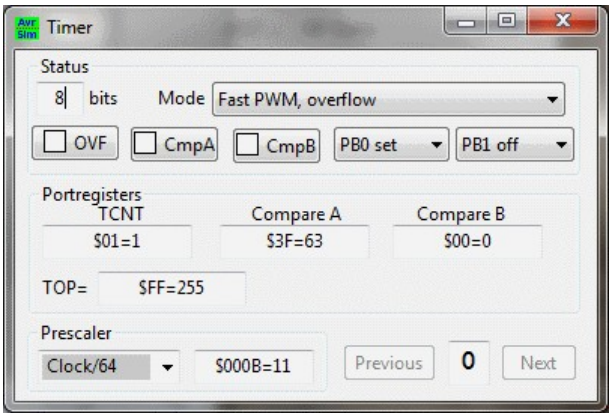
wird durch vier geteilt (= 63) und in das Compare-A-Portregister OCR0A des Timers geschrieben, was aber erst beim nächstfolgenden PWM-Zyklus die Pulsweite der Duo-LED und damit deren Helligkeit bestimmt.

Table 11-8. Waveform Generation Mode Bit Description

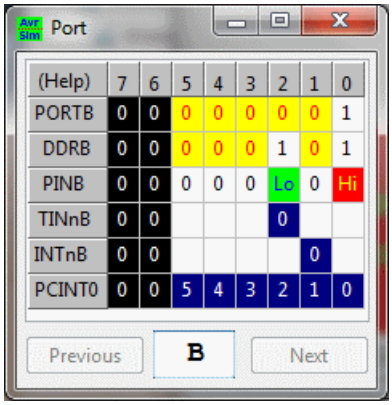
Mode	WGM2	WGM1	WGM0	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on(1)(2)
3	0	1	1	Fast PWM	0xFF	TOP	MAX



Das verzögerte Beschreiben von OCR0A stellt sicher, dass über jeden PWM-Zyklus hinweg der gleiche Vergleichswert verwendet wird und nicht mittendrin ein Wechsel des Wertes erfolgt, wann auch immer der AD-Wandler fertig ist.



Der Timerstand TOP wurde erreicht und der Vergleichswert wurde neu eingestellt:



Hier wurde der ursprüngliche erste Compare-Match-A-Wert von 128 erstmals überschritten. Der I/O-Pin PB0 wurde dadurch gesetzt, die LED wurde auf grün geschaltet, was sie auch bis zum Erreichen des TOP-Wertes bleibt.

6,89 ms sind seit der Initialisierung vergangen, wenn der erste Compare-Match-A erreicht ist. Das entspricht

$$64 * (128 + 1) / 1.200.000 \text{ s}$$

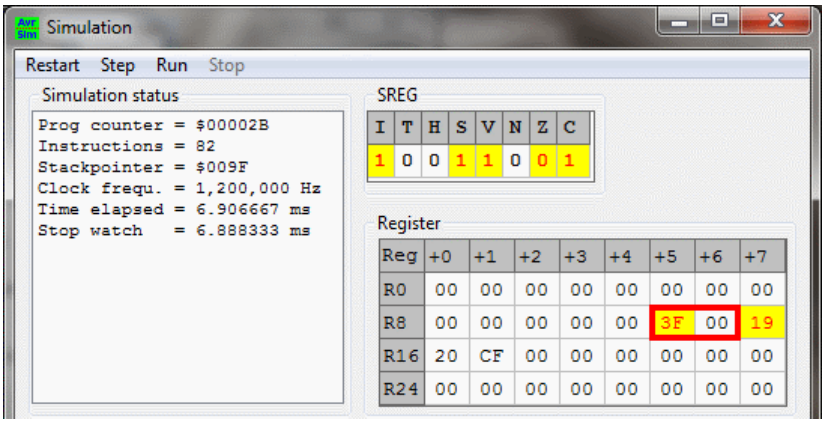
und ist korrekt.

Der gesamte PWM-Zyklus dauert

64 * 256 / 1.200.000 =
13,65 ms

oder einer PWM-Frequenz von 73,24 Hz. Das ist schnell genug dass es vom menschlichen Auge nicht als Flackern wahrgenommen werden kann.

Das Registerpaar R14:R13 enthält übrigens das AD-Wandler-Resultat geteilt durch vier.



Mit Hilfe der Simulation kann Quellcode genau getestet werden. Fehler im Quellcode werden gefunden und können korrigiert werden. Auch das gesamte Timing der Abläufe kann kontrolliert und, falls nötig, auch korrigiert werden.

Top	Home	Einführung ADC	Einführung PCINT	Hardware	Hell1	Hell2	Hell3	Hell4

8.5 Helligkeitsregelung mit Farbwechsel

8.5.1 Aufgabe 2

Jetzt soll nicht nur die rote, sondern auch die Helligkeit der grünen Lampe einstellbar sein. Die Umschaltung der Lampenfarbe soll mit dem Taster erfolgen. In beiden Fällen soll das Rechtsdrehen des Potis zu größerer Helligkeit führen.

8.5.2 Entprellen

Das Umschalten der LED-Farbe von Rot auf Grün und zurück, ist ein Leichtes: wenn das PORTB2-Bit oder PINB2 auf Eins steht, wird in PORTB2 Null geschrieben, und umgekehrt.

Wie wir das Drücken der Taste feststellen können, haben wir schon in der Einführung zum PCINT gesehen. Bei dieser Aufgabe wäre aber das Prellen des Tasters fatal. Je nach Anzahl an Schwarmimpulsen kommt beim Umschalten, immer wenn beim PCINT der Taster-Eingang auf Null steht, mal Rot, mal Grün raus. Das wäre unsauber, deshalb brauchen wir einen Mechanimus, der nach einem Tastendruck nachfolgende Impulse des Tasteneinganges unterdrückt. Dazu brauchen wir Folgendes:

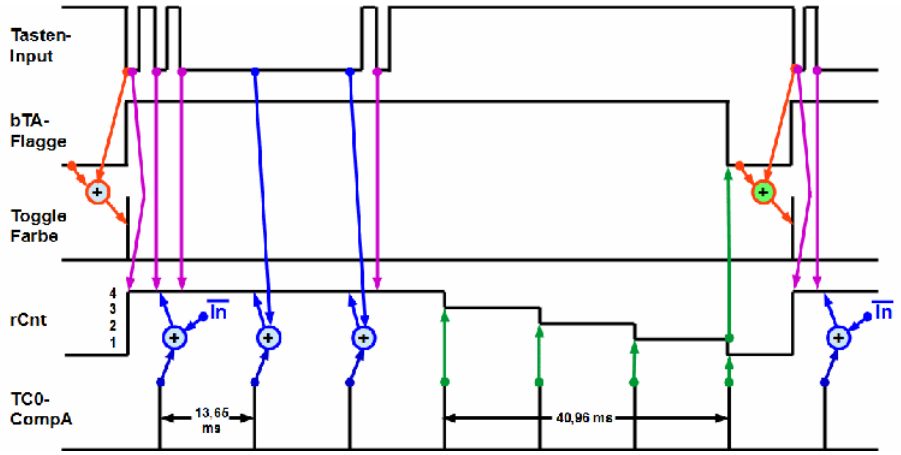
- eine Flagge, die nach erkanntem Tastendruck erst mal jede weitere Aktivität am Tasteneingang abwehrt, und
- die erst nachdem die Taste wieder losgelassen wurde und nach einer angemessenen Wartezeit zurückgesetzt wird.

Als Zeitmesser könnten wir wieder jede Menge Schleifenzeugs einbauen, aber das wäre unter dem bislang erreichten Programmierniveau. Für die Zeitmessung stehen uns schon zwei Quellen zur Verfügung:

- der ADC-Interrupt, der mit 721 Hz Wiederholfrequenz, also alle 1,39 ms zuschlägt,
- den TC0, der das PWM-Signal mit 73 Hz erzeugt und dessen Vergleicherwert im Fast-Modus bei 255 mit einem Interrupt alle 13,68 ms verwendbar wäre.

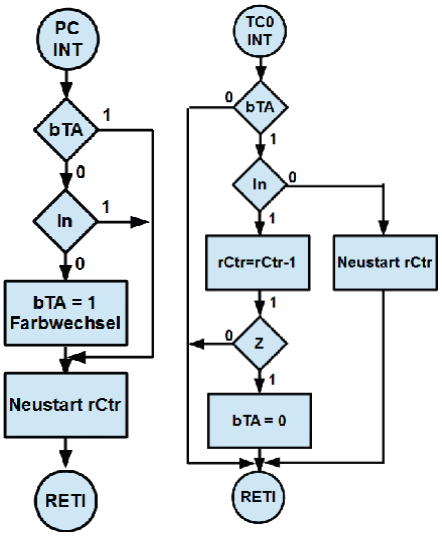
Da wir den ADC-Interrupt mit dieser zusätzlichen Aufgabe nicht behelligen wollen, nehmen wir dafür den TC0-CompareA-Interrupt. Da 13 ms für die Prell-Erkennung etwas zu kurz wäre, verwenden wir noch einen Abwärtszähler, um wenigstens 40 ms Ruhe am Tasteneingang festzustellen.

Damit kriegen wir die nachfolgenden Zeitbeziehungen zwischen Tastensignal, Inaktivitätsflagge bTA, Farbumschaltung und dem TC0-Interrupt.



Ist die Taste gedrückt (Tasteneingang geht auf Null), während die Flagge Null ist, wechselt die LED ihre Farbe und die Flagge wird gesetzt. Der Interruptzähler wird auf seinen Anfangswert 4 gesetzt.

Bei jedem nachfolgenden Null-Impuls am Eingangspin wird der Interruptzähler wieder auf vier gesetzt. das stellt sicher, dass die Flagge erst dann wieder gelöscht wird, wenn mindestens 40 ms oder vier Compare-Interrupts nach dem letzten Tasten-Null-Signal vergangen sind.



Bei jedem PWM-Interrupt wird geprüft, ob die bTA-Flagge gesetzt ist. Wenn nicht, gibt es nichts weiter zu tun. Wenn sie Eins ist, entscheidet der Zustand am Tasteneingang über den weiteren Ablauf. Ist dieser inaktiv (=Eins), dann wird der Zähler abwärts gezählt. Ist der Zähler daraufhin Null, wird die bTA-Flagge zurückgesetzt, der Wartezeitraum ist damit abgelaufen. Ist der Tastereingang aktiv (=Null), dann wird der Wartezeitraum wieder auf vier gesetzt.

Damit sind die Ablaufdiagramme für die beiden Interrupts klar. Das sind sie. Durch die Zusammenarbeit beider ist sichergestellt, dass mit der Taste kein Kuddelmuddel passiert.

8.5.3 Programm 2

Jetzt nutzen wir das ADLAR-Bit des ADC, um die Schieberei und Rotiererei des 10-Bit-Wertes aus dem ADC loszuwerden. Außerdem nutzen wir bei der Umschaltung der Farben die Umpolarisierung des OC0A-Ausgangs, um rot und grün korrekt anzusteuern.

Das hier ist das Programm (zum Quellcode im asm-Format geht es hier).

```
;                                              ; verursachen koennen.
;*********************************************** ;
; * Helligkeitsregler rot/gruen mit ADC und Taste ; ---------- Register ---------------------
; * (C)2016 by www.gsc-elektronic.net            ; frei: R0 .. R14
; *********************************************** .def rSreg = R15 ; Sicherungsregister
;                                                .def rmp   = R16 ; Vielzweckregister
.NOLIST                                          .def rimp  = R17 ; Vielzweckregister Interrupts
.INCLUDE "tn13def.inc"                           .def rFlag = R18 ; Flaggenregister
.LIST                                              .equ bTA = 0 ; Taste-Aktiv-Flagge
;                                                .def rCnt  = R19 ; Zaehler fuer Prellunterdrück
; ---------- Programmablauf --------------       ; frei: R20 .. R31
;                                                ;
; Liest laufend die Spannung am Potentiometer-     ; ---------- Ports ------------------------
; eingang mit dem ADC ein und schreibt die         .equ pOut = PORTB ;  Ausgabeport
; obersten 8 Bit des Ergebnisses in das            .equ pDir = DDRB  ;  Richtungsport
; Vergleichsregister des Timers.                   .equ pIn  = PINB  ;  Leseport
; Der Timer steuert die Helligkeit der LED         .equ bRAO = PORTB2 ; Ausgabe Anode rote LED
; im 8-Bit-PWM-Modus.                              .equ bRAD = DDB2   ; Richtung Anode rote LED
; Wird die Taste gedrueckt, wechselt die Duo-      .equ bRAI = PINB2  ; Lesen Anode rote LED
; LED ihre Farbe von rot nach gruen und zurueck.   .equ bRKO = PORTB0 ; Ausgabe Kathode rote LED
; Durch entsprechende Massnahmen wird sicher-      .equ bRKD = DDB0   ; Richtung Kathode rote LED
; gestellt, dass nachfolgende Tastensignale        .equ bTaO = PORTB3 ; Pullup-Bit Taste
; durch Prellen keine erneute Farbumkehr           .equ bTaI = PINB3  ; Inputport Taste
```

```
.equ bTaE = PCINT3 ; PCINT-Maskenbit
.equ bAdD = ADC2D  ; Input-Disable ADC-Pin
;
; ---------- Timing ----------------------
; Prozessortakt = 1.200.000 Hz
; ADC-Vorteiler =       128
; ADC-Zyklen    =        13
; Messfrequenz  =       721 Hz
; TC0-Vorteiler =        64
; PWM-Stufen    =       256
; PWM-Frequenz  =        73 Hz
; TC0-Int-Zeit  =        13,65 ms
;
; ----------- Konstanten ------------------
.equ cTakt  = 1200000 ; Prozessortakt
.equ cPresc = 64 ; TC0-Vorteiler
.equ cPwm   = 256 ; PWM-Taktdauer
.equ cPrell = 50 ; ms Prellzeit Taster
.equ cFPwm  = cTakt/cPresc/cPwm ; Frequenz PWM Hz
; Berechnung mit Rundung
.equ cTPwm  = (1000+cFPwm/2)/ cFPwm ; Taktdauer
  ; PWM in ms
.equ cCnt   = (cPrell+cTPwm/2) / cTPwm
  ; Zaehlimpulse Prellen
;
; ---------- Rest- und Interruptvektoren ---
.CSEG ; Assemblieren in den Flashspeicher (Code
    ; Segment)
.ORG 0 ; Adresse auf Null (Reset- und
    ; Interruptvektoren beginnen bei Null)
  rjmp Start ; Reset Vektor, Sprung zur
    ; Initiierung
  reti ; INT0-Int, nicht aktiv
  rjmp PcIntIsr ; PCINT-Int, aktiv
  rjmp TC0OvfIsr ; TIM0_OVF, aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  rjmp AdcIsr ; ADC-Int, aktiv
;
; Interrupt-Service-Routinen, mit Anzahl Takte
;
; PCINT wird von der sich schliessenden Taste
; ausgeloest.
; Durch Abfrage der Flagge bTA wird sicher-
; gestellt, dass die Ausloesung nur erfolgt,
; wenn die Karenzzeit vorausgehender Tasten-
; ereignisse abgelaufen ist. Tastensignale
; innerhalb der Karenzzeit verlaengern die
; Karenzzeit.
; Ist die Karenzzeit abgelaufen, wird die
; Farbe der LED umgekehrt indem die Polari-
; taeten der PWM-Ausgaenge OC0A und OC0B
; umgekehrt werden.
;
PcIntIsr: ; Interrupt-Service-Routine PCINT
  in rSreg,SREG ; Sichern Statusregister, +1 = 1
  sbrc rFlag,bTA ; Uerspringe wenn Blockade
    ; inaktiv, +1/2 = 2/3
  rjmp PcIntIsrSet ; Setze Zaehler neu, +2 = 4
  sbic pIn,bTaI ; Ueberspringe wenn Taste = 0
    ; +1/2 = 4/5
  rjmp PcIntIsrSet ; Setze Zaehler neu ; +2 = 6
  sbr rFlag,1<<bTA ; Flagge setzen, +1 = 6
  sbic pIn,bRAI ; Ueberspringe wenn Anode Rot =
    ; 0, +1/2 = 7/8
  rjmp PcIntIsrGruen ; Kathode Gruen = 1, +2 = 9
  sbi pOut,bRAO ; Setze LED Gruen, +2 = 10
  ldi rimp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)|
(1<<WGM00) ; Fast/clr on match, +1 = 11
  out TCCR0A,rimp ; an Timer-Kontrollport A, +1 =
    ; 12
  rjmp PcIntIsrSet ; Setze Zaehler neu, +2 = 14
PcIntIsrGruen:
  cbi pOut,bRAO ; Setze LED Rot, + 2 = 11
  ldi rimp,(1<<COM0A1)|(1<<WGM01)|(1<<WGM00)
    ; Fast/set on match, +1=12
  out TCCR0A,rimp ; an Timer-Kontrollport A, +1 =
    ; 13
PcIntIsrSet:
  ldi rCnt,cCnt ; starte Zaehler neu, + 1 =
    ; 5/7/15/14
  out SREG,rSreg ; Wiederherstellen SREG, +1 =
    ; 6/8/16/15
  reti ; Rueckkehr vom Interrupt, + 4 =
    ; 10/12/20/19
;
; TC0 Overflow
; Wird vom Timer TC0 am Ende des PWM-Zyklusses
; ausgeloest.
; Vermindert bei gesetzter bTA-Flagge den Zaehler
; fuer die Karenzzeit und setzt bei Erreichen von
; Null die bTA-Flagge zurueck. Ist der Tastenein-
; gang Low, wird die Karenzzeit auf vier gesetzt.
;
TC0OvfIsr: ; Interrupt Service Routine TC0-
    ;Overflow
  in rSreg,SREG ; Sichern Statusregister, +1 = 1
  sbrs rFlag,bTa ; Ueberspringe wenn bTA-Flagge
   ; 1, +1/2 = 2/3
  rjmp TC0OvfIsrRet ; Beenden, +2 = 4
  sbis pIn,bTaI ; Ueberspringe wenn Taste = 1,
    ; +1/2 = 4/5
  rjmp TC0OvfIsrNeu ; Neustart Zaehler, +2 = 6
  dec rCnt ; vermindere Zaehler, + 1 = 6
  brne TC0OvfIsrRet ; noch nicht Null,
    ; zurueckkehren, +1/2 = 7/8
  cbr rFlag,1<<bTa ; bTa-Flagge loeschen, +1 = 8
  rjmp TC0OvfIsrRet ; Zurueck, +2 = 10
TC0OvfIsrNeu:
  ldi rCnt,cCnt ; Neustart Abwaertszaehler, +1= 7
TC0OvfIsrRet:
  out SREG,rSreg ; Wiederherst. SREG, +1=5/9/8
  reti ; Rueckkehr vom Interrupt, + 4 = 9/13/12
;
; ADC-Ready Int
; Liest die obersten 8 Bit des ADC-Werts ein,
; schreibt ihn in das Vergleichsregister A
; des Timers und startet die AD-Wandlung neu.
;
AdcIsr: ; Interrupt Service Routine ADC
  ; ADC-Ergebnis lesen
  in rimp,ADCH ; Lese ADC-Ergebnis MSB, +1 = 1
  ; neuen PWM-Wert setzen
  out OCR0A,rimp ; in Compare Register A, +1 = 2
  ; Neustart ADC (Teiler 128, Interrupt)
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0); +1=3
  out ADCSRA,rimp ; in ADC-Kontrollregister,+1= 4
  reti ; +4 = 8
;
; ---------- Hauptprogramm-Init ------------
Start:
  ; Stapel anlegen
  ldi rmp,LOW(RAMEND) ; Stapelzeiger auf RAMEND
  out SPL,rmp ; in Stapelport
  ; LEDs konfigurieren und einschalten
  ldi rmp,(1<<bRAD)|(1<<bRKD) ; Portpins auf
    ; Ausgabe
  out pDir,rmp ; in Richtungsregister
  ldi rmp,(1<<bRKO)|(1<<bTaO) ; LED auf Rot,
    ; Tasten-Pullup
  out pOut,rmp ; in Port-Outputregister
  ; Timer als 8-Bit-PWM
  ldi rmp,0x80 ; halbe Helligkeit
  out OCR0A,rmp ; in Compare Register A
  ldi rmp,(1<<COM0A1)|(1<<WGM01)|(1<<WGM00)
  out TCCR0A,rmp ; in Kontrollregister A
  ldi rmp,(1<<CS01)|(1<<CS00) ; Vorteiler 64
  out TCCR0B,rmp ; in Kontrollregister B
  ldi rmp,1<<TOIE0 ; Overflow Interrupt
  out TIMSK0,rmp ; in Timer-Int-Maske
```

```
  ; ADC konfigurieren: Left adjust, Signaleingang
    ; = ADC2
  ldi rmp,(1<<ADLAR)|(1<<MUX1) ; in Register
  out ADMUX,rmp ; in ADC-Multiplexer-Port
  ; ADC-Eingangstreiber abschalten
  ldi rmp,1<<bAdD ; Disable Porttreiber
  out DIDR0,rmp ; in Disable-Port
  ; ADC einschalten und starten
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; in ADC-Kontrollregister
  ; PCINT-Enable
  ldi rmp,1<<bTaE ; Tasten-Interrupts
  out PCMSK,rmp ; in PCINT-Maskenregister
  ldi rmp,1<<PCIE ; PCINT einschalten
  out GIMSK,rmp ; in Extern Interrupt Register
  ; Schlafmodus und Interrupts
  ldi rmp,1<<SE ; Schlafen ermoeglichen
  out MCUCR,rmp ; in Kontrollregister
  sei ; Interrupts zulassen
; Programmschleife
Schlafen:
  sleep ; schlafen legen
  nop ; Nach Aufwecken durch Int
  rjmp Schlafen ; wieder schlafen legen
;
; Ende Quellcode
;
```

Zunächst ergibt die genaue Zählung der Anzahl Takte in den Interrupt-Service-Routinen maximal 20 Takte oder 20/1,2 = 16,7 µs Dauer. Das bleibt deutlich unter den Millisekunden, die typisch für das Prellen sind. Es bleibt auch weit unter der Dauer für einen PWM-Zyklus des Timers. Die Routinen sind auch kurz genug, so dass eine Verlegung von Teilen in das Hauptprogramm unnötig ist.

Neue Instruktionen kommen darin nicht vor.

Wenn wir uns bei diesen Abläufen vorstellen, dies alles mittels genau konstruierter Schleifen zu erledigen, wird uns in jedem Fall schummrig vor Augen. Deshalb sind Interrupts so einfach wie hilfreich.

Noch ein Wort zur Programmierung der Aufgabe in Hochsprachen: Die ist ziemlich unmöglich, weil wir schon beim Timing in immense Schwierigkeiten laufen. Das Verweben zweier Interrupt-Service-Routinen miteinander dürfte oberste Programmierkunst nötig machen. Hochsprachen sind eben in Wirklichkeit nicht einfacher und für komplexe Aufgaben geeigneter als Assembler. Sie sind einfach nur ganz weit weg von der verfügbaren Hardware und super-schwerfällig. Wenn es flexibel zugehen muss, eröffnet Assembler eben doch viel mehr Möglichkeiten.

Top	Home	Einführung ADC	Einführung PCINT	Hardware	Hell1	Hell2	Hell3	Hell4

8.6 Helligkeitsregelung dynamisch

8.6.1 Aufgabe 3

Bei dieser Aufgabe steigt bzw. fällt die Helligkeit der LED automatisch. Die Geschwindigkeit, mit der das geschieht, wird durch das Poti bestimmt. Der Taster schaltet zwischen roter und grüner LED um. Die Umschaltung von steigender auf fallende Helligkeit und umgekehrt erfolgt bei maximaler bzw. minimaler Helligkeit.

8.6.2 Lösung

Zusätzlich muss im TCO-Interrupt, der nach Ablauf eines PWM-Zyklusses eintritt, nicht nur das Timing des Tastendrucks ausgewertet werden. Es muss auch ausgewertet werden, ob der Vergleichswert erhöht/vermindert werden muss. Damit die Geschwindigkeit geregelt werden kann, muss hier ein Abwärtszähler hinzugebastelt werden. Sein Anfangswert bestimmt darüber, nach wieviel PWM-Zyklen diese Erhöhung/Erniedrigung erfolgt. Dieser Anfangswert wird vom AD-Wandler gesetzt, indem das 8-Bit-MSB des ADC (mit gesetztem ADLAR-Bit) durch 16 geteilt wird. Damit bei einem Wert von Null keine ewige Verlängerung eintritt (ein Abwärtszähler mit Null zu Beginn würde erst nach 256 Durchläufen wieder Null werden), wird zu dem Wert einfach eine Eins dazugezählt. Der Anfangswert kann daher zwischen 1 und 16 liegen.

Aus Erfahrung dauert so ein Durchlauf trotzdem noch ewig lang, weil er 256 einzelne Stufen durchläuft. Wir erhöhen daher den Zählertakt um das Achtfache (Vorteiler = 8 statt 64). Das ergibt vernünftige und merkliche Dynamik bei der LED.

Ansteigende und fallende Helligkeit lässt sich durch Verändern des OCR0A-Wertes realisieren. Die Feststellung, ob bei ansteigender Helligkeit der OCR0A-Wert von 255 auf Null und bei fallender Helligkeit der OCR0A-Wert von Null auf 255 wechseln würde und dann eine Richtungsumkehr vorgenommen werden muss, sind etwas längerwierige Entscheidungen. Dieser Teil wird daher in die Hauptprogrammschleife verlegt, Bearbeitungsbedarf wird mit einer Flagge signalisiert.

In der Hauptprogrammschleife wird auch die Farbe der LED eingestellt. Diese steht einfach als ein Bit im Flaggenregister. Der Tastendruck kehrt dieses Bit um, der Farbwechsel wird aber erst nach Ablauf der vorgewählten Anzahl PWM-Durchläufe und durch die Bearbeitung im Hauptprogramm an die LED weitergegeben.

Damit sind alle Elemente des Programmes geklärt und es kann ans Kodieren gehen.

8.6.3 Programm 3

Das hier ist das Programm (zum Quellcode im asm-Format geht es hier).

```
;                                            ; PWM-Stufen    =       256
; ******************************************* ; PWM-Frequenz  =       585 Hz
; * Helligkeitsregler rot/gruen mit ADC und Taste ; TC0-Int-Zeit  =         1,7 ms
; * (C)2016 by www.gsc-elektronic.net         ; Zaehlzeit min =         1,7 ms
; ******************************************* ; Zaehlzeit max =        28,9 ms
;                                            ; Auf-/Ab-Zyklus (min)=   0,88 s
.NOLIST                                      ;                (max)=  14,8 sec
.INCLUDE "tn13def.inc"                       ;
.LIST                                        ; ----------- Konstanten ------------------
;                                            .equ cTakt  = 1200000 ; Prozessortakt
; -------- Programmablauf ------------       .equ cPresc = 8 ; TC0-Vorteiler
;                                            .equ cPwm   = 256 ; PWM-Taktdauer
; Eine Duo-LED wird mit dem Taster von       .equ cPrell = 50 ; ms Prellzeit Taster
; rot auf gruen umgestellt und zurueck       .equ cFPwm  = cTakt/cPresc/cPwm ; Frequenz PWM Hz
; (mit Prellunterdrueckung). Die Hellig-     ; Berechnung mit Runden
; keit nimmt dabei zu und ab. Die Ge-        .equ cTPwm  = (1000+cFPwm/2)/ cFPwm ; Taktdauer
; schwindigkeit der Zu- und Abnahme wird         ; PWM in ms
; von der Potentiometerstellung bestimmt.    .equ cCnt   = (cPrell+cTPwm/2) / cTPwm
;                                                ; Zaehlimpulse Prellen
; ---------- Register ---------------------  ;
; frei: R0 .. R12                            ; ---------- Rest- und Interruptvektoren ---
.def rCyc  = R13 ; Zykluszaehler             .CSEG ; Assemblieren in den Flashspeicher (Code
.def rAdc  = R14 ; ADC-Ergebnis                  ; Segment)
.def rSreg = R15 ; Sicherungsregister        .ORG 0 ; Adresse auf Null (Reset- und
.def rmp   = R16 ; Vielzweckregister              ; Interruptvektoren beginnen bei Null)
.def rimp  = R17 ; Vielzweckregister Interrupts    rjmp Start ; Reset Vektor, Sprung zur
.def rFlag = R18 ; Flaggenregister                 ; Initiierung
  .equ bTA = 0 ; Taste-Aktiv-Flagge          reti ; INT0-Int, nicht aktiv
  .equ bCy = 1 ; Behandlungsflagge Zyklusende rjmp PcIntIsr ; PCINT-Int, aktiv
  .equ bAb = 2 ; Abwaertszaehlen             rjmp TC0OvfIsr ; TIM0_OVF, aktiv
  .equ bGn = 3 ; LED gruen gewaehlt          reti ; EE_RDY-Int, nicht aktiv
.def rCnt  = R19 ; Zaehler fuer              reti ; ANA_COMP-Int, nicht aktiv
    ;  Prellunterdrueckung                   reti ; TIM0_COMPA-Int, nicht aktiv
; frei: R20 .. R31                           reti ; TIM0_COMPB-Int, nicht aktiv
;                                            reti ; WDT-Int, nicht aktiv
; ---------- Ports ------------------------   rjmp AdcIsr ; ADC-Int, aktiv
.equ pOut = PORTB ;  Ausgabeport             ;
.equ pDir = DDRB  ;  Richtungsport           ; Interrupt-Service-Routinen, mit Anzahl Takte
.equ pIn  = PINB  ;  Leseport                ;
.equ bRAO = PORTB2 ; Ausgabe Anode rote LED  ; PCINT Interrupt
.equ bRAD = DDB2   ; Richtung Anode rote LED ; Wird vom Tastendruck ausgeloest.
.equ bRAI = PINB2  ; Lesen Anode rote LED    ; Ist die bTA-Flagge gesetzt, wird die Karenz-
.equ bRKO = PORTB0 ; Ausgabe Kathode rote LED ; zeit neu gestartet. Ist der Tasteneingang
.equ bRKD = DDB0   ; Richtung Kathode rote LED ; Eins, ebenfalls. Wenn beides nicht, wird die
.equ bTaO = PORTB3 ; Pullup-Bit Taste        ; bTA-Flagge gesetzt und die Farbflagge inver-
.equ bTaI = PINB3  ; Inputport Taste         ; tiert.
.equ bTaE = PCINT3 ; PCINT-Maskenbit         ;
.equ bAdD = ADC2D  ; Input-Disable ADC-Pin   PcIntIsr: ; Interrupt-Service-Routine PCINT
;                                              in rSreg,SREG ; Sichern Statusregister, +1 = 1
; ---------- Timing -----------------------    sbrc rFlag,bTA ; Uerspringe wenn Blockade
; Prozessortakt = 1.200.000 Hz                   ; inaktiv, +1/2 = 2/3
; ADC-Vorteiler =       128                    rjmp PcIntIsrSet ; Setze Zaehler neu, +2 = 4
; ADC-Zyklen    =        13                    sbic pIn,bTaI ; Uebersprinege wenn Taste = 0
; Messfrequenz  =       721 Hz                    ; +1/2 = 4/5
; TC0-Vorteiler =         8                    rjmp PcIntIsrSet ; Setze Zaehler neu ; +2 = 6
```

```
  sbr rFlag,1<<bTA ; Flagge setzen, +1 = 6
  ldi rimp,1<<bGn ; Kehre Flagge Gruen um, +1 = 7
  ; EOR kehrt alle Bits in rFlag um, die in rimp
  ; Eins sind (hier das bGn-Bit)
  eor rFlag,rimp ; aus rot wird gruen, aus gruen
    ; rot, +1 = 8
PcIntIsrSet:
  ldi rCnt,cCnt ; starte Zaehler neu, + 1 = 5/7/9
  out SREG,rSreg ; Wiederherstellen SREG, +1 =
    ; 6/8/10
  reti ; Rueckkehr vom Interrupt, + 4 = 10/12/14
;
; TC0 Overflow Interrupt
;
; Wird vom Timer am Ende jedes PWM-Zyklusses
; ausgeloest.
; Falls die bTA-Flagge gesetzt ist, wird
; der Tasteneingang abgefragt. Ist dieser
; High, wird der Karenzzaehler vermindert
; und bei Erreichen von Null die bTA-Flagge
; geloescht.
; Ist die bTA-Flagge nicht gesetzt oder
; ist ist der Karenzzaehler noch nicht
; Null wird ein Zykluszaehler abwaerts
; gezaehlt. Ist dieser Null, wird die
; bCy-Flagge gesetzt, die nach dem
; SLEEP bearbeitet wird.
;
TC0OvfIsr: ; Interrupt Service Routine TC0-
    ; Overflow
  in rSreg,SREG ; Sichern Statusregister, +1 = 1
  sbrs rFlag,bTa ; Ueberspringe wenn bTA-Flagge
    ; 1, +1/2 = 2/3
  rjmp TC0OvfIsrDwn ; Beenden, +2 = 4
  sbis pIn,bTaI ; Ueberspringe wenn Taste = 1,
    ; +1/2 = 4/5
  rjmp TC0OvfIsrNeu ; Neustart Zaehler, +2 = 6
  dec rCnt ; vermindere Zaehler, + 1 = 6
  brne TC0OvfIsrDwn ; noch nicht Null, weiter,
    ; +1/2 = 7/8
  cbr rFlag,1<<bTa ; bTa-Flagge loeschen, +1 = 8
  rjmp TC0OvfIsrDwn ; weiter, +2 = 10
TC0OvfIsrNeu:
  ldi rCnt,cCnt ; Neustart Abwaertszaehler, +1= 7
TC0OvfIsrDwn:
  dec rCyc ; zaehle Zyklen abwaerts, +1= 5/9/11/8
  brne TC0OvfIsrRet ; noch nicht Null, +1/2=
    ; 6/7/12/13/9/10
  mov rCyc,rAdc ; Neustart Downzaehler, +1=
    ; 7/13/10
  sbr rFlag,1<<bCy ; Setze Behandlungsflagge, +1
    ; = 8/14/11
TC0OvfIsrRet:
  out SREG,rSreg ; Wiederherstellen
Statusregister, +1 = 8/13/11/9/15/12
  reti ; Rueckkehr vom Interrupt, + 4 =
    ; 12/17/15/13/19/16
;
; ADC Ready Interrupt
;
; Wird vom AD-Wandler am Ende jedes Wandlungs-
; zyklusses ausgeloest.
; Liest die oberen 8 Bits des Ergebnisses ein,
; teilt dieses durch 16 und legt es im
; Register rAdc ab. Die Wandlung wird neu
; gestartet.
;
AdcIsr: ; Interrupt Service Routine ADC
  ; ADC-Ergebnis lesen
  in rAdc,ADCH ; Lese ADC-Ergebnis MSB, +1 = 1
  lsr rAdc ; teile durch 16, +1 = 2
  lsr rAdc ; +1 = 3
  lsr rAdc ; +1 = 4
  lsr rAdc ; +1 = 5
  inc rAdc ; +1 = 6
  ; Neustart ADC (Teiler 128, Interrupt)
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0); +1=7
  out ADCSRA,rimp ; in ADC-Kontrollregister,+1= 8
  reti ; +4 = 12
;
; ---------- Hauptprogramm-Init ------------
Start:
  ; Stapel anlegen
  ldi rmp,LOW(RAMEND) ; Stapelzeiger auf RAMEND
  out SPL,rmp ; in Stapelport
  ; LEDs konfigurieren und einschalten
  ldi rmp,(1<<bRAD)|(1<<bRKD) ; Portpins auf
    ; Ausgabe
  out pDir,rmp ; in Richtungsregister
  ldi rmp,(1<<bRKO)|(1<<bTaO) ; LED auf Rot,
     ; Tasten-Pullup
  out pOut,rmp ; in Port-Outputregister
  ; Startbedingungen
  clr rFlag ; Flaggen auf Null
  ldi rmp,0x10 ; kurzer Zyklus
  mov rAdc,rmp ; in ADC-Register
  mov rCyc,rmp ; in Zykluszaehler
  ; Timer als 8-Bit-PWM
  ldi rmp,0x80 ; halbe Helligkeit
  out OCR0A,rmp ; in Compare Register A
  ldi rmp,(1<<COM0A1)|(1<<WGM01)|(1<<WGM00)
  out TCCR0A,rmp ; in Kontrollregister A
  ldi rmp,(1<<CS01) ; Vorteiler 8
  out TCCR0B,rmp ; in Kontrollregister B
  ldi rmp,1<<TOIE0 ; Overflow Interrupt
  out TIMSK0,rmp ; in Timer-Int-Maske
  ; ADC konfigurieren: Left adjust, Signaleingang
    ; = ADC2
  ldi rmp,(1<<ADLAR)|(1<<MUX1) ; ADLAR und ADC
    ; Pin in Register
  out ADMUX,rmp ; in ADC-Multiplexer-Port
  ; ADC-Eingangstreiber abschalten
  ldi rmp,1<<bAdD ; Disable Porttreiber
  out DIDR0,rmp ; in Disable-Port
  ; ADC einschalten und starten
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; in ADC-Kontrollregister
  ; PCINT-Enable
  ldi rmp,1<<bTaE ; Tasten-Interrupts
  out PCMSK,rmp ; in PCINT-Maskenregister
  ldi rmp,1<<PCIE ; PCINT einschalten
  out GIMSK,rmp ; in Extern Interrupt Register
  ; Schlafmodus und Interrupts
  ldi rmp,1<<SE ; Schlafen ermoeglichen
  out MCUCR,rmp ; in Kontrollregister
  sei ; Interrupts zulassen
; Programmschleife
Schlafen:
  sleep ; schlafen legen
  nop ; Nach Aufwecken durch Int
  sbrc rFlag,bCy ; Behandlungsflagge Zyklusende
  rcall Zyklus ; Behandle Flagge
  rjmp Schlafen ; wieder schlafen legen
;
; Zyklus: Angestossen vom Ablauf jedes Zyklusses
;
; Loescht die gesetzte Flagge. Wenn die Farbflag-
; ge gruen ist werden die PWM-Ausgaenge OC0A und
; OC0B auf gruen eingestellt. Wenn nicht auf rot.
; Ist die Abwaertsflagge nicht gesetzt, wird der
; Vergleichswert A der PWM erhoeht, wenn gesetzt
; erniedrigt. Falls dabei 0xFF bzw. 0x00 erreicht
; wird, wird das Richtungsbit umgekehrt und bei
; dem entsprechenden Anfangswert neu begonnen.
;
Zyklus: ; Flagge bearbeiten, mit Takten
  cbr rFlag,1<<bCy ; Flagge ruecksetzen, +1 = 1
  sbrc rFlag,bGn ; Gruene LED?, +1/2 = 2/3
  rjmp ZyklusGruen ; Ja schalte LED gruen, +2 = 4
  sbi pOut,bRAO ; schalte LED rot, +2 = 5
  ldi rmp,(1<<COM0A1)|(1<<WGM01)|(1<<WGM00);+1= 6
  out TCCR0A,rmp ; +1 = 7
```

```
  rjmp ZyklusRichtung ; Richtungsumkehr, +2 = 9
ZyklusGruen:
  cbi pOut,bRAO ; schalte LED gruen, +2 = 6
  ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)|
(1<<WGM00); +1 = 7
  out TCCR0A,rmp ; +1 = 8
ZyklusRichtung:
  in rmp,OCR0A ; PWM-Vergleichswert lesen, +1 =
    ; 10/9
  sbrc rFlag,bAb ; ueberspringe wenn Abwaerts
    ; Null, +1/2 = 11/12/10/11
  rjmp ZyklusAbwaerts ; Nach Abwaerts ; +2= 13/12
  ; Vergleichswert aufwaerts
  inc rmp ; PWM eine Stufe aufwaerts, +1 = 13/12
  brne ZyklusSet ; setzen neuen Vergleichswert,
    ; +1/2 = 14/14
    sbr rFlag,1<<bAb ; setze Abwaertsflagge, +1= 15
    ldi rmp,0xFF ; setze auf hoechsten Wert, +1= 16
    rjmp ZyklusSet ; setze neuen Wert, +2 = 18
ZyklusAbwaerts:
  subi rmp,1 ; eins abziehen, +1 = 14/13
  brcc ZyklusSet ; setze Wert wenn kein
    ; Ueberlauf, +1/2 = 15/16/14/15
  cbr rFlag,1<<bAb ; aufwaerts zaehlen, +1= 16/15
  clr rmp ; bei Null starten, +1 = 17/16
ZyklusSet:
  out OCR0A,rmp ; neuen Vergleichswert setzen, +1
    ; = 15/15/19/17/16/18/17
  ret ; fertig, +4 = 19/19/23/21/20/22/21
;
; Ende Quellcode
;
```

Die folgenden Instruktionen sind neu:

- MOV Register,Register: kopiert den Inhalt des zweiten Registers in das erste,
- EOR Register,Register: exklusiv-oder beider Register, Ergebnis in das erste Register (kehrt für alle Bits im ersten Register den Wert um, die im zweiten Register an der gleichen Position Eins gesetzt sind).
- BRCC Label: verzweigt zum Label, wenn das Übertragbit gelöscht ist.
- INC Register: addiert eine Eins zum Register (erhöhen).
- SUBI Register,Konstante: subtrahiere die Konstante vom Register (R16..R31).

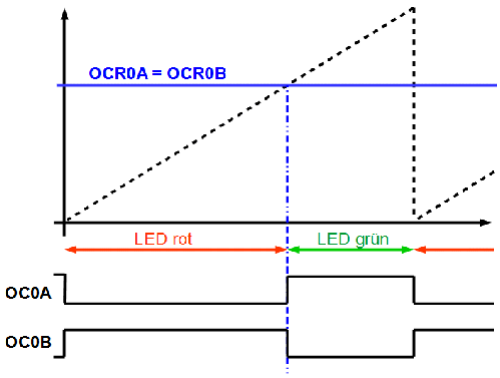
Im Vergleich zum vorherigen Programm läuft der Timer jetzt acht mal schneller. Dazu wurde nur die Zeile mit dem Timer-Init geändert. Bei der Berechnung der Konstanten im Definitionskopf wurde 64 durch 8 getauscht. Die Konstante, mit der die Inaktivitätszeit am Tasteneingang festgelegt wird, hat sich dadurch erhöht, denn jetzt muss mehr gezählt werden, um die 50 ms abzuwarten. Die Konstante cCnt, die jetzt zur Anwendung kommt ist cCnt=25. Das ist der Vorteil, solche Konstanten zu Beginn im Kopf zu definieren: der Änderungsbedarf im Programmcode ist sehr viel geringer.

Um herauszufinden, wie gross cCnt denn jetzt ist, hilft das Listing des Studio-Assemblers nicht weiter. Mit meinem Assembler gavrasm kriegt man mit dem Aufrufparameter -S am Ende des Listings eine Aufstellung aller Konstanten und ihrer Dezimal- und Hexwerte. Man kann es aber auch mit dem Taschenrechner nachrechnen ...

8.7 Helligkeitsregelung rot/grün

8.7.1 Aufgabe 3

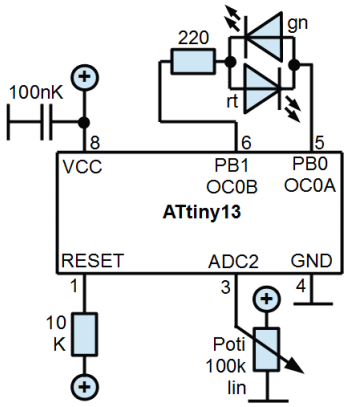
Das ist eine Bonusaufgabe. Was passiert mit der Farbe der LED, wenn wir schnell zwischen Rot und Grün hin- und herwechseln? Dazu ist eine PWM so zu schalten, dass der Regler den Anteil beider Farben an der Gesamtzeit der PWM variiert. Steigende Spannungen am Poti sollen den Rotanteil erhöhen.



8.7.2 Lösung

Zunächst kriegen wir das mit der bestehenden Hardware kaum elegant hin. Idealerweise lassen wir dazu die beiden PWMs im ATtiny13, A und B, gegenläufig laufen. Die Signalausgänge OCR0A und OCR0B liefern bei

richtiger Programmierung ein gegenläufiges invertiertes Signal, das die Farben der LED umschaltet.



Da der OC0B-Anschluss benötigt wird, muss der zweite Anschluss der Duo-LED an Pin 6 verlegt werden. Der Taster wird hier nicht gebraucht, er stört aber auch nicht.

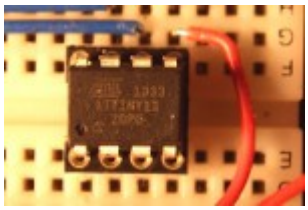
8.7.3 Programm 4

Das hier ist das Programm (zum Quellcode im asm-Format geht es hier). Da die gesamte PWM-Schalterei vollständig vom Timer übernommen wird, brauchen wir nur für den ADC-Wandler eine Interrupt-Service-Routine.

```
;
; ********************************
; * Duo-LED im Gegentakt ATtiny13 *
; * (C)2016 by gsc-elektronic.net *
; ********************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; ------ Programmablauf ----------
;
; Liest den Potistand mit dem ADC
; ein und schaltet die Duo-LED mit
; dem TC0-PWM schnell zwischen rot
; und gruen um. Der Rot-Anteil
; steigt mit zunehmender Potispan-
; nung.
;
; -------- Register ---------------
; frei: R0 .. R15
.def rmp = R16 ; Vielzweckregister
.def rimp = R17 ; Vielzweckregister Interrupts
;
; -------- Ports ------------------
.equ pDir = DDRB ; Port-Ausgaenge
.equ bARD = DDB1 ; Rote Anode
.equ bKRD = DDB0 ; Rote Kathode
;
; -------- Timing -----------------
;  Takt         = 1200000 Hz
;  Prescaler    =      64
;  PWM-Stufen   =     256
;  PWM-Frequenz =      73 Hz
;
; -- Reset- und Interruptvektoren -
.CSEG ; Code Segment
.ORG 0 ; Bei 0 beginnen
  rjmp Start ; Reset Vektor, Sprung zur
    ; Initiierung
  reti ; INT0-Int, nicht aktiv
  reti ; PCINT-Int, nicht aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  rjmp AdcIsr ; ADC-Int, aktiv
;
; ----- Interrupt Service Routinen -----
;
AdcIsr:
  in rimp,ADCH ; Lese ADC-Ergebnis MSB
  out OCR0A,rimp ; in Vergleichsregister A
  out OCR0B,rimp ; in Vergleichsregister B
  ; Neustart ADC (Teiler 128, Interrupt)
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rimp ; in ADC-Kontrollregister
  reti
;
; ----- Hauptprogramm Init -------------
Start:
  ; Stapel einrichten
  ldi rmp,LOW(RAMEND) ; auf SRAM-Ende
  out SPL,rmp ; in Stackregister
  ; Ausgaenge initiieren
  ldi rmp,(1<<bARD)|(1<<bKRD) ; Ausgaenge
  out pDir,rmp ; in Richtungsregister
  ; Vergleicher initiieren
  ldi rmp,0x80 ; auf halbe/halbe
  out OCR0A,rmp ; in Vergleichsregister A
  out OCR0B,rmp ; in Vergleichsregister B
  ; Timer als PWM mit A- und B-Outputsteuerung
  ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<COM0B1)|
(1<<WGM01)|(1<<WGM00)
  out TCCR0A,rmp ; in Kontrollregister A
  ; Timer mit 64 starten
  ldi rmp,(1<<CS01)|(1<<CS00) ; Prescaler 64
  out TCCR0B,rmp ; in Kontrollregister B
  ; ADC konfigurieren: Left adjust, Signaleingang
    ; = ADC2
  ldi rmp,(1<<ADLAR)|(1<<MUX1) ; ADLAR und ADC-
    ; Pin in Register
  out ADMUX,rmp ; in ADC-Multiplexer-Port
  ; ADC-Eingangstreiber abschalten
  ldi rmp,1<<ADC2D ; Disable Porttreiber
  out DIDR0,rmp ; in Disable-Port
  ; ADC einschalten und starten
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; in ADC-Kontrollregister
  ; Schlafen einschalten
  ldi rmp,1<<SE ; Sleep Enable
  out MCUCR,rmp ; in General Kontrollregister
  ; Interrupts einschalten
  sei ; set I-Flagge
```

```
Schleife:                                   ;
  sleep ; schlafen legen                    ; Ende Quellcode
  nop ; nach Aufwachen                      ;
  rjmp Schleife ; wieder schlafen legen
```

Hier gibt es keine neuen Instruktionen.



Lektion 9: Tongenerator mit AD-Wandler, Tontabellen, Multiplikation

Der OC0A-Ausgang dient hier als variabler Tonerzeuger mit Poti-Einstellung der Tonhöhe. Außerdem werden hier 8-Bit-Zahlen miteinander multipliziert und Musikstücke abgespielt.

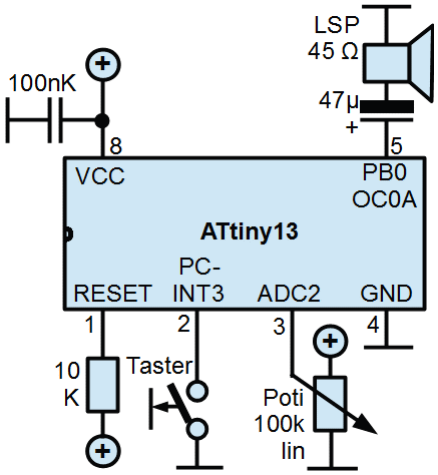
9.0 Übersicht

9.1 Einführung in die Tonerzeugung

Das Erzeugen von Tönen kennen wir im Grunde schon, weil es nichts anderes als eine blinkende LED ist, nur mit höheren Frequenzen im Niederfrequenzbereich (NF) bis 20 kHz (für Fledermäuse bis 40 kHz). Wir lernen daher hier nichts Neues über Timer, nur über den Anschluss von Lautsprechern, das Multiplizieren und den Umgang mit Tabellen.

9.2 Hardware, Bauteile und Aufbau

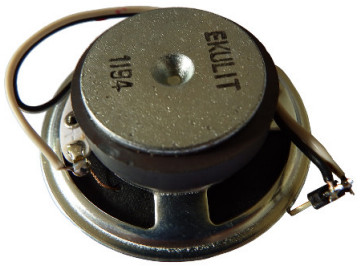
9.2.1 Die Schaltung



Zur Tonausgabe wird ein Lautsprecher benötigt, der an OC0A angeschlossen wird. Der Elko mit 47 µF entkoppelt den Gleichstrom und lässt nur die Pegelwechsel an OC0A durch.

Taster und Potentiometer sind wie gehabt angeschlossen.

9.2.2 Die Bauteile



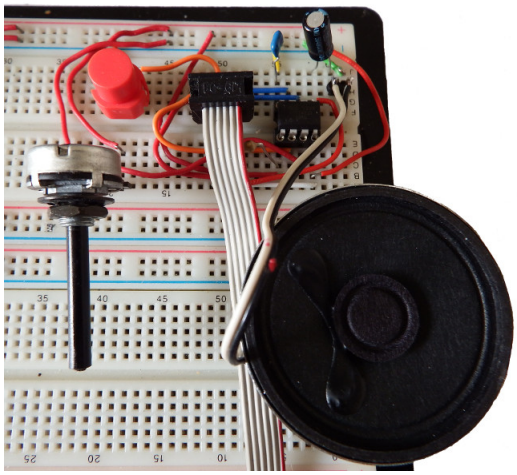
9.2.2.1 Der Lautsprecher

Das ist der Lautsprecher. Er hat 45 Ω Impedanz, damit die Lautstärke ordentlich hoch ist. Die beiden Anschlüsse kriegen einen Kabelanschluss mit Pins angelötet, die in das Breadboard passen. Die Polarität ist für unsere Zwecke egal, da sie nur akustische Folgen hat.

9.2.2.2 Der Elko

Das hier ist ein Elko. Der Minuspol ist auf dem Gehäuse gekennzeichnet, der Pluspol-Anschlussdraht ist der längere.

9.2.3 Der Aufbau



Der Anschluss des Lautsprechers erfolgt an Pin 5 über den Elko.

Damit kann es mit dem Herumtönen losgehen.

9.3 Aufgabe 1: Tonhöhenregelung

9.3.1 Einfache Aufgabe 1

Bei der Aufgabe 1 sollen Töne auf dem Lautsprecher ausgegeben werden, deren Tonhöhe mit dem Poti geregelt werden kann. Töne zwischen 300 Hz und 75 kHz ("Fledermausschreck") sollen erzeugt werden. Der Ton soll nur ausgegeben werden, wenn die Taste gedrückt ist.

9.3.2 Lösung

9.3.2.1 Frequenzbereiche

Es ist klar, dass hier der CTC-Modus des Timers verwendet werden muss. Der Ausgang muss, wenn der Ton gehört werden soll, torkeln (von Null auf Eins und zurück auf Null). Da für jede Schwingung zwei CTC-Durchläufe nötig sind, ist die erzeugte Frequenz halb so groß. Der Frequenzbereich bei verschiedenen Takten und Vorteilern überstreicht folgende Bereiche:

Takt	Vorteiler	OCR0A = 0	OCR0A = 255
9,6 MHz	1	4,8 MHz	18,75 kHz
	8	600 kHz	2,34 kHz
	64	75 kHz	292,5 Hz
	256	18,75 kHz	73,1 Hz
	1024	4,69 kHz	18,3 Hz
1,2 MHz	1	600 kHz	2,34 kHz
	8	75 kHz	292,5 Hz
	64	9,38 kHz	36,6 Hz
	256	2,35 kHz	9,15 Hz
	1024	586 Hz	2,29 Hz

Der hörbare Bereich lässt sich bei 1,2 MHz Takt mit einem Vorteiler von 8 gut überdecken.

9.3.2.2 AD-Werte und OCR0A-Werte

Je höher die Spannung am Poti ist, desto höher soll der Ton sein. Da der OCR0A-Wert sich umgekehrt verhält (je höher der OCR0A-Wert desto niedriger die Frequenz), muss entweder das Potentiometer umgekehrt angeschlossen werden oder eine Umkehr der gemessenen Werte erfolgen. Eine Softwarelösung dafür wäre, den gemessenen Wert von 0xFF abzuziehen. Das ginge mit

```
  ldi Register1,0xFF
  sub Register1,Register2 ; Register2 = Messwert
  mov Register2,Register1
```

Die Instruktion SUB Register,Register zieht das zweite Register vom ersten ab und schreibt das Ergebnis in das erste Register.

Das Umkehren aller Bits in einem Register kann der Prozessor aber von sich aus schon (siehe Quelltext und die Erläuterung darunter), wir können uns also diese "Von-Hinten-durch-die-Brust-ins-Auge-Lösung" sparen.

9.3.3 Programm

Das hier ist das Programm (zum Quelltext im asm-Format geht es hier).

```
;
; ****************************************
; * Tonerzeugung mit Taster und Regelung     *
; * (C)2016 by http://www.gsc-elektronic.net *
; ****************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; --------- Programmablauf -----------------
;
; Die Spannung am Poti-Eingang wird mit dem
; ADC staendig eingelesen und in das Ver-
; gleichsregister des Timers TC0 geschrieben.
; Mit dem PCINT am Tasteneingang wird fest-
; gestellt, ob die Taste gedrueckt ist. Wenn
; ja wird der Timer auf Toggle eingestellt,
; wenn nicht wird OC0A auf Clear eingestellt.
;
; --------- Register -------------------------
; frei: R0 .. R14
.def rSreg = R15 ; Sichern Statusregister
.def rmp = R16 ; Vielzweckregister
.def rimp = R17 ; Vielzweckregister Interrupts
; frei: R18 .. R31
;
; --------- Ports ----------------------------
.equ pOut = PORTB ; Ausgabeport
.equ pDir = DDRB ; Richtungsport
.equ pInp = PINB ; Eingangsport
.equ bLspD = DDB0 ; Lautsprecherausgang
.equ bTasO = PORTB3 ; Pullup Tasteneingang
.equ bTasI = PINB3 ; Tasteninputpin
.equ bAdID = ADC2D ; ADC-Input-Disable
;
; --------- Timing ---------------------------
; Takt = 1200000 Hz
; Vorteiler = 8
; CTC-TOP-Bereich = 0 .. 255
; CTC-Teiler-Bereich = 1 .. 256
; Toggle-Teiler = 2
; Frequenzbereich: 75 kHz .. 293 Hz
;
; --------- Reset- und Interruptvektoren -----
.CSEG ; Assemblieren ins Code-Segment
.ORG 0 ; Start bei Null
  rjmp Start ; Reset Vektor, Sprung zu Init
  reti ; INT0-Int, nicht aktiv
  rjmp PcIntIsr ; PCINT-Int, aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  rjmp AdcIsr ; ADC-Int, aktiv
;
; ---------- Interrupt Service Routinen -----
;
; PCINT Interrupt
; Wird von Tastenereignissen ausgeloest.
; Falls Taste gedrueckt, wird der Timer
; auf Torkeln gestellt, wenn nicht wird
; der Timer auf Clear gestellt.
;
PcIntIsr: ; PCINT-Interrupt Tasten-Interrupt
  sbic pInp,bTasI ; Ueberspringe bei Taste = 0
  rjmp PcIntIsrAus ; Taste ist nicht gedrueckt
  ldi rimp,(1<<COM0A0)|(1<<WGM01) ; Toggle, CTC-A
  out TCCR0A,rimp ; in Kontrollregister A
  rjmp PcIntIsrRet ; zurueck
PcIntIsrAus:
  ldi rimp,(1<<COM0A1)|(1<<WGM01) ; Clear, CTC-A
  out TCCR0A,rimp
PcIntIsrRet:
  reti
;
; ADC Ready Interrupt
;
; Wird vom AD-Wandler bei abgeschlossener
; Wandlung ausgeloest.
; Liest die obersten 8 Bit des AD-Wandlers
; und schreibt es in das Vergleichregister
; A des Timers. Die naechste Wandlung wird
; angestossen.
;
AdcIsr: ; ADC-Interrupt
  in rSreg,SREG ; sichern Statusregister
  in rimp,ADCH ; lese MSB Ergebnis
  com rimp ; Wert umkehren
  out OCR0A,rimp ; in CTC-TOP-Register
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rimp ; in ADC-Kontrollregister A
  out SREG,rSreg ; wiederherstellen Status
  reti
;
; ---------- Programmstart und Init ----------
Start:
```

```
  ; Stapel einrichten
  ldi rmp,LOW(RAMEND) ; SRAM-Ende
  out SPL,rmp ; in Stackzeiger
  ; In- und Output-Ports
  ldi rmp,1<<bLspD ; Lautsprecherausgang Richtung
  out pDir,rmp ; in Richtungsregister
  ldi rmp,1<<bTasO ; Pullup am Tastenport
  out pOut,rmp ; in Ausgangsregister
  ; Timer als CTC konfigurieren
  ldi rmp,(1<<COM0A1)|(1<<WGM01) ; Clear, CTC-A
  out TCCR0A,rmp ; in Kontrollregister A
  ldi rmp,1<<CS01 ; Vorteiler = 8, Timer starten
  out TCCR0B,rmp ; in Kontrollregister B
  ; AD-Wandler konfigurieren und starten
  ldi rmp,(1<<ADLAR)|(1<<MUX1) ; Linksjust., ADC2
  out ADMUX,rmp ; in ADC-MUX
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; in Kontrollregister A, starten
  ; PCINT fuer Tasteneingang
  ldi rmp,1<<PCINT3 ; PB3-Int ermoeglichen
  out PCMSK,rmp ; in PCINT-Maskenregister
  ldi rmp,1<<PCIE ; PCINT ermoeglichen
  out GIMSK,rmp ; in Interrupt-Maskenregister
  ; Schlafen ermoeglichen
  ldi rmp,1<<SE ; Schlafen, Idle-Modus
  out MCUCR,rmp ; in MCU-Kontrollregister
  ; Interrupts einschalten
  sei
; ------------ Hauptprogramm-Schleife --------
Schleife:
  sleep ; schlafen legen
  nop ; Aufwachen
  rjmp Schleife ; wieder schlafen legen
;
; Ende Quellcode
;
```

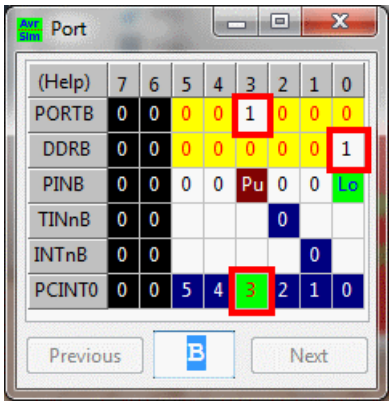
An neuen Instruktionen gibt es hier:

- COM Register: kehrt alle Bits im Register um, aus 0x00 wird 0xFF, aus FF 00. Zieht den Registerinhalt von 0xFF ab und legt das Ergebnis im Register ab. Heißt auch Einerkomplement.

Damit haben wir eine tonhöhen-regulierbare Morsetaste gebastelt.

Top	Home	Einführung	Hardware	Tonhöhe	Tonleiter	Melodie

9.3.4 Simulation des Programmes mit avr_sim

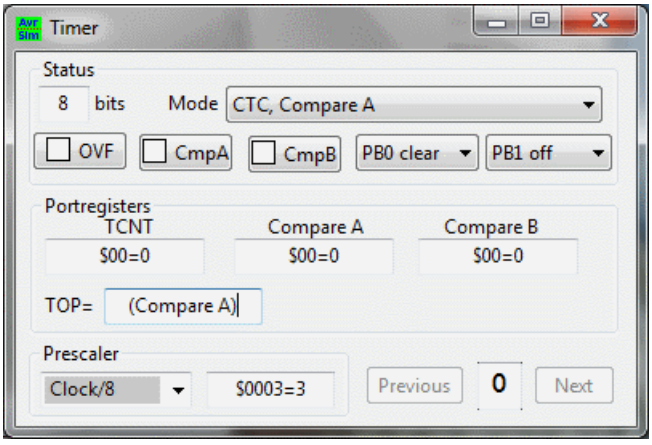


Die Simulation erfolgt mit avr_sim wie folgt.

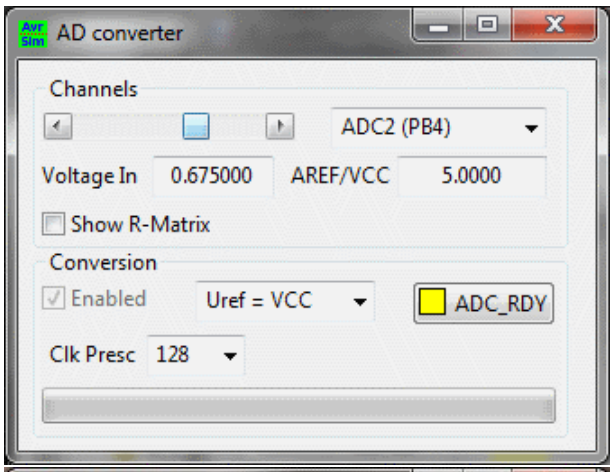
Das sind die Einstellungen nach der Initphase. Der I/O-Port PB0 ist als Ausgang konfiguriert, er treibt den Lautsprecher an. Sein Ausgang ist Low.

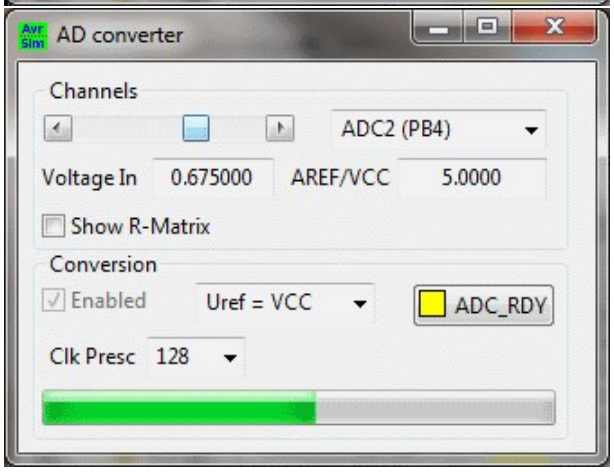
Das Eingangsbit PB3, an das die Taste angeschlossen ist, hat den Pullup-Widerstand eingeschaltet und ist High, solange die Taste nicht gedrückt ist.

Ebenfalls am PB3 ist die PCINT-Maske und der PCINT-Interrupt-Enable gesetzt. Bei Tastendrücken und beim Loslassen wird daher der PCINT ausgelöst.



Der Timer TC0 ist im CTC-Modus, weshalb er nach Erreichen des Vergleichswertes in Compare-Match-A zurücksetzt. Der Ausgang PB0 wird bei Erreichen des Compare-Match-A-Werts auf Null gesetzt, welches PB0 auf Null lässt und den Lautsprecher stumm macht. Der Vorteiler ist auf acht gesetzt, was den Timer mit 1.200.000 / 8 = 150 kHz Takt ansteuert.





Der AD-Wandler ist aktiv und wird mit einem Vorteiler von 128 getaktet. Das bedeutet pro Wandlung bei 13 Taktzyklen

13 * 128 / 1.200.000 = 1,386 ms

Wandlerzeit.

Seine Referenzspannung ist die Betriebsspannung (5 V). Der eingestellte Spannungswert 0,675 V sollte zu einem Wandlungsergebnis von

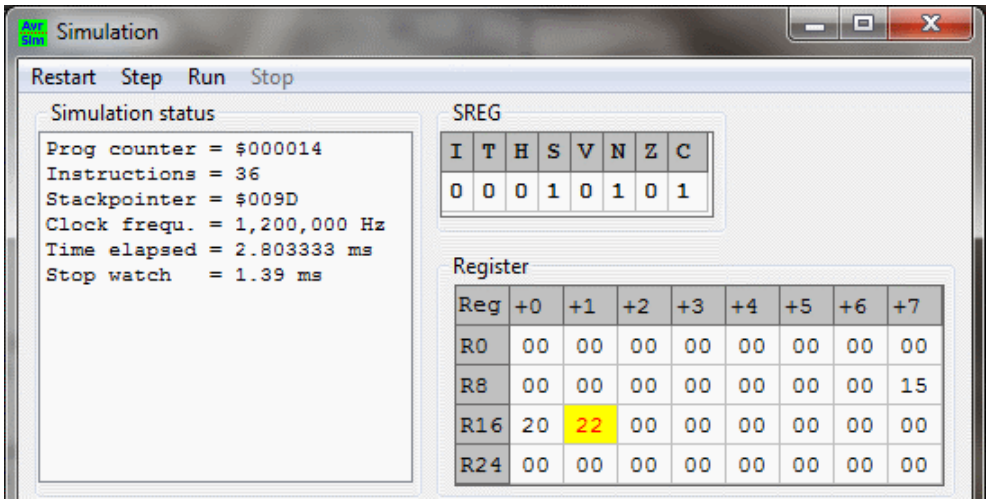
1.024 * 0,675 / 5,0 = 138

führen. Ist das ADLAR-Bit gesetzt, was hier der Fall ist, liefert das MSB in ADCH 138 / 4 = 34 (hexadezimal 0x22).

Das ADC-Ready-Interrupt-Enable-Bit ist ebenfalls gesetzt, der AD-Wandler führt nach jedem Wandlervorgang einen Interrupt aus.

Die erste Wandlung hat begonnen, was man an der angezeigten Fortschrittsanzeige erkennt.

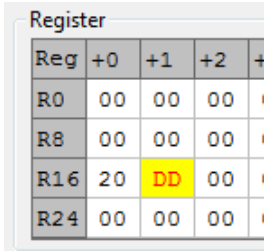
Die erste AD-Wandlung hat einigen Fortschritt genommen. Da die erste Wandlung etwa 1,38 ms dauert (und sogar ein bisschen länger, weil der AD-Wandler frisch eingeschaltet wurde) bleibt der Prozessor im Schlafzustand.



Nach Abschluss der Wandlung ist der ADC-Ready-Interrupt ausgelöst worden und das MSB des ADC-Ergebnisses wird mit der Instruktion

```
in rimp,ADCH
```

nach R17 eingelesen.

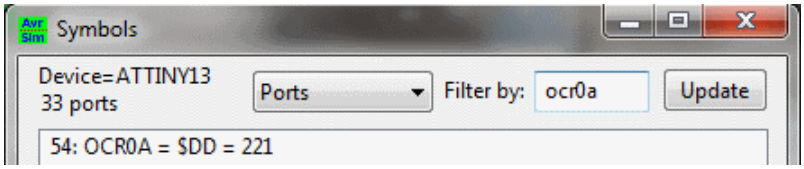


Die Umkehr der Bits des Ergebnisses mit der Instruktion *com rimp* liefert 0xFF - 0x22 = 0xDD oder dezimal 221.

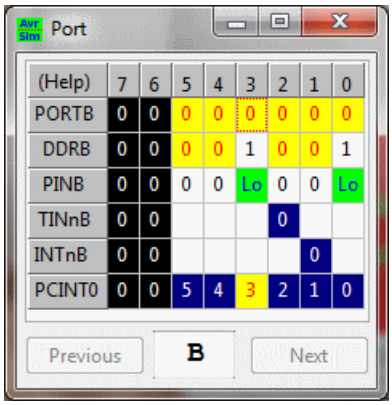
Mit

```
out OCR0A,R17
```

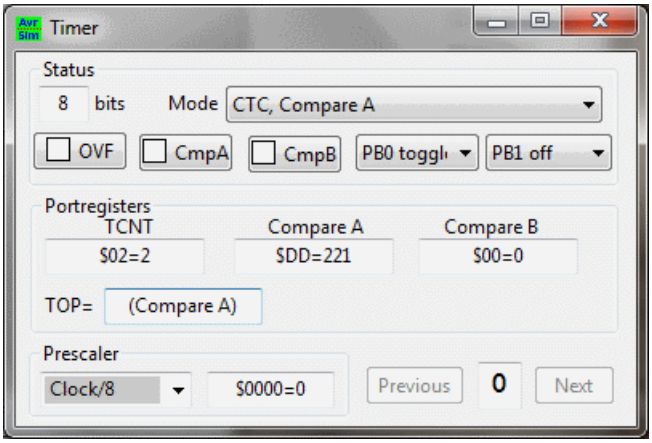
wird dieses invertierte Ergebnis in



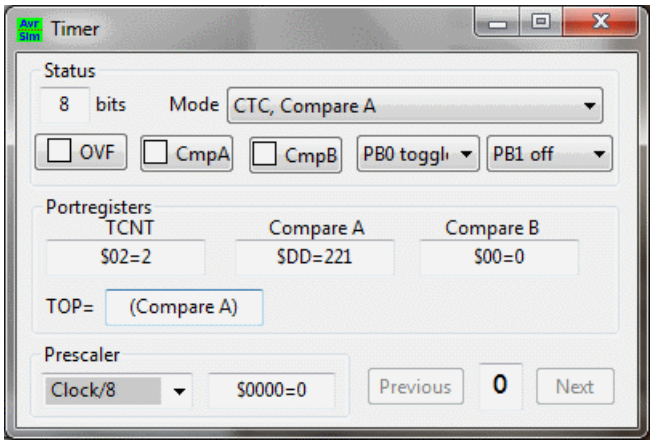
das Vergleichsregister Compare A geschrieben. Damit tritt jetzt alle (221 + 1) * 8 / 1.200.000 = 1,48 ms ein Compare-Match-A ein. Das führt noch nicht zu einem hörbaren Ereignis, da PB0 bei Erreichen des Compare-Match-A-Wertes noch immer so eingestellt ist, dass der Lautsprecherausgang auf Null stehen bleibt.



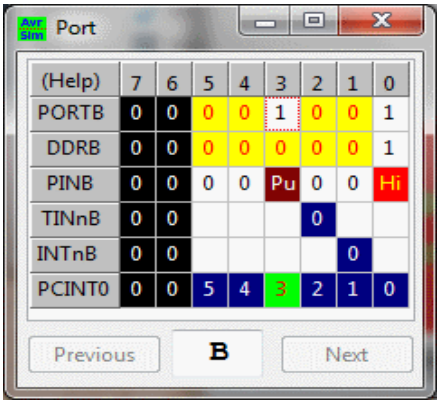
Das ändert sich erst, wenn wir den PCINT-Interrupt ausführen. Das kriegen wir hin, indem wir auf das Bit PCINT3 in der untersten Zeile der Portdarstellung klicken. Der PCINT-Interrupt wird nach vier Wartezyklen ausgeführt, falls kein anderer Interrupt ausgeführt wird und kein höherwertigerer zur Ausführung ansteht (was nur ein INT0 sein könnte, der aber hier gar nicht enabled ist).



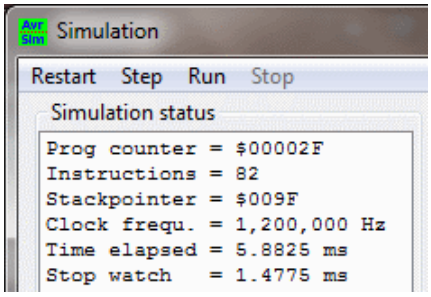
Innerhalb der PCINT-Interrupt-Service-Routine schalten die Instruktionen *ldi rimp, (1<<COM0A0)|(1<<WGM01)* und *out TCCR0A,rimp* das Torkeln des PB0-Pins ein, aber nur falls der Tasteneingang Null (Taste gedrückt) ist. Ist er hingegen Eins, wird der PB0-Ausgang wieder auf "Clear" geschaltet.



Nach Erreichen des Compare-Match-A-Wertes (und dem Rücksetzen des Timers auf Null) ändert sich das Portbit PORTB0 nun tatsächlich und der Lautsprecher kriegt den ersten Schlag versetzt.



Der Pin PB0 ist jetzt High, der Elko vor dem Lautsprecher wird schlagartig auf Plus geladen, der Ladestrom fließt durch den Lautsprecher und bewegt die Membran.



Da das CTC-Ereignis nach 1,4775 ms eintrat, entsprechen jeweils zwei solche Ereignisse (Ein- und Ausschalten des Elko/Lautsprechers) einer hörbaren Frequenz von 338 Hz, also ein Ton irgendwo zwischen e^1 und f^1 (siehe nächstes Kapitel).

9.4 Aufgabe 2: Die Tonleiter

9.4.1 Aufgabenstellung

In diesem Teil der Lektion soll die Tonleiter gespielt werden, das Potentiometer soll der Tonauswahl dienen.

9.4.2 Die Tonleiter

Leider ist es so, dass Musik in unserer Hemisphäre nur mit ganz bestimmten Frequenzen geht. Sanft wechselnde Tonfrequenzen sind bei unserem Hörvermögen und musikalischem Empfinden total out und werden eher als störend empfunden. Wir brauchen daher eine Tonleitertabelle, die nur die hierzulande zulässigen Töne umfasst. Das sind die in der nachfolgenden Tabelle stehenden Frequenzen.

Ton	Freq.(Hz)	Prescaler	CTC	Ist(Hz)	Delta(%)	#
a	440	8	170	441,18	0,27%	0
h	493,883	8	152	493,42	-0,09%	1
c	523,251	8	143	524,48	0,23%	2
d	587,330	8	128	585,94	-0,24%	3
e	659,255	8	114	657,89	-0,21%	4
f	698,456	8	107	700,93	0,35%	5
g	783,991	8	96	781,25	-0,35%	6
a'	880	8	85	882,35	0,27%	7
h'	987,766	8	76	986,84	-0,09%	8
c'	1.046,502	8	72	1.041,67	-0,46%	9
d'	1.174,660	8	64	1.171,88	-0,24%	10
e'	1.318,510	8	57	1.315,79	-0,21%	11
f'	1.396,912	8	54	1.388,89	-0,57%	12
g'	1.567,982	8	48	1.562,50	-0,35%	13

Ton	Freq.(Hz)	Prescaler	CTC	Ist(Hz)	Delta(%)	#
A	1.760	8	43	1.744,19	-0,90%	14
H	1.975,532	8	38	1.973,68	-0,09%	15
C	2.093,004	8	36	2.083,33	-0,46%	16
D	2.349,320	8	32	2.343,75	-0,24%	17
E	2.637,020	1	228	2.631,58	-0,21%	18
F	2.793,824	1	215	2.790,70	-0,11%	19
G	3.135,964	1	191	3.141,36	0,17%	20
A'	3.520	1	170	3.529,41	0,27%	21
H'	3.951,064	1	152	3.947,37	-0,09%	22
C'	4.186,008	1	143	4.195,80	0,23%	23
D'	4.698,640	1	128	4.687,50	-0,24%	24
E'	5.274,040	1	114	5.263,16	-0,21%	25
F'	5.587,648	1	107	5.607,48	0,35%	26
G'	6.271,928	1	96	6.250,00	-0,35%	27
A''	7.040	1	85	7.058,82	0,27%	28

Um das dem Timer beizubringen, sind für eine Taktfrequenz von 1,2 MHz auch noch die optimalen Werte für den Vorteiler und die CTC-Werte angegeben. Da die Timermimik diese Töne nicht genau trifft, sind die tatsächlich erzeugten Frequenzen und die Abweichungen vom korrekten Ton auch noch gleich mit angegeben.

Um nun den Timer dazu zu kriegen, genau nur diese Frequenzen zu treffen, brauchen wir eine Tonleitertabelle, die dem Prozessor diese Werte irgendwie beibringt. Da die Werte für die einzelnen Oktaven (von a nach a', von a' nach A, von A nach A' und von A' nach A'') sich immer um den Faktor 2 unterscheiden, könnten wir die CTC-Werte auch berechnen, aber das würde bei den höheren Frequenzen oberhalb D etwas ungünstig, weil die optimalerweise mit einem Vorteiler von 1 statt 8 erzeugt werden. Unsere Tabelle enthält daher sowohl den CTC-Wert als auch den Vorteiler für alle vier Oktaven.

9.4.3 Tabellen und ihre Platzierung

Unsere Tabelle braucht 2*29 = 58 Bytes Länge. Es gibt prinzipiell drei Orte im Prozessor, an denen wir eine solche Tabelle ablegen könnten:

1. den SRAM-Speicher. Er umfasst im ATtiny13 64 Bytes, wäre also ziemlich voll und es könnten Konflikte mit dem Stapel ins Haus stehen, der ja auch das SRAM benutzt.
2. das EEPROM. Es umfasst ebenfalls 64 Bytes. Es würde passen, wäre aber recht gefüllt.
3. den Flash-Speicher. Er bietet 512 Worte oder 1.024 Bytes Platz, das wäre also reichlich und böte also Raum für weitere Oktaven.

9.4.3.1 Tabelle im SRAM

Im ersten Fall, der Ablage im SRAM-Speicher, gibt es keine andere Möglichkeit, unsere Tabellenwerte hineinzuschreiben als jeden Wert einzeln mittels der Instruktion "STS Adresse,Register" dort abzulegen. Dazu müssten wir folgendes programmieren:

```
ldi R16,8 ; Prescaler-Wert
sts 0x60,R16 ; speichere an Adresse 0x0060 im SRAM
ldi R16,170 ; CTC-Wert
sts 0x60+1,R16 ; speichere an Adresse 0x0061 im SRAM
[...]
ldi R16,1 ; Prescaler-Wert
sts 0x60+56,R16 ; speichere an Adresse 0x0098 im SRAM
ldi R16,85 ; CTC-Wert
sts 0x61,R16 ; speichere an Adresse 0x0099 im SRAM
```

Jedes Wertepaar würde vier Instruktionen benötigen, von denen zwei (STS) auch noch Zwei-Wort-Instruktionen sind (sechs Worte pro Paar). Selbst wenn wir diese Instruktionen etwas vereinfachen, indem wir die Instruktion "ST Z+,Register" verwenden, wäre auch das mühsam. ST Z+ geht folgendermaßen:

```
ldi ZH,HIGH(0x0060) ; Zeiger Z auf SRAM-Startadresse, MSB
ldi ZL,LOW(0x0060) ; dto., LSB
ldi R16,8 ; Prescaler-Wert
st Z+,R16 ; speichern R16 im SRAM und Adresse in Z erhoehen
ldi R16,170 ; CTC-Wert
st Z+,R16 ; an naechste Adresse
[...]
ldi R16,1 ; Prescaler-Wert
st Z+,R16 ; speichere R16 im SRAM und Adresse in Z erhoehen
ldi R16,85 ; CTC-Wert
st Z,R16 ; speichere R16 an letzter Adresse im SRAM
```

Das Speichern mit automatischer Adresserhöhung "ST Z+,Register" wird beim letzten zu schreibenden Wert geändert in "ST Z,Register", die Adresse in Z wird dann nicht mehr erhöht.

Das Umgekehrte, nämlich Adresse vermindern, gibt es auch, aber etwas anders. Mit "ST -Z,Register" vermindert sich die Adresse zuerst und das Schreiben erfolgt an die schon erniedrigte Adresse. Was sich norwegische Sprachkonstrukteure so alles ausdenken, um das Leben von Anfängern ein wenig bunter zu machen.

Mal abgesehen von der langweiligen Tipperei ist auch diese Lösung alles andere als elegant und praktikabel. Das SRAM ist einfach nicht der richtige Ort, um so was wie eine lange Tabelle komfortabel abzulegen.

9.4.3.2 Tabelle im EEPROM

Der zweite Ort, die Tabelle abzulegen, das EEPROM, bietet schon etwas komfortablere Bedingungen. Hier lautet die Konstruktion:

```
.ESEG
.ORG 0
.db 8,170
[...]
.db 1,85
.CSEG
```

Die Assemblerdirektive ".ESEG" bewirkt, dass der Assembler das nachfolgende im EEPROM-Segment ablegt, nicht im Programmspeicher. Ab jetzt werden alle Inhalte separat gehandhabt und in eine Datei mit der Endung ".eep" geschrieben. Der Inhalt dieser Datei kann mit der Brennsoftware direkt in das EEPROM des Chips geschrieben werden. Wenn die Tabelle irgendwo zwischen dem Code steht, wird mit ".CSEG" abschließend wieder auf Programmcode-Erzeugung umgeschaltet.

Das ".ORG 0"bewirkt, dass an der EEPROM-Adresse 0 mit der Tabelle begonnen wird.

Die einzelnen ".DB"-Direktiven bewirken, dass die mit Kommata getrennten folgenden Zahlen nacheinander im EEPROM an die nachfolgenden Adressen abgelegt werden. Texte kann man mit der Direktive .DB "Text" in das EEPROM ablegen. Gespeichert werden die ASCII-Codes des

Textes, Buchstabe für Buchstabe.

Wie wir an die Werte im EEPROM wieder herankommen, kommt erst in einer späteren Lektion dran.

Das ist schon wesentlich komfortabler, aber immer noch nicht so elegant wie das Folgende.

9.4.3.3 Tabelle im Programmspeicher

Jetzt wird es etwas komplizierter, weil der Programmspeicher ja 16-bittig ausgelegt ist und ganze Worte speichert. Mit

```
Tabelle:
.db 8,170
[...]
.db 1,85
```

passiert jetzt Folgendes:

- Die 8 wird als LSB, die 170 als MSB in ein 16-bittiges Wort verwandelt und an der nächsten Adresse im Programmspeicher abgelegt. Die Adresse ist mit dem Label "Tabelle:" festgehalten.
- Alle nachfolgenden ".DB" legen Worte an den nachfolgenden Adressen ab.

Es ist klar, dass pro ".DB" immer ganze Worte abgelegt werden. Schreibt man also ".DB 1", wird effektiv 0x0001 abgelegt. Da die Anzahl abzulegender Bytes in diesem Fall nur Eins und daher ungerade ist, quittiert das der Assembler mit einer Warnung, er habe ein weiteres Byte (0x00) hinzugefügt, um eine geradzahlige Anzahl an Bytes zu erhalten. Grundsätzlich gilt, dass zuerst das LSB, dann das MSB befüllt wird.

Die gleiche Warnung resultiert, wenn wir den Text "ABC" mit .DB "ABC" im Flashspeicher ablegen wollen. Auch hier wird noch ein Nullbyte angefügt, wenn wir nicht .DB "ABC_" hinschreiben.

Es gibt noch eine zweite Möglichkeit, um die Tabelle zu füllen:

```
Tabelle:
.dw 8+170*256
[...]
.dw 1+85*256
```

Das erzeugt direkt Worte und legt sie in der Tabelle ab. Hier haben wir es in der Hand darüber zu bestimmen, was LSB und was MSB wird.

Wie kriegen wir nun die so erzeugte Tabelle wieder ausgelesen? Um das erste Byte zu lesen, formulieren wir Folgendes:

```
  ldi ZH,HIGH(2*Tabelle) ; LSB-Zeiger in Z
  ldi ZL,LOW(2*Tabelle)
  lpm ; Load from Program Memory
```

Das gelesene LSB, in unserem Fall die 8, kommt mit LPM nun in das Register R0. Um es woandershin zu laden, schreiben wir LPM Register, Z:

```
  ldi ZH,HIGH(2*Tabelle) ; LSB-Zeiger in Z
  ldi ZL,LOW(2*Tabelle)
  lpm R16,Z ; Load from Program Memory nach R16
```

Um das MSB zu lesen, könnten wir formulieren:

```
  ldi ZH,HIGH(2*Tabelle+1) ; MSB-Zeiger in Z
  ldi ZL,LOW(2*Tabelle+1)
  lpm R16,Z ; Load from Program Memory nach R16
```

Damit ist auch klar, weshalb das Label Tabelle mit zwei malgenommen werden muss: das unterste Bit der Leseadresse dient der LSB/MSB-Auswahl.

Um beide (oder mehr) Bytes nacheinander zu lesen, kann man

```
  ldi ZH,HIGH(2*Tabelle) ; MSB-Zeiger in Z
  ldi ZL,LOW(2*Tabelle)
  lpm XL,Z+ ; Load LSB from Program Memory nach XL
  lpm XH,Z+ ; Load MSB from Program Memory nach XH
```

schreiben. Das LPM Register,Z+ erhöht die Adresse im Zeigerpaar Z automatisch nach dem Lesen und zeigt schon mal auf das nächste Byte. Rückwärts geht es auch, mit "LPM -Z,Register", aber nicht beim ATtiny13. XL und XH sind die beiden Einzelregister des Zeigerpaares X. XL, XH, YL, YH, ZL und ZH sind übrigens in der Datei "*def.inc" definiert, weshalb bei der Verwendung ohne diese Typdefinition in der def.inc eine Fehlermeldung resultiert. Das hat historische Gründe, weil die ersten AVR-Typen noch gar keine Zeigerregister hatten.

Damit ist unsere Notentabelle klar: hier ist sie.

```
Notentabelle:                   .db 1<<CS01,  71 ; c'   #9      .db 1<<CS00, 214 ; F   #19
.db 1<<CS01, 169 ; a    #0      .db 1<<CS01,  63 ; d'  #10      .db 1<<CS00, 190 ; G   #20
.db 1<<CS01, 151 ; h    #1      .db 1<<CS01,  56 ; e'  #11      .db 1<<CS00, 169 ; A'  #21
.db 1<<CS01, 142 ; c    #2      .db 1<<CS01,  53 ; f'  #12      .db 1<<CS00, 151 ; H'  #22
.db 1<<CS01, 127 ; d    #3      .db 1<<CS01,  47 ; g'  #13      .db 1<<CS00, 142 ; C'  #23
.db 1<<CS01, 113 ; e    #4      .db 1<<CS01,  42 ; A   #14      .db 1<<CS00, 127 ; D'  #24
.db 1<<CS01, 106 ; f    #5      .db 1<<CS01,  37 ; H   #15      .db 1<<CS00, 113 ; E'  #25
.db 1<<CS01,  95 ; g    #6      .db 1<<CS01,  35 ; C   #16      .db 1<<CS00, 106 ; F'  #26
.db 1<<CS01,  84 ; a'   #7      .db 1<<CS01,  31 ; D   #17      .db 1<<CS00,  95 ; G'  #27
.db 1<<CS01,  75 ; h'   #8      .db 1<<CS00, 227 ; E   #18      .db 1<<CS00,  84 ; A'' #28
```

Die Tabelle nimmt jetzt 29 Worte im Flash-Speicher ein. Das sind 5,7 % des Speichers und ist verträglich.

Wenn wir jetzt die beiden Bytes der zehnten Note holen und in den Timer schreiben wollen, lautet der Code dafür so:

```
ldi R16,10 ; zehnte Note
lsl R16 ; Note mal zwei (2 Bytes pro Note)
ldi ZH,HIGH(2*Notentabelle)
ldi ZL,LOW(2*Notentabelle)
add ZL,R16 ; addiere Note zu Zeiger
ldi R16,0 ; CLR wuerde Carry-Flagge loeschen
adc ZH,R16 ; addiere eventuelles Carry
lpm R0,Z+ ; lese Vorteiler
out TCCR0B,R0 ; schreibe in Timer-Kontrollport B
lpm R0,Z ; lese CTC-Wert
out OCR0A,R0 ; schreibe in CTC-Vergleichswert
```

Neu ist die Instruktion ADD, die zwei Register addiert. Auch die Instruktion ADC ist neu, es addiert und zählt noch das Carry-Bit (Übertragsbit) dazu. ADD/ADC können zur 16-Bit-Addition verwendet werden.

Damit wäre das Auslesen und Verwenden der Notentabelle für unseren Fall gelöst.

9.4.4 Einführung in die Multiplikation

Damit hätten wir das Problem mit den Noten gelöst, laufen aber geradewegs in ein anderes: unser AD-Wandler liefert Werte zwischen 0 und 1.023 (ohne ADLAR) bzw. 0 und 255 (mit ADLAR). Unsere Notentabelle hat aber nur 29 Noten. Wir könnten jetzt alle Werte oberhalb von 28 einfach auf 28 setzen und das Problem wäre schon gelöst. So richtig schön ist diese Lösung nicht, weil sich die 29 Noten dann auf den unteren 2,8 % (10-Bit-ADC) bzw. 11 % (8-Bit-ADC) des Potentiometers eng drängeln und der obere Teil arg langweilig immer nur A'' liefert.

Irgendwie müssen wir die ankommenden 1.023 oder 255 zu 28 verkleinern. Der C-Programmierer ist da jetzt fein raus: er teilt den Wert einfach durch 36,5 bzw. durch 9,1 und rundet. Und schwupp-die-wupp hat er sich die Fließkommabibliothek in seinen Code geholt. Und die alleine ist schon so groß, dass sie nicht mehr in den Flashspeicher passt. Der C-Programmierer steigt jetzt auf einen ATxmega um, mit reichlich Speicher, damit sich seine Monsterbibliothek darin pudelwohl fühlt. Wir ersetzen solche Monster aber gerne mit Intelligenz und denken uns was Passendes dafür aus.

Die Aufgabe lautet (mit ADLAR, 10 Bit Auflösung nicht nötig) eigentlich:

$$\text{Ergebnis} = 29 * \text{ADC} / 256$$

Und teilen durch 256 ist in der digitalen Prozessorwelt ja sowas von einfach: einfach das LSB des Multiplikationsergebnisses wegstreichen und nur das MSB nehmen.

9.4.4.1 Einfachstmultiplikation

Bleibt die Multiplikation des ADC-Wertes mit 29. Das ist eine einfache Aufgabe: einfach den ADC-Wert 29 mal aufaddieren. Z. B. so:

```
  in R0,ADCH ; Wert aus dem ADC in R0 lesen, +1 = 1
  clr R2 ; R2:R1 ist das Ergebnis, +1 = 2
  clr R1 ; +1 = 3
  ldi R16,29 ; Multiplikation mit 29, +1 = 4
Schleife:
  add R1,R0 ; Dazu zaehlen, +29*1 = 33
  brcc NachUeberlauf ; kein Ueberlauf, +29*1 = 62
  inc R2 ; MSB eins hoeher, +29*1 = 91
NachUeberlauf:
  dec R16 ; abwaerts zaehlen, + 29*1 = 130
  brne Schleife ; noch mal addieren, +28*2 + 1 = 187
```

Die 187 Takte, die benötigt werden, sind nicht so arg lange. Bei 1,2 MHz sind das 156 µs, weniger als eine einzige NF-Schwingung. Aber es geht schneller.

9.4.4.2 Schneller multiplizieren

Multiplizieren mit zwei ist in der Binärwelt besonders einfach und schnell. Wir könnten die nächste Zweierpotenz ansteuern und dann dazuzählen oder abziehen. Das ginge so:

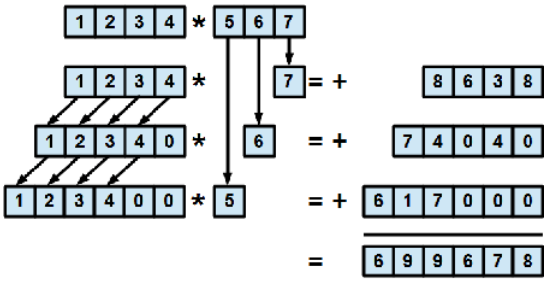
```
  in R0,ADCH ; Wert aus dem ADC in R0 lesen, +1 = 1
  clr R2 ; R2:R1 ist das Ergebnis, +1 = 2
  mov R1,R0 ; Wert einmal kopieren, +1 = 3
  lsl R1 ; LSB mit zwei malnehmen, +1 = 4
  rol R2 ; MSB mit zwei malnehmen und Carry hineinschieben, +1 = 5
  lsl R1 ; mit vier malnehmen, +1 = 6
  rol R2 ; +1 = 7
  lsl R1 ; mit acht malnehmen, +1 = 8
  rol R2 ; +1 = 9
  lsl R1 ; mit 16 malnehmen, +1 = 10
  rol R2 ; +1 = 11
  lsl R1 ; mit 32 malnehmen, +1 = 12
  rol R2 ; +1 = 13
  sub R1,R0 ; einmal abziehen, +1 = 14
  brcc KeinCarry1 ; kein Carry, +1/2 = 15/16
  dec R2 ; MSB vermindern, +1 = 16
KeinCarry1:
  sub R1,R0 ; zweimal abziehen, +1 = 17
  brcc KeinCarry2 ; kein Carry, +1/2 = 18/19
  dec R2 ; MSB vermindern, +1 = 19
KeinCarry2:
  sub R1,R0 ; dreimal abziehen, +1 = 20
  brcc KeinCarry3 ; kein Carry, +1/2 = 21/22
  dec R2 ; MSB vermindern, +1 = 22
```

Neu ist die Instruktion ROL Register, ROtate Left. Sie ist das Pendant zu ROR, rollt aber das Übertragbit nach links in das Register und Bit 7 des Registers in das Übertragsbit.

Das ist mehr als acht mal kürzer als die Primitivmultiplikation. Aber es geht noch schneller.

9.4.4.3 Noch schneller multiplizieren

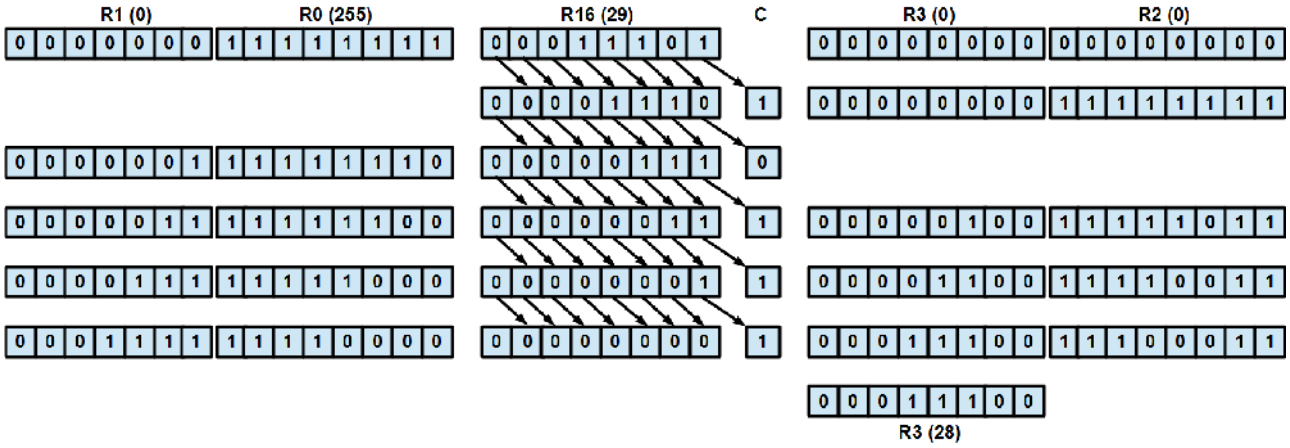
Die bisherigen Lösungen sind eng auf die konkrete Aufgabe zugeschnitten. Die echte binäre Multiplikation, anwendbar auf alle Zahlen mit je acht Bit, ist aber gar nicht so kompliziert, dass auch C-Programmierer sie erlernen können, wenn sie mal nicht auf Riesenbibliotheken und Riesenchips umschalten wollen.



Die binäre Multiplikation ist sogar einfacher als die Dezimalmultiplikation. Die geht bekanntlich so. Die erste Zahl wird mit den Einern der zweiten Zahl malgenommen. Dann wird die erste Zahl um eine Stelle nach links geschoben, mit den Zehnern der zweiten Zahl malgenommen und zum Ergebnis addiert. Dann erneutes Linksschieben und Malnehmen mit den Hunderten. Die Summe, das Ergebnis der Multiplikation ist fertig.

Die binäre Multiplikation ist noch einfacher, weil es ja nur zwei Ziffern gibt. Entweder wird also die links geschobene Zahl addiert (Ziffer ist Eins) oder nicht (Ziffer ist Null).

Die Multiplikation von 255 mit 29 zeigt das Bild.



Die zweite Zahl in R16 wird rechts geschoben, dadurch gelangt die nächste Ziffer in das Carry-Bit im Statusregister (C). Ist das eine Null, wird die erste Zahl nicht addiert. Ist es eine Eins, wird sie addiert (16-Bit-Addition mit Übertrag). Dann wird die erste Zahl um eine Stelle als links geschoben (16-Bit-Linksschieben). Es folgt Rechtsschieben des nächsten Bits in der zweiten Zahl, Nichtaddieren/Addieren, etc. Sind alle Einsen herausgeschoben, ist die Multiplikation beendet.

So sieht der Code aus.

```
  in R0,ADCH ; lese MSB vom ADC als LSB, 1
  clr R1 ; MSB leeren, +1 = 2
  clr R2 ; Ergebnis LSB leeren, +1 = 3
  clr R3 ; dto., MSB, +1 = 4
  ldi R16,29 ; Multiplikant setzen, +1 = 5
Schleife:
  lsr R16 ; niedrigstes Bit in Carry, +5*1 = 10
  brcc NachAddieren ; C = Null, +1*2+4*1 = 16
  add R2,R0 ; addieren LSB, +5*1 = 21
  adc R3,R1 ; addieren MSB mit Carry, +5*1 = 21
NachAddieren:
  lsl R0 ; Linksschieben erste Zahl, MSB, +5*1 = 26
  rol R1 ; Carry in MSB, +5*1 = 31
  tst R16 ; Ende erreicht?, +1*5 = 36
  brne Schleife ; noch Einsen vorhanden, +4*2+1*1 = 45
```

Das Ergebnis (28) steht im Register R3.

Ok, das sind 45 Takte, also mehr als die Zweierpotenzlösung. Dafür funktioniert die Routine mit jeder beliebigen 8-Bit-Zahl in gleicher Weise. Und ist so einfach, dass es auch denkfaule C-Programmierer intellektuell bewältigen könnten.

Damit ist unser Problem gelöst, wie aus 255 nur noch 28 werden. Und ganz ohne Teilen und Fließkomma-Bibliothek. Fast jedes rechnerische Problem lässt sich damit lösen, wenn man es irgendwie zu einer Multiplikation umgebogen kriegt.

9.4.5 Das Programm

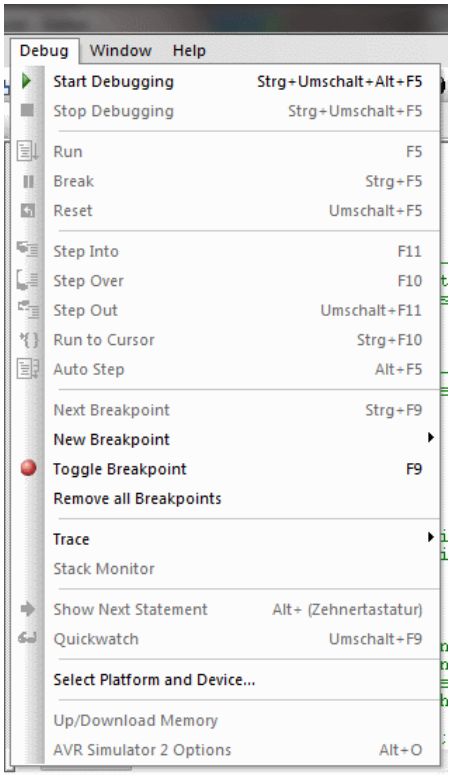
Damit haben wir alle Grundlagen zusammen, um loszulegen. Das Programm steht unten (hier geht es zum Quellcode im asm-Format).

```
;
; ************************************
; * Tonleiter-Toene mit dem ATtiny13  *
; * (C)2016 by www.gsc-elektronic.net *
; ************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; -------- Programmablauf -------------
;
; Der AD-Wandler ermittelt staendig die
; Spannung am Potentiometereingang,
; wandelt das Ergebnis durch Multipli-
; kation mit 29 und Teilen des Ergebnis-
; ses durch 256 in die Nummer eines Tons
; zwischen 0 und 28 um. Fuer diesen Ton
; wird aus einer Notentabelle die Timer-
; vorteiler- und die Vergleichsregister-
; A-Einstellung ermittelt und in den
; Timer 0 geschrieben.
; Die Taste schaltet die Tonausgabe an
; und aus.
;
; --------- Register ------------------
; Verwendet: R0 fuer LPM und Berechnungen
; Verwendet: R1 fuer Berechnungen
.def rMultL = R2 ; Multiplikator, LSB
.def rMultH = R3 ; dto., MSB
; frei: R4 .. R14
.def rSreg = R15 ; Statusregister sichern
.def rmp = R16 ; Vielzweckregister
.def rimp = R17 ; Vielzweckregister Interrupts
.def rFlag = R18 ; Flaggenregister
  .equ bAdcR = 0 ; ADC-Wert eingelesen
; frei R18 .. R29
; Verwendet: R31:R30, ZH:ZL fuer LPM
;
; --------- Ports ---------------------
.equ pOut = PORTB ; Ausgabeport
.equ pDir = DDRB ; Richtungsport
.equ pInp = PINB ; Eingangsport
.equ bLspD = DDB0 ; Lautsprecherausgang
.equ bTasO = PORTB3 ; Pullup Tasteneingang
.equ bTasI = PINB3 ; Tasteninputpin
.equ bAdID = ADC2D ; ADC-Input-Disable
;
; --------- Timing --------------------
; Takt = 1200000 Hz
; Vorteiler = 1 und 8
; CTC-TOP-Bereich = 0 .. 255
; CTC-Teiler-Bereich = 1 .. 256
; Toggle-Teiler = 2
; Frequenzbereich: 600 kHz .. 293 Hz
;
; ---- Reset- und Interruptvektoren ---
.CSEG ; Assemblieren in den Code-Bereich
.ORG 0 ; An den Anfang
  rjmp Start ; Rest-Vektor, Init
  reti ; INT0-Int, nicht aktiv
  rjmp PcIntIsr ; PCINT-Int, aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  rjmp AdcIsr ; ADC-Int, aktiv
;
; ---------- Interrupt Service Routinen -----
PcIntIsr: ; PCINT-Interrupt Tasten-Interrupt
  sbic pInp,bTasI ; Ueberspringe bei Taste = 0
  rjmp PcIntIsrAus ; Taste ist nicht gedrueckt
  ; Tonausgabe einschalten
  ldi rimp,(1<<COM0A0)|(1<<WGM01) ; Toggle, CTC-A
  out TCCR0A,rimp ; in Kontrollregister A
  rjmp PcIntIsrRet ; zurueck
PcIntIsrAus:
  ; Tonausgabe ausschalten
  ldi rimp,(1<<COM0A1)|(1<<WGM01) ; Clear, CTC-A
  out TCCR0A,rimp
PcIntIsrRet:
  reti
;
; ADC Ready Interrupt
;
; Wird vom AD-Wandler ausgeloest, wenn die Wand-
; lung beendet ist.
; Liest die oberen 8 Bit des Ergebnisses in das
; Register rMultL und setzt die bAdcR-Flagge.
; Startet die naechste Wandlung.
;
AdcIsr: ; ADC-Interrupt
  in rSreg,SREG ; sichern Statusregister
  in rMultL,ADCH ; lese MSB Ergebnis
  sbr rFlag,1<<bAdcR ; Flagge Neuer ADC-Wert
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rimp ; in ADC-Kontrollregister A
  out SREG,rSreg ; wiederherstellen Statusregister
  reti
;
; ---------- Programmstart und Init ----------
Start:
  ; Stapel einrichten
  ldi rmp,LOW(RAMEND) ; SRAM-Ende
  out SPL,rmp ; in Stackzeiger
  ; In- und Output-Ports
  ldi rmp,1<<bLspD ; Lautsprecherausgang Richtung
  out pDir,rmp ; in Richtungsregister
  ldi rmp,1<<bTasO ; Pullup am Tastenport
  out pOut,rmp ; in Ausgangsregister
  ; Timer als CTC konfigurieren
  ldi rmp,(1<<COM0A1)|(1<<WGM01) ; Clear, CTC-A
  out TCCR0A,rmp ; in Kontrollregister A
  ; Vorteiler und Timerstart erfolgt durch ADC-
    ; Interrupt
  ; AD-Wandler konfigurieren und starten
  ldi rmp,(1<<ADLAR)|(1<<MUX1) ; Linksjustieren,
    ; ADC2
  out ADMUX,rmp ; in ADC-MUX
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; in Kontrollregister A, starten
  ; PCINT fuer Tasteneingang
  ldi rmp,1<<PCINT3 ; PB3-Int ermoeglichen
  out PCMSK,rmp ; in PCINT-Maskenregister
  ldi rmp,1<<PCIE ; PCINT ermoeglichen
  out GIMSK,rmp ; in Interrupt-Maskenregister
  ; Schlafen ermoeglichen
  ldi rmp,1<<SE ; Schlafen, Idle-Modus
```

```
  out MCUCR,rmp ; in MCU-Kontrollregister
  ; Interrupts einschalten
  sei
; ------------ Hauptprogramm-Schleife --------
Schleife:
  sleep ; schlafen legen
  nop ; Aufwachen
  sbrc rFlag,bAdcR ; Ueberspringe wenn Adc-Flagge
Null
  rcall AdcCalc ; Umrechnung von ADC-Wert in Ton
  rjmp Schleife ; wieder schlafen legen
;
; ------------ AD-Wert setzen ----------------
; ADC-Wert in Tonhoehe wandeln und Timer starten
; AD-Wert ist in rMultL
; Ausgeloest von der gesetzten Flagge bAdcR, die
; der AD-Wandler nach jeder Wandlung setzt.
;
AdcCalc:
  cbr rFlag,1<<bAdcR ; Flagge zuruecksetzen
  ; ADC-Wert mit 29 multplizieren
  clr rMultH ; MSB loeschen
  clr R0 ; Ergebnis LSB Null setzen
  clr R1 ; dto., MSB
  ldi rmp,29 ; Anzahl Toene plus Eins
AdcCalcShift:
  lsr rmp ; niedrigstes Bit in Carry
  brcc AdcCalcNachAdd
  add R0,rMultL ; addiere LSB zum Ergebnis
  adc R1,rMultH ; addiere MSB mit Carry
AdcCalcNachAdd:
  lsl rMultL ; mal zwei schieben
  rol rMultH ; in MSB rollen und mal zwei
  tst rmp ; rmp schon leergeschoben?
  brne AdcCalcShift
  ; Ton aus Tabelle holen
  lsl R1 ; Tonnummer mal zwei
  ldi ZH,HIGH(2*Tonleitertabelle) ; Z auf Tabelle
  ldi ZL,LOW(2*Tonleitertabelle)
  add ZL,R1 ; Tonnummer addieren
  ldi rmp,0 ; Ueberlauf addieren
  adc ZH,rmp
  lpm R0,Z+; Tabellenwert LSB in R0 einlesen
  out TCCR0B,R0 ; in Timerregister B
  lpm ; Tabellenwert MSB in R0 einlesen
  out OCR0A,R0 ; in Vergleichsregister A
  ret ; fertig
;
; ------- Tonleitertabelle -----------
Tonleitertabelle:
.db 1<<CS01, 169 ; a #0
.db 1<<CS01, 151 ; h #1
.db 1<<CS01, 135 ; cis #2
.db 1<<CS01, 127 ; d #3
.db 1<<CS01, 113 ; e #4
.db 1<<CS01, 101 ; fis #5
.db 1<<CS01, 90 ; gis #6
.db 1<<CS01, 84 ; a' #7
.db 1<<CS01, 75 ; h' #8
.db 1<<CS01, 67 ; cis' #9
.db 1<<CS01, 63 ; d' #10
.db 1<<CS01, 56 ; e' #11
.db 1<<CS01, 50 ; fis' #12
.db 1<<CS01, 44 ; gis' #13
.db 1<<CS01, 42 ; A #14
.db 1<<CS01, 37 ; H #15
.db 1<<CS01, 33 ; CIS #16
.db 1<<CS01, 31 ; D #17
.db 1<<CS00, 226 ; E #18
.db 1<<CS00, 204 ; FIS #19
.db 1<<CS00, 181 ; GIS #20
.db 1<<CS00, 169 ; A' #21
.db 1<<CS00, 151 ; H' #22
.db 1<<CS00, 135 ; CIS' #23
.db 1<<CS00, 127 ; D' #24
.db 1<<CS00, 113 ; E' #25
.db 1<<CS00, 101 ; FIS' #26
.db 1<<CS00, 90 ; GIS' #27
.db 1<<CS00, 84 ; A'' #28
;
; Ende Quellcode
;
```

Neben den schon eingeführten Instruktionen LPM mit allen seinen Untervarianten sind keine neuen Instruktionen verwendet.

Noch ein wichtiger Hinweis: Am Pin 5 befindet sich jetzt der Elko und ein niederohmiger Lautsprecher. Beim Programmieren steht dies den hochfrequenten Programmierimpulsen im Weg, wir kriegen Fehlermeldungen beim Programmieren. Vor dem Programmieren also einfach den Elko abziehen und nach dem Programmieren wieder reinstecken.



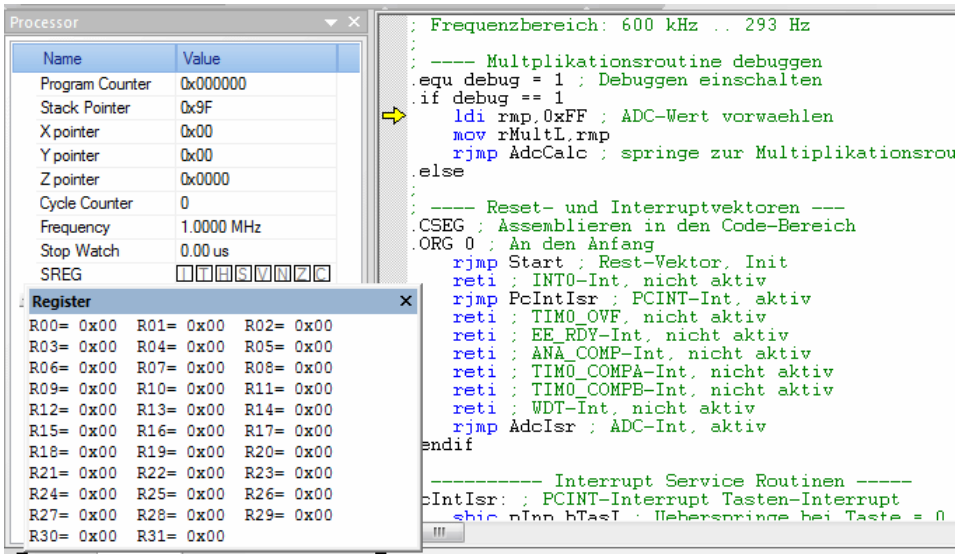
9.4.6 Debuggen mit dem Studio

Wenn wir solche Routinen schreiben wie die Multiplikationsroutine oder das Auslesen aus der Tabelle wüssten wir schon gerne, ob da auch das richtige Ergebnis herauskommt. Spätestens dann, wenn auf Tastendruck aus dem Lautsprecher nichts heraustönt. Es gibt eine Möglichkeit, dem Prozessor bei der Arbeit zuzugucken und jeden Schritt einzeln zu beobachten. Sie heißt Simulator und ist im Studio eingebaut.

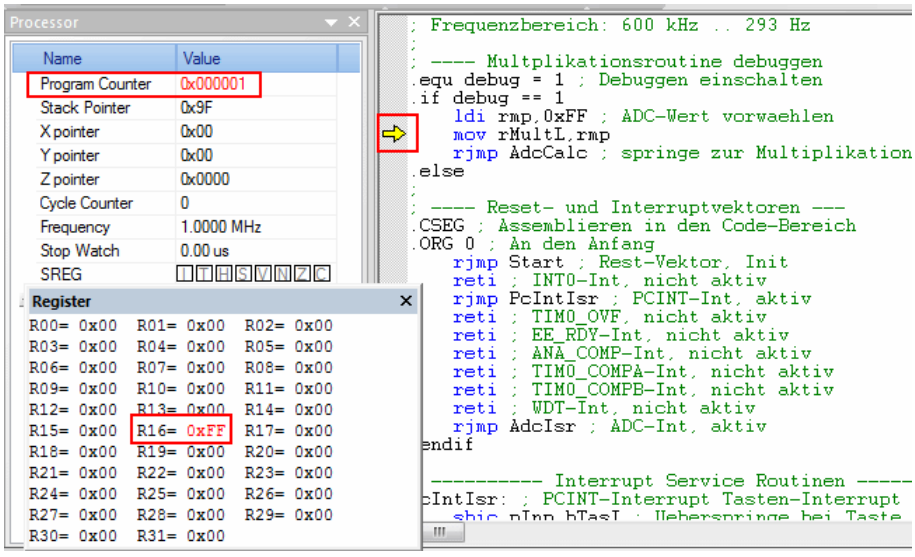
Um den Simulator zu starten, drücken wir im Debug-Menue einfach "Start Debugging". Bevor wir das tun, fügen wir in unseren Quellcode die folgenden Zeilen ein:

```
; ---- Multplikationsroutine debuggen

.equ debug = 1 ; Debuggen einschalten
.if debug == 1
  ldi rmp,0xFF ; ADC-Wert vorwaehlen
```

```
  mov rMultL,rmp
  rjmp AdcCalc ; springe zur Multiplikationsroutine
.else
;
; ---- Reset- und Interruptvektoren ---
.CSEG ; Assemblieren in den Code-Bereich
.ORG 0 ; An den Anfang
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0-Int, nicht aktiv
  rjmp PcIntIsr ; PCINT-Int, aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  reti ; TIM0_COMPA-Int, nicht aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  rjmp AdcIsr ; ADC-Int, aktiv

.endif
;
```



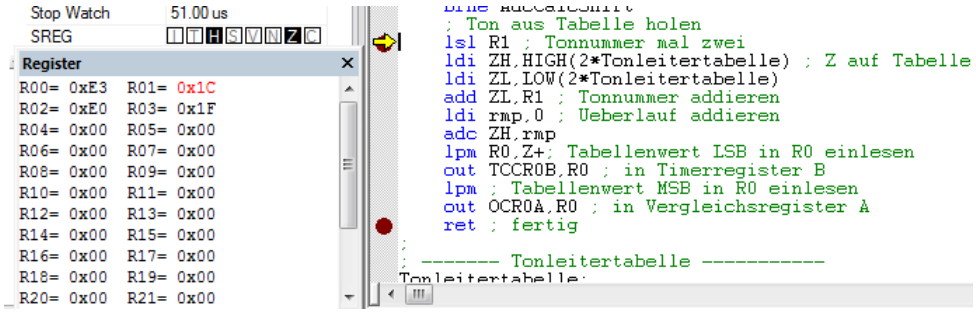
Die Zeilen maskieren mit dem Schalter "debug" die Reset- und Vektortabelle: Ist der Schalter == 1, dann assembliert der Assembler die drei markierten Zeilen, die Reset- und Interruptvektortabelle nicht. Damit springt die Simulation mit dem Wert 0xFF im Register rMultL direkt in die Multiplikationsroutine. Der Grund für diese Maskierung der Reset- und Vektortabelle ist die ".ORG"-Direktive: sie würde beim Assemblieren zu einem Fehler führen, weil die drei eingefügten Instruktionen die Ausführungsadresse erhöhen und mit ".ORG" immer nur vorwärts gesprungen werden kann. Alternativ könnten wir auch die Multiplikationsroutine herauskopieren und als "stand alone" assemblieren, weil uns der Rest des Programmes ja eigentlich nicht interessiert.



Nach dem Start des Simulators bietet dieser eine Vielzahl an Informationen zum Zustand. Man kann den Inhalt von Registern beobachten (links unten), man kann eine Stopuhr starten, die Instruktionen und Ausführungszeiten werden angezeigt (links Mitte). Der Cursor in gelb (rechts oben) zeigt auf die erste ausführbare Instruktion.

Mit "Step into" (F11) wird die Ausführung der Instruktion simuliert. Ausgeführt wurde "ldi rmp,0xFF", mit rmp = R16. In der Registerliste ist der geänderte Wert rot markiert, der Programmzähler steht auf 0001 und der Cursor steht auf der nächsten Instruktion.

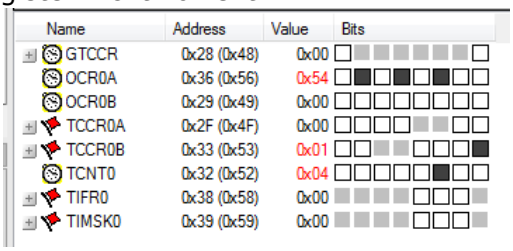
Um nicht jede Instruktion einzeln ausführen zu müssen, können wir "Breakpoints" setzen. Dazu setzt man den Cursor in die betreffende Zeile, drückt die rechte Maustaste und wählt "Toggle breakpoint". Dann führt der Simulator mit "Run" im Debug-Menue alle Instruktionen aus, bis er auf einen solchen Breakpoint stößt und hält dann an.



In unserem Fall sind das zwei Punkte: das Ende der Multiplikationsroutine und das Ende des Schreibens in die Timerregister. Im ersten Fall steht im Register R1 0x1C, was dezimal 28 bedeutet.

```
    clr R0 ; Ergebnis LSB Null setzen
    clr R1 ; dto., MSB
    ldi rmp,29 ; Anzahl Toene plus Eins
AdcCalcShift:
    lsr rmp ; niedrigstes Bit in Carry
    brcc AdcCalcNachAdd
    add R0,rMultL ; addiere LSB zum Ergebnis
    adc R1,rMultH ; addiere MSB mit Carry
AdcCalcNachAdd:
    lsl rMultL ; mal zwei schieben
    rol rMultH ; in MSB rollen und mal zwei
    tst rmp ; rmp schon leergeschoben?
    brne AdcCalcShift
    ; Ton aus Tabelle holen
    lsl R1 ; Tonnummer mal zwei
    ldi ZH,HIGH(2*Tonleitertabelle) ; Z auf Tabelle
    ldi ZL,LOW(2*Tonleitertabelle)
    add ZL,R1 ; Tonnummer addieren
    ldi rmp,0 ; Ueberlauf addieren
    adc ZH,rmp
    lpm R0,Z+; Tabellenwert LSB in R0 einlesen
    out TCCR0B,R0 ; in Timerregister B
    lpm ; Tabellenwert MSB in R0 einlesen
    out OCR0A,R0 ; in Vergleichsregister A
    ret ; fertig
```

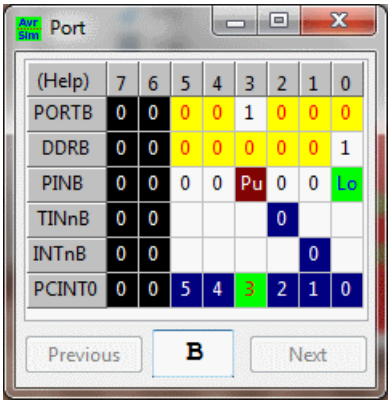
Das ist der Zustand des Timers beim zweiten Breakpoint. Die beiden Register, die beschrieben wurden (TCCR0B und OCR0A) zeigen die richtigen Werte aus der Notentabelle (0x01 und 0x54). Da wir das Timer-Init im Hauptprogramm übersprungen haben, sind die anderen Timer-Register nicht korrekt.



Mit diesem Werkzeug kommen wir Fehlern in den eigenen Programmen auf die Spur.

Top	Home	Einführung	Hardware	Tonhöhe	Tonleiter	Melodie

9.4.7 Simulation mit avr_sim

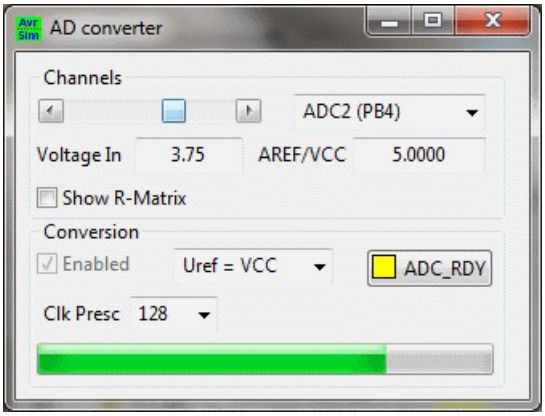


Die Simulation mit avr_sim zeigt die folgenden Abläufe. Simuliert wird die Multiplikationsroutine und der Zugriff auf die Notentabelle mit LPM.

Um die Hardware zu initiieren steppen wir durch die ersten Instruktionen bis die Interrupts mit *SEI* ermöglicht werden.

Wie im ersten Fall wird der Pin PB0 als Ausgang geschaltet und mit Low-Potential initiiert. PB3 hat wie immer seinen Pull-up-Widerstand eingeschaltet und ist in der PCINT-Maske als Interruptquelle bei allen Flankenwechseln nominiert.

Der AD-Wandler misst wieder laufend die Spannung am ADC2-Eingang, was dem I/O-Pin PB4 entspricht. Die simulierte Spannung ist diesmal 3,75 V, was nach der Wandlung 3,75 / 5,00 * 1.023 = 767 oder linksadjustiert 191 dezimal oder 0xBF entspricht.



Nach Erreichen des ADC-Ready-Interrupts hat die Interrupt-Service-Routine die obersten 8 Bits in das Register R2 geschrieben und die bAdcR-Flagge gesetzt. Ihre Bearbeitung findet ausserhalb der Interrupt-Sevice-Routine, nach dem Aufwachen vom *SLEEP* statt, weil sie die Interrupts zu lange blockieren würde. In der Routine AdcCalc: erfolgt die Umrechnung von ADC-Werten in Noten. Die folgenden Instruktionen zeigen zuerst die Multiplikation, um aus dem ADC-Wert einen Wert von 0 bis 29 zu fabrizieren:

```
clr rMultH ; MSB loeschen
clr R0 ; Ergebnis LSB Null setzen
clr R1 ; dto., MSB
ldi rmp,29 ; Anzahl Toene plus Eins
```

SREG

I	T	H	S	V	N	Z	C
1	0	0	1	1	0	0	1

Register

Reg	+0	+1	+2	+3
R0	00	00	BF	00
R8	00	00	00	00
R16	0E	CF	00	00
R24	00	00	00	00

Register

Reg	+0	+1	+2
R0	00	00	BF
R8	00	00	00
R16	1D	CF	00
R24	00	00	00

clr rMultH (R3) und *clr R0/R1* löschen erstmal das High-Byte für das Multiplizieren und das Ergebnis. *ldi rmp,29* wird auf die Anzahl der Notentöne gesetzt. Nun kann die Multiplikation beginnen.

Die Anweisung

```
lsr rmp ; niedrigstes Bit in Carry
```

schiebt R16 nach rechts und das niedrigste Bit in R16 in die Carry-Flagge im Statusregister. In diesem Fall ist es eine Eins.

Register

Reg	+0	+1	+2	+3
R0	BF	00	7E	01
R8	00	00	00	00
R16	0E	CF	00	00
R24	00	00	00	00

Register

Reg	+0	+1	+2
R0	BF	00	BF
R8	00	00	00
R16	0E	CF	00
R24	00	00	00

Weil es eine Eins ist, addieren die folgenden Instruktionen

```
brcc AdcCalcNachAdd
add R0,rMultL ; addiere LSB zum
Ergebnis
adc R1,rMultH ; addiere MSB mit Carry
```

bei gesetzter C-Flagge die 16-Bit-Zahl in R3:R2 zum Ergebnis in R1:R0. Da es 16-Bits sind, addiert *adc* eventuelle Überläufe (Carry) aus der unteren Addition mit dazu.

Die beiden Instruktionen

```
lsl rMultL ; mal zwei schieben
rol rMultH ; in MSB rollen und mal zwei
```

schieben den Multiplikator in R3:R2 um eine Position nach links und multiplizieren damit den Inhalt mit zwei. Die Instruktion *rol* schiebt noch das aus dem unteren Byte stammende Carry in das MSB ein.

SREG

I	T	H	S	V	N	Z	C
1	0	0	0	0	0	0	0

Register

Reg	+0	+1	+2	+3
R0	BF	00	7E	01
R8	00	00	00	00
R16	07	CF	00	00
R24	00	00	00	00

Register

Reg	+0	+1	+2	+3
R0	BF	00	FC	02
R8	00	00	00	00
R16	07	CF	00	00
R24	00	00	00	00

Weil R16 noch immer Bits zum Multiplizieren enthält, wird es wieder rechts geschoben (geteilt durch 2) und das niedrigste Bit in das Carry geschoben. Diesmal ist Carry 0 und das Addieren von R3:R2 zum Ergebnis in R1:R0 entfällt.

Register

Reg	+0	+1	+2	+3
R0	A3	15	F0	0B
R8	00	00	00	00
R16	00	CF	00	00
R24	00	00	00	00

Dann wird R3:R2 erneut mit Shift Left und Rotate Left mit zwei malgenommen.

Es gibt immer noch Einsen in rmp, daher müssen wir weiter durch zwei teilen und Bits in das Carry schieben. Diesmal ist es wieder eine Eins.

SREG

I	T	H	S	V	N	Z	C
1	0	0	1	1	0	0	1

Register

Reg	+0	+1	+2	+3
R0	BF	00	FC	02
R8	00	00	00	00
R16	03	CF	00	00
R24	00	00	00	00

Wieder wird daher R3:R2 zu R1:R0, in 16-Bit-Manier mit *ADD* und *ADC*, addiert.

Register

Reg	+0	+1	+2	+3
R0	BB	03	FC	02
R8	00	00	00	00
R16	03	CF	00	00
R24	00	00	00	00

Die nachfolgende Multiplikation von R3:R2 mit zwei ist nicht dargestellt.

SREG

I	T	H	S	V	N	Z	C
1	0	0	1	1	0	0	1

Register

Reg	+0	+1	+2	+3
R0	BB	03	F8	05
R8	00	00	00	00
R16	01	CF	00	00
R24	00	00	00	00

Es wird langsam langweilig, aber rmp muss noch mal rechts geschoben werden. Weil dabei eine Eins herausrollt muss wieder addiert werden.

Register

Reg	+0	+1	+2	+3
R0	B3	09	F8	05
R8	00	00	00	00
R16	**01**	CF	00	00
R24	00	00	00	00

Das nächste Addieren von R3:R2 zu R1:R0 erfolgt hier.

Register

Reg	+0	+1	+2	+3
R0	**B3**	**09**	F0	0B
R8	00	00	00	00
R16	01	CF	00	00
R24	00	00	00	00

Und die nächste Multiplikation mit 2 für R3:R2 hier.

SREG

I	T	H	S	V	N	Z	C
1	0	0	1	1	0	1	1

Register

Reg	+0	+1	+2	+3
R0	B3	09	F0	0B
R8	00	00	00	00
R16	00	CF	00	00
R24	00	00	00	00

Nun rollt die letzte Eins in das Carry-Flag und wir sind fast fertig mit der Multiplikation.

Register

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	A3	2A	E0	17	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	CF	00	00	00	00	00	00
R24	00	00	00	00	00	00	92	00

Die letzte Addition von R3:R2 zu R1:R0. Die nachfolgende Multiplikation mit zwei ist eigentlich nicht mehr nötig, weil R16 ja schon leer ist.

Simulation status

```
Prog counter = $00002C
Instructions = 86
Stackpointer = $009F
Clock frequ. = 1,200,000 Hz
Time elapsed = 1.471667 ms
Stop watch   = 55.833333 us
```

Die Stoppuhr wurde benutzt um die Dauer der Multiplikation zu messen. Sie zeigt 55 µs an, was ziemlich schnell ist für die vielen Schieben-, Rotieren- und Addieren-Befehle. Jedenfalls kein vernünftiger Grund für das Wechseln auf einen ATmega (mit eingebauter schneller Hardware-Multiplikation) oder für das Hinzuladen einer ausgiebigen Rechenbibliothek. Das würde die gerade mal zwölf Instruktionen auf einige Hundert bis Tausend aufblasen und zu einem anderen Prozessortyp zwingen, weil dafür der Flashspeicher vom ATtiny13 zu klein ist. Arme C-Programmierer: ersetzen Intelligenz durch Blow-Up-Code!

Nach der Multiplikation ist die zu spielende Note 0x15 oder dezimal 21 im MSB des Ergebnisses in R1. Nun müssen wir das zu unserer Notentabelle addieren, um die Notenparameter aus der Tabelle ablesen zu können. Die Instruktionen

```
lsl R1 ; Tonnummer mal zwei
; Z auf Tabelle
ldi ZH,HIGH(2*Tonleitertabelle)
ldi ZL,LOW(2*Tonleitertabelle)
```

Register

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	A3	2A	E0	17	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	CF	00	00	00	00	00	00
R24	00	00	00	00	00	00	BC	00

nehmen das Ergebnis erst mal mal zwei, weil die Notentabelle pro Eintrag aus zwei Bytes besteht. Dann wird Z auf die Adresse des Tabellenanfangs (mal zwei wegen Low- und High-Byte) gestellt.

Die Tonleitertabelle ist hier:

```
150: ; ------- Tonleitertabelle -----------
151: Tonleitertabelle:
152: .db 1<<CS01, 169 ; a    #0
     000049 A902
153: .db 1<<CS01, 151 ; h    #1
     00004A 9702
```

Ihre Adresse ist also 0x000049, was sich zu 0x0092 in Z verdoppelt, um auf die Worttabelle byteweise zugreifen zu können.

Nun wird der verdoppelte Wert in R1 zur Tabellenadresse 0x0092 in Z addiert.

```
add ZL,R1 ; Tonnummer addieren
ldi rmp,0 ; Ueberlauf addieren
adc ZH,rmp
```

ZL und R1 werden normal addiert, das Ergebnis in ZL abgelegt. Um den Fall abzudecken, dass sich die Tabelle über die Bytegrenze erstreckt, wird ein eventuelles Carry noch zur MSB hinzu addiert. Um die Null und das Carry zu addieren, darf nicht *clr rmp* verwendet werden, denn das würde das Carry-Flag löschen. *ldi rmp,0* beeinflusst das Carry-Flag hingegen nicht. Z zeigt jetzt auf 0x00BC, was der Note an Adresse 0x00005E (mal zwei) entspricht. Dort steht korrekterweise Note #21:

```
173: .db 1<<CS00, 169 ; A'   #21
     00005E A901
```

Daher wird der Prescaler nun auf CS00 = 1 (Vorteiler = 1) und der Vergleichswert A auf 169 gesetzt. Das bedeutet eine Frequenz von

$$f = 1.200.000 / 1 / (169 + 1) / 2 = 3.529{,}4 \text{ Hz}$$

und entspricht ungefähr der Note A' (Sollwert: 3.520 Hz.)

Register

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	01	2A	E0	17	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	CF	00	00	00	00	00	00
R24	00	00	00	00	00	00	BD	00

Die Instruktionen

```
lpm R0,Z+; Tabellenwert LSB in R0
      ; einlesen
out TCCR0B,R0 ; in Timerregister B
```

lesen das LSB aus der Tabelle, erhöhen die Adresse Z und schreiben das gelesene Byte in das Timer-Kontrollregister B.

Nun kommt noch das zweite *lpm*, diesmal ohne irgendwas. Das liest das Byte an der schon erhöhten Adresse in das Register R0 (0xA9 oder dezimal 129). Das braucht dann nur noch ins Vergleichsregister A geschrieben zu werden und fertig ist der Lack.

```
lpm ; Tabellenwert MSB in R0 einlesen
out OCR0A,R0 ; in Vergleichsregister A
```

Register

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	A9	2A	E0	17	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	CF	00	00	00	00	00	00
R24	00	00	00	00	00	00	BD	00

Nun steht der Spielerei von Kammerton A' nix mehr im Weg herum, wenn wir per PCINT das Torkeln des PB0-Ausganges einschalten würden.

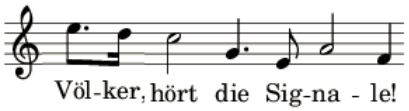
9.5 Aufgabe 3: Musikstück

9.5.1 Aufgabe

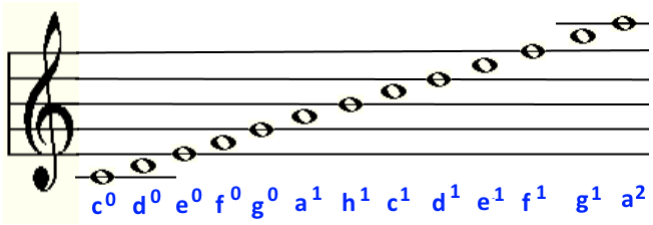
Auf Tastendruck soll der Prozessor "Völker hört die Signale" abspielen.

9.5.2 Das Musikstück

Das hier ist zu spielen.



Dieses sind die Notenhöhen, mit denen wir die Melodie in unsere Tonleitertabelle übersetzen müssen. Damit wir die Melodie in eine Tabelle übersetzen können wäre es angenehm, wenn wir diese Tabelle nicht mit Zahlen zwischen 0 und 28 sondern mit benannten Symbolen generieren könnten. Dazu wären die Noten mit ".EQU Notenname = Notennummer" zu definieren. Notennamen sollen mit "n" beginnen, dann den Ton mit "a" bis "g" und danach die Oktave mit "0" bis "4" angeben, also ungefähr so:




```
; Notensymbole      |.equ nf0 = 5       |.equ ne1 = 11      |.equ nD2 = 17      |.equ nC3 = 23
.equ na0 = 0        |.equ ng0 = 6       |.equ nf1 = 12      |.equ nE2 = 18      |.equ nD3 = 24
.equ nh0 = 1        |.equ na1 = 7       |.equ ng1 = 13      |.equ nF2 = 19      |.equ nE3 = 25
.equ nc0 = 2        |.equ nh1 = 8       |.equ nA2 = 14      |.equ nG2 = 20      |.equ nF3 = 26
.equ nd0 = 3        |.equ nc1 = 9       |.equ nH2 = 15      |.equ nA3 = 21      |.equ nG3 = 27
.equ ne0 = 4        |.equ nd1 = 10      |.equ nC2 = 16      |.equ nH3 = 22      |.equ nA4 = 28
```

Die in den Notenregeln verwendeten Klein- und Großbuchstaben sind in Assembler nicht verwendbar, weil dieser zwischen beiden nicht unterscheidet.

Mit diesen Definitionen lautet unsere Tontabelle für dieses Musikstück nun:

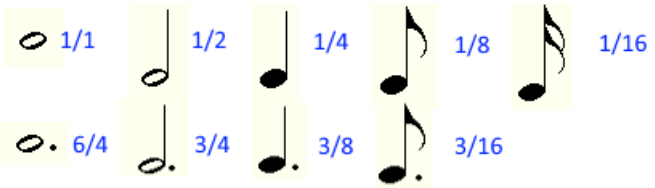
```
Musik:
.db ne1,nd1,nc1,ng0,ne0,na1,nf0,0xFF
```

Eine handhabbare Kodierung.

Das angefügte 0xFF signalisiert, dass damit die Musik zu Ende ist, weil sonst der Prozessor den gesamten Inhalt seines Flashspeichers als Musik interpretieren und abspielen würde.

9.5.3 Tondauer

Die Notensymbolik kodiert die Dauer, über die diese Noten ertönen, wie in der nebenstehenden Grafik gezeigt. Wir brauchen daher als zusätzliche Information noch die Dauern 1/2, 1/4, 3/8 und 1/8. In Achteln ausgedrückt: 1 (1/8), 2 (1/4), 3 (3/8) und 4 (1/2). Das bräuchte zwei Bits und wir könnten die Dauer in der Tabelle in die Noten dazukodieren, z. B. so:



```
.equ bLow = 1<<5 ; Dauerkodierung, low Bit
.equ bHigh = 1<<6 ; Dauerkodierung, high Bit
.equ d12 = bLow + bHigh ; 1/2, beide Dauerbits gesetzt
.equ d14 = bHigh ; 1/4, oberes Dauerbit gesetzt
.equ d38 = bLow ; 3/8, unteres Dauerbit gesetzt
.equ d18 = 0 ; 1/8, weder oberes noch unteres Dauerbit gesetzt
Musik:
.db ne1+d38,nd1+d14,nc1+d12,ng0+d38,ne0+d18,na1+d12,nf0+d14,0xFF
```

Um diese Noten zu dekodieren, würden wir folgendes programmieren:

```
  ; Z-Zeiger auf Musiktabelle setzen
  ldi ZH,HIGH(2*Musik) ; Zeiger auf Melodietab.
  ldi ZL,LOW(2*Musik)
  ; Notenbyte lesen
  lpm R17,Z ; lese erste Note aus Tabelle in R17
  ; Musikende feststellen
  cpi R17,0xFF ; Ende Musik?
  breq Musikaus ; Musik ausschalten
  ; Notendauer ermitteln
  andi R17,0b01100000 ; Bits 5 und 6 isolieren
  ldi R16,1 ; Anzahl Achtel Dauer
  sbrc R17,5 ; Bit 5 auswerten
  subi R16,-1 ; eins hinzuzählen (addi-Ersatz)
  sbrc R17,6 ; Bit 6 auswerten
  subi R17,-2 ; zwei hinzuzählen
  ; Note in Timertakt und Timer-CTC wandeln
  lpm R17,Z+ ; Note noch mal lesen, Zeiger hoeher
  andi R17,0b10011111 ; die Dauerbits loeschen
  [Wandeln und Note ausgeben]
Musikaus:
  [Musikausgabe abschalten]
```

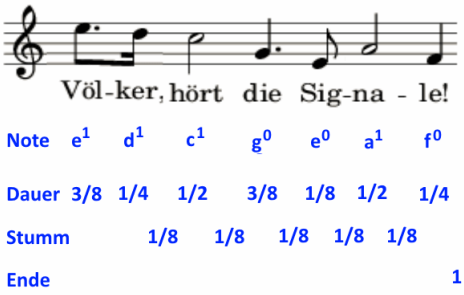
Wir könnten die Dauer aber auch als zweites Byte hinzufügen, was die Notenauswertung vereinfacht, aber die Länge der Notentabelle verdoppelt. Da wir mit dieser einfachen Melodie kaum an Flashgrenzen stoßen werden, kodieren wir das so:

```
Musik: ; LSB: Note oder FF, MSB: Dauer in Achtel
.db ne1,3,nd1,2,nc1,4,ng0,3,ne0,1,na1,4,nf0,2,0xFF,0xFF
```

Jetzt muss noch ein weiteres 0xFF angefügt werden, um wieder geradzahlig zu werden.

9.5.4 Tonpausen

Noten abspielen bedeutet nicht nur Töne, sondern auch Pausen zwischen den Tönen. Zwischen der ersten und der zweiten Note nicht, aber zwischen allen anderen. Wir brauchen daher neben dem Endezeichen 0xFF auch noch ein Pausenzeichen und wählen dafür 0xFE. Unsere Tabelle sieht daher so aus:




```
Musik: ; LSB: Note oder FF, MSB: Dauer in Achtel
.db ne1,3,nd1,2,0xFE,1,nc1,4,0xFE,1,ng0,3
.db 0xFE,1,ne0,1,0xFE,1,na1,4,0xFE,1,nf0,2,0xFF,0xFF
```

9.5.5 Spieldauer von Noten

Es kommt aber noch etwas hinzu, wenn wir mit der Tonleiter Musik abspielen wollen. Bei den Frequenzen zwischen 440 und 7.040 Hz liegen prozessortechnisch sehr unterschiedlich lange CTC-Sequenzen zugrunde. Bei 440 Hz entsprechen 880 CTC-Durchläufe einer Sekunde Dauer, bei 7.040 Hz aber 14.080. Die Anzahl Durchläufe pro Sekunde Spieldauer ist umgekehrt proportional zum CTC-Wert, wie er in der Tonleitertabelle steht. Wir könnten uns dem mit zwei Möglichkeiten nähern:

- Wir teilen 600.000 durch den den Vorteilerwert und den CTC-Wert und kriegen die Anzahl zu durchlaufender CTC-Zyklen pro Sekunde heraus, oder
- wir fügen diese Zahl unserer Tonleitertabelle hinzu und lesen sie mit LPM aus.

Wir sind faul, noch nicht reif zum Dividieren von 24-Bit-Ganzzahlen durch 8-Bit-Zahlen in Assembler und möchten dem Prozessor diese Prozedur nicht zumuten. Der C-Programmierer hat dieses Problem nicht und importiert seine Fließkomma-Bibliothek, um festzustellen, dass er doch einen ATxmega braucht.

Die Umformulierung unserer Notentabelle mit der Dauer des jeweiligen Tones in Achtel-Sekunden sieht dann so aus:

```
Notentabelle_Dauer:
.DB 1<<CS01, 169, 111, 0 ; a  #0
.DB 1<<CS01, 151, 124, 0 ; h  #1
.DB 1<<CS01, 142, 132, 0 ; c  #2
.DB 1<<CS01, 127, 148, 0 ; d  #3
.DB 1<<CS01, 113, 166, 0 ; e  #4
```

```
.DB 1<<CS01, 106, 177, 0 ; f  #5
.DB 1<<CS01,  95, 197, 0 ; g  #6
.DB 1<<CS01,  84, 223, 0 ; a' #7
.DB 1<<CS01,  75, 250, 0 ; h' #8
.DB 1<<CS01,  71,   8, 1 ; c' #9
.DB 1<<CS01,  63,  42, 1 ; d' #10
.DB 1<<CS01,  56,  79, 1 ; e' #11
.DB 1<<CS01,  53,  98, 1 ; f' #12
.DB 1<<CS01,  47, 143, 1 ; g' #13
.DB 1<<CS01,  42, 190, 1 ; A  #14
.DB 1<<CS01,  37, 251, 1 ; H   #15
.DB 1<<CS01,  35,  24, 2 ; C   #16
.DB 1<<CS01,  31,  93, 2 ; D   #17
.DB 1<<CS00, 227, 149, 2 ; E   #18
.DB 1<<CS00, 214, 189, 2 ; F   #19
.DB 1<<CS00, 190,  21, 3 ; G   #20
.DB 1<<CS00, 169, 120, 3 ; A'  #21
.DB 1<<CS00, 151, 225, 3 ; H'  #22
.DB 1<<CS00, 142,  32, 4 ; C'  #23
.DB 1<<CS00, 127, 157, 4 ; D'  #24
.DB 1<<CS00, 113,  47, 5 ; E'  #25
.DB 1<<CS00, 106, 135, 5 ; F'  #26
.DB 1<<CS00,  95,  43, 6 ; G'  #27
.DB 1<<CS00,  84, 250, 6 ; A'' #28
; Stumme Achteldauer
.DB (1<<CS01)|(1<<CS00), 256, 18, 0 ; Pause #254
```

Unsere Tabelle hat jetzt pro Ton vier Bytes oder zwei Worte und wir können uns der Programmierung nähern.

9.5.4 Programmablaufstruktur

Bei dieser Aufgabe muss

- bei einem Tastendruck entschieden werden, ob das Musikstück schon läuft. Wenn nicht, wird eine Startflagge gesetzt,
- die abzuspielende Note aus der Musiktabelle gelesen werden,
- die Tonhöhe dieser Note (der Vorteiler, der CTC-Wert), aus der Notentabelle gelesen werden und dem Timer mitgeteilt,
- die Dauer, über die die Note zu spielen ist, aus der Musiktabelle geholt werden muss (ganze Noten, halbe Noten, Viertelnoten, Achtelnoten), umgerechnet und einem 16-Bit-Zähler zugeführt werden,
- bei Pausen (0xFE) die Tonausgabe abgeschaltet, andernfalls eingeschaltet werden,
- das Ende des Stücks (0xFF) festgestellt werden, worauf der Timer abgeschaltet und nachfolgende Tastendrücke wieder zuzulassen sind.

Innerhalb der Interrupt-Service-Routinen werden folgende Dinge zu erledigen sein:

- PCINT: Abfrage der Polarität des Tasteneingangs, wenn Eins beenden, wenn Null dann T-Flagge abfragen, wenn ebenfalls Null dann Startflagge setzen.
- TC0-Compare-A-Int: 16-Bit-Zähler um Eins vermindern, wenn Null, dann Nullflagge setzen.

In der Hauptprogrammschleife werden beide Flaggen ausgewertet und entsprechend behandelt (Musikstück neu starten, nächste Note auswerten).

9.5.5 Programm

Das hier ist das fertige Programm (hier geht es zum Quellcode im asm-Format).

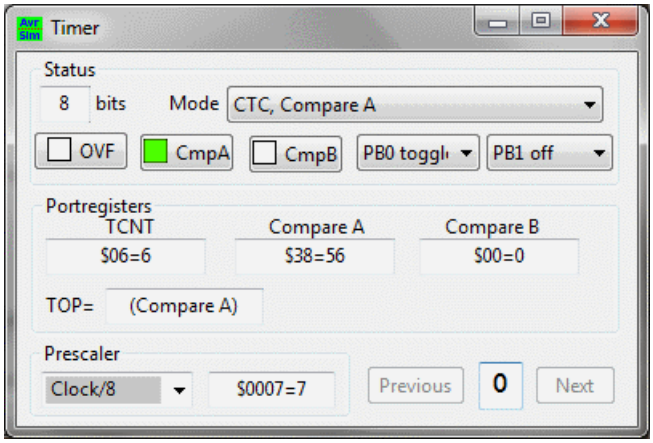
```
;
; ************************************
; * Musikstueck abspielen mit ATtiny13 *
; * (C)2016 by www.gsc-elektronic.net  *
; ************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; --------- Programmablauf ------------
;
; Beim Schliessen des Tasters wird ein PCINT
; ausgeloest und, falls das Abspielen der
; Melodie nicht schon im Gange ist, mit dem
; Einlesen und Abspielen der ersten Note der
; Melodie begonnen. Sind alle Noten und Pau-
; sen gespielt, kann von vorne begonnen wer-
; den.
;
; ------- Register ---------------------
; frei: R0 .. R14
.def rSreg = R15 ; Sichern Statusregister
.def rmp = R16 ; Vielzweckregister
.def rFlag = R17 ; Flaggenregister
  .equ bStart = 0 ; Starten Musikstueck
  .equ bNote = 1 ; naechste Note spielen
; frei: R18 .. R23
.def rCtrL = R24 ; 16-Bit-Zaehler CTC-Durchlaeufe
.def rCtrH = R25
; benutzt: X, XH:XL fuer Notendauerberechnung,
    ; Sichern Z
; benutzt: Y, YH:YL fuer Achteldauer
; benutzt: Z, ZH:ZL fuer Lesen aus
    ; Programmspeicher, Musikzeiger
;
; ------ Ports -------------------------
.equ pOut = PORTB ; Ausgabeport
.equ pDir = DDRB ; Richtungsport
.equ pInp = PINB ; Eingangsport
.equ bLspD = DDB0 ; Lautsprecherausgang
.equ bTasO = PORTB3 ; Pullup Tasteneingang
.equ bTasI = PINB3 ; Tasteninputpin
;
; --------- Timing --------------------
; Takt = 1200000 Hz
```

```
; Vorteiler = 1, 8, 64
; CTC-TOP-Bereich = 0 .. 255
; CTC-Teiler-Bereich = 1 .. 256
; Toggle-Teiler = 2
; Frequenzbereich: 600 kHz .. 36 Hz
;
; ---- Reset- und Interruptvektoren ---
.CSEG ; Assemblieren in den Code-Bereich
.ORG 0 ; An den Anfang
  rjmp Start ; Rest-Vektor, Init
  reti ; INT0-Int, nicht aktiv
  rjmp PcIntIsr ; PCINT-Int, aktiv
  reti ; TIM0_OVF, nicht aktiv
  reti ; EE_RDY-Int, nicht aktiv
  reti ; ANA_COMP-Int, nicht aktiv
  rjmp TC0CAIsr ; TIM0_COMPA-Int, aktiv
  reti ; TIM0_COMPB-Int, nicht aktiv
  reti ; WDT-Int, nicht aktiv
  reti ; ADC-Int, nicht aktiv
;
; ---- Reset- und Interruptvektoren ---
;
; PCINT Interrupt
; Wird von Tastenaenderungen ausgeloest. Ist
; der Tasteingang High, erfolgt nichts. Ist
; er Low, wird geprueft, ob die T-Flagge
; schon gesetzt ist. Wenn nicht, wird die
; Flagge bStart gesetzt.
;
PcIntIsr: ; PCINT Taste
  in rSreg,SREG ; Status retten
  sbic pInp,bTasI ; Ueberspringe bei Null
  rjmp PcIntIsrRet ; Fertig
  brts PcIntIsrRet ; T-Flagge gesetzt
  sbr rFlag,1<<bStart ; Start-Flagge setzen
PcIntIsrRet:
  out SREG,rSreg ; Status wieder herstellen
  reti
;
TC0CAIsr: ; Timer-CTC-A-Int
  in rSreg,SREG ; Status retten
  sbiw rCtrL,1 ; 16-Bit-Zaehler abwaerts
  brne TC0CAIsrRet ; noch nicht Null
  sbr rFlag,1<<bNote ; Flagge setzen
TC0CAIsrRet:
  out SREG,rSreg ; Status wieder herstellen
  reti
;
; ---- Programmstart, Init -------------
Start:
  ; Stapel initiieren
  ldi rmp,LOW(RAMEND) ; Stapelzeiger auf Ende
  out SPL,rmp
  ; T-Flagge loeschen
  clt ; Inaktiv
  ; Portpins einstellen
  ldi rmp,1<<bLspD ; Lautsprecherausgang Richtung
  out pDir,rmp ; in Richtungsregister
  ldi rmp,1<<bTasO ; Pullup am Tastenport
  out pOut,rmp ; in Ausgangsregister
  ; Timer als CTC starten
  ldi rmp,(1<<COM0A1)|(1<<WGM01) ; Clear, CTC-A
  out TCCR0A,rmp ; in Kontrollregister A
  ; Vorteiler, Timerstart und Int erfolgt durch
    ; Startroutine
  ; PCINT fuer Tasteneingang
  ldi rmp,1<<PCINT3 ; PB3-Int ermoeglichen
  out PCMSK,rmp ; in PCINT-Maskenregister
  ldi rmp,1<<PCIE ; PCINT ermoeglichen
  out GIMSK,rmp ; in Interrupt-Maskenregister
  ; Schlafen ermoeglichen
  ldi rmp,1<<SE ; Schlafen, Idle-Modus
  out MCUCR,rmp ; in MCU-Kontrollregister
  ; Interrupts einschalten
  sei
; ---- Hauptprogramm-Schleife -----------
Schleife:
  sleep ; schlafen legen
  nop ; Aufwachen
  sbrc rFlag,bStart ; Ueberspringe wenn Start-
    ; Flagge Null
  rcall Musikstart ; Musikausgabe starten
  sbrc rFlag,bNote ; Note ausgeben
  rcall NoteAus ; Note ausgeben
  rjmp Schleife ; wieder schlafen legen
;
; ----- Behandlungsroutinen -------------
;
; Musikstart: wird von der gesetzten Flagge
; bStart ausgeloest. Startet die Melodieausgabe
; mit der ersten Note.
;
Musikstart: ; Ausgabe Musikstueck starten
  cbr rFlag,1<<bStart ; Flagge loeschen
  set ; setze T-Flagge
  ldi ZH,HIGH(2*Melodie) ; Zeiger auf Musikstueck
  ldi ZL,LOW(2*Melodie)
  rcall NoteSpielen ; gib die Note aus
  ldi rmp,1<<OCIE0A ; Interrupts einschalten
  out TIMSK0,rmp ; in Interrupt Maske
  ret
;
; NoteAus: wird von der gesetzten Flagge bNote
; ausgeloest. Gibt die naechste Note aus
;
NoteAus: ; naechste Note ausgeben
  cbr rFlag,1<<bNote ; Flagge loeschen
  rcall NoteSpielen ; naechste Note ausgeben
  ret
;
; Notespielen:
; Wird von den Flaggen bStart und bNote aufgeru-
; fen und spielt die Note, auf die das Doppel-
; register Z zeigt.
; Ist das Ende der Melodie erreicht (Note =
; 0xFF), wird das T-Flag geloescht, der TC0-
; Interrupt aus- und der Lautsprecherausgang
; stumm geschaltet.
; Sind noch Noten und Pausen zu spielen, wird
; a) die zu spielende Notennummer und
; b) die Dauer in Achtelnotendauern gelesen.
; Ist die Notennummer eine Pause (Note = 0xFE),
; wird die Tonausgabe stumm geschaltet.Sonst
; wird die Lautsprecherausgabe eingeschaltet und
; fuer die Note die Noteneinstellungen aus der
; Notentabelle gelesen und an den Timer ausge-
; geben. Die Dauer der Note bzw. der Pause
; werden in die Anzahl CTC-Zyklen umgerechnet
; und in das Doppelregister R25:R24 geschrieben.
;
NoteSpielen: ; Spiele die Note auf die Z zeigt
  lpm rmp,Z+ ; lese Note
  cpi rmp,0xFF ; stelle Tabellenende fest
  brne NoteSpielen1 ; Nicht Ende
  ; Musikstueck ist zu Ende
  ldi rmp,(1<<COM0A1)|(1<<WGM01) ; Clear, CTC-A
  out TCCR0A,rmp ; in Kontrollregister A
  clr rmp ; Null
  out TIMSK0,rmp ; TC0-Int abschalten
  pop rmp ; entferne Ruecksprungadresse vom Stapel
  pop rmp
  clt ; T-Flagge loeschen
  ret
NoteSpielen1: ; Nicht zu Ende
  cpi rmp,0xFE ; Pause?
  brne NoteSpielen2
  ; Pause, stumm LSP schalten
  ldi rmp,(1<<COM0A1)|(1<<WGM01) ; Clear, CTC-A
  out TCCR0A,rmp ; in Kontrollregister A
  ldi rmp,(1<<CS01)|(1<<CS00) ; Vorteiler = 64
  out TCCR0B,rmp ; in Kontrollregister B
  ldi rmp,255 ; CTC auf Hoechstwert
  ldi rCtrL,18 ; Zaehler auf Achteldauer
  ldi rCtrH,0
```

```
  lpm R16,Z+ ; Dauerbyte ueberlesen
  ret
NoteSpielen2: ; Normale Note
  mov XH,ZH ; Musikzeiger sichern
  mov XL,ZL
  ldi ZH,HIGH(2*Notentabelle_Dauer) ; Zeiger auf
    ; Tontabelle
  ldi ZL,LOW(2*Notentabelle_Dauer)
  lsl rmp ; mal zwei
  lsl rmp ; mal vier
  add ZL,rmp ; zum Zeiger addieren
  ldi rmp,0
  adc ZH,rmp
  lpm rmp,Z+ ; lese Vorteiler
  out TCCR0B,rmp ; in Timer
  lpm rmp,Z+ ; lese CTC-Wert
  out OCR0A,rmp ; in Vergleicher A
  lpm YL,Z+ ; lese Achtel-Dauer nach Y
  lpm YH,Z+
  mov ZH,XH ; Zeiger auf Musik wieder herstellen
  mov ZL,XL
  lpm rmp,Z+ ; Lese Laengen-Byte
  mov XH,YH ; Einfache Achtel-Laenge
  mov XL,YL
NoteSpielen3:
  dec rmp ; Laengen-Byte abwaerts
  breq NoteSpielen4 ; fertig
  add XL,YL ; Achteldauer addieren
  adc XH,YH
  rjmp NoteSpielen3
NoteSpielen4:
  mov rCtrL,XL ; in Zaehler
  mov rCtrH,XH
  ldi rmp,(1<<COM0A0)|(1<<WGM01) ; Toggle, CTC-A
  out TCCR0A,rmp ; in Kontrollregister A
  ret
;
; Notentabelle mit Dauer
Notentabelle_Dauer:
.DB 1<<CS01, 169, 111, 0 ; a  #0
.DB 1<<CS01, 151, 124, 0 ; h  #1
.DB 1<<CS01, 142, 132, 0 ; c  #2
.DB 1<<CS01, 127, 148, 0 ; d  #3
.DB 1<<CS01, 113, 166, 0 ; e  #4
.DB 1<<CS01, 106, 177, 0 ; f  #5
.DB 1<<CS01,  95, 197, 0 ; g  #6
.DB 1<<CS01,  84, 223, 0 ; a' #7
.DB 1<<CS01,  75, 250, 0 ; h' #8
.DB 1<<CS01,  71,   8, 1 ; c' #9
.DB 1<<CS01,  63,  42, 1 ; d' #10
.DB 1<<CS01,  56,  79, 1 ; e' #11
.DB 1<<CS01,  53,  98, 1 ; f' #12
.DB 1<<CS01,  47, 143, 1 ; g' #13
.DB 1<<CS01,  42, 190, 1 ; A  #14
.DB 1<<CS01,  37, 251, 1 ; H   #15
.DB 1<<CS01,  35,  24, 2 ; C   #16
.DB 1<<CS01,  31,  93, 2 ; D   #17
.DB 1<<CS00, 227, 149, 2 ; E   #18
.DB 1<<CS00, 214, 189, 2 ; F   #19
.DB 1<<CS00, 190,  21, 3 ; G   #20
.DB 1<<CS00, 169, 120, 3 ; A'  #21
.DB 1<<CS00, 151, 225, 3 ; H'  #22
.DB 1<<CS00, 142,  32, 4 ; C'  #23
.DB 1<<CS00, 127, 157, 4 ; D'  #24
.DB 1<<CS00, 113,  47, 5 ; E'  #25
.DB 1<<CS00, 106, 135, 5 ; F'  #26
.DB 1<<CS00,  95,  43, 6 ; G'  #27
.DB 1<<CS00,  84, 250, 6 ; A'' #28
;
; ---- Notensymbole ----------
; Notensymbole
.equ na0 = 0
.equ nh0 = 1
.equ nc0 = 2
.equ nd0 = 3
.equ ne0 = 4
.equ nf0 = 5
.equ ng0 = 6
.equ na1 = 7
.equ nh1 = 8
.equ nc1 = 9
.equ nd1 = 10
.equ ne1 = 11
.equ nf1 = 12
.equ ng1 = 13
.equ nA2 = 14
.equ nH2 = 15
.equ nC2 = 16
.equ nD2 = 17
.equ nE2 = 18
.equ nF2 = 19
.equ nG2 = 20
.equ nA3 = 21
.equ nH3 = 22
.equ nC3 = 23
.equ nD3 = 24
.equ nE3 = 25
.equ nF3 = 26
.equ nG3 = 27
.equ nA4 = 28
;
; ---- Melodie -----------------
Melodie: ; LSB: Note oder FF, MSB: Dauer in
;  Achtel
;   Völ-  ker          hört         die
.db ne1,3,nd1,2,0xFE,1,nc1,4,0xFE,1,ng0,3,0xFE,1
;   Sig-         na-          le!
.db ne0,1,0xFE,1,na1,4,0xFE,1,nf0,2,0xFF,0xFF
;
;
; Ende Quellcode
;
```

Im Quelltext gibt es keine neuen Instruktionen.

9.5.6 Simulation mit avr_sim



Simulation verwendet avr_sim um die Ausführung der ersten beiden Noten der Melodie zu verifizieren.

Die erste Note der Melodie, nämlich Note ne1 mit

```
; ---- Notensymbole ----------
.equ ne1 = 11
```

und

```
Notentabelle_Dauer:
.DB 1<<CS01,  56,  79, 1 ; e' #11
```

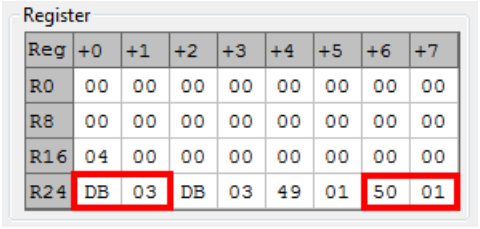
ist in den Timer TC0 geladen. Der Ausgang PB0 mit dem Lautsprecher torkelt. TC0 ist als CTC mit Compare-Match A als TOP-Wert konfiguriert, der auf 56 eingestellt ist. Der Vorteiler steht auf acht und die CTC-Zeit bzw. die Schwingfrequenz beträgt

$$t_{CTC} = 8 * (56 + 1) / 1{,}2 = 380\ \mu s$$
$$f = 1.200.000 / 8 / (56 + 1) / 2 = 1.315{,}8\ Hz$$

Die Dauer der Note in Achteln ist in der Tabelle mit 73 und 1 definiert. Das bedeutet 1 * 256 + 73 = 329 CTC-Zyklen. Die "3", die in der Melodietabelle auf die Note ne1 folgt, sagt: Dauer = 3 Achtel, also werden die 329 mit drei malgenommen. Das ergibt 987 CTC-Zyklen Dauer oder eine Dauer von

$$t_{3/8} = 987 * 380\ \mu s = 375\ ms.$$

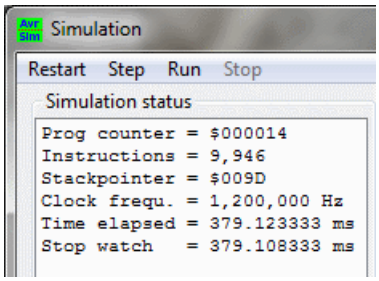
Das entspricht auffällig genau 3/8 Sekunden.


Register

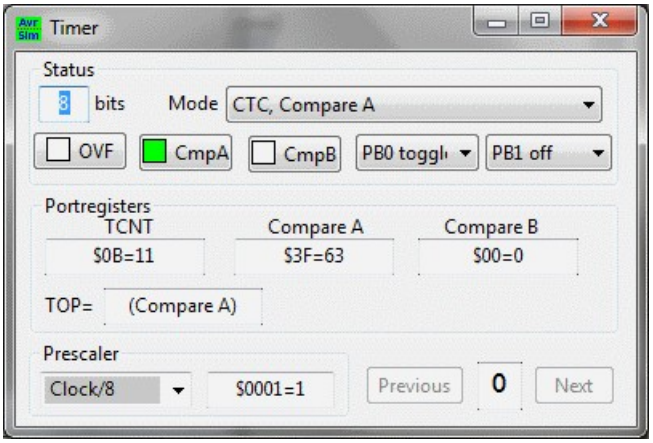
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	04	00	00	00	00	00	00	00
R24	DB	03	DB	03	49	01	50	01

Die Dauer, 987 oder 0x03DB, steht im Doppelregister R25:R24 und wird von der TC0-Interrupt-Service-Routine bei jedem Compare-Match A abwärts gezählt.

Z zeigt auf die nächste Note der Melodie im Flashspeicher bei Adresse 0x00A8.



Dies ist die Zeit, wenn das Registerpaar R25:R24 Null erreicht und die nächste Note gespielt werden soll. Die 379 ms sind fast genau.



Die zweite Note der Melodie wird jetzt in den TC0-Timer geladen. Der Vorteiler ist wieder 8, der Compare-Match-A-Wert ist nun 63. Das entspricht

$$f_2 = 1.200.000 / 8 / (63 + 1) / 2 = 1.171\ Hz$$

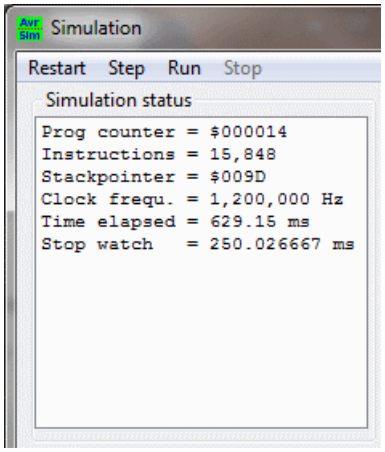
und liegt nahe genug bei den 1.173 Hz von Note nd1. Die Dauer der Note ist jetzt auf zwei Achtel eingestellt:

$$t = 2 * 8 * (63 + 1) * (1 * 256 + 42) / 1.200.000 = 254{,}3 \text{ ms}$$

Der Zählerwert in R25:R24 ist ok, das Doppelregister Z zeigt schon auf die dritte Note (eine Pause).

Register

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	40
R16	42	00	00	00	00	00	00	00
R24	4A	02	4A	02	25	01	52	01



Wie vorherberechnet sind 250 ms am Ende der zweiten Note vergangen.

Die dritte Note ist eine Pause (Code: 0xFE). Der einzige Unterschied zwischen einer Note und einer Pause ist, dass der Ausgang OC0A nicht auf Torkeln sondern auf Clear eingestellt wird, was den Ausgang auf Dauer-Null einstellt.

Simulation ist ein wertvolles Werkzeug, mit dem selbst komplexe Abläufe anschaulich beobachtet und analysiert werden können.

Top	Home	Einführung	Hardware	Tonhöhe	Tonleiter	Melodie

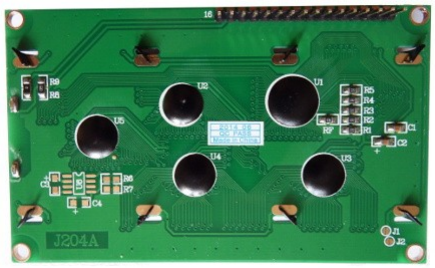
Lektion 10: Lcd-Anzeige am ATtiny24

Um eine LCD an einen AVR anzuschließen, brauchen wir ein paar Pins mehr. Die folgenden Experimente verwenden daher einen ATtiny24. Die LCD wird mit zwei Methoden angesteuert: mit festen Warteschleifen und mit Auswertung des Busy-Flags.

10.0 Übersicht

10.1 Einführung in LCD-Anzeigen

10.1.1 Allgemeines über LCD-Anzeigen



Es gibt unzählige Farben und Formen von LCDs. Grau, grün, gelb, blau, da ist für jeden Geschmack etwas dabei. Es gibt Text- und Graphikpixel-Anzeigen. Dies hier befasst sich mit Textanzeigen. Diese sind am einfachsten anzusteuern und bieten für die meisten Zwecke auch ausreichend Möglichkeiten der Darstellung.

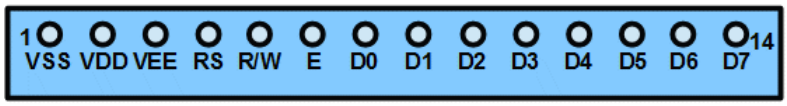
Textanzeigen arbeiten meist mit einer Pixelmatrix von 5x8, was für die Erkennung von Zeichen auch aus einiger Entfernung völlig ausreicht.

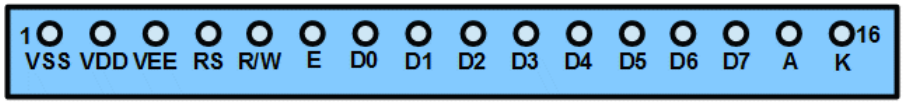
Der Vorteil von LCD-Anzeigen (gegenüber z. B. Siebensegmentanzeigen) ist der niedrige Strombedarf von etwa 1 bis 2 mA. Für Batteriebetrieb ist das ideal. Um den gleichen Kontrast zu erzielen ist bei Siebensegment mindestens das 20-fache nötig. Außerdem ist Textdarstellung mit Siebensegment grausig, und bei 8 oder gar 16 Zeichen ist ein arger Hardwareverhau nötig. Das entfällt bei LCDs weitgehend.

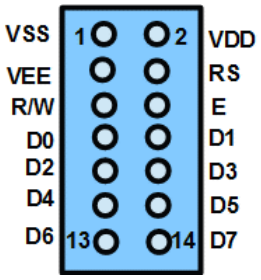
Auch bei der Anzahl Zeilen und Zeichen herrscht große Vielfalt. Einzeilige LCD mit acht Zeichen sind ideal für kleine Messgeräte, da stören mehr Zeichen eher. Zweizeilige LCD mit 8, 16 oder 20 Zeichen pro Zeile können schon eine ganze Menge darstellen. Für mehr gibt es vierzeilige LCD mit 20 oder 24 Zeichen.

10.1.2 Interfaces von LCD-Anzeigen

Die Anschlüsse von LCD sind quasi normiert, d. h. nahezu alle verwenden die gleiche Reihenfolge von Anschlüssen. Es gibt aber Varianten, z. B. 18-pinnige.







Die einzelnen Pins bedeuten folgendes:

Pin	Name	Bedeutung
1	VSS	Betriebsspannung, Minus
2	VDD	Betriebsspannung, Plus, 3 oder 5 Volt
3	VEE	Regelspannung für Kontrast
4	RS	Register Select, 0=Control, 1=Daten
5	R/W	Read/Write, 0=Schreiben in LCD, 1=Lesen von LCD
6	E	Enable, 0=inaktiv, 1=aktiv/Lesen/Schreiben
7	D0	Datenbit 0
8	D1	Datenbit 1
9	D2	Datenbit 2
10	D3	Datenbit 3
11	D4	Datenbit 4
12	D5	Datenbit 5
13	D6	Datenbit 6
14	D7	Datenbit 7
15	A	Hintergrundbeleuchtung Anode
16	K	Hintergrundbeleuchtung Kathode

Die zweireihige Steckervariante eignet sich für Verbindungen mit Flachbandkabel. Abweichende Pinfolgen sind den Datenblättern zu entnehmen.

Die Regelspannung für die Kontrasteinstellung wird meistens mit einem Trimmer eingestellt, der die Betriebsspannung teilt. Er wird auf maximalen Kontrast justiert. Liegen hier 0 Volt oder die Betriebsspannung an, sind die Zeichen nicht zu erkennen.

Bei den meisten LCDs bewirkt das Einschalten der Betriebsspannung und die Kontrastregelung alleine, d. h. ohne Prozessortätigkeit, dass Zeichenpositionen erkennbar werden, bei mehrzeiligen LCDs bleiben einzelne Zeilen aber auch leer (dunkel). Immerhin kann man daran erkennen, dass die Betriebsspannung korrekt angeschlossen ist (wenn nicht: LCD in die Tonne treten) und ob die Kontrastregelung funktioniert.

10.1.2.1 8-Bit-Interface

Im Startzustand, bei Anlegen der Betriebsspannung, sind LCDs auf die Kommunikation im 8-Bit-Modus eingestellt. Bei jedem Schreib- und Lesevorgang werden zwischen Prozessor und LCD acht Datenbits übertragen. Alle Datenbit-Eingänge liegen übrigens per default auf High-Pegel.

10.1.2.2 4-Bit-Interface

Mit einem speziellen Befehl lassen sich LCDs in den 4-Bit-Modus umschalten. In diesem Modus werden nur die obersten vier Datenbits zur Kommunikation verwendet (Pin-Spar- und Verkabe-

lungsvermeidungs-Modus). Dafür müssen alle Datenbytes in zwei Portionen an die LCD gesendet und von der LCD gelesen werden. Nur die Anschlüsse D4 bis D7 werden verwendet. Zuerst werden die oberen vier Bits gesendet/gelesen, dann die unteren vier Bits.

10.1.3 Ansteuerung von LCD-Anzeigen

10.1.3.1 Initiierung

Mit folgendem Ablauf werden LCDs initiiert, d. h. erstmalig in Betrieb genommen. In allen Fällen muss der Anschluss RS auf Null gehalten werden. Die Übertragung erfolgt durch Aktivieren des LCD-Eingangs E für mindestens 1 µs Dauer. "x" bedeutet, dass diese Bits egal sind.

- Nach dem Einschalten der Betriebsspannung ist zunächst zu warten, bis der LCD-Prozessor seinen Aufwärmvorgang abgeschlossen hat. Bevor das erfolgt, gelingt keine Kommunikation mit der LCD. Der Zeitraum wird unterschiedlich lang angegeben, 50 ms sind in jedem Fall dafür ausreichend.
- Als nächstes wird der Function Set durchgeführt. Er lautet 0b001L.NFxx.
  - → Ist ein 8-Bit-Interface angeschlossen, dann ist L=1. N gibt die Anzahl der Zeilen an (0: einzeilig, 1: mehrzeilig), F den Zeichenfont (0: 5x8).
  - → Ist ein 4-Bit-Interface angeschlossen, dann ist zunächst drei mal 0b0011.xxxx an die LCD zu senden, um sie sicher in den 8-Bit-Modus zu schalten. Erst dann erfolgt die Umschaltung auf den 4-Bit-Modus durch Ausgabe von 0b0010.xxxx, noch im vorher eingestellten 8-Bit-Modus. Nachfolgend erfolgt die Einstellung von N und F durch erneute Ausgabe von 0b0010.NFxx, jetzt aber im 4-Bit-Verfahren (Ausgabe von 0b0010 zuerst, dann von 0bNFxx). Die Umschaltung dauert etwas mehr als 1 ms, 5 ms Warten sind ausreichend.
- Die nachfolgenden Befehle benötigen etwa 40 µs für die Ausführung. Ab jetzt ist es aber auch möglich, das Busy-Flag abzufragen. Dazu wird der Datenport auf Eingang geschaltet (Richtungsbits löschen) und der R/W-Eingang auf Eins gesetzt. Mit Aktivieren des LCD-Eingangs E gibt die LCD den Zustand des Busy-Flags (Bit 7) und die aktuelle Adresse der Zeichenausgabe (Bit 6 bis 4) zurück. Im 4-Bit-Modus sind zwei Aktivierungen des E-Eingangs nötig, bei den unteren vier Bits sendet die LCD die Bits 3 bis 0 der aktuellen Adresse. Ist das Busy-Flag Null, ist die interne Verarbeitung des vorausgehenden Befehls abgeschlossen und die nächste Operation kann erfolgen.
  - → Als nächstes erfolgt mit 0b0000.DCBx das Einschalten der Anzeige (mit D=1), das Einschalten des Cursors (mit C=1) und das Blinken des Cursors (mit B=1).
  - → Mit dem Befehl 0b0000.0001 wird die Anzeige gelöscht.
  - → Mit dem Befehl 0b0000.00IS wird Auto-Increment der Displayadresse (mit I=1) und der Schiebemodus des Displays (mit S=1) eingestellt.
  - → Mit dem Befehl 0b0000.001x wird die Ausgabeadresse auf das erste Zeichen der ersten Zeile gesetzt und Verschiebungen (bei S=1) aufgehoben. Dieser Befehl benötigt etwas mehr als 1 ms, 5 ms sind ausreichend.

Damit ist die Initiierung abgeschlossen und es kann an das Ausgeben von Zeichen beginnen.

10.1.3.2 Datenausgabe

Um Daten zur LCD zu senden, muss der RS-Eingang der LCD auf Eins, der R/W- Eingang auf Null gesetzt werden. Die nachfolgend an die LCD ausgegebenen ASCII-Zeichen erscheinen bei Auto-Indent (I=1) nacheinander auf dem Display. Um die Adresse des Displays zu ändern, bei der das nächste Zeichen erscheinen soll, sind mit RS=0 und R/W=0 folgende Zahlen auszugeben (bei einer mehrzeiligen LCD mit N Zeichen pro Zeile):

- Zeile 1: 0x80 + (Spalte - 1),
- Zeile 2: 0xC0 + (Spalte - 1),
- Zeile 3: 0x80 + N + (Spalte - 1),
- Zeile 4: 0xC0 + N + (Spalte -1).

10.1.4 Beleuchtung von LCD-Anzeigen

LCD-Anzeigen sind (besonders im Dunklen) besser ablesbar (und auch schöner), wenn sie eine Hintergrundbeleuchtung haben. Um diese anzusteuern, wird der K-Eingang auf Minus gebracht und über den A-Eingang ein Strom auf die Hintergrund-LEDs gegeben. Die Höhe des Stroms ist im Datenblatt spezifiziert, für die hier verwendete vierzeilige LCD sind 70 mA angegeben.

Die folgenden Bilder zeigen, dass oberhalb von 5 mA Strom kein sichtbarer Effekt eintritt.

I = 2,1 mA, U = 2,9 V

I = 7,4 mA, U = 3,4 V

I = 12,0 mA, U = 3,8 V

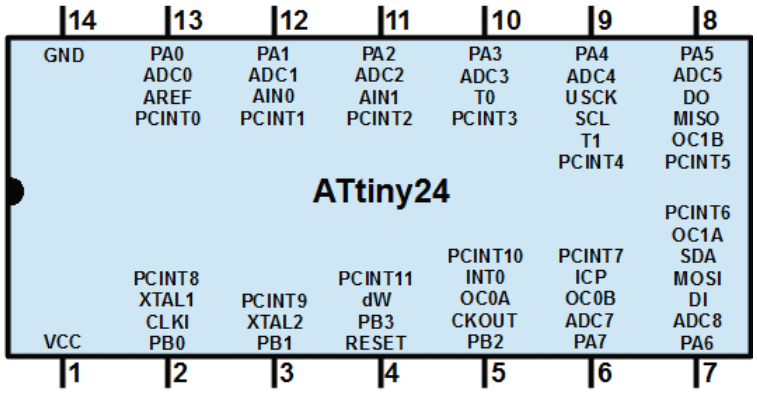
I = 18,3 mA, U = 4,3 V

Der Effekt für die Batteriehaltbarkeit oder die Akkulaufzeit dürfte bei dem spezifizierten Strom von 70 mA relevanter sein als der erzielbare optische Effekt.

Top	Home	Einführung	ATtiny 24	Hardware	LCD Zeit	LCD busy	LCD Zeichen

10.2 Einführung in den ATtiny24

Der ATtiny24 ist ein 14-poliger Prozessortyp. Wie die Vielfachbelegung seiner Pins zeigt, ist er nicht nur mit mehr Pins sondern auch mit viel mehr interner Hardware ausgestattet. Der Port A ist mit allen acht Bits zugänglich, zusätzlich sind drei Bits des Ports B vorhanden. Wenn man auf die ISP-Programmierung verzichtet, kann noch ein weiterer Bit des Ports verwendet werden, der defaultmäßig den Reset-Eingang bildet.



Acht Anschlüsse lassen sich als AD-

Wandler-Eingänge ADC0 bis ADC7 verwenden.

Hinter OC0A und OC0B steckt ein 8-Bit-Timer, hinter OC1A und OC1B ein 16-Bit-Timer. Alle vier Pins können als Takt- oder PWM-Ausgänge verwendet werden.

An die Pins 2 und 3 kann ein externer Quarz angeschlossen werden und mit den Fuses eingeschaltet werden.

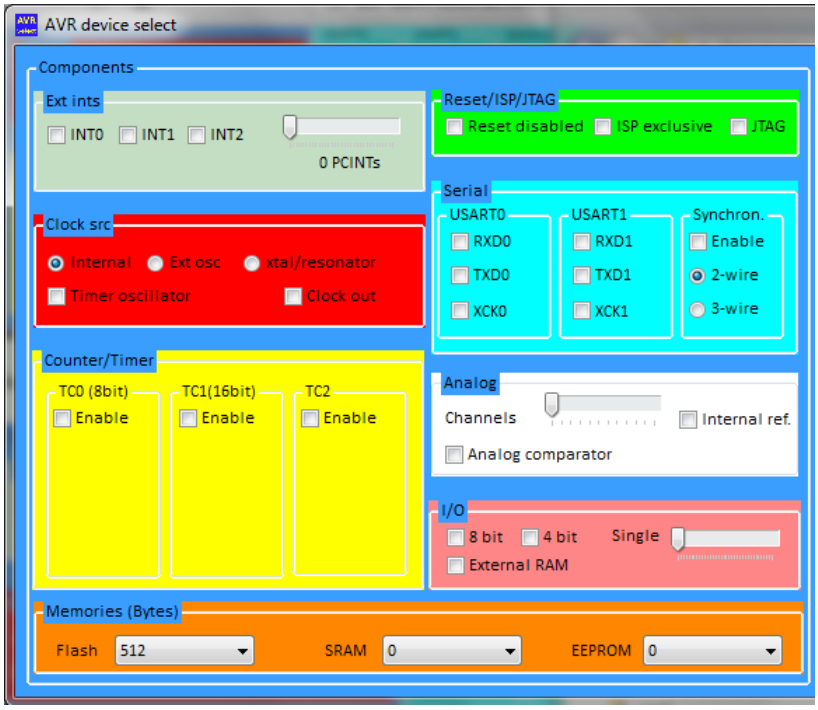
Alle Pins können mit zwei PCINT (0 .. 7, 8 .. 10/11) auf Pegelwechsel überwacht werden.

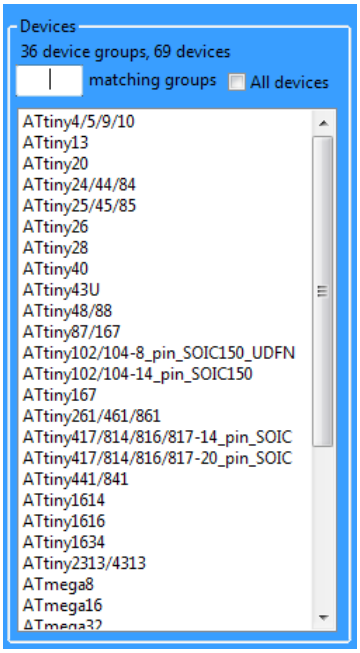
Natürlich kann mit USCK, MISO, MOSI und RESET auch eine ISP-Programmierung erfolgen.

Der ATtiny24 hat einen internen 8 MHz-Oszillator. Mit der defaultmäßig gesetzten CDIV8-Fuse taktet dieser mit 1 MHz, kann aber, wie schon beim ATtiny13 gezeigt wurde, mit der Fuse auf höhere und mit CLKPR auf niedrigere Taktraten umgestellt werden.

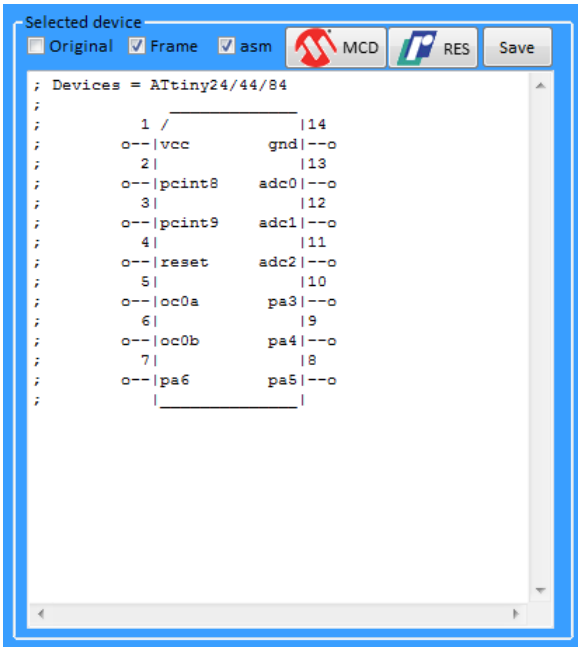
Wem also wie uns die Pins beim ATtiny13 zu wenige sind, kann zu diesem Typ greifen. Die Mehrfachbelegung der Pins macht deutlich, worum es bei der optimalen Auswahl des AVR-Typs geht: man muss ein klares Bild von dem haben, was man in seiner Schaltung alles an prozessorinterner Hardware benötigt, welche Pins man für welche Zwecke benötigt und bei welchen Pins man optimalerweise eine bestimmte Reihenfolge benötigt, z. B. für unsere vier Bits Datenbus zur LCD. Natürlich könnte man dazu beliebige Pins in beliebigen Ports verwenden, nur muss man dann jedes Bit eines ASCII-Zeichens an den richtigen Pin bringen.

Das Softwaretool avr_select kann bei der Auswahl helfen, da es die interne Hardware aller Tiny- und Megatypen kennt. Es ist als Lazarus-Quellcode sowie als fertig kompilierte Linux- und Windows-64-Bit-Version kostenlos erhältlich. Mit ihm lassen sich bequem Hardwareanforderungen zusammenklicken.





Diejenigen AVR-Typen, die diese Hardwareanforderungen erfüllen, werden in einer Typenliste angezeigt und können dort auch ausgewählt werden.



Wird ein Typ ausgewählt, erscheint seine Pinbelegung, schon mal vorbelegt mit den intern belegten Hardwarezuordnungen. Die advanced version zeigt direkte Links zu dem Typ bei Microchip Direct (z. B. mit den Handbüchern dieses Typs) und bei Reichelt an.

Die Pinbelegung kann editiert und als Textdatei auf die Festplatte geschrieben werden.

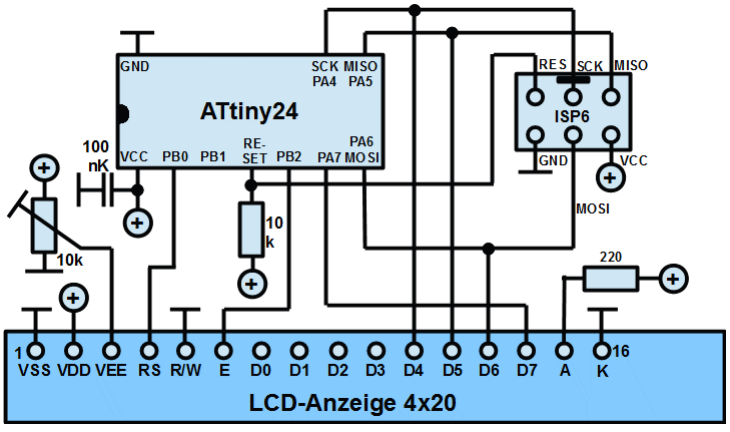
Hat man weder über die Hardware noch über solche Vorbedingungen einen Überblick, neigt man eher zur Überdimensionierung. Hat man sich nämlich einmal für einen zu kleinen Typ entschieden und braucht dann noch zusätzlich ein oder zwei Portpins mehr oder benötigt unbedingt einen OC0A- oder OC1B-Ausgang, muss man nicht nur einige Zeilen im Programm ändern. Die Interrupt-Vektor-Tabelle anzupassen ist noch der geringere Aufwand, manchmal ist dazu aber auch ein völliges Neudesign der Schaltung nötig. Auf so was lässt man sich ein, wenn man halt keinen Elefanten (Arduino) in den eigenen kleinen Porzellanladen reinlassen will.

Das Thema Mehrfachnutzungen von Pins ist im Design auch ein Thema. Da wir ISP zum Programmieren benutzen wollen, sind die Pins USCK, MOSI und MISO sowieso schon gesetzt. Da sich deren Nutzung durchaus mit der Nutzung als Datenbus zur LCD verträgt (wenn der LCD-Eingang R/W nicht dauerhaft auf Lesen steht), stören sich diese beiden Mehrfachverwendungen nicht. Die Verwendung des USCK-Pins z. B. für die Messung von Analogspannungen oder für die Ansteuerung eines Motors würde hingegen zahlreiche Fragen aufwerfen.

10.3 Hardware, Bauteile und Aufbau

10.3.1 Schaltbild

Das Schaltbild zeigt, wie die LCD vom ATtiny24 angesteuert wird:

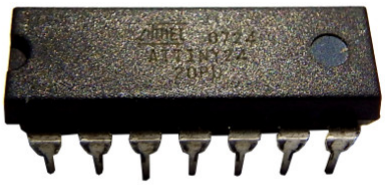


- Die Pins PB0 (LCD-RS) und PB2 (LCD-E) steuern die Kontrollpins der LCD, der Datenbus wird vom oberen Nibble (Nibble = 4-Bit) des Datenports PA bedient. Der R/W-Eingang liegt dauerhaft auf Minus, da wir bei der ersten Aufgabe den Read-Modus der LCD nicht verwenden.
- Die ISP-Schnittstelle ist an die entsprechenden Ports angeschlossen, die drei Signale werden parallel auch vom LCD-Datenbus verwendet.
- Die Hintergrundbeleuchtung der LCD wird über einen 220 Ω-Widerstand mit einem sparsamen Strom angetrieben.

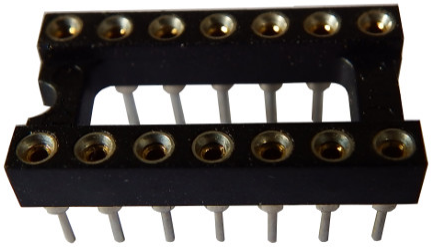
10.3.2 Bauteil Lcd-Anzeige

Die LCD wird mit einer 16-poligen Stiftleiste ausgestattet, damit sie in die Löcher unseres Breadboards passt. Wer eine andere LCD verwendet, muss sich hier was einfallen lassen.

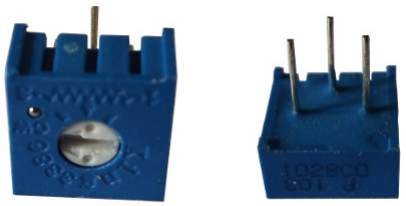
10.3.3 Bauteile ATtiny24 und 14-polige Fassung



Das hier ist der 14-polige ATtiny24, der in nebenstehende Fassung passt.

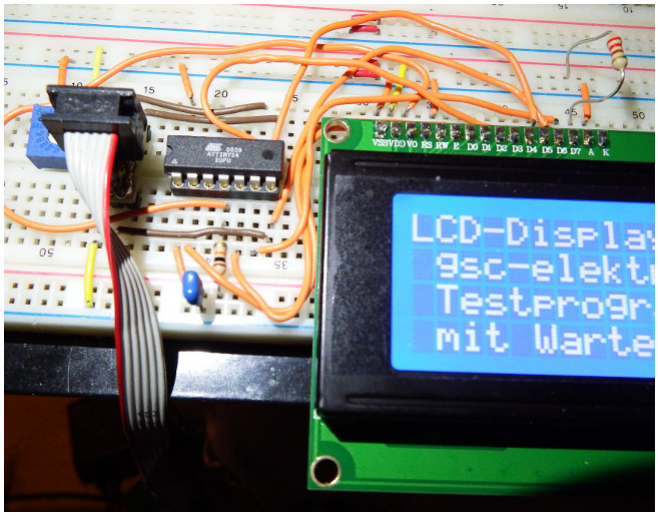


10.3.3 Bauteil Trimmer



Der Trimmer dient der Kontrasteinstellung der LCD. Die beiden Anschlüsse im Vordergrund des rechten Bilds sind die Endpunkte der Widerstandsbahn, der mittlere Pin im Hintergrund ist der Schleifer.

10.3.4 Aufbau



So wird der Tiny24 und die LCD-Anzeige aufgebaut.

Noch ein Hinweis zu solchen Breadboards: die Kontakte solcher Boards leiern mit der Zeit aus, wenn man z. B. dicke Stiftleisten über längere Zeit in den gleichen Löchern stecken lässt. Bei Einfachstboards ohne metallene Unterlage kann sich die Plastikunterlage herausdrücken, die Kontaktsicherheit ist dann gleich Null. Das kann üble Wackelkontakte zur Folge haben. Besonders bei gedrehten IC-Fassungen sind die Pins sehr dünn und kurz. Das kann den Tod von Prozessoren oder einzelner Portpins verursachen, wenn die Versorgungsleitung einen Wackler hat. Bei Verdacht mit einem Durchgangsprüfer oder einem Ohmmeter den tatsächlichen Kontakt lieber noch mal prüfen.

10.3.5 Alternativer Aufbau

Wer dieses oder die nächsten Experimente lieber auf kompaktere Weise aufbauen möchte und die vielen Zuleitungen zur LCD fest verdrahtet sehen möchte, kann sich das Board hier als gedruckte Platine bauen. Alle Programmbeispiele dieser und der nachfolgenden Lektionen laufen darauf ohne Änderungen.

10.4 Ansteuerung der LCD mit Warteschleifen

10.4.1 Zeitbedarf und Konstruktion von Warteschleifen

Die folgenden Zeitbedarfe sind bei der Ansteuerung der LCD zu beachten:

- Initiierung: 50 ms
- Rücksetzen der Anzeige: 5 ms
- Ausgabe von Zeichen, einfache Kontrollbefehle: 40 µs

Sinnvollerweise läuft die Warteschleife parallel zur Tätigkeit der LCD, also nach Absetzen des jeweiligen Befehls.

Für alle genannten Zeiten ist eine 16-Bit-Warteschleife geeignet, wie sie in der Lektion 3 vorgestellt wurde. Die Formulierung

```
  ldi ZH,HIGH(n)
  ldi ZL(n)
Schleife:
  sbiw ZL,1
  brne Schleife
```

verzögert um t = 2 + 4* (n-1) + 3 = 4 * n + 1 Takte, bei 1 Mhz Takt 4 * n + 1 µs. Zusammen mit dem Aufruf der Verzögerungsroutine, einer "RCALL"-Instruktion (3 Takte), einer "RET"-Instruktion (4 Takte) und einem Sprung zur Zählroutine "RJMP" mit 2 Takten ergeben sich t = 4 * n + 10 Takte. Soll das Registerpaar ZH:ZL vor dem Zählen noch gesichert und danach wieder hergestellt werden (2 * "PUSH" und "POP" zu je 2 Takten ergeben sich t = 4 * n + 18 Takte.

```
  ; rcall warte50ms ; + 3 Takte
warte50ms:
  push ZH ; + 2 Takte
  push ZL ; + 2 Takte
  ldi ZH,HIGH(n) ; + 1 Takt
  ldi ZL,LOW(n) ; + 1 Takt
  rjmp warte ; + 2 Takte
warte:
  sbiw ZL,1 ; + 2 Takte
  brne warte ; +1/2 Takte
  pop ZL ; + 2 Takte
  pop ZH ; + 2 Takte
  ret ; + 4 Takte
```

Kehrt man die Formel um, um n zu ermitteln, lautet die Formel (mit t in µs bei 1 MHz Takt):

$$n = (t - 18) / 4$$

mit Aufrunden: $n = (t - 16) / 4$

Für 50 ms ergeben sich

$$n = (50.000 - 16) / 4 = 12.496$$

Auf ähnliche Weise lassen sich die Zählkonstanten für andere Zeiten oder Taktraten errechnen.

10.4.2 Aufgabenstellung

Auf der LCD sollen auf den vier Zeilen (alternativ: 1, 2 oder 3 Zeilen) mit 20 (alternativ: 8, 16, 20, 24) Zeichen einer Meldung ausgegeben werden. Die Ausführungszeiten sollen mit Warteschleifen überbrückt werden.

10.4.3 Programm

Das hier ist das Programm (hier geht es zum Quellcode im asm-Format). Es ist linear angelegt, weil zum Initiieren und zur Textausgabe an die LCD keine Interrupts benötigt werden.

Bitte beachten: Wird hier das tn24_lcd-Board verwendet oder ist der R/W-Eingang der LCD nicht auf GND sondern an PB1 angeschlossen ist.

```
;
; ************************************
; * LCD-Display 4*20 am ATtiny24, 4-Bit *
; * (C)2016 by www.gsc-elektronic.net   *
; ************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----------- Programmablauf ------------
;
; Initiiert die an den ATtiny24 angeschlossene
; 4*20-LCD mit Verzoegerungswarteschleifen
; und gibt dann einen Text auf der LCD aus.
;
; ----------- Ports, Portpins -----------
; LCD-Kontrollport
.equ pLcdCO   = PORTB  ; LCD-Kontrollport-Ausgabe
.equ pLcdCR   = DDRB   ; LCD-Kontrollportrichtung
.equ bLcdCOE  = PORTB2 ; LCD Enable Pin Output
.equ bLcdCRE  = DDB2   ; LCD Enable Pin Richtung
.equ bLcdCORS = PORTB0 ; LCD RS Pin Output
.equ bLcdCRRS = DDB0   ; LCD RS Pin Richtung
.equ bLcdCORW = PORTB1 ; LCD RW Pin Output
.equ bLcdCRRW = DDB1   ; LCD RW Pin Richtung
; LCD-Datenport
.equ pLcdDO   = PORTA  ; LCD-Datenport-Ausgabe
.equ pLcdDI   = PINA   ; LCD-Datenport-Eingabe
.equ pLcdDR   = DDRA   ; LCD-Datenport-Richtung
.equ mLcdDR   = 0xF0   ; LCD-Datenport-
    ; richtungsmaske
;
; ------- Register ----------------------
; frei: R0 .. R15
.def rmp = R16 ; Vielzweckregister
.def rmo = R17 ; weiteres Vielzweckregister
.def rLine = R18 ; Zeilenzaehler LCD
; frei R19 .. R29
; verwendet: R31:R30, ZH:ZL, fuer Zaehlen
;
; ------- Konstanten --------------------
.equ Takt = 1000000 ; Default-Taktfrequenz
;
; -------- Programmstart, Init ----------
.CSEG ; Code Segment
.ORG 0 ; Start bei 0
  ; Stapel Init fuer Unterprogramme
  ldi rmp,LOW(RAMEND) ; Init Stapel
  out SPL,rmp ; in Stapelzeiger
  ; Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
; Wenn der R/W-Eingang der LCD nicht auf GND
; liegt sondern an PB1 angeschlossen ist (wie
; dies bei den folgenden Experimenten und beim
; tn24_lcd-Board der Fall ist), muss die folgende
; Instruktion hinzugefuegt werden:
; sbi DDRB,DDB1 ; Portbit auf Ausgabe
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDR ; Datenport-Ausgabemaske
  out pLcdDR,rmp ; auf Ausgabe
  ; LCD initiieren
  rcall LcdInit
  ; Text auf LCD ausgeben
  ldi ZH,HIGH(2*Texttabelle)
  ldi ZL,LOW(2*Texttabelle)
  rcall LcdText ; Text ab ZH:ZL ausgeben
  ; Sleep enable
  ldi rmp,1<<SE
  out MCUCR,rmp
Schleife:
  sleep ; schlafen
  rjmp Schleife
;
; Ausgabetext auf LCD
Texttabelle:
.db "LCD-Display ATtiny24",0x0D,0xFF ; Zeile 1
.db " gsc-elektronic.net ",0x0D,0xFF ; Zeile 2
.db " Testprogramm 4-Bit ",0x0D,0xFF ; Zeile 3
.db " mit Warteschleifen",0x00 ; Zeile 4
;
; --------- LCD-Ansteuerung Init ----------
LcdInit:
  ; Warte 50 ms bis LCD hochgefahren ist
  rcall Warte50ms ; Karenzzeit 50 ms
  ; Versetze in 8-Bit-Modus (drei Mal)
  ldi rmp,0x30 ; 8-Bit-Modus
  rcall LcdC8Byte ; Schreibe im 8-Bit-Mode
  rcall Warte5ms ; Warte 5 ms
  ldi rmp,0x30
  rcall LcdC8Byte
  rcall Warte5ms
  ldi rmp,0x30
  rcall LcdC8Byte
  rcall Warte5ms
  ; Umstellung auf 4-Bit-Modus
  ldi rmp,0x20 ; Schalte in 4-Bit-Modus um
  rcall LcdC8Byte
  rcall Warte5ms
  ; Funktionseinstellungen LCD
  ldi rmp,0x28 ; 4-Bit-Modus, 4 Zeilen, 5*7
  rcall LcdC4Byte
  rcall Warte5ms
  ldi rmp,0x0F ; Display ein, Blinken
  rcall LcdC4Byte
  rcall Warte5ms
  ldi rmp,0x01 ; Display loeschen
  rcall LcdC4Byte
  rcall Warte5m-*s
  ldi rmp,0x06 ; Autoindent
  rcall LcdC4Byte
  rjmp Warte40us
;
; Ausgabe von Text auf der LCD
;   Z zeigt auf Text im Flash
LcdText:
  clr rLine ; Zeilenzaehler
LcdText1:
  lpm rmp,Z+ ; lese Zeichen aus Flash
  cpi rmp,0 ; Ende Ausgabe?
  breq LcdTextRet ; nein
  cpi rmp,0xFF ; Fuellzeichen?
  breq LcdText1 ; ja, naechstes Zeichen
  cpi rmp,0x0D ; Zeilenwechsel?
  brne LcdText2 ; Kein Zeilenwechsel
  inc rLine ; Naechste Zeile
  mov rmp,rLine ; Setze Zeile
  rcall LcdLineSet ; Stelle Zeile ein
```

```
  rjmp LcdText1 ; weiter mit Zeichen
LcdText2: ; Ausgabe Zeichen
  rcall LcdD4Byte ; Zeichen ausgeben
  rcall Warte40us ; Warten
  rjmp LcdText1 ; und weiter
LcdTextRet:
  ret ; Fertig
;
; Setzt den Ausgabecursor auf den
; Zeilenanfang der ausgewaehlten Zeile
;   Zeile: rmp 0 .. 3
LcdLineSet:
  cpi rmp,1 ; Zeile 2?
  brcs LcdLineSet1 ; nach Zeile 1
  breq LcdLineSet2 ; nach Zeile 2
  cpi rmp,2 ; Zeile 3?
  breq LcdLineSet3 ; nach Zeile 3
  rjmp LcdLineSet4 ; nach Zeile 4
LcdLineSet1:
  ldi rmp,0x80 ; Zeile 1
  rjmp LcdLineSetRmp
LcdLineSet2:
  ldi rmp,0xC0 ; Zeile 2
  rjmp LcdLineSetRmp
LcdLineSet3:
  ldi rmp,0x80+20 ; Zeile 3
  rjmp LcdLineSetRmp
LcdLineSet4:
  ldi rmp,0xC0+20 ; Zeile 4
  rjmp LcdLineSetRmp
LcdLineSetRmp:
  rcall LcdC4Byte ; Ausgabe Kontrollwort
  rjmp Warte40us
;
; Datenwort-Ausgabe im 4-Bit-Modus
;   Daten in rmp
LcdD4Byte:
  sbi pLcdCO,bLcdCORS ; setze RS-Bit
  rjmp Lcd4Byte ; gib Byte in rmp aus
;
; Kontrollwort-Ausgabe im 4-Bit-Modus
;   Daten in rmp
LcdC4Byte:
  cbi pLcdCO,bLcdCORS ; loesche RS-Bit
; Ausgabe Byte im 4-Bit-Modus
Lcd4Byte:
  push rmp ; rmp auf Stapel legen
  andi rmp,0xF0 ; unteres Nibble loeschen
  in rmo,pLcdDI ; Lese Data-Input-Port
  andi rmo,0x0F ; oberes Nibble loeschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  pop rmp ; rmp wieder herstellen
  andi rmp,0x0F ; oberes Nibble loeschen
  swap rmp ; Nibble vertauschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  ret ; fertig
;
; Kontrollwort-Ausgabe im 8-Bit-Modus
;   Datenbyte in rmp
LcdC8Byte:
  cbi pLcdCO,bLcdCORS ; RS-Bit loeschen
  andi rmp,0xF0 ; unteres Nibble loeschen
  in rmo,pLcdDI ; Lese Data-Input-Port
  andi rmo,0x0F ; oberes Nibble loeschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  ret ; fertig
;
; ------ Warteroutinen ------------------
Warte50ms: ; Warteroutine 50 ms
.equ c50ms = 50000
.equ n50ms = (c50ms-16)/4
;   rcall: + 3
  push ZH ; + 2
  push ZL ; + 2
  ldi ZH,HIGH(n50ms) ; + 1
  ldi ZL,LOW(n50ms) ; + 1
  rjmp Warte ; + 2, gesamt = 11
Warte5ms: ; Warteroutine 5 ms
.equ c5ms = 5000
.equ n5ms = (c5ms-16)/4
  push ZH
  push ZL
  ldi ZH,HIGH(n5ms)
  ldi ZL,LOW(n5ms)
  rjmp Warte
Warte100us: ; Warteroutine 100us
.equ c100us = 100
.equ n100us = (c100us-16)/4
  push ZH
  push ZL
  ldi ZH,HIGH(n100us)
  ldi ZL,LOW(n100us)
  rjmp Warte
Warte40us: ; Warteroutine 40us
.equ c40us = 40
.equ n40us = (c40us-16)/4
  push ZH
  push ZL
  ldi ZH,HIGH(n40us)
  ldi ZL,LOW(n40us)
  rjmp Warte
; Warteroutine Z Takte
Warte: ; Warteschleife, Takte=4*(n-1)+11=4*n+7
  sbiw ZL,1 ; + 2
  brne Warte ; + 1 / 2
  pop ZL ; + 2
  pop ZH ; +2
  ret ; + 4, Gesamt=4*n+18
;
```

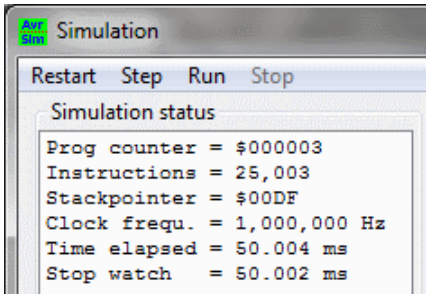
In dem Programm ist eine neue Instruktion verwendet, "SWAP Register". Sie vertauscht die unteren vier Bits 0 bis 3 in einem Register mit den oberen vier Bits 4 bis 7. Wir brauchen diese Instruktion, weil wir ja die oberen vier Bits zuerst auf den LCD-Datenbus bringen müssen (der im oberen Port-A-Bereich liegt) und danach die unteren vier Bits auf den oberen Port-A-Bereich legen müssen. Um die unteren vier Bits zu den oberen zu machen, könnten wir auch vier mal Linksschieben (mit "LSL Register"), aber ANDI und SWAP sind schneller.

10.4.4 Simulieren der Warteroutinen

Hier macht es Sinn, die Warteroutinen per Simulation zu verifizieren. Also starten wir avr_sim, füttern den Quellcode in den Simulator und fügen direkt hinter der Initiierung des Stapelzeigers die folgenden Instruktionen ein:

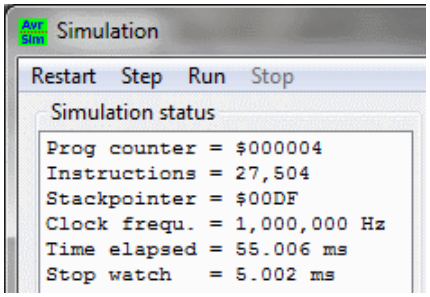
```
rcall Wait50ms
rcall Wait5ms
rcall Wait100us
rcall Wait40us
nop
```

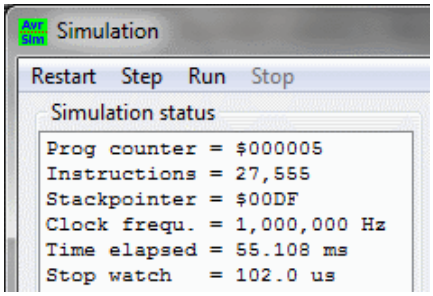
Nach dem Assemblieren setzen wir Breakpunkte auf alle diese Instruktionen und löschen die Stoppuhr nach jedem ausgeführten *rcall*.



Die 50 ms-Unterroutine ist recht exakt, nur zwei Taktzyklen länger als berechnet.

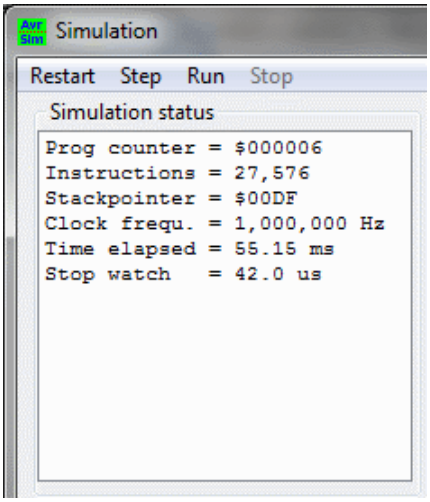
Dasselbe bei der Routine für 5 ms Verzögerung.





Und wiederum auch für 100 µs, ebenfalls mit zwei Rundungstakten.

Und, es wird schon fast langweilig, dasselbe für die 40 µs-Schleife.

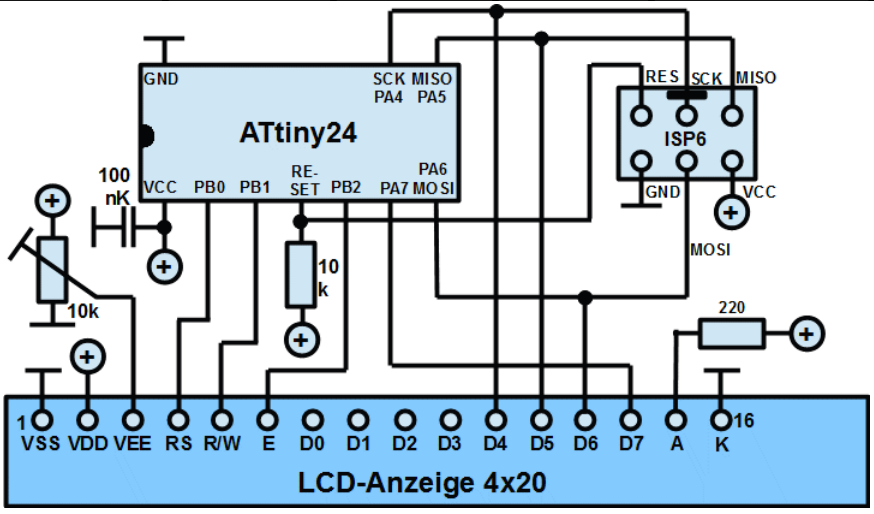


Es sieht so aus, als ob die Formel ausreichend korrekt und jedenfalls für den Zweck der Verzögerung bei der LCD-Ansteuerung gut geeignet ist.

10.5 Ansteuerung im Busy-Modus

10.5.1 Abfragen des Busy-Flags

Für das Abfragen des Busy-Flags muss der LCD-Datenbus auf Schreiben (R/W = 1) gesetzt werden, die ATtiny-Eingänge des Datenbuses müssen vorher als Eingänge



konfiguriert werden. Das setzt voraus, dass ein weiterer Pin den R/W-Eingang schaltet. Die R/W-Steuerung ist in diesem Schaltbild an PB1 realisiert.

10.5.2 Aufgabenstellung

Die Aufgabe ist die gleiche (Textausgabe auf der LCD), nur soll jetzt soweit als möglich das Busy-Bit der LCD zur Zeitsteuerung verwendet werden.

10.5.3 Programm

Das hier ist das Programm (hier geht es zum Quellcode).

```
;
; ********************************************
; * LCD-Display 4*20 am ATtiny24, 4-Bit mit Busy
; * (C)2016 by http://www.gsc-elektronic.net
; ********************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----------- Programmablauf ------------
;
; Initiiert das 4*20-LCD am ATtiny24 im
; Busy-Modus (Abfrage des Busy-Flags der
; LCD statt Verzoegerungsroutinen) und
; gibt einen Text auf der LCD aus.
;
; ----------- Ports, Portpins -----------
; LCD-Kontrollport
.equ pLcdCO   = PORTB  ; LCD-Kontrollport-Ausgabe
.equ pLcdCR   = DDRB   ; LCD-Kontrollportrichtung
.equ bLcdCOE  = PORTB2 ; LCD Enable Pin Output
.equ bLcdCRE  = DDB2   ; LCD Enable Pin Richtung
.equ bLcdCORS = PORTB0 ; LCD RS Pin Output
.equ bLcdCRRS = DDB0   ; LCD RS Pin Richtung
.equ bLcdCORW = PORTB1 ; LCD R/W Pin Output
.equ bLcdCRRW = DDB1   ; LCD R/W Pin Richtung
; LCD-Datenport
.equ pLcdDO   = PORTA  ; LCD-Datenport-Ausgabe
.equ pLcdDI   = PINA   ; LCD-Datenport-Eingabe
.equ pLcdDR   = DDRA   ; LCD-Datenport-Richtung
.equ mLcdDRW  = 0xF0   ; LCD-Datenport-Maske
    ; Schreiben
.equ mLcdDRR  = 0x00   ; LCD-Datenport-Maske
    ; Lesen
;
; ------- Register ----------------------
; frei: R0 .. R15
.def rmp = R16 ; Vielzweckregister
.def rmo = R17 ; weiteres Vielzweckregister
.def rLine = R18 ; Zeilenzaehler LCD
.def rLese = R19 ; Leseergebnis vom LCD-Datenport
; frei R20 .. R29
; verwendet: R31:R30, ZH:ZL, fuer Zaehlen
;
; ------- Konstanten --------------------
.equ Takt = 1000000 ; Default-Taktfrequenz
;
; -------- Programmstart, Init ----------
.CSEG ; Code Segment
.ORG 0 ; Start bei 0
  ; Stapel Init fuer Unterprogramme
  ldi rmp,LOW(RAMEND) ; Init Stapel
  out SPL,rmp ; in Stapelzeiger
  ; Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
Schreiben
  out pLcdDR,rmp ; auf Ausgabe
  ; LCD initiieren
  rcall LcdInit
  ; Text auf LCD ausgeben
  ldi ZH,HIGH(2*Texttabelle)
  ldi ZL,LOW(2*Texttabelle)
  rcall LcdText ; Text ab ZH:ZL ausgeben
  ; Sleep enable
  ldi rmp,1<<SE
  out MCUCR,rmp
Schleife:
  sleep ; schlafen
  rjmp Schleife
;
; Ausgabetext auf LCD
Texttabelle:
.db "LCD-Display ATtiny24",0x0D,0xFF ; Zeile 1
.db " gsc-elektronic.net ",0x0D,0xFF ; Zeile 2
.db " Testprogramm 4-Bit ",0x0D,0xFF ; Zeile 3
.db "mit Busy-Flag lesen",0x00 ; Zeile 4
;
; --------- LCD-Ansteuerung Init ----------
LcdInit:
  ; Warte 50 ms bis LCD hochgefahren ist
  rcall Warte50ms ; Karenzzeit 50 ms
  ; Versetze in 8-Bit-Modus (drei Mal)
  ldi rmp,0x30 ; 8-Bit-Modus
  rcall LcdC8Byte ; Schreibe im 8-Bit-Mode
  rcall Warte5ms ; Warte 5 ms
  ldi rmp,0x30
  rcall LcdC8Byte
  rcall Warte5ms
  ldi rmp,0x30
  rcall LcdC8Byte
  rcall Warte5ms
  ; Umstellung auf 4-Bit-Modus
  ldi rmp,0x20 ; Schalte in 4-Bit-Modus um
  rcall LcdC8Byte
  rcall Warte5ms
  ; Funktionseinstellungen LCD
  ldi rmp,0x28 ; 4-Bit-Modus, 4 Zeilen, 5*7
  rcall LcdC4Byte ; Byte an Kontrollregister LCD
  ldi rmp,0x0F ; Display ein, Blinken
  rcall LcdC4Byte ; Byte an Kontrollregister LCD
  ldi rmp,0x01 ; Display loeschen
  rcall LcdC4Byte ; Byte an Kontrollregister LCD
  ldi rmp,0x06 ; Autoindent
  rjmp LcdC4Byte ; Byte an Kontrollregister LCD
;
; Ausgabe von Text auf der LCD
;   Z zeigt auf Text im Flash
LcdText:
  clr rLine ; Zeilenzaehler
LcdText1:
  lpm rmp,Z+ ; lese Zeichen aus Flash
  cpi rmp,0 ; Ende Ausgabe?
  breq LcdTextRet ; nein
  cpi rmp,0xFF ; Fuellzeichen?
  breq LcdText1 ; ja, naechstes Zeichen
```

```
  cpi rmp,0x0D ; Zeilenwechsel?
  brne LcdText2 ; Kein Zeilenwechsel
  inc rLine ; Naechste Zeile
  mov rmp,rLine ; Setze Zeile
  rcall LcdLineSet ; Stelle Zeile ein
  rjmp LcdText1 ; weiter mit Zeichen
LcdText2: ; Ausgabe Zeichen
  rcall LcdD4Byte ; Zeichen ausgeben
  rjmp LcdText1 ; und weiter
LcdTextRet:
  ret ; Fertig
;
; Setzt den Ausgabecursor auf den
; Zeilenanfang der ausgewaehlten Zeile
;   Zeile: rmp 0 .. 3
LcdLineSet:
  cpi rmp,1 ; Zeile 2?
  brcs LcdLineSet1 ; nach Zeile 1
  breq LcdLineSet2 ; nach Zeile 2
  cpi rmp,2 ; Zeile 3?
  breq LcdLineSet3 ; nach Zeile 3
  rjmp LcdLineSet4 ; nach Zeile 4
LcdLineSet1:
  ldi rmp,0x80 ; Zeile 1
  rjmp LcdLineSetRmp
LcdLineSet2:
  ldi rmp,0xC0 ; Zeile 2
  rjmp LcdLineSetRmp
LcdLineSet3:
  ldi rmp,0x80+20 ; Zeile 3
  rjmp LcdLineSetRmp
LcdLineSet4:
  ldi rmp,0xC0+20 ; Zeile 4
  rjmp LcdLineSetRmp
LcdLineSetRmp:
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
;
; Datenwort-Ausgabe im 4-Bit-Modus
;   Daten in rmp
LcdD4Byte:
  rcall LcdBusy ; warte bis busy = Null
  sbi pLcdCO,bLcdCORS ; setze RS-Bit
  rjmp Lcd4Byte ; gib Byte in rmp aus
;
; Kontrollwort-Ausgabe im 4-Bit-Modus
;   Daten in rmp
LcdC4Byte:
  rcall LcdBusy ; warte bis busy = Null
  cbi pLcdCO,bLcdCORS ; loesche RS-Bit
; Ausgabe Byte im 4-Bit-Modus mit Busy
Lcd4Byte:
  push rmp ; rmp auf Stapel legen
  andi rmp,0xF0 ; unteres Nibble loeschen
  in rmo,pLcdDI ; Lese Data-Input-Port
  andi rmo,0x0F ; oberes Nibble loeschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  pop rmp ; rmp wieder herstellen
  andi rmp,0x0F ; oberes Nibble loeschen
  swap rmp ; Nibble vertauschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  ret ; fertig
;
; Warte bis Busy Null
LcdBusy:
  push rmp ; rette rmp
  ldi rmp,mLcdDRR ; Lesemaske
  out pLcdDR,rmp ; in Richtungsregister
  cbi pLcdCO,bLcdCORS ; loesche RS
  sbi pLcdCO,bLcdCORW ; R/W setzen
LcdBusy1:
  sbi pLcdCO,bLcdCOE ; setze LCD-Enable
  nop
  in rLese,pLcdDI ; lese oberes Nibble
  cbi pLcdCO,bLcdCOE ; loesche LCD-Enable
  andi rLese,0xF0 ; unteres Nibble loeschen
  sbi pLcdCO,bLcdCOE ; setze LCD-Enable
  nop
  in rmp,pLcdDI ; lese unteres Nibble
  cbi pLcdCO,bLcdCOE ; loesche LCD-Enable
  andi rmp,0xF0 ; loesche unteres Nibble
  swap rmp ; oberes/unteres Nibble tauschen
  or rLese,rmp ; oberes und unteres Nibble
  sbrc rLese,7 ; ueberspringe bei Busy=0
  rjmp LcdBusy1 ; wiederhole bis Busy=0
  cbi pLcdCO,bLcdCORW ; R/W loeschen
  ldi rmp,mLcdDRW ; Schreibmaske
  out pLcdDR,rmp ; in Richtungsregister
  pop rmp ; rmp wieder herstellen
  ret ; zurueck
;
; Kontrollwort-Ausgabe im 8-Bit-Modus
;   Datenbyte in rmp
LcdC8Byte:
  cbi pLcdCO,bLcdCORS ; RS-Bit loeschen
  andi rmp,0xF0 ; unteres Nibble loeschen
  in rmo,pLcdDI ; Lese Data-Input-Port
  andi rmo,0x0F ; oberes Nibble loeschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  ret ; fertig
;
; ------ Warteroutinen ------------------
Warte50ms: ; Warteroutine 50 ms
.equ c50ms = 50000
.equ n50ms = (c50ms-16)/4
;   rcall: + 3
  push ZH ; + 2
  push ZL ; + 2
  ldi ZH,HIGH(n50ms) ; + 1
  ldi ZL,LOW(n50ms) ; + 1
  rjmp Warte ; + 2, gesamt = 11
Warte5ms: ; Warteroutine 5 ms
.equ c5ms = 5000
.equ n5ms = (c5ms-16)/4
  push ZH
  push ZL
  ldi ZH,HIGH(n5ms)
  ldi ZL,LOW(n5ms)
  rjmp Warte
; Warteroutine Z Takte
Warte: ; Warteschleife, Takte=4*(n-1)+11= 4*n + 7
  sbiw ZL,1 ; + 2
  brne Warte ; + 1 / 2
  pop ZL ; + 2
  pop ZH ; +2
  ret ; + 4, Gesamt=4*n+18
;
```

In diesem Programm sind keine neuen Instruktionen enthalten.

10.6 Neue Zeichen definieren

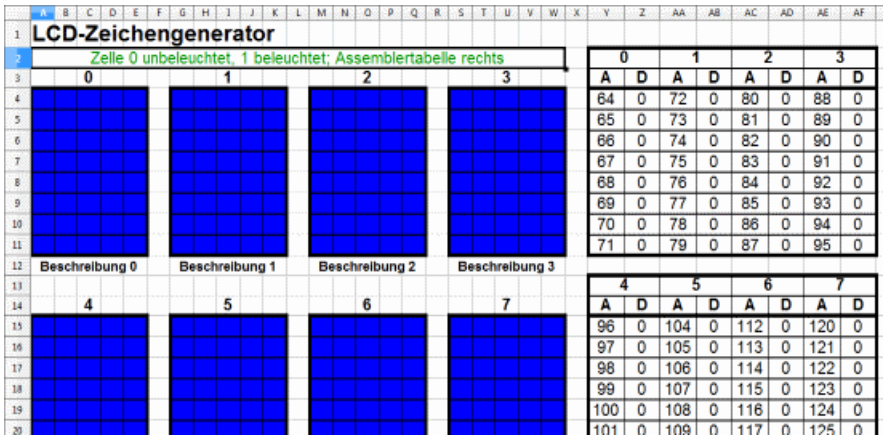
10.6.1 Der Zeichengenerator in LCDs

Alle LCD bieten die Möglichkeit, eigene Zeichen zu definieren, allerdings nur die Zeichen mit dem Code 0 bis 7. Sie sind defaultmässig belegt, ihr Inhalt kann aber überschrieben werden.

Das Überschreiben funktioniert so:

1. Ein Byte mit dem Aufbau 0b01NNNZZZ wird an den Kontrollport der LCD gesendet (mit RS = Null). Darin bedeutet NNN die Zeichennummer, die definiert werden soll (0 bis 7). ZZZ ist die Zeile des Zeichens, 0 ist die oberste Zeile und 7 die unterste.
2. Danach wird ein Byte mit dem Aufbau 0b000BBBBB an den Datenport der LCD gesendet (mit RS = Eins), worin B die fünf Pixel dieser Zeile angibt (vorderstes Bit = linkestes Pixel).
3. Für alle acht Zeilen des Zeichens ist ein Paar aus Adresse und Daten zu schreiben.

10.6.2 Software für das Erzeugen neuer Zeichen



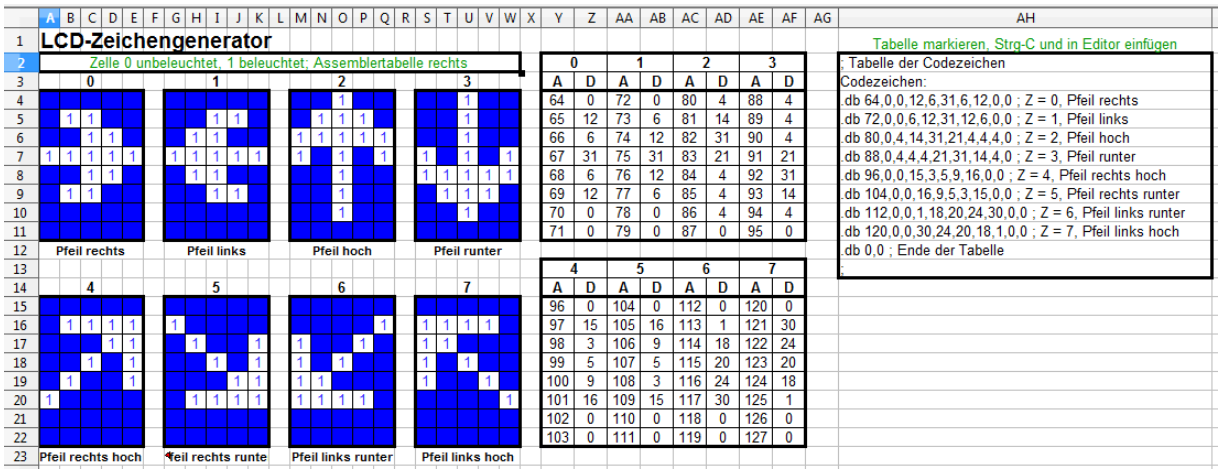
0		1		2		3	
A	D	A	D	A	D	A	D
64	0	72	0	80	0	88	0
65	0	73	0	81	0	89	0
66	0	74	0	82	0	90	0
67	0	75	0	83	0	91	0
68	0	76	0	84	0	92	0
69	0	77	0	85	0	93	0
70	0	78	0	86	0	94	0
71	0	79	0	87	0	95	0

4		5		6		7	
A	D	A	D	A	D	A	D
96	0	104	0	112	0	120	0
97	0	105	0	113	0	121	0
98	0	106	0	114	0	122	0
99	0	107	0	115	0	123	0
100	0	108	0	116	0	124	0
101	0	109	0	117	0	125	0

Um sich das Entwerfen solcher Zeichen zu erleichtern, kann man eine Tabellenkalkulation verwenden.

So lassen sich komfortabel Zeichen entwerfen: hier ist eine Tabellenkalkulation am Werk (im OpenOffice-Format hier, im Excel-Format hier). Um ein Pixel weiss zu machen, trägt man in die Zelle eine Eins ein, um es nicht leuchten zu lassen eine Null. Die Adressen- und Daten in Dezimalformat zeigt die Tabelle rechts an.

Daraus macht die Tabellenkalkulation auf der rechten Seite direkt eine assemblergerechte Tabelle. Sie beginnt mit der ersten Adresse, ab der das Zeichen abgelegt werden soll, gefolgt von einer Null (geradzahlig ist bei Tabellen immer Pflicht). Darauf folgen die acht Datenzeilen des Zeichens. Sind alle gewünschten Zeichen definiert (es können auch weniger als acht sein), kommt eine Null für den Abbruch. Die Tabelle kann mit Strg-C/Strg-V direkt in den Assembler-Quellcode eingefügt werden.



0		1		2		3	
A	D	A	D	A	D	A	D
64	0	72	0	80	4	88	4
65	12	73	6	81	14	89	4
66	6	74	12	82	31	90	4
67	31	75	31	83	21	91	21
68	6	76	12	84	4	92	31
69	12	77	6	85	4	93	14
70	0	78	0	86	4	94	4
71	0	79	0	87	0	95	0

4		5		6		7	
A	D	A	D	A	D	A	D
96	0	104	0	112	0	120	0
97	15	105	16	113	1	121	30
98	3	106	9	114	18	122	24
99	5	107	5	115	20	123	20
100	9	108	3	116	24	124	18
101	16	109	15	117	30	125	1
102	0	110	0	118	0	126	0
103	0	111	0	119	0	127	0

Tabelle markieren, Strg-C und in Editor einfügen

```
; Tabelle der Codezeichen
Codezeichen:
.db 64,0,0,12,6,31,6,12,0,0 ; Z = 0, Pfeil rechts
.db 72,0,0,6,12,31,12,6,0,0 ; Z = 1, Pfeil links
.db 80,0,4,14,31,21,4,4,4,0 ; Z = 2, Pfeil hoch
.db 88,0,4,4,4,21,31,14,4,0 ; Z = 3, Pfeil runter
.db 96,0,0,15,3,5,9,16,0,0 ; Z = 4, Pfeil rechts hoch
.db 104,0,0,16,9,5,3,15,0,0 ; Z = 5, Pfeil rechts runter
.db 112,0,0,1,18,20,24,30,0,0 ; Z = 6, Pfeil links runter
.db 120,0,0,30,24,20,18,1,0,0 ; Z = 7, Pfeil links hoch
.db 0,0 ; Ende der Tabelle
```

Den Zeichengenerator mit den Pfeilen gibt es im OpenOffice-Format hier, im M$-Excel-Format

hier.

10.6.3 Aufgabenstellung

Definiere die Zeichen "Pfeil nach rechts", "Pfeil nach links", "Pfeil nach oben", "Pfeil nach unten", "Pfeil nach rechts oben", "Pfeil nach rechts unten", "Pfeil nach links unten" und "Pfeil nach links oben" und lasse diese auf der LCD auf Zeile 4 erscheinen.

10.6.4 Das Programm

10.6.4.1 Umbauten im Programm

Um die Aufgabe zu lösen, kann das vorherige Programm als Basis verwendet werden. Folgende Umbauten sind dazu nötig:

- Zwischen dem Initiieren der LCD und der Textausgabe muss ein neuer Programmteil eingebaut werden, der die neuen Zeichen in die LCD schreibt.
- Der Text zur Ausgabe muss für die neuen Zeichen 0 bis 7 umgeschrieben werden. Da bei der bisherigen Textausgaberoutine die Null das Beenden der Ausgabe bewirkt hat, muss für die Beendigung der Ausgabe die Null durch ein anderes Zeichen ersetzt werden (ich finde das FE dafür eine gute Wahl).

10.6.4.1 Der Code

Hier ist der Quellcode (zum Quellcode im asm-Format geht es hier).

```
;                                               .def rLine = R18 ; Zeilen bei LcdText, Zaehler
;***********************************************    ; bei LcdChars
; * LCD-Display 4*20 am tiny24/4-Bit/Busy/Zeichen .def rLese = R19 ; Leseergebnis vom LCD-Datenport
; * (C)2016 by http://www.gsc-elektronic.net     .def rAddr = R20 ; Zeilenadresse LCD-Zeichen
; *********************************************** ; frei R21 .. R29
;                                               ; verwendet: R31:R30, ZH:ZL, fuer Zaehlen
.NOLIST                                         ;
.INCLUDE "tn24def.inc"                          ; ------- Konstanten --------------------
.LIST                                           .equ Takt = 1000000 ; Default-Taktfrequenz
;                                               ;
; ----------- Programmablauf ------------       ; -------- Programmstart, Init ----------
;                                               .CSEG ; Code Segment
; Initiiert die an den ATtiny24 angeschlossene  .ORG 0 ; Start bei 0
; 4*20-LCD im Busy-Modus, schreibt acht neue      ; Stapel Init fuer Unterprogramme
; Zeichen in die LCD (Richtungszeichen) und       ldi rmp,LOW(RAMEND) ; Init Stapel
; stellt diese dar.                               out SPL,rmp ; in Stapelzeiger
;                                                 ; Port-Ausgaenge initiieren
; ----------- Ports, Portpins -----------         ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
; LCD-Kontrollport                                out pLcdCR,rmp ; Kontrollport-Ausgaenge
.equ pLcdCO   = PORTB  ; LCD-Kontrollport-Ausgabe clr rmp ; Ausgaenge aus
.equ pLcdCR   = DDRB   ; LCD-Kontrollportrichtung out pLcdCO,rmp ; an Kontrollport
.equ bLcdCOE  = PORTB2 ; LCD Enable Pin Output    ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
.equ bLcdCRE  = DDB2   ; LCD Enable Pin Richtung    ;Schreiben
.equ bLcdCORS = PORTB0 ; LCD RS Pin Output        out pLcdDR,rmp ; auf Ausgabe
.equ bLcdCRRS = DDB0   ; LCD RS Pin Richtung      ; LCD initiieren
.equ bLcdCORW = PORTB1 ; LCD R/W Pin Output       rcall LcdInit
.equ bLcdCRRW = DDB1   ; LCD R/W Pin Richtung     ; Zeichen definieren
; LCD-Datenport                                   rcall LcdChars ; Neue Zeichen definieren
.equ pLcdDO   = PORTA  ; LCD-Datenport-Ausgabe    ; Text auf LCD ausgeben
.equ pLcdDI   = PINA   ; LCD-Datenport-Eingabe    ldi ZH,HIGH(2*Texttabelle)
.equ pLcdDR   = DDRA   ; LCD-Datenport-Richtung   ldi ZL,LOW(2*Texttabelle)
.equ mLcdDRW  = 0xF0   ; LCD-Datenport-Maske      rcall LcdText ; Text ab ZH:ZL ausgeben
    ; Schreiben                                   ; Sleep enable
.equ mLcdDRR  = 0x00   ; LCD-Datenport-Maske      ldi rmp,1<<SE
    ; Lesen                                       out MCUCR,rmp
;                                               Schleife:
; ------- Register ----------------------         sleep ; schlafen
; frei: R0 .. R15                                 rjmp Schleife
.def rmp = R16 ; Vielzweckregister              ;
.def rmo = R17 ; weiteres Vielzweckregister     ; Ausgabetext auf LCD
```

```
Texttabelle:
.db "LCD-Display ATtiny24",0x0D,0xFF ; Zeile 1
.db " gsc-elektronic.net ",0x0D,0xFF ; Zeile 2
.db "Eigene Zeichen hier:",0x0D,0xFF ; Zeile 3
.db " ",0x00," ",0x01," ",0x02," ",0x03 ; Zeile 4
    ; links
.db " ",0x04," ",0x05," ",0x06," ",0x07 ; Zeile 4
    ; rechts
.db 0xFE,0xFE ; Ende des Texts
;
; --------- LCD-Ansteuerung Init ----------
LcdInit:
  ; Warte 50 ms bis LCD hochgefahren ist
  rcall Warte50ms ; Karenzzeit 50 ms
  ; Versetze in 8-Bit-Modus (drei Mal)
  ldi rmp,0x30 ; 8-Bit-Modus
  rcall LcdC8Byte ; Schreibe im 8-Bit-Mode
  rcall Warte5ms ; Warte 5 ms
  ldi rmp,0x30
  rcall LcdC8Byte
  rcall Warte5ms
  ldi rmp,0x30
  rcall LcdC8Byte
  rcall Warte5ms
  ; Umstellung auf 4-Bit-Modus
  ldi rmp,0x20 ; Schalte in 4-Bit-Modus um
  rcall LcdC8Byte
  rcall Warte5ms
  ; Funktionseinstellungen LCD
  ldi rmp,0x28 ; 4-Bit-Modus, 4 Zeilen, 5*7
  rcall LcdC4Byte ; Byte an Kontrollregister LCD
  ldi rmp,0x0F ; Display ein, Blinken
  rcall LcdC4Byte ; Byte an Kontrollregister LCD
  ldi rmp,0x01 ; Display loeschen
  rcall LcdC4Byte ; Byte an Kontrollregister LCD
  ldi rmp,0x06 ; Autoindent
  rjmp LcdC4Byte ; Byte an Kontrollregister LCD
;
; Eigene Zeichen definieren
LcdChars:
  ldi ZH,HIGH(2*Codezeichen) ; Zeiger  Zeichentab
  ldi ZL,LOW(2*Codezeichen)
LcdChars1:
  lpm rAddr,Z ; Lese Adresse
  tst rAddr ; auf Null pruefen
  breq LcdChars3 ; fertig
  adiw ZL,2 ; auf naechstes Zeichen
  ldi rLine,8
LcdChars2:
  mov rmp,rAddr ; Adresse setzen
  rcall LcdC4Byte ; an LCD
  lpm rmp,Z+ ; lese Daten
  rcall LcdD4Byte ; Ausgabe an LCD
  inc rAddr ; Adresse erhoehen
  dec rLine ; Zaehler abwaerts
  brne LcdChars2 ; noch welche
  rjmp LcdChars1 ; naechstes Zeichen
LcdChars3:
  ret ; fertig

; Tabelle der Codezeichen
Codezeichen:
.db 64,0,0,12,6,31,6,12,0,0 ; Z = 0, Pfeil rechts
.db 72,0,0,6,12,31,12,6,0,0 ; Z = 1, Pfeil links
.db 80,0,4,14,31,21,4,4,4,0 ; Z = 2, Pfeil hoch
.db 88,0,4,4,4,21,31,14,4,0 ; Z = 3, Pfeil runter
.db 96,0,0,15,3,5,9,16,0,0 ; Z = 4, Pfeil rechts
    ; hoch
.db 104,0,0,16,9,5,3,15,0,0 ; Z = 5, Pfeil rechts
    ; runter
.db 112,0,0,1,18,20,24,30,0,0 ; Z = 6, Pfeil
    ; links runter
.db 120,0,0,30,24,20,18,1,0,0 ; Z = 7, Pfeil
    ; links hoch
.db 0,0 ; Ende der Tabelle
;
; Ausgabe von Text auf der LCD
;   Z zeigt auf Text im Flash
LcdText:
  rcall LcdLineSet1 ; Auf erste Zeile
  clr rLine ; Zeilenzaehler
LcdText1:
  lpm rmp,Z+ ; lese Zeichen aus Flash
  cpi rmp,0xFE ; Ende Ausgabe?
  breq LcdTextRet ; ja
  cpi rmp,0xFF ; Fuellzeichen?
  breq LcdText1 ; ja, naechstes Zeichen
  cpi rmp,0x0D ; Zeilenwechsel?
  brne LcdText2 ; Kein Zeilenwechsel
  inc rLine ; Naechste Zeile
  mov rmp,rLine ; Setze Zeile
  rcall LcdLineSet ; Stelle Zeile ein
  rjmp LcdText1 ; weiter mit Zeichen
LcdText2: ; Ausgabe Zeichen
  rcall LcdD4Byte ; Zeichen ausgeben
  rjmp LcdText1 ; und weiter
LcdTextRet:
  ret ; Fertig
;
; Setzt den Ausgabecursor auf den
; Zeilenanfang der ausgewaehlten Zeile
;   Zeile: rmp 0 .. 3
LcdLineSet:
  cpi rmp,1 ; Zeile 2?
  brcs LcdLineSet1 ; nach Zeile 1
  breq LcdLineSet2 ; nach Zeile 2
  cpi rmp,2 ; Zeile 3?
  breq LcdLineSet3 ; nach Zeile 3
  rjmp LcdLineSet4 ; nach Zeile 4
LcdLineSet1:
  ldi rmp,0x80 ; Zeile 1
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
LcdLineSet2:
  ldi rmp,0xC0 ; Zeile 2
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
LcdLineSet3:
  ldi rmp,0x80+20 ; Zeile 3
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
LcdLineSet4:
  ldi rmp,0xC0+20 ; Zeile 4
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
;
; Datenwort-Ausgabe im 4-Bit-Modus
;   Daten in rmp
LcdD4Byte:
  rcall LcdBusy ; warte bis busy = Null
  sbi pLcdCO,bLcdCORS ; setze RS-Bit
  rjmp Lcd4Byte ; gib Byte in rmp aus
;
; Kontrollwort-Ausgabe im 4-Bit-Modus
;   Daten in rmp
LcdC4Byte:
  rcall LcdBusy ; warte bis busy = Null
  cbi pLcdCO,bLcdCORS ; loesche RS-Bit
; Ausgabe Byte im 4-Bit-Modus mit Busy
Lcd4Byte:
  push rmp ; rmp auf Stapel legen
  andi rmp,0xF0 ; unteres Nibble loeschen
  in rmo,pLcdDI ; Lese Data-Input-Port
  andi rmo,0x0F ; oberes Nibble loeschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  pop rmp ; rmp wieder herstellen
  andi rmp,0x0F ; oberes Nibble loeschen
  swap rmp ; Nibble vertauschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
```

```
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  ret ; fertig
;
; Warte bis Busy Null
LcdBusy:
  push rmp ; rette rmp
  ldi rmp,mLcdDRR ; Lesemaske
  out pLcdDR,rmp ; in Richtungsregister
  cbi pLcdCO,bLcdCORS ; loesche RS
  sbi pLcdCO,bLcdCORW ; R/W setzen
LcdBusy1:
  sbi pLcdCO,bLcdCOE ; setze LCD-Enable
  nop
  in rLese,pLcdDI ; lese oberes Nibble
  cbi pLcdCO,bLcdCOE ; loesche LCD-Enable
  andi rLese,0xF0 ; unteres Nibble loeschen
  sbi pLcdCO,bLcdCOE ; setze LCD-Enable
  nop
  in rmp,pLcdDI ; lese unteres Nibble
  cbi pLcdCO,bLcdCOE ; loesche LCD-Enable
  andi rmp,0xF0 ; loesche unteres Nibble
  swap rmp ; oberes/unteres Nibble tauschen
  or rLese,rmp ; oberes und unteres Nibble
  sbrc rLese,7 ; ueberspringe bei Busy=0
  rjmp LcdBusy1 ; wiederhole bis Busy=0
  cbi pLcdCO,bLcdCORW ; R/W loeschen
  ldi rmp,mLcdDRW ; Schreibmaske
  out pLcdDR,rmp ; in Richtungsregister
  pop rmp ; rmp wieder herstellen
  ret ; zurueck
;
; Kontrollwort-Ausgabe im 8-Bit-Modus
;   Datenbyte in rmp
LcdC8Byte:
  cbi pLcdCO,bLcdCORS ; RS-Bit loeschen
  andi rmp,0xF0 ; unteres Nibble loeschen
  in rmo,pLcdDI ; Lese Data-Input-Port
  andi rmo,0x0F ; oberes Nibble loeschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  ret ; fertig
;
; ------ Warteroutinen ------------------
Warte50ms: ; Warteroutine 50 ms
.equ c50ms = 50000
.equ n50ms = (c50ms-16)/4
;   rcall: + 3
  push ZH ; + 2
  push ZL ; + 2
  ldi ZH,HIGH(n50ms) ; + 1
  ldi ZL,LOW(n50ms) ; + 1
  rjmp Warte ; + 2, gesamt = 11
Warte5ms: ; Warteroutine 5 ms
.equ c5ms = 5000
.equ n5ms = (c5ms-16)/4
  push ZH
  push ZL
  ldi ZH,HIGH(n5ms)
  ldi ZL,LOW(n5ms)
  rjmp Warte
; Warteroutine Z Takte
Warte: ; Warteschleife, Takte = 4*(n-1)+11 = 4*n +
7
  sbiw ZL,1 ; + 2
  brne Warte ; + 1 / 2
  pop ZL ; + 2
  pop ZH ; +2
  ret ; + 4, Gesamt=4*n+18
;
```

Und das hier ist der Lohn der ganzen Arbeit: jede Menge schöne Pfeile.

Hier wurde statt *CPI* mal "*TST* Register" verwendet. Neu ist *ADIW* Register,N, was die Konstante N zum Doppelregister hinzuzählt (geht nur mit R24, R26, R28 und R30).

Top	Home	Einführung	ATtiny 24	Hardware	LCD Zeit	LCD busy	LCD Zeichen

Noch ein Hinweis: Alle hier beschriebenen Modi (4/8-Bit-Ansteuerung, Warten/Busy, Spezialzeichenerzeugung) sind in einer fortgeschrittenen LCD-Include-Datei zusammengeführt, die man recht einfach in jeden Assembler-Quellcode einfügen kann, nach Bedarf konfigurieren und die alle Aufrufe zur Verfügung stellt, die man zur Ansteuerung von LCDs braucht. Sie ist unter dieser URL beschrieben und hier als Include-Datei herunterladbar.

Lektion 11: EEPROM mit LCD am ATtiny24

Neu in dieser Lektion ist der lesende und schreibende Zugriff auf das interne EEPROM und die Darstellung von Dezimalzahlen auf der LCD.

11.0 Übersicht

11.1 Einführung in das EEPROM

11.1.1 Das EEPROM als nichtflüchtiger Speicher

Das prozessorinterne statische RAM ist flüchtig, das heißt sein Inhalt übersteht ein Abschalten der Betriebsspannung nicht. Startet der Prozessor dann neu, muss er den Inhalt des SRAM wieder auf vorprogrammierte Anfangswerte setzen.

Das eingebaute EEPROM ist hingegen nichtflüchtig, d. h. sein Inhalt bleibt auch ohne Betriebsspannungsversorgung über nahezu beliebig lange Zeiträume erhalten. Typische Inhalte, für die das sinnvoll ist, sind z. B. verstellbare Weckzeitpunkte einer Uhr, voreingestellte und vom Anwender einstellbare Laufzeiten oder versteckte Einschaltzähler für elektrische Geräte.

Da EEPROM-Speicher nur eine gewisse Anzahl Schreibvorgänge verkraften, sind sie als Erweiterung des SRAM ungeeignet. Sie sind nicht gedacht für einige 100 Schreibvorgänge pro Sekunde. Da pro Schreibvorgang ohnehin etwa 3 ms nötig sind, kriegt man mehr schon gar nicht hin.

Der ATtiny24 hat 128 Byte EEPROM. Es startet an Adresse 0x0000 und endet daher bei 0x007F.

11.1.2 Das EEPROM programmieren

Das EEPROM wird beim Löschen ("Erase") des Flash-Speichers ebenfalls gelöscht. Ebenso wie beim Flash-Speicher bedeutet Löschen das vollständige Beschreiben aller EEPROM-Zellen mit Einsen (0xFF). Die meisten und die neueren AVR-Typen erlauben es jedoch, dieses doppelte Löschen abzuschalten. Dazu muss im Fuses-Bereich die Fuse "EESAVE" gesetzt werden. Bei gesetztem Fusebit ist das Ändern des EEPROM-Inhaltes nur über das explizierte Programmieren und über Schreiboperationen des Prozessors möglich.

Programmieren des EEPROMs geht folgendermaßen. Im Assemblercode wird folgendes geschrieben:

```
.ESEG ; Umschalten zum EEPROM-Segment
.ORG 0 ; Bei Adresse 0 beginnen
.db 0,1,2,3,4 ; Zahlen schreiben
.db "01234" ; Text schreiben
```

Die Direktive ".ESEG" bewirkt, dass die erzeugten Bytes der folgenden Operationen in das EEPROM-Segment ausgegeben werden. Sie landen beim Assemblieren in einer Datei mit Namen "Quellcode.eep", die im Intel-Hex-Format kodiert ist. Ihr Inhalt kann mit einem einfachen Texteditor inspiziert werden.

Da das EEPROM byteweise organisiert ist, kann jede beliebige Anzahl Bytes in das EEPROM-

Segment geschrieben werden, bis es voll ist. Mit ".ORG Adresse" kann bewirkt werden, dass nur der vom Segment definierte Bereich des EEPROMs beschrieben wird und andere Inhalte des EEPROMs unverändert erhalten bleiben.

Im EEPROM-Segment sind nur ".DB"- und ".DW"-Direktiven zulässig.

Zum eigentlichen Schreiben des erzeugten Inhalts in das EEPROM öffnet man die Tools im Studio, wählt im EEPROM-Programmierunterfenster durch Klicken auf das kleine Quadrat mit "..." die erzeugte ".eep"-Datei aus. Ihr Inhalt wird mit "WRITE" in das EEPROM übertragen.

Im gleichen Programmierdialog kann man den Inhalt des EEPROMs auch auslesen oder verifizieren.

11.1.3 Vom Programm aus das EEPROM auslesen

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	EECR
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	X	X	0	0	X	0	

Das Auslesen von Inhalten des EEPROMs umfasst folgende Einzelschritte:

1. Durch Abfragen des EEPE-Bits im EEPROM-Kontrollregister EECR ist sicherzustellen, dass das EEPROM zum Lesen bereit ist.
2. Die zu lesende EEPROM-Adresse wird in die beiden Register EEARH (MSB) und EEARL (LSB) zu schreiben. Selbst wenn der AVR nur bis zu 256 Byte hat, sollte das EEARH mit Nullen beschrieben werden.
3. Dann wird das Read Enable Bit EERE im Kontrollregister Eins gesetzt. Dies blockiert die Tätigkeit des Prozessors für vier Takte und der Inhalt des EEPROMs an der eingestellten Adresse kann danach sofort aus dem Datenregister "EEDR" ausgelesen werden.

Bei nachfolgenden Lesevorgängen muss nur die LSB-Adresse neu geschrieben werden, solange sich das MSB nicht ändert.

12.1.4 Daten vom Programm aus in das EEPROM schreiben

Das Schreiben geht folgendermaßen:

1. Zunächst ist wie beim Lesen sicherzustellen, dass Schreibvorgänge abgeschlossen sind.
2. Durch Schreiben von Null in das EECR sicherstellen, dass EEPM1 und EEPM0 sowie EERE gelöscht sind (M1 und M0 = Null: lösche die Zelle zuerst und beschreibe sie danach in einem Schritt).
3. MSB und LSB der Adresse in die Adressregister EEARH und EEARL schreiben.
4. Das zu programmierende Byte in das Datenregister EEDR schreiben.
5. Das EEMPE-Bit im EEPROM-Kontrollregister auf Eins setzen. Dieses Bit löscht sich nach vier Taktzyklen selbst.
6. Innerhalb der vier Taktzyklen das EEPE-Bit auf Eins setzen. Der Programmiervorgang dauert 3,2 ms, danach setzt sich das EEPE-Bit selbst zurück.

Bei gestartem Programmiervorgang (nicht vorher!) kann durch Setzen des EERIE-Bits das Auslösen des EE_RDY-Interrupts ausgelöst werden. Dieses Bit muss nach Abschluss der Programmiervorgänge unbedingt und sofort wieder gelöscht werden, da das schreibbereite EEPROM sonst einen Dauerinterrupt auslöst und den Prozessor blockiert.

Top	Home	EEPROM	Dezimalumwandlung	Hardware	Bytezähler	Wortzähler

11.2 Einführung in die Dezimalumwandlung

Da wir jetzt eine LCD haben und schon in diesem Kapitel gerne Zahlen in lesbarer Form ausgeben wollen, muss es jetzt sein: wir lernen Binärzahlen in Dezimalzahlen umzuwandeln.

11.2.1 Die Primitivstversion

Um eine 8-Bit-Binärzahl in eine Dezimalzahl zu verwandeln, könnten wir mit "CPI rZahl,200" zunächst abfragen, ob die erste Ziffer eine 2 ist. Wenn ja, käme eine ASCII-Zwei (hex 32, dezimal 50) zur Anzeige. Wenn nicht, testen wir auf 100, was bei Erfolg eine ASCII-1 produziert. Wenn nicht, ist es eine ASCII-0 oder, wenn wir führende Nullen unterdrücken wollen, ein Leerzeichen (hex 20, dezimal 32) an die LCD zu senden.

Nun käme die zweite Ziffer dran. War die Zahl größer oder gleich 200, müssten wir von der Zahl mit "SUBI rZahl,200" die 200 abziehen. Analog bei größer oder gleich 100. Um die zweite Ziffer zu ermitteln, müssten wir nun 90, 80, 70 bis herunter zu 10 vergleichen. Das würde eine lustige, aber langwierige Angelegenheit. Wers nicht glaubt, kann ja mal den Assemblercode dafür in die Tasten hauen und mit dem Studio simulieren, ob es auch richtig herauskommt.

Die dritte, letzte Ziffer ist dann relativ einfach: zum Rest wird einfach hex 32 addiert und zur Anzeige gesendet. Hingeschrieben: "ADDI rZahl,0x32" und reingefallen: ADDI gibt es in AVR-Assembler gar nicht. Der Kenner schreibt stattdessen "SUBI rZahl,-0x32", was selbiges tut.

11.2.2 Die bessere Version

Es gibt natürlich komfortablere Versionen der Umwandlungsaufgabe. Die besteht darin, mit einem Zähler festzustellen, wie oft man 100 bzw. bei der zweiten Ziffer 10 von der Zahl abziehen kann. Das kann so gehen:

```
; die Binaerzahl ist in R0
  ldi R16,100
  clr R1 ; R1 ist Zaehler
Zaehlen1:
  cp R0,R16 ; vergleiche mit 100
  brcs Ziffer1 ; Ueberlauf, Ziffer fertig
  sub R0,R16 ; 100 abziehen
  inc R1 ; Zaehler erhoehen
  rjmp Zaehlen1 ; weiter vergleichen
Ziffer1:
  ldi R16,'0' ; ASCII-Null
  add R16,R1 ; Ergebnis dazu zaehlen
  ; Ziffer 1 ausgeben
  ldi R16,10 ; jetzt die Zehner
  clr R1 ; bei Null anfangen
Zaehlen2:
  cp R0,R16 ; vergleiche mit 10
  brcs Ziffer2 ; Ueberlauf, Ziffer fertig
  sub R0,R16 ; 10 abziehen
  inc R1 ; Zaehler erhoehen
  rjmp Zaehlen2 ; weiter vergleichen
Ziffer2:
  ldi R16,'0' ; ASCII-Null
  add R16,R1 ; Zaehler addieren
  ; Ziffer 2 ausgeben
  ldi R16,'0' ; ASCII-Null
  add R16,R0 ; Addiere Zahlenrest
  ; Ziffer 3 ausgeben
```

Neu ist die Instruktion CP Register,Register, die das zweite Register temporär vom ersten subtrahiert und die entsprechenden Flaggen setzt.

Die beiden rot markierten Teile kommen doppelt vor, wir könnten diesen Teil als Unterfunktion formulieren und mit RCALL aufrufen.

11.2.3 Die 16-Bit-Version

Haben wir eine 16-Bit-Zahl in zwei Registern, dann muss das Vergleichen und Abziehen im 16-Bit-Modus erfolgen. Das geht dann so:

```
; die Binaerzahl ist in R1:R0
  ldi ZH,HIGH(10000) ; Zehntausender
  ldi ZL,LOW(10000)
  rcall Zaehlen
  ; Ziffer 1 ausgeben
  ldi ZH,HIGH(1000) ; Tausender
  ldi ZL,LOW(1000)
  rcall Zaehlen
  ; Ziffer 2 ausgeben
  ldi ZH,HIGH(100) ; Hunderter
  ldi ZL,LOW(100)
  rcall Zaehlen
  ; Ziffer 3 ausgeben
  ldi ZL,LOW(10) ; Zehner
  rcall Zaehlen
  ; Ziffer 4 ausgeben
  ldi R16,'0'
  add R16,R0
  ; Ziffer 5 ausgeben
  ; fertig
; Unterprogramm
Zaehlen:
  clr R16 ; R16 ist Zaehler
Zaehlen1:
  sub R0,ZL ; ziehe LSB ab
  sbc R1,ZH ; ziehe MSB ab
  brcs Zaehlen2 ; Uebertrag
  inc R16
  rjmp Zaehlen1
Zaehlen2:
  add R0,ZL ; addiere LSB
  adc R1,ZH ; addiere MSB
  subi R16,-'0' ; addiere ASCII-Null
  ret ; zurueck
```

Neue Instruktionen hier:

- SBC Register,Register: subtrahiere das zweite Register plus das Carry-Bit vom Ersten.

Das Problem ist gelöst, nur hat die Lösung noch einen Makel: führende Nullen würden als Nullen ausgegeben.

Damit ist das Instrumentarium und die Methode klar, um selbst 24-Bit- oder 32-Bit-Zahlen zu handhaben.

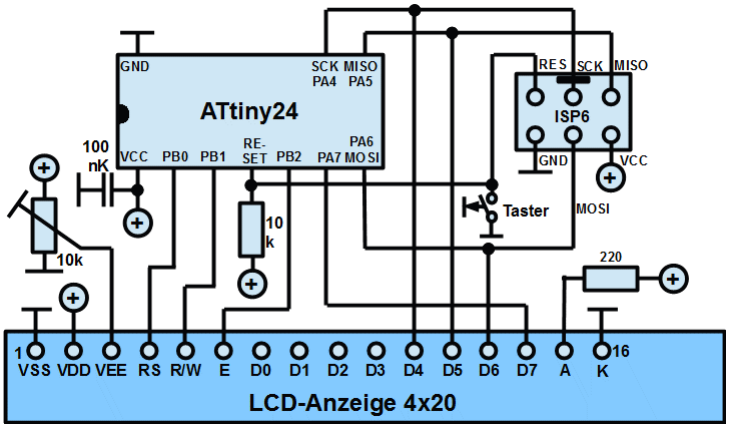
Top	Home	EEPROM	Dezimalumwandlung	Hardware	Bytezähler	Wortzähler

11.3 Aufgabe, Hardware, Bauteile und Aufbau

11.3.1 Aufgabe

Die folgende Aufgabe ist zu lösen: Bei jedem Reset und beim Anlegen der Betriebsspannung ist ein Zähler zu erhöhen und sein alter und neuer Stand auf der LCD auszugeben.

11.3.2 Schaltbild



Damit wir nicht ständig die Betriebsspannung abziehen müssen, montieren wir einen Resettaster.

Die Bauteile kennen wir alle schon, die Montage ist auch simpel.

11.3.3 Alternativer Aufbau

Wer dieses oder die nächsten Experimente lieber auf kompaktere Weise aufbauen möchte und die vielen Zuleitungen zur LCD fest verdrahtet sehen möchte, kann sich das Board hier als gedruckte Platine bauen. Alle Programmbeispiele dieser und der nachfolgenden Lektionen laufen darauf ohne Änderungen.

Top	Home	EEPROM	Dezimalumwandlung	Hardware	Bytezähler	Wortzähler

11.4 Bytezähler

11.4.1 LCD-Routinen als Include-Datei

So langsam wird das Einfügen der immer gleichen LCD-Routinen langweilig. Wir nehmen diese Routinen aus dem Programm heraus, fügen noch die eine oder andere zusätzliche Routine hinzu, stellen die Eigenschaften in einem aussagekräftigen Kopf dar und formulieren so eine Include-Datei. Im eigentlichen Programm fügen wir diese dann mit .INCLUDE "Dateiname.inc" an einer freien Stelle in den Code ein. Der Assembler tut dann so, als ob dort enthaltenen Zeilen

im Quellcode stünden.

Das hier ist die perfekte 4-Bit-Busy-LCD-Include-Datei (zum Quellcode im asm-Format geht es hier), ohne Pfadangabe muss die Include Routine in den gleichen Ordner in dem die aufrufende .asm-Datei liegt).

```
;                                                 rcall Warte50ms ; Karenzzeit 50 ms
; ***********************************************   ; Versetze in 8-Bit-Modus (drei Mal)
; * Include Code zur 4-Bit-Ansteuerung einer LCD*   ldi rmp,0x30 ; 8-Bit-Modus
; * im Busy-Flag-Modus, mit Basisroutinen       *   rcall LcdC8Byte ; Schreibe im 8-Bit-Mode
; * (C)2016 by www.gsc-elektronic.net           *   rcall Warte5ms ; Warte 5 ms
; ***********************************************   ldi rmp,0x30
;                                                   rcall LcdC8Byte
; Der Include-Code enthaelt Basisroutinen fuer      rcall Warte5ms
; die 4-Bit-Ansteuerung einer 4*20-LCD am           ldi rmp,0x30
; ATtiny24 im Busy-Modus (bidirektionaler           rcall LcdC8Byte
; Datenbus).                                        rcall Warte5ms
;                                                   ; Umstellung auf 4-Bit-Modus
; Diese angegebenen Register sind im Haupt-         ldi rmp,0x20 ; Schalte in 4-Bit-Modus um
; programm zu definieren:                           rcall LcdC8Byte
;                                                   rcall Warte5ms
; -------- Verwendete Register ------------------   ; Funktionseinstellungen LCD
; rmp, rmo, rLine, rLese, R0                        ldi rmp,0x28 ; 4-Bit-Modus, 4 Zeilen, 5*7
;                                                   rcall LcdC4Byte ; Byte an Kontrollregister LCD
; Die folgenden Routinen werden zur Verfuegung      ldi rmp,0x0F ; Display ein, Blinken
; gestellt:                                         rcall LcdC4Byte ; Byte an Kontrollregister LCD
;                                                   ldi rmp,0x01 ; Display loeschen
; -------- Routinen -----------------------------   rcall LcdC4Byte ; Byte an Kontrollregister LCD
; Routine  Macht        Aufrufparameter  Register   ldi rmp,0x06 ; Autoindent
; -------  ------------ ---------------- --------   rjmp LcdC4Byte ; Byte an Kontrollregister LCD
; LcdInit  Initiieren   Keine            rmp,rmo  ;
;                                        rLese    ; Ausgabe von Text im Flash auf der LCD
; LcdChars Erzeuge ei-  ZH:ZL Zeichen-   rmp,rmo  ;   Z zeigt auf Text im Flash
;          gene Zeichen tabelle          rLese,R0 LcdText:
; LcdText  Gib den Text ZH:ZL Texttabelle rmp,rmo   rcall LcdLine1 ; auf Zeile 1
;          im Flash in  0D=Naechste Zeile rLine   LcdTextC: ; Setze an aktuelle Position fort
;          der Tabelle  FF=Ignoriere      rLese     clr rLine ; Zeilenzaehler
;          in Zeile 1   FE=Ende Tabelle            LcdText1:
;          aus                                       lpm rmp,Z+ ; lese Zeichen aus Flash
; LcdTextC Gib den Text (wie LcdText)    (dto.)      cpi rmp,0xFE ; Ende Ausgabe?
;          im Flash an                               breq LcdTextRet ; nein
;          der aktuellen                             cpi rmp,0xFF ; Fuellzeichen?
;          Position aus                              breq LcdText1 ; ja, naechstes Zeichen
; LcdSRam  Gib den Text XH:XL: zeigt auf  rmp,rmo    cpi rmp,0x0D ; Zeilenwechsel?
;          im SRAM aus  Position im SRAM  rLese      brne LcdText2 ; Kein Zeilenwechsel
;                       ZH:ZL: Lcd-Position          inc rLine ; Naechste Zeile
;                       rmp: Anzahl Zeichen          mov rmp,rLine ; Setze Zeile
; LcdPos   Setze Ausga- ZH: Zeile 0..3    rmp,rmo    rcall LcdLine ; Stelle Zeile ein
;          beposition   ZL: Spalte 0..19  rLese      rjmp LcdText1 ; weiter mit Zeichen
; LcdLine  Zeile ein-   rmp: Zeile 0..3   rmp,rmo LcdText2: ; Ausgabe Zeichen
;          stellen,                       rLese      rcall LcdD4Byte ; Zeichen ausgeben
;          Spalte=0                                  rjmp LcdText1 ; und weiter
; LcdLineN Zeile ein-   N: Zeile 1..4     rmp,rmo LcdTextRet:
;          stellen,                       rLese      ret ; Fertig
;          Spalte=0                                ;
;                                                  ; Ausgabe von Text im SRAM auf dem LCD
; ---------Ports und Portpins der LCD ----------- ;   ZH = Zeile 0..3; ZL = Spalte 0..19,
; LCD-Kontrollport                                 ;   XH:XL = Adresse im SRAM, rmp: Anzahl
.equ pLcdCO   = PORTB  ; LCD-Kontrollport-Ausgabe LcdSram:
.equ pLcdCR   = DDRB   ; LCD-Kontrollp.-Richtung    mov R0,rmp ; kopieren Anzahl
.equ bLcdCOE  = PORTB2 ; LCD Enable Pin Output      rcall LcdPos ; setze LCD-Position
.equ bLcdCRE  = DDB2   ; LCD Enable Pin Richtung  LcdSram1:
.equ bLcdCORS = PORTB0 ; LCD RS Pin Output          ld rmp,X+ ; lese Ausgabe-Byte
.equ bLcdCRRS = DDB0   ; LCD RS Pin Richtung        rcall LcdD4Byte ; auf LCD ausgeben
.equ bLcdCORW = PORTB1 ; LCD R/W Pin Output         dec R0 ; Zaehler abwaerts
.equ bLcdCRRW = DDB1   ; LCD R/W Pin Richtung       brne LcdSram1 ; noch Zeichen
; LCD-Datenport                                     ret ; Fertig
.equ pLcdDO   = PORTA  ; LCD-Datenport-Ausgabe   ;
.equ pLcdDI   = PINA   ; LCD-Datenport-Eingabe   ; Setzt den Ausgabecursor auf den
.equ pLcdDR   = DDRA   ; LCD-Datenport-Richtung  ; Zeilenanfang der ausgewaehlten Zeile
.equ mLcdDRW  = 0xF0   ; LCD-Datenp. Schreiben   ;   Zeile: rmp 0 .. 3
.equ mLcdDRR  = 0x0F   ; LCD-Datenp. Lesen       LcdLine:
;                                                   cpi rmp,1 ; Zeile 2?
; --------- LCD-Ansteuerung Init ----------        brcs LcdLine1 ; nach Zeile 1
LcdInit:                                            breq LcdLine2 ; nach Zeile 2
  ; Warte 50 ms bis LCD hochgefahren ist             cpi rmp,2 ; Zeile 3?
```

```
  breq LcdLine3 ; nach Zeile 3
  rjmp LcdLine4 ; nach Zeile 4
LcdLine1:
  ldi rmp,0x80 ; Zeile 1
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
LcdLine2:
  ldi rmp,0xC0 ; Zeile 2
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
LcdLine3:
  ldi rmp,0x80+20 ; Zeile 3
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
LcdLine4:
  ldi rmp,0xC0+20 ; Zeile 4
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
;
; Setzt den Ausgabecursor auf die Position
; in ZH:ZL, ZH ist Zeile (0..3), ZL ist Spalte
LcdPos:
  ldi rmp,0x80 ; Setze Zeile 1
  cpi ZH,1 ; Zeile 2?
  brcs LcdPos1 ; Zeile = 1
  ldi rmp,0xC0 ; Setze Zeile 2
  breq LcdPos1 ; Zeile = 2
  ldi rmp,0x80+20 ; Setze Zeile 3
  cpi ZH,2 ; Zeile 3?
  breq LcdPos1 ; Zeile = 3
  ldi rmp,0xC0+20 ; Zeile = 4
LcdPos1:
  add rmp,ZL ; addiere Spalte
  rjmp LcdC4Byte ; Ausgabe Kontrollwort
;
; Eigene Zeichen definieren
LcdChars:
  lpm ; Lese Adresse
  tst R0 ; auf Null pruefen
  breq LcdChars2 ; fertig
  adiw ZL,2 ; ueberlese Fuellbyte
  ldi rLine,8
LcdChars1:
  mov rmp,R0 ; Adresse setzen
  rcall LcdC4Byte ; an LCD
  lpm rmp,Z+ ; lese Daten
  rcall LcdD4Byte ; Ausgabe an LCD
  inc R0 ; Adresse erhoehen
  dec rLine ; Zaehler abwaerts
  brne LcdChars1 ; noch welche
  rjmp LcdChars ; naechstes Zeichen
LcdChars2:
  ret ; fertig
;
; Datenwort-Ausgabe im 4-Bit-Modus
;   Daten in rmp
LcdD4Byte:
  rcall LcdBusy ; warte bis busy = Null
  sbi pLcdCO,bLcdCORS ; setze RS-Bit
  rjmp Lcd4Byte ; gib Byte in rmp aus
;
; Kontrollwort-Ausgabe im 4-Bit-Modus
;   Daten in rmp
LcdC4Byte:
  rcall LcdBusy ; warte bis busy = Null
  cbi pLcdCO,bLcdCORS ; loesche RS-Bit
; Ausgabe Byte im 4-Bit-Modus mit Busy
Lcd4Byte:
  push rmp ; rmp auf Stapel legen
  andi rmp,0xF0 ; unteres Nibble loeschen
  in rmo,pLcdDO ; Lese Data-Input-Port
  andi rmo,0x0F ; oberes Nibble loeschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  pop rmp ; rmp wieder herstellen
  andi rmp,0x0F ; oberes Nibble loeschen
  swap rmp ; Nibble vertauschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  ret ; fertig
;
; Warte bis Busy Null
LcdBusy:
  push rmp ; rette rmp
  in rmp,pLcdDR
  andi rmp,mLcdDRR ; Lesemaske
  out pLcdDR,rmp ; in Richtungsregister
  cbi pLcdCO,bLcdCORS ; loesche RS
  sbi pLcdCO,bLcdCORW ; R/W setzen
LcdBusy1:
  sbi pLcdCO,bLcdCOE ; setze LCD-Enable
  nop
  in rLese,pLcdDI ; lese oberes Nibble
  cbi pLcdCO,bLcdCOE ; loesche LCD-Enable
  andi rLese,0xF0 ; unteres Nibble loeschen
  sbi pLcdCO,bLcdCOE ; setze LCD-Enable
  nop
  in rmp,pLcdDI ; lese unteres Nibble
  cbi pLcdCO,bLcdCOE ; loesche LCD-Enable
  andi rmp,0xF0 ; loesche unteres Nibble
  swap rmp ; oberes/unteres Nibble tauschen
  or rLese,rmp ; oberes und unteres Nibble
  sbrc rLese,7 ; ueberspringe bei Busy=0
  rjmp LcdBusy1 ; wiederhole bis Busy=0
  cbi pLcdCO,bLcdCORW ; R/W loeschen
  in rmp,pLcdDR
  ori rmp,mLcdDRW
  out pLcdDR,rmp ; in Richtungsregister
  pop rmp ; rmp wieder herstellen
  ret ; zurueck
;
; Kontrollwort-Ausgabe im 8-Bit-Modus
;   Datenbyte in rmp
LcdC8Byte:
  cbi pLcdCO,bLcdCORS ; RS-Bit loeschen
  andi rmp,0xF0 ; unteres Nibble loeschen
  in rmo,pLcdDO ; Lese Data-Input-Port
  andi rmo,0x0F ; oberes Nibble loeschen
  or rmp,rmo ; unteres und oberes Nibble
  out pLcdDO,rmp ; an Datenport LCD
  nop ; ein Takt warten
  sbi pLcdCO,bLcdCOE ; LCD-Enable aktivieren
  nop ; ein Takt warten
  cbi pLcdCO,bLcdCOE ; LCD-Enable aus
  ret ; fertig
;
; ------ Warteroutinen ------------------
Warte50ms: ; Warteroutine 50 ms
.equ c50ms = 50000
.equ n50ms = (c50ms-16)/4
;   rcall: + 3
  push ZH ; + 2
  push ZL ; + 2
  ldi ZH,HIGH(n50ms) ; + 1
  ldi ZL,LOW(n50ms) ; + 1
  rjmp Warte ; + 2, gesamt = 11
Warte5ms: ; Warteroutine 5 ms
.equ c5ms = 5000
.equ n5ms = (c5ms-16)/4
  push ZH
  push ZL
  ldi ZH,HIGH(n5ms)
  ldi ZL,LOW(n5ms)
  rjmp Warte
; Warteroutine Z Takte
Warte: ; Warten, Takte = 4*(n-1)+11 = 4*n + 7
  sbiw ZL,1 ; + 2
  brne Warte ; + 1 / 2
  pop ZL ; + 2
  pop ZH ; +2
```

```
  ret ; + 4, Gesamt=4*n+18
;
```

```
; Ende Include
;
```

Mit dieser Includedatei werden die nachfolgenden Programme sehr viel kürzer.

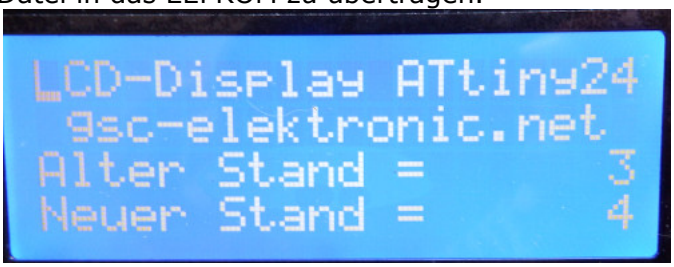
11.4.2 Das Programm

Das Programm für die Lösung der Aufgabe ist nachfolgend dargestellt (zum Quellcode im asm-Format geht es hier, zusätzlich wird die LCD-Include-Datei benötigt).

```
;
; ********************************************
; * EEPROM lesen und beschreiben mit ATtiny24 LCD
; * (C)2016 by http://www.gsc-elektronic.net
;*********************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ------- Programmablauf ----------------
;
; Initiiert die 4*20-LCD am ATtiny24, liest
; einen 8-Bit-Zaehler aus dem EEPROM, er-
; hoeht ihn und schreibt den neuen Stand
; in das EEPROM. Zeigt alten und neuen
; Stand auf der LCD an.
;
; ------- Ports, Portbits ---------------
; (Alle LCD-Ports sind in der Include-Routine)
;
; ------- Register ----------------------
; benutzt: R0 lokal von Zeichendefinitionsroutine
; frei: R1 .. R15
.def rmp = R16 ; Vielzweckregister
.def rmo = R17 ; weiteres Vielzweckregister
.def rLine = R18 ; Zeilenzaehler LCD
.def rLese = R19 ; Leseergebnis vom LCD-Datenport
.def rEep = R20 ; Datenbyte fuer EEPROM-Zaehler
; frei R21 .. R25
; benutzt: XH:XL R27:R26 Zeiger
; frei YH:YL R29:R28
; verwendet: R31:R30, ZH:ZL, fuer Zaehlen und LCD
;
; ------- Konstanten --------------------
.equ Takt = 1000000 ; Default-Taktfrequenz
.equ EepAdresse = 0 ; Lesen und Schreiben EEPROM
;
; ------- SRAM --------------------------
.DSEG ; in Datensegment
.ORG 0x0060 ; an den Beginn
Zahl: ; Umwandlung Byte in ASCII-Ziffernfolge
.BYTE 3 ; braucht drei Zeichen
;
; -------- EEPROM-Startinhalt -----------
.ESEG ; EEPROM-Segment
.ORG EepAdresse
.db 0 ; Init EEPROM-Zaehler
;
; -------- Programmstart, Init ----------
.CSEG ; Code Segment
.ORG 0 ; Start bei 0
  ; Stapel Init fuer Unterprogramme
  ldi rmp,LOW(RAMEND) ; Init Stapel
  out SPL,rmp ; in Stapelzeiger
  ; Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
    ; Schreiben
  out pLcdDR,rmp ; auf Ausgabe
  ; LCD initiieren
  rcall LcdInit
  ; Text auf LCD ausgeben
  ldi ZH,HIGH(2*Texttabelle)
  ldi ZL,LOW(2*Texttabelle)
  rcall LcdText ; Text ab ZH:ZL ausgeben
  ; Lese Byte von EEPROM und gib in Zeile 3 aus
  rcall EepRead
  inc rEep ; Erhoehe EEPROM-Byte
  rcall EepWrite ; in EEPROM schreiben
  ; Sleep enable
  ldi rmp,1<<SE
  out MCUCR,rmp
Schleife:
  sleep ; schlafen
  rjmp Schleife
;
; Ausgabetext auf LCD
Texttabelle:
.db "LCD-Display ATtiny24",0x0D,0xFF ; Zeile 1
.db " gsc-elektronic.net ",0x0D,0xFF ; Zeile 2
.db "Alter Stand =       ",0x0D,0xFF ; Zeile 3
.db "Neuer Stand =      ",0xFE ; Zeile 4
;
; EEPROM-Byte lesen
EepRead:
  ; Warten bis EEPROM bereit ist
  sbic EECR, EEPE ; EEPROM-Enable-Bit abfragen
  rjmp EepRead ; noch nicht fertig
  ; EEPROM-Lese-Adresse setzen
  ldi rmp,HIGH(EepAdresse) ; MSB in Adressreg.
  out EEARH,rmp ; fuer ATtiny24/44 unnoetig, da
    ; weniger als 257 Byte EEPROM
  ldi rmp,LOW(EepAdresse) ; LSB
  out EEARL,rmp ; in LSB-Adressregister
  ; EERE-Bit im EEPROM-Kontrollregister setzen
  sbi EECR,EERE
  ; Lese Byte aus Datenregister
  in rEep,EEDR
  ; Gelesene Zahl in ASCII umwandeln
  mov R0,rEep ; Zahl in R0 kopieren
  ldi XH,HIGH(Zahl) ; SRAM-Puffer-Zeiger fuer
    ; Ergebnis
  ldi XL,LOW(Zahl) ; dto. LSB
  rcall Dezimal ; Wandle in Dezimalzahl um
  ldi XH,HIGH(Zahl) ; SRAM-Puffer-Zeiger fuer
    ; Ausgabe
  ldi XL,LOW(Zahl) ; dto., LSB
  ldi ZH,2 ; in Zeile 3
  ldi ZL,17 ; in Spalte 17
  ldi rmp,3 ; drei Zeichen
  rcall LcdSram ; Aufruf Include-Routine
  ret
;
; In EEPROM schreiben
EepWrite:
  ; Warten bis EEPROM fertig ist
  sbic EECR, EEPE ; Enable-Bit im Kontrollreg.
  rjmp EepWrite ; noch nicht fertig
  ; EEPROM-Programmiermodus setzen
  ldi rmp,(0<<EEPM1)|(0<<EEPM0) ; loeschen und
    ; schreiben
  out EECR,rmp ; in EEPROM-Kontrollregister
```

```
  ; Adresse in Adressenregister
  ldi rmp,HIGH(EepAdresse) ; MSB
  out EEARH,rmp ; bei ATtiny24/44 nicht noetig
  ldi rmp,LOW(EepAdresse) ; LSB
  out EEARL,rmp
  ; Zu schreibender Inhalt in Datenregister
  out EEDR,rEep ; schreiben in Datenregister
  ; Memory Program enable EEMPE setzen
  sbi EECR,EEMPE ; Schreiben ermoeglichen
  ; EEPROM schreiben mit EEPE = Eins
  sbi EECR,EEPE ; schreiben starten, 3,4 ms Dauer
  ; neuen Inhalt auf LCD ausgeben
  mov R0,rEep ; Zahl in R0 kopieren
  ldi XH,HIGH(Zahl) ; SRAM-Puffer-Zeiger fuer
    ; Ergebnis
  ldi XL,LOW(Zahl) ; dto. LSB
  rcall Dezimal ; Wandle in Dezimalzahl um
  ldi XH,HIGH(Zahl) ; SRAM-Puffer-Zeiger fuer
    ; Ausgabe
  ldi XL,LOW(Zahl) ; dto., LSB
  ldi ZH,3 ; in Zeile 4
  ldi ZL,17 ; in Spalte 17
  ldi rmp,3 ; drei Zeichen
  rcall LcdSram ; Aufruf Include-Routine
  ret
;
; Wandle R0 in eine Dezimalzahl um
;   XH:XL = Position im SRAM
Dezimal:
  set ; Setze T-Flagge fuer fuehrende Nullen
  ldi rmp,100 ; dezimal 100
  clr R1 ; R1 ist Ergebnis
Dezimal100:
  cp R0,rmp ; Kleiner als 100?
  brcs Dezimal2 ; ja
  sub R0,rmp ; Abziehen 100
  inc R1 ; Ergebnis um Eins erhoehen
  rjmp Dezimal100 ; weiter abziehen
Dezimal2: ; fuehrende Nullen unterdruecken
  tst R1 ; ist Ergebnis Null?
  brne Dezimal3 ; nein
  ldi rmp,' ' ; Leerzeichen
  rjmp Dezimal4 ; weiter
Dezimal3: ; keine fuehrende Null
  clt ; Flagge loeschen
  ldi rmp,0x30 ; ASCII-Null
  add rmp,R1 ; Ergebnis addieren
Dezimal4: ; Ergebnis speichern
  st X+,rmp ; in SRAM speichern
  ; Zehner ermitteln
  ldi rmp,10 ; dezimal 10
  clr R1 ; Ergebnis loeschen
Dezimal10: ; Zehnerschleife
  cp R0,rmp ; Vergleiche mit 10
  brcs Dezimal5 ; schon fertig
  sub R0,rmp ; Abziehen
  inc R1
  rjmp Dezimal10 ; Zehnerschleife
Dezimal5: ; Zehner sind ermittelt
  brtc Dezimal6 ; fuehrende Nullflagge ist aus
  tst R1 ; fuehrende Null
  brne Dezimal6 ; keine Null
  ldi rmp,' ' ; Leerzeichen
  rjmp Dezimal7 ; direkt ausgeben
Dezimal6:
  ldi rmp,'0' ; ASCII-Null
  add rmp,R1 ; Ergebnis addieren
Dezimal7:
  st X+,rmp ; Zehner speichern
  ; Einer sind uebrig
  ldi rmp,'0' ; ASCII-Null
  add rmp,R0 ; Rest addieren
  st X+,rmp ; in SRAM speichern
  ret
;
; Lcd4Busy-Routinen hinzufuegen
.include "Lcd4Busy.inc"
;
; Ende Quellcode
;
```

Beim Programmieren nicht vergessen, nach dem Brennen des Programmes die erzeugte .eep-Datei in das EEPROM zu übertragen.



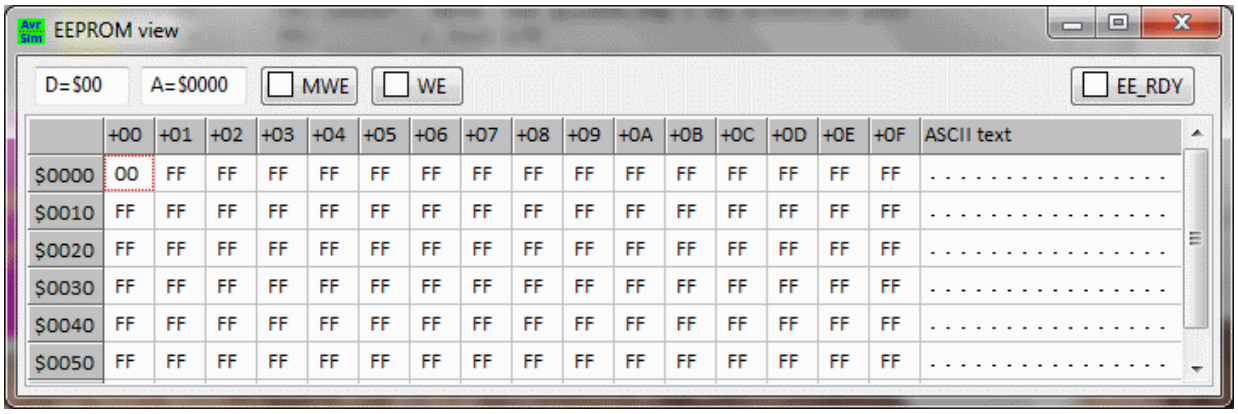
Das hier ist das perfekte Ergebnis. Sieht professionell aus, oder?

Top	Home	EEPROM	Dezimalumwandlung	Hardware	Bytezähler	Wortzähler

11.4.3 Die Simulation

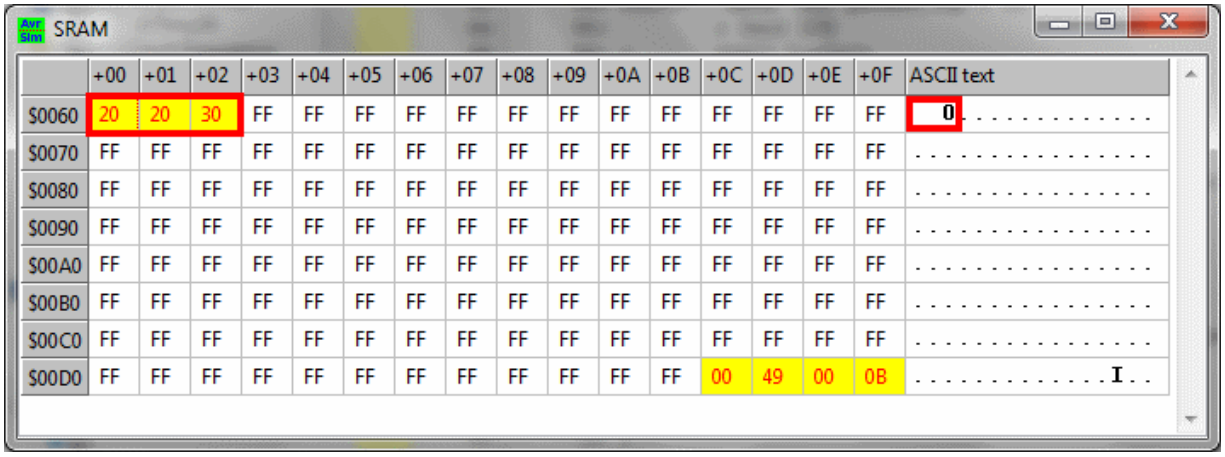
Mit einer Simulation lässt sich der korrekte Ablauf der Funktionen verifizieren. Im Folgenden wird avr_sim verwendet um den Quellcode zu erproben.

Zuvor werden im Quellcode alle Aufrufe der LCD-Routinen (LcdInit, LcdText, LcdSram) durch auskommentieren der Zeilen mit ";" unwirksam gemacht. Zum Assemblieren des Quellcodes muss sich dennoch die LCD-Include-Datei im gleichen Pfad befinden. Dann kann assembliert werden.



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0000	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .

Das ist der EEPROM-Inhalt wenn die Simulation beginnt. Der Zähler bei Adresse 0x0000 ist auf Null gesetzt, der Rest des EEPROMs ist gelöscht (0xFF).



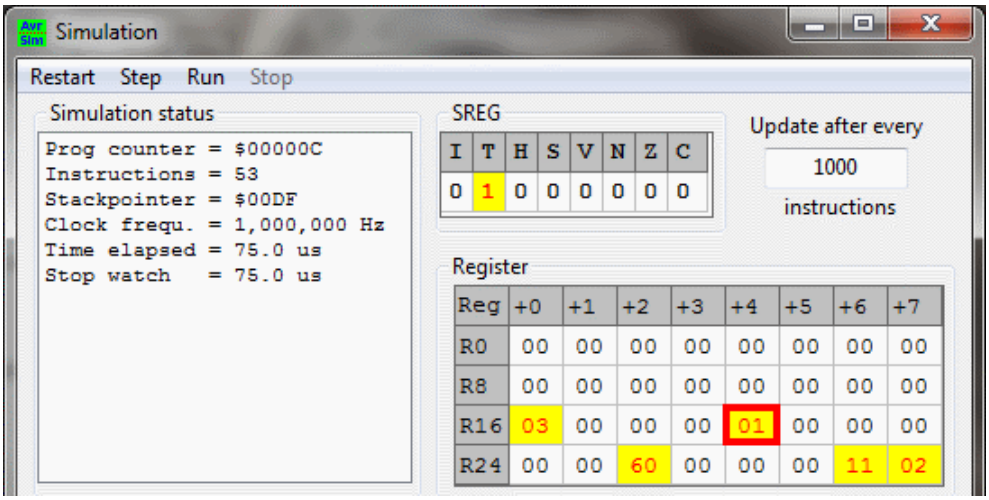
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	20	20	30	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	0 . . . . .
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	49	00	0B	. . . . . I . .

Der Zählerinhalt des EEPROMs bei Adresse 0x0000 wurde gelesen. Hier war der Inhalt 0x00. Das wird jetzt in eine Dezimalzahl verwandelt und dessen ASCII-Kodierung in das SRAM geschrieben.

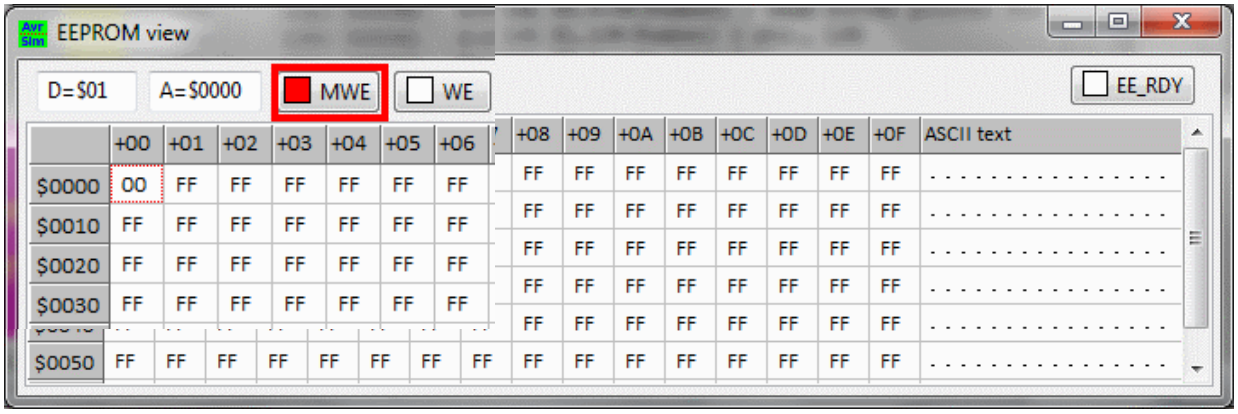
Die ersten beiden Zeichen sind Leerzeichen (Platzhalter fuer Hunderter und Zehner), dann kommt die ASCII-Null. Diese drei Bytes werden in die LCD geschrieben (was hier nicht simuliert wird).

Der Zähler rEep in R20 wurde hier um Eins erhöht.

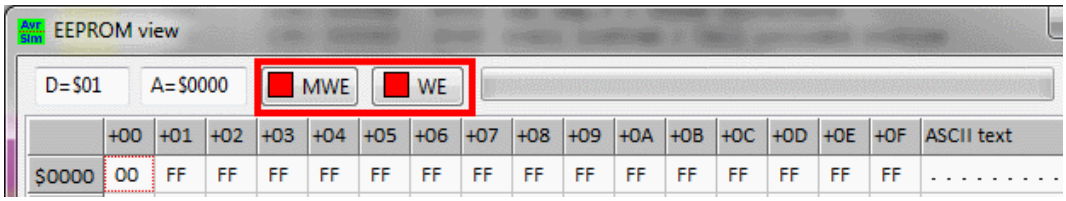
Nun muss der erhöhte Zähler in das EEPROM zurückgeschrieben werden. Um das zu tun, werden die EEPROM-Portregister für die Adresse und das Datenbyte beschrieben (hier markiert).



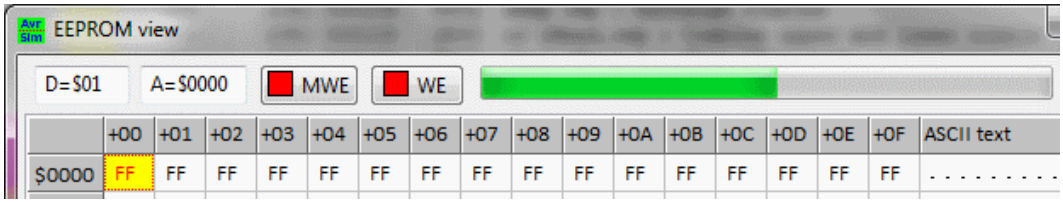
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	03	00	00	00	01	00	00	00
R24	00	00	60	00	00	00	11	02



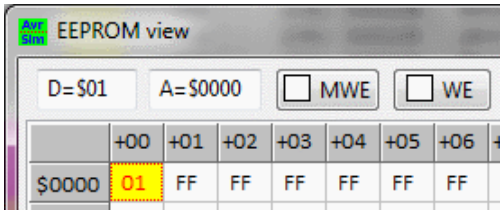
Zuerst wird dann das Bit Master Write Enable gesetzt.



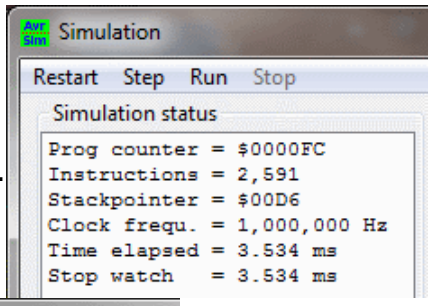
Innerhalb von vier Taktzyklen des Prozessors muss dann das Bit Write Enable gesetzt werden. Die Programmierung startet nun, die Fortschrittsanzeige wird gezeigt.



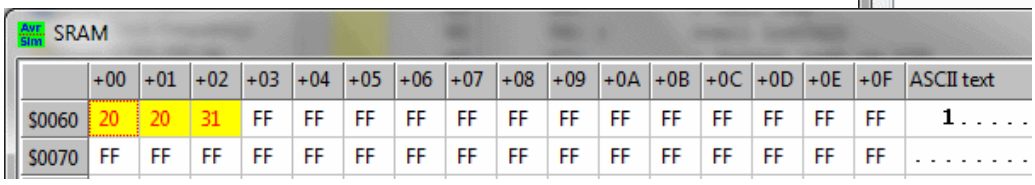
Die erste Stufe des Schreibprozesses ist das Löschen der Zelle, auf die das Adressregister zeigt. Ungefähr zur halben Zeit des Schreibablaufs ist der Inhalt gelöscht (alle Bits auf Eins).



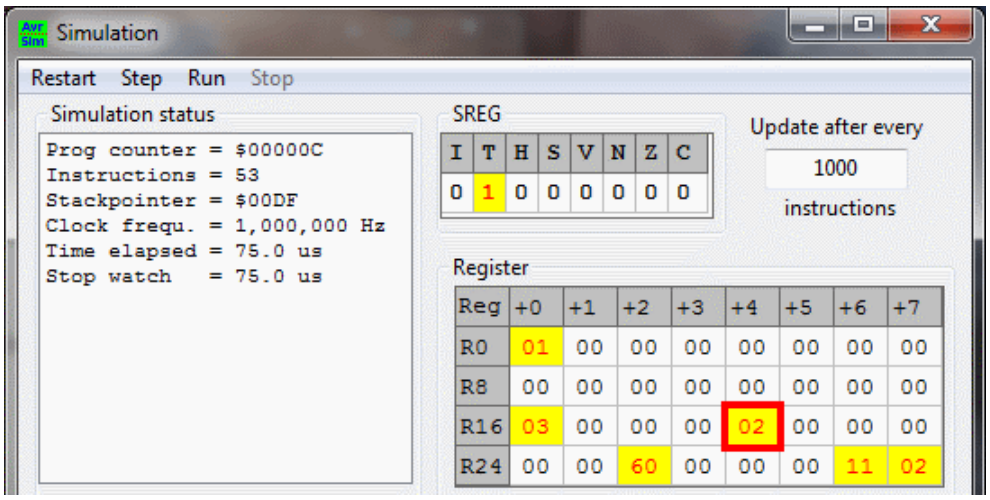
Nachdem der Schreibprozess abgeschlossen ist, hat sich der Inhalt der Zelle an Adresse 0x0000 geändert.



Die Zeit, die zum Schreiben benötigt wurde, beträgt ca. 3,5 ms.



Nach dem Neustart der Simulation, ohne Neubeschreibung des EEPROMs, ist der Zähler Eins, wird in ASCII umgewandelt und in das SRAM geschrieben.



Natürlich führt die zweite Erhöhung zu "2" in rEep in R20. Und so weiter, und so weiter, ...

11.5 Wortzähler

11.5.1 Aufgabe

Das vorherige Programm ist so umzubauen, dass nach Erreichen eines definierten Zählerstands der Chip seine weitere Mitarbeit einstellt (geplante Obsoleszenz). Damit auch höhere Grenzen eingestellt werden können, ist dafür ein 16-Bit-Zähler zu verwenden. Bei Erreichen der Grenze ist in Zeile 4 der LCD eine entsprechende Meldung auszugeben.

11.5.2 Das Programm dazu

Dies hier ist das Programm dazu (zum Quellcode im asm-Format geht es hier, zusätzlich wird die LCD-Include-Datei benötigt).

```
;                                          .def rmo = R17 ; weiteres Vielzweckregister
; ****************************************  .def rLine = R18 ; Zeilenzaehler LCD
; * EEPROM-Wort lesen und schreiben mit LCD  *  .def rLese = R19 ; Leseergebnis vom LCD-Datenport
; * (C)2016 by http://www.gsc-elektronic.net *  .def rEepH = R20 ; MSB Datenbyte fuer EEPROM-
; ****************************************      ; Zaehler
;                                          .def rEepL = R21 ; dto., LSB
.NOLIST                                    ; frei R22 .. R25
.INCLUDE "tn24def.inc"                     ; benutzt: XH:XL R27:R26 Zeiger
.LIST                                      ; frei YH:YL R29:R28
;                                          ; verwendet: R31:R30, ZH:ZL, fuer Zaehlen, LCD
; ------- Programmablauf -----------------  ;
;                                          ; ------- Konstanten --------------------
; Initiiert die vierzeilige LCD am ATtiny24  .equ Takt = 1000000 ; Default-Taktfrequenz
; im 4-Bit-Busy-Modus. Liest einen 16-Bit-   .equ EepAdresse = 0 ; Lese- und Schreibadresse
; Zaehler aus dem EEPROM, erhoeht ihn,           ; EEPROM
; schreibt ihn wieder zurueck und stellt     .equ Obsoleszenz = 3 ; Obsoleszenzkriterium
; beide Zaehlerstaende auf der LCD dar.          ; 1..65535
; Erreicht der Zaehler eine voreingestellte  ;
; Zahl, erscheint eine Ende-Meldung.         ; ------- SRAM --------------------------
;                                          .DSEG ; in Datensegment
; ------- Ports, Portbits ---------------  .ORG 0x0060 ; an den Beginn
; (Alle LCD-Ports sind in der Include-Routine Zahl: ; Umwandlung Byte in ASCII-Ziffernfolge
; definiert)                                 .BYTE 5 ; braucht fuenf Zeichen
;                                          ;
; ------- Register ----------------------  ; -------- EEPROM-Startinhalt -----------
; benutzt: R0 bis R3 lokal von der           .ESEG ; EEPROM-Segment
    ; Zeichendefinition und Dezimal          .ORG EepAdresse
; frei: R4 .. R15                            .db LOW(32767),HIGH(32767) ; Init EEPROM-Zaehler
.def rmp = R16 ; Vielzweckregister         ;
```

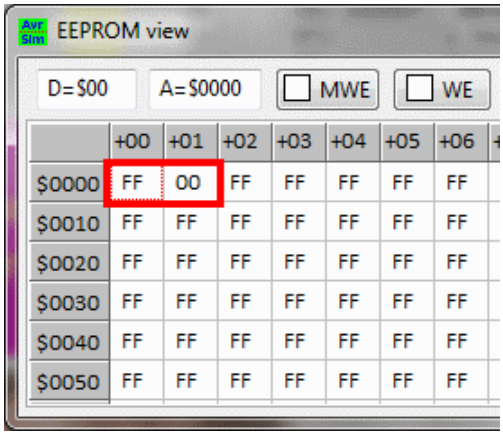
```
; -------- Programmstart, Init ----------
.CSEG ; Code Segment
.ORG 0 ; Start bei 0
  ; Stapel Init fuer Unterprogramme
  ldi rmp,LOW(RAMEND) ; Init Stapel
  out SPL,rmp ; in Stapelzeiger
  ; Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
Schreiben
  out pLcdDR,rmp ; auf Ausgabe
  ; LCD initiieren
  rcall LcdInit
  ; Text auf LCD ausgeben
  ldi ZH,HIGH(2*Texttabelle)
  ldi ZL,LOW(2*Texttabelle)
  rcall LcdText ; Text ab ZH:ZL ausgeben
  ; Lese Wort von EEPROM und gib in Zeile 3 aus
  rcall EepReadW
  inc rEepL ; Erhoehe EEPROM-Byte
  brne MsbOk ; Kein Ueberlauf
  inc rEepH
MsbOk:
  ; Obsolenz pruefen
  rcall Obsolenz ; Ende-Meldung ausgeben
  brcc Obsolet ; Geraet ist obsolet
  rcall EepWriteW ; in EEPROM schreiben und in
    ; Zeile 4
Obsolet:
  ; Sleep enable
  ldi rmp,1<<SE
  out MCUCR,rmp
Schleife:
  sleep ; schlafen
  rjmp Schleife
;
; Ausgabetext auf LCD
Texttabelle:
.db "LCD-Display ATtiny24",0x0D,0xFF ; Zeile 1
.db " gsc-elektronic.net ",0x0D,0xFF ; Zeile 2
.db "Alter Stand =       ",0x0D,0xFF ; Zeile 3
.db "Neuer Stand =      ",0xFE ; Zeile 4
;
; Obsolenz prufen
Obsolenz:
  cpi rEepL,LOW(Obsoleszenz) ; vergleiche mit
    ; Konstante
  ldi rmp,HIGH(Obsoleszenz)
  cpc rEepH,rmp ; mit Uebertrag
  brcs ObsolenzRet
  ; Ende der Mitarbeit erreicht
  rcall LcdLine4 ; in Zeile 4
  ldi ZH,HIGH(2*ObsoletText) ; Text
  ldi ZL,LOW(2*ObsoletText)
  rcall LcdTextC
  clc ; Carry-Flagge loesen
ObsolenzRet:
  ret
; Text der Meldung
ObsoletText:
.db "Hoechstzahl erreicht",0xFE,0xFE
;
; EEPROM-Wort lesen
EepReadW:
  ; Warten bis EEPROM bereit ist
  sbic EECR, EEPE ; EEPROM-Enable-Bit abfragen
  rjmp EepReadW ; noch nicht fertig
  ; EEPROM-Lese-Adresse setzen
  ldi rmp,HIGH(EepAdresse) ; MSB in Adressreg.
  out EEARH,rmp ; fuer ATtiny24/44 unnoetig, da
    ; weniger als 257 Byte EEPROM
  ldi rmp,LOW(EepAdresse) ; LSB
  out EEARL,rmp ; in LSB-Adressregister
  ; EERE-Bit im EEPROM-Kontrollregister setzen
  sbi EECR,EERE
  ; Lese Byte aus Datenregister
  in rEepL,EEDR ; in LSB
  cbi EECR,EERE ; lesen aus
  ldi rmp,LOW(EepAdresse+1) ; Adresse naechstes
    ; Byte
  out EEARL,rmp ; in Adressregister
  sbi EECR,EERE ; lesen einschalten
  in rEepH,EEDR ; MSB aus Datenregister lesen
  ; Gelesene Zahl in ASCII umwandeln
  mov R0,rEepL ; Zahl in R1:R0 kopieren
  mov R1,rEepH
  ldi XH,HIGH(Zahl) ; SRAM-Puffer-Zeiger fuer
    ; Ergebnis
  ldi XL,LOW(Zahl) ; dto. LSB
  rcall DezimalW ; Wandle Datenwort in
    ; Dezimalzahl um
  ldi XH,HIGH(Zahl) ; SRAM-Puffer-Zeiger fuer
    ; Ausgabe
  ldi XL,LOW(Zahl) ; dto., LSB
  ldi ZH,2 ; in Zeile 3
  ldi ZL,15 ; in Spalte 17
  ldi rmp,5 ; drei Zeichen
  rcall LcdSram ; Aufruf Include-Routine
  ret
;
; In EEPROM schreiben
EepWriteW:
  ; Warten bis EEPROM fertig ist
  sbic EECR,EEPE ; Enable-Bit im Kontrollregister
  rjmp EepWriteW ; noch nicht fertig
  ; EEPROM-Programmiermodus setzen
  ldi rmp,(0<<EEPM1)|(0<<EEPM0) ; loeschen und
    ; schreiben
  out EECR,rmp ; in EEPROM-Kontrollregister
  ; Adresse in Adressenregister
  ldi rmp,HIGH(EepAdresse) ; MSB
  out EEARH,rmp ; bei ATtiny24/44 nicht noetig
  ldi rmp,LOW(EepAdresse) ; LSB
  out EEARL,rmp
  ; Zu schreibender Inhalt in Datenregister
  out EEDR,rEepL ; schreiben in Datenregister
  ; Memory Program enable EEMPE setzen
  sbi EECR,EEMPE ; Schreiben ermoeglichen
  ; EEPROM schreiben mit EEPE = Eins
  sbi EECR,EEPE ; schreiben starten, 3,4 ms Dauer
EepWrite1:
  sbic EECR,EEPE ; warten bis fertig
  rjmp EepWrite1 ; noch nicht fertig
  ldi rmp,LOW(EepAdresse+1) ; MSB
  out EEARL,rmp
  ; Zu schreibender Inhalt in Datenregister
  out EEDR,rEepH ; schreiben in Datenregister
  ; Memory Program enable EEMPE setzen
  sbi EECR,EEMPE ; Schreiben ermoeglichen
  ; EEPROM schreiben mit EEPE = Eins
  sbi EECR,EEPE ; schreiben starten, 3,4 ms Dauer
  ; neuen Inhalt auf LCD ausgeben
  mov R0,rEepL ; Zahl in R1:R0 kopieren
  mov R1,rEepH
  ldi XH,HIGH(Zahl) ; SRAM-Puffer-Zeiger fuer
   ; Ergebnis
  ldi XL,LOW(Zahl) ; dto. LSB
  rcall DezimalW ; Wandle in Dezimalzahl um
  ldi XH,HIGH(Zahl) ; SRAM-Puffer-Zeiger fuer
    ; Ausgabe
  ldi XL,LOW(Zahl) ; dto., LSB
  ldi ZH,3 ; in Zeile 4
  ldi ZL,15 ; in Spalte 17
  ldi rmp,5 ; drei Zeichen
  rcall LcdSram ; Aufruf Include-Routine
  ret
;
; Wandle R1:R0 in eine Dezimalzahl um
;   XH:XL = Position im SRAM
;   verwendet R3:R2, ZH:ZL
DezimalW:
```

```
  set ; Setze T-Flagge fuer fuehrende Nullen
  ldi ZH,HIGH(2*DezTab) ; Z ist Zeiger auf
  ldi ZL,LOW(2*DezTab) ; Dezimaltabelle
DezimalW1:
  lpm R2,Z+ ; Lese LSB aus Dezimaltabelle
  tst R2 ; ist LSB Null?
  breq DezimalWEiner ; ja, beendet
  lpm R3,Z+ ; Lese MSB aus Dezimaltabelle
  clr rmp ; rmp ist Zaehler
DezimalW2:
  sub R0,R2 ; LSB subtrahieren
  sbc R1,R3 ; MSB subtrahieren
  brcs DezimalW3 ; zu viel, Uebertrag
  inc rmp ; Ergebnis erhoehen
  rjmp DezimalW2 ; weiter subtrahieren
DezimalW3:
  add R0,R2 ; Subtrahieren rueckgaengig
  adc R1,R3
  brtc DezimalW6 ; keine fuehrenden Nullen mehr
    ; unterdruecken
  tst rmp ; noch eine Null?
  brne DezimalW5 ; nein
  ldi rmp,' ' ; Leerzeichen ausgeben
  rjmp DezimalW7 ; direkt ausgeben
DezimalW5:
  clt ; T-Flagge ausschalten
DezimalW6:
  subi rmp,-'0' ; ASCII-Null addieren
DezimalW7:
  st X+,rmp ; in SRAM
  rjmp DezimalW1 ; naechstniedrige Dezimalstelle
DezimalWEiner:
  ldi rmp,'0' ; ASCII-Null
  add rmp,R0 ; Einer addieren
  st X+,rmp ; in SRAM speichern
  ret
;
; Dezimaltabelle
DezTab:
.dw 10000 ; Zehntausender
.dw 1000 ; Tausender
.dw 100 ; Hunderter
.dw 10  ; Zehner
.dw 0   ; Tabellenende
;
; Lcd4Busy-Routinen hinzufuegen
.include "Lcd4Busy.inc"
;
; Ende Quellcode
;
```

Neu ist die Instruktion *CPC* Register,Register, die das zweite Register plus das Übertragsbit temporär vom ersten Register abzieht und die Flaggen setzt.

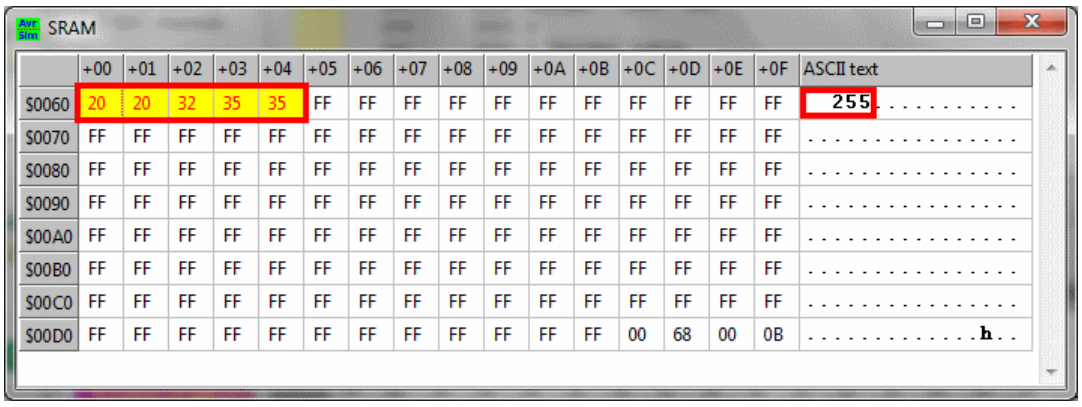
11.5.3 Simulation

Die Simulation wird wieder mit avr_sim vorgenommen. Die Aufrufe der LCD-Routinen werden vor der Simulation wieder auskommentiert.



	+00	+01	+02	+03	+04	+05	+06
$0000	FF	00	FF	FF	FF	FF	FF
$0010	FF	FF	FF	FF	FF	FF	FF
$0020	FF	FF	FF	FF	FF	FF	FF
$0030	FF	FF	FF	FF	FF	FF	FF
$0040	FF	FF	FF	FF	FF	FF	FF
$0050	FF	FF	FF	FF	FF	FF	FF

Das EEPROM hat den 16-Bit-Zähler in 0x0000 und 0x0001 auf 0x00FF gesetzt (siehe Definition im .ESEG-Segment).



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	20	20	32	35	35	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	255
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	68	00	0B	.....h..

Die Dezimalwandlung im SRAM umfasst nun fünf Bytes weil Zahlen bis 65.535 auftreten können.

Register

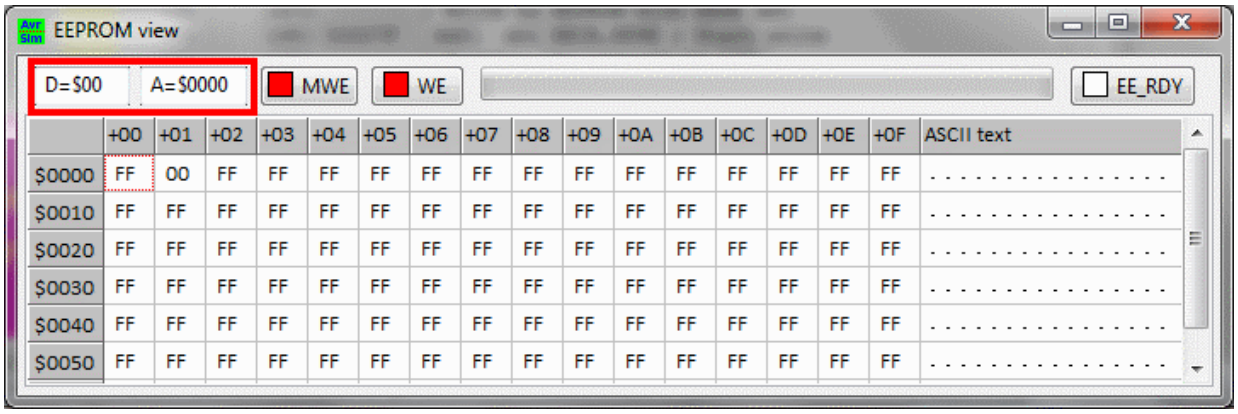
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	05	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	05	00	00	00	00	FF	00	00
R24	00	00	60	00	00	00	0F	02

Die beiden Zählerbytes wurden hier aus dem EEPROM gelesen und in die Register R20:R21 kopiert, mit dem höherwertigen Byte (MSB, most significant byte) in R20.

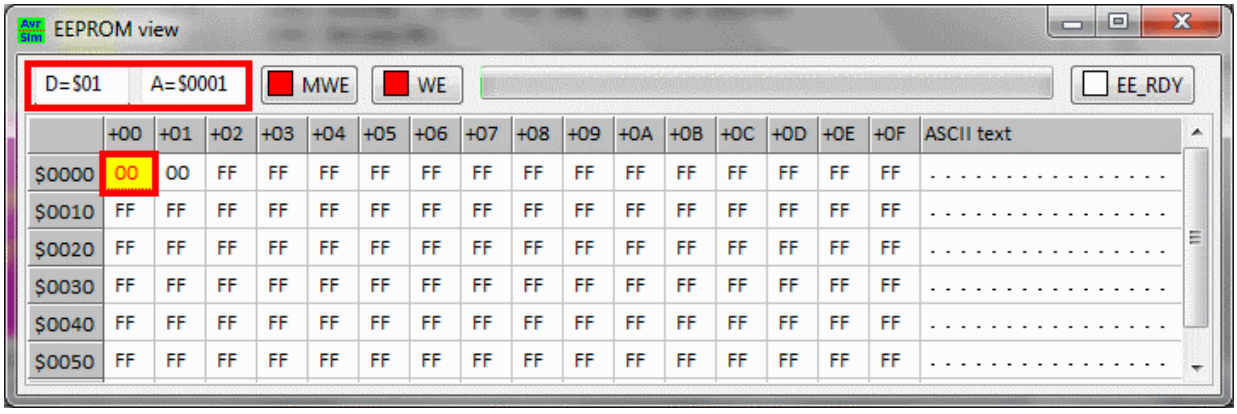
Register

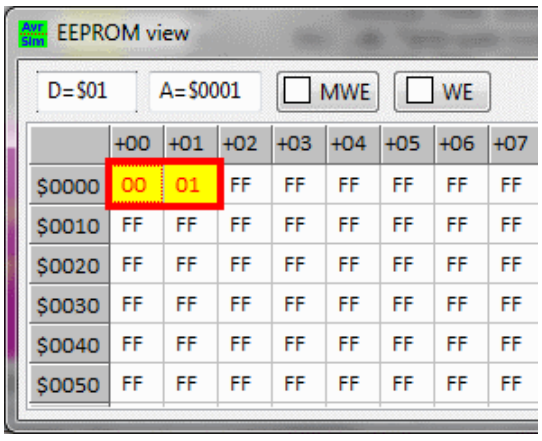
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	05	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	FF	00	00	00	01	00	00	00
R24	00	00	60	00	00	00	0F	02

Der Wert des Zählers wurde erhöht und in Dezimalformat in das SRAM geschrieben. Als Erstes wird das LSB des erhöhten Zählers in das EEPROM geschrieben, an die Adresse 0x0000.



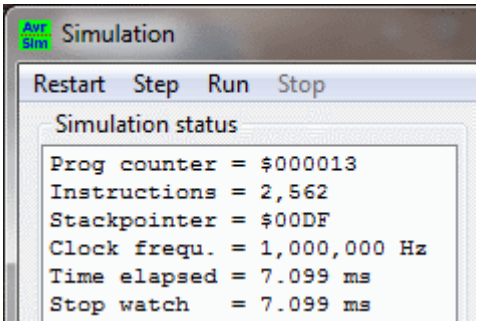
Nach erfolgter Beendigung des Schreibprozesses wird das MSB in das EEPROM geschrieben.

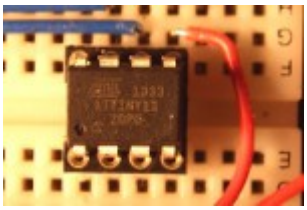




Beide Bytes sind jetzt im EEPROM.

Rund sieben Millisekunden sind vergangen bis beide Bytes in das EEPROM geschrieben sind.





Lektion 12: IR-Empfänger und Sender

Hier mal was richtig Praktisches: mit einem IR-Empfänger Infrarot-Fernbedienungssignale erkunden und mit einem Infrarotsender auch noch selber produzieren. Um die Zahl der Programmvarianten gering zu halten kommt hier in großem Umfang das bedingte Assemblieren zum Einsatz.

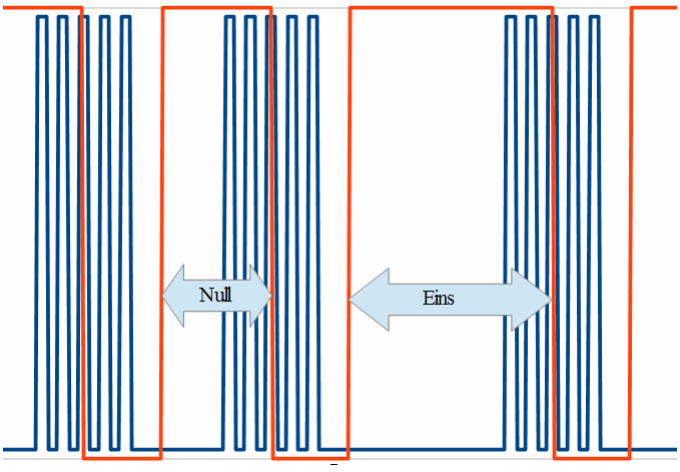
12.0 Übersicht

12.1 Einführung in Infrarotsignale

Die kleinen formschönen Schächtelchen, die auf dem Wohnzimmertisch herumliegen und auf unseren Tastendruck harren, sind große Unbekannte. Kaum jemand weiß oder versteht genau, wie die es fertigkriegen, den Fernseher oder den CD-Spieler vom Sofa aus zu bedienen. Früher, als es diese Schächtelchen noch nicht gab, musste man die müden Knochen noch erheben und am Gerät selbst Knöpfe drücken. Die Fernbedienung hat es möglich gemacht liegen zu bleiben und von dort aus Sekundärknöpfchen zu drücken.

Infrarot sagt schon, dass die mit Wärmestrahlung arbeiten und wir die Signale gar nicht sehen können. Da die verwendeten Leuchtdioden auch nur maximal 250 mW versenden, merken wir es nicht mal auf der Hautoberfläche. Aber selbst wenn wir die sehen könnten, würden wir nur wenig sehen, denn das spielt sich in wenigen Millisekunden ab, dann ist der Fernseher auf Kanal 15 verstellt.

Damit sich das Infrarotsignal von anderen Infrarotstrahlern, die auch noch im Wohnzimmer herumstrahlen (wie z. B. ein heißer Tee), gehörig unterscheidet, wird es gehörig zerhackt. Das kann der Tee nicht und hilft, den Tee von der Fernbedienung zu unterscheiden. Das Zerhacken geht mit Fledermausfrequenzen um die 35 bis 56 kHz. Im Empfänger wird diese Frequenz aus dem analogen Messsignal herausgefiltert, so dass die Fernsteuerung auch in schwüler Sommerhitze nur den Kanal 15 erkennt. So ein Ausschnitt aus einem IR-Signal sieht folglich so aus.



Während die IR-LED aus ist, wird vom Empfänger (rot) kein Signal erkannt, er geht auf inaktiv (high). Die LED wird mit einer Frequenz von z. B. 40 kHz ein- und ausgeschaltet (12,5 µs an, 12,5 µs aus). Nach einer Einschwingzeit folgt das Empfängersignal, es wird aktiv (low). Bei Datensignalen steckt die Information in dem Zeitraum,

über den die LED aus bleibt. Die Anzahl dieser Wartezeiten zwischen zwei aktiven LED-Signalen liegt bei 15 bis 30 für eine binäre Null und bei 50 bis 130 für eine binäre Eins.

Sind vom IR-Empfänger genügend Signale erkannt, schaltet er ein, in Sendepausen wieder aus. Die Dauer der erkannten Signalfolge, die Pausendauer und die Signaldauer, ist entscheidend für die übertragenen Signale.

Um Batteriestrom zu sparen, arbeiten die meisten Hersteller von Fernsteuerungen so: die Pausendauer, nicht die Signaldauer bestimmt über den Inhalt. Die Signaldauer ist kurz, die Pausendauer ist entweder genauso kurz (Null) oder mehr als doppelt so lang (Eins).

Warum hier immer "z. B." steht liegt daran, dass jeder Hersteller sein eigenes Süppchen kocht und es keinerlei Normierung gibt. Nicht nur sind als Frequenzen 30, 33, 36, 36,7, 38, 40 und 56 kHz in Gebrauch, auch die Zusammensetzung und Dauer der Zeichenfolgen sind höchst individuell. Das Ziel der Hersteller ist eher darauf gerichtet, dass mit der Vielfalt jede Verwechslung vermieden wird als danach, ähnliche Geräte mit einer ähnlichen z. B. 64-bittigen Signalfolge zu steuern. Entsprechend wächst die Anzahl der Schächtelchen auf dem Wohnzimmertisch mit jeder Geräteanschaffung ins Uferlose.

Wir werden uns mit der Abwesenheit von Normen in dieser Lektion noch praktisch befassen.

12.2 Einführung in die bedingte Programmierung

Mit der Version 2 des ATMEL-Assemblers wurde die Möglichkeit eröffnet, mittels Direktiven dem Assembler mitzuteilen, er möge bestimmte Teile des Quellcodes in Abhängigkeit von Bedingungen assemblieren oder nicht. Dies macht dann Sinn, wenn die Grundstruktur eines Programmes die Gleiche bleibt und sich nur begrenzte ausgewählte Teile verändern.

12.2.1 Setzen von Bedingungen

Mit der folgenden Formulierung wird der nachfolgende Programmbestandteil nur dann übersetzt, wenn der Schalter Eins ist:

```
.equ Schalter = 1 ; definiere den Schalter
; [...]
.if Schalter == 1
        ; [assembliere diesen Teil]
        .endif
; [...]
```

Das doppelte Gleichheitszeichen in der .IF-Bedingung kennzeichnet eine logische Entscheidung, deren Ergebnis nur wahr (Nicht Null) oder falsch (Null) sein kann. Wieder ist es wichtig zu verstehen, dass die Entscheidung nur beim Assemblieren gefällt wird und mit der Prozessortätigkeit nur so viel zu tun hat als sich der erzeugte Maschinencode ändert. Entscheidungen während des Programmlaufs werden mit bedingten Sprungbefehlen wie BRNE, BREQ, BRCC oder BRNC vorgenommen, nicht mit .IF.

Einem .IF MUSS immer auch ein .ENDIF folgen, am Ende des Quellcodes noch offene Bedingungen führen zu einem Fehler.

Falls man das Gegenteil abfragen möchte, kommen zwei Methoden in Frage:

```
.if Schalter != 1
        ; [assembliere diesen Teil, wenn Schalter aus]
        .endif
; [...]
```

oder selbiges auch

```
.if Schalter == 0
        ; [assembliere diesen Teil]
        .endif
```

```
; [...]
```

Die Bedingung != steht für Ungleich.

12.2.2 Wenn-Dann-Sonst-Alternativen

Falls man mit dem Schalter zwischen zwei Alternativen auswählen möchte kann das mit der .ELSE-Direktive tun. Die folgende Formulierung reagiert auf unterschiedliche Polaritäten eines Eingangspins:

```
.equ Schalter = 1
; [...]
.if Schalter == 1
        sbic PINB,PINB0 ; ueberspringe wenn Eingangspin Null ist
        .else
        sbis PINB,PINB0 ; ueberspringe wenn Eingangspin Eins ist
        .endif
        rjmp PinbedingungNichtEingehalten
```

Der Code hinter .ELSE wird dann vom Assembler assembliert, wenn die .IF-Bedingung nicht eingehalten ist. Das kann leicht unübersichtlich werden, weil man hier Assembler-Bedingungen und bedingte Sprünge vermengt. In solchen Fällen kann es hilfreich sein, in das vom Assembler erzeugte Listing zu schauen, um die Verständnisaufgabe um eine Dimension zu reduzieren.

Manchmal kann es sinnvoll sein, mit der .ELSE-Bedingung eine weitere Schalterbedingung zu setzen. Dann macht man das mit .ELIF. Allerdings entsteht dabei die Gefahr, dass keine der beiden Bedingungen gegeben ist, dann wird weder der eine noch der andere Code assembliert. Das muss der Programmierer durchdenken und entscheiden. Ein Beispiel, mit der ein Code für die beiden Typen ATtiny13 und ATtiny24 fit gemacht werden soll:

```
.equ cTyp == 13 ; kann 13 oder 24 sein
; [...]
.if cTyp == 13
        ; Reset- und Int-Vektortabelle ATtiny13
        .elif cTyp == 24
        ; Reset- und Int-Vektortabelle ATtiny24
        .endif
```

Wenn der User jetzt weder 13 noch 24 einträgt, dann gibt es gar keinen Reset- und Interruptvektor im erzeugten Code. Da passieren dann lustige Effekte.

Um dies zu vermeiden, gibt es bei meinem Assembler gavrasm die zusätzliche Direktive .IFDEVICE Typ. Hier kann man explizit formulieren .IFDEVICE "ATtiny13" oder .IFDEVICE ATTINY13.

12.2.3 Andere hilfreiche Direktiven

Um im Fall, dass im obigen Beispiel im Kopf weder 13 noch 24 angegeben ist, zu reagieren kann man mit

```
.if (cTyp != 13) && (cTyp != 24)
        .error "Falscher Typ!"
        .endif
```

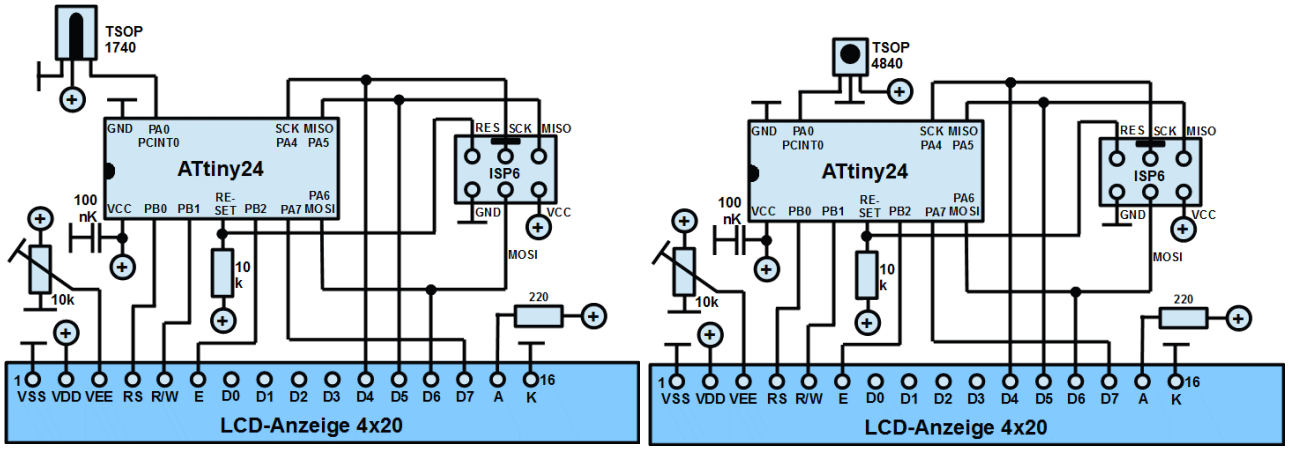
eine Fehlermeldung auslösen. Das kann man auch verwenden, um einen Abbruch zu erzwingen, wenn eine Konstante bestimmte Werte überschreitet (z. B. wenn ein Wert größer als 255 wird). Soll nur eine Meldung ausgegeben werden, die Assemblierung aber fortgesetzt werden, hilft ein .MESSAGE "Messagetext".

12.3 Hardware, Bauteile und Aufbau

12.3.1 Schaltbilder

Die Schaltbilder für die beiden Typen von IR-Empfangsmodulen sind geringfügig unteschiedlich.

Außer den IR-Empfängern ändert sich an der Schaltung sonst nichts.

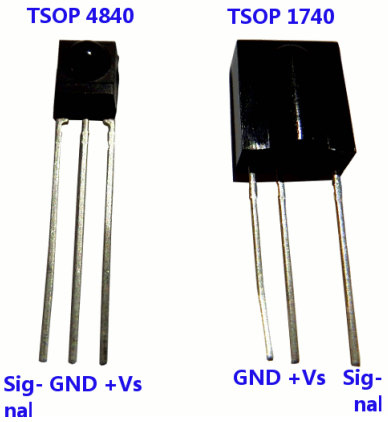


12.3.2 Bauteile: die IR-Empfänger

Die beiden IR-Empfänger sind für unterschiedliche Fernbedienungen erforderlich. TSOP4840 eignet sich eher für ältere, TSOP1740 eher für neuere Fernbedienungen. In der Praxis habe ich keine relevanten Unterschiede festgestellt.

Die Anschlussbelegung der beiden Typen sind unterschiedlich. Der Signalausgang ist mit dem PA0-Eingang des ATtiny24 zu verbinden. Das war es schon.

Beide sind Aktiv-Low, das heißt bei aktiv erkanntem Signal gehen die Ausgänge auf Null.



12.3.3 Aufbau

Der Anbau des Bauteils ist trivial.

12.3.4 Alternativer Aufbau

Wer dieses oder die nächsten Experimente lieber auf kompaktere Weise aufbauen möchte und die vielen Zuleitungen zur LCD fest verdrahtet sehen möchte, kann sich das Board hier als gedruckte Platine bauen. Alle Programmbeispiele dieser und der nachfolgenden Lektionen laufen darauf ohne Änderungen.

Top	Home	IR	.IF	Hardware	Messen	IR-TX	Daten-TX	3-Kanal TX/RX

12.4 Messen von IR-Signalen

12.4.1 Aufgabe

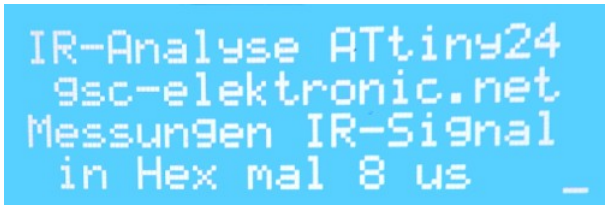
Neben der reinen Neugier, was denn das schwarze Schächtelchen so umtreibt, wenn eine Taste gedrückt wird, kann es sinnvoll sein, die Signale vorhandener Fernsteuerungen zu analysieren. Z. B. wenn wir eine Fernsteuerung klonen wollen, damit wir klammheimlich der sich auf dem Sofa lümmelnden jungen Dame ihren Lieblingssender wegzappen möchten, obwohl sie über die unumstrittene Hoheit über die Fernbedienung verfügt. Also machen wir in diesem Teil ein paar Portionen Erkundungssoftware. Die hilft uns auch, den im nächsten Teil gebauten IR-Sender zu testen und zu schauen, was der so macht.

12.4.2 Startsignale

12.4.2.1 Aufgabe

Die erste Analyseaufgabe ist es, die Startsignale von Fernbedienungen sichtbar zu machen.

12.4.2.2 Beispiele für IR-Signalfolgen



So sieht die Software unten nach dem Starten aus. Alle nachfolgende Software gibt ihre Messwerte in hexadezimalem Format aus. Wozu das gut sein soll, wird weiter unten klar. Wer Werte weiterverarbeiten will, kann einen Hex-Taschenrechner oder eine Tabellenkalkulation wie Open Office dazu verwenden, um aus den Krickeln Zahlen zu machen.

Alle Zahlen sind mit 8 malzunehmen, weil der zur Zeitmessung verwendete Teiler mit einem Vorteiler von 8 und einem Prozessortakt von 1 MHz läuft.

Die Software wartet jetzt auf Signale vom IR-Empfangsgerät. Sind 16 Pegelwechsel eingetroffen, werden diese gemessenen Zeiten dargestellt.

Das Signal beginnt mit einer langen, inaktiven Pause, mit 0x1696 * 8 µs Dauer. Dann folgen 0x0232 * 8 µs eine aktive LED, für etwa den gleichen Zeitraum eine inaktive Pause. Danach folgen Pausenperioden von jeweils entweder 0xC8 * 8 µs oder 0x3F * 8 µs Dauer und dazwischen aktive Signale von 0x49 * 8 µs Dauer. Sieht ziemlich einfach aus.

Signal#	H/L	TV	µs	N	L+H	HDR	µs	N	L+H	Kamera	µs	N	L+H
1	H	1696	46.256	3700		2404	73.760	5.901		13B0	40.320	3.226	
	L	0232	4.496	360		01B0	3.456	276		045B	8.920	714	
2	H	0229	4.424	354		00D2	1.680	134		021D	4.328	346	
	L	0049	584	47	176	003E	496	40	68	004F	632	51	176
3	H	00C9	1.608	129		002C	352	28		00C3	1.560	125	
	L	0049	584	47	176	003A	464	37	137	004B	600	48	174
4	H	00C9	1.608	129		009C	1.248	100		00C5	1.576	126	
	L	0049	584	47	175	0041	520	42	68	004B	600	48	86
5	H	00C8	1.600	128		0029	328	26		003C	480	38	
	L	0048	576	46	86	003E	496	40	69	004F	632	51	175
6	H	003F	504	40		002D	360	29		00C2	1.552	124	
	L	0049	584	47	87	003D	488	39	67	004B	600	48	86
7	H	003F	504	40		002C	352	28		003C	480	38	
	L	0048	576	46	86	003C	480	38	66	004C	608	49	87
8	H	003F	504	40		002C	352	28		003B	472	38	
	L	0048	576	46		003D	488	39		004C	608	49	
Durchschnitt				64				40				64	
Durchschnitt L				47				39				49	
Durchschnitt H				129				100				125	

Das hier sieht schon etwas verrückter aus, es stammt von einem HD-DVD-Recorder. Der Signalkopf hat eine lange High-Periode, eine etwas kürzere Low-Periode, eine etwa halb so lange High-Periode und eine kurze Low-Periode. Die Signalpausen (High) liegen alle zwischen 0x0029 und 0x002D, ein High-Signal liegt bei 0x009C. Die aktiven Signale liegen zwischen 0x003A und 0x0041.

Das hier sind die Signale einer Kamerasteuerung. Sie ähnelt den beiden obigen Beispielen, nur die Dauern sind völlig unterschiedlich.

Das sind die dargestellten Beispieldaten in einer Tabellenkalkulation. Die Köpfe dauern mit 50, 76 bzw. 50 ms am längsten, Datenbits brauchen je nach Fernbedienung 0,7 bis 1,0 ms.

Es ist noch N angegeben. Das wären die Anzahl Timerdurchläufe, um per CTC das Ein- und Ausschalten der LED mit 40 kHz zu bewerkstelligen.

12.4.2.3 Das Programm

Prinzipiell misst das Programm den Zeitraum zwischen ansteigenden und abfallenden sowie die zwischen abfallenden und ansteigenden Flanken. Dazu wird der 16-Bit-Timer TC1 verwendet, der ausgelesen und wieder auf Null gesetzt wird.

12.4.2.4 Programmstruktur

Das Programm arbeitet in folgenden Stufen:

- Zuerst wird wie immer der Stapel eingerichtet, die LCD-Ports initiiert, die LCD im 4-Bit-Busy-Modus initiiert, die Spezialzeichen in die LCD gebracht und der Starttext dargestellt. Der Zeiger Y wird auf den Pufferanfang für empfangene Signale im SRAM gesetzt. Der Timer 1 wird im Normalmodus mit einem Vorteiler von 8 gestartet, er dient der Zeitmessung. Am Mess-

eingang mit dem IR-Empfänger werden Pin Change Interrupts ermöglicht.

- Tritt ein Level Change ein, wird zunächst seine Polarität festgestellt: ist der Eingang auf Null, dann war der IR-Empfänger vor dem Pin Change inaktiv. Um zusammengehörige High/Low-Signale zusammen im SRAM abzulegen, wird hier ein Trick angewendet: mit dem versetzten Speichern mit STD Y+N,Register erfolgt die Ablage der gemessenen Zeiten nur an vorgewählten Plätzen im SRAM. Die Ablagereihenfolge ist dabei: MSB-Inaktiv, LSB-Inaktiv, MSB Aktiv, LSB Aktiv. Der Trick ist deshalb nötig, weil bei einem 16-Bit-Timer IMMER das LSB zuerst gelesen werden muss, weil mit dem Lesen auch das zugehörige MSB in einen Zwischenspeicher abgelegt wird. Würde nach dem nächsten Takt das MSB vielleicht gar nicht mehr zu dem gelesenen LSB passen, weil der Timer schon weiter und möglicherweise übergelaufen ist. Um die Anpasserei des Zeigers zu vermeiden, wenn das LSB zuerst zu speichern ist, ist STD ganz hilfreich. Erst wenn alle vier Bytes abgelegt sind, wird Y wieder um vier Bytes nach oben verschoben und die nächsten vier Messwerte abgeholt. Ist der Puffer mit Messwerten gefüllt, wird die Ausgabeflagge gesetzt. Am Ende der Service Routine wird der TC0-Zähler rückgesetzt.
- Die Ausgaberoutine gibt die im Puffer liegenden Daten formatiert aus.

12.4.2.5 Programm 1

Hier ist das Programm (den Quelltext im asm-Format gibt es hier, zusätzlich wird die LCD-Include-Datei benötigt).

```
;                                              .def rmo   = R17 ; Vielzweckregister
; ********************************************* .def rLine = R18 ; LCD-Zeilenzaehler
; * IR-Empfaenger mit ATtiny24 + LCD, Langmessung .def rLese = R19 ; LCD-Register
; * (C)2016 by http://www.elektronic.net        .def rimp  = R20 ; Interrupt-Vielzweck
; ********************************************* .def rFlag = R21 ; Flaggenregister
;                                                  .equ bUpd = 0 ; Update-Flagge
.NOLIST                                          .equ bShf = 1 ; Shift-Rueckwaerts
.INCLUDE "tn24def.inc"                         ; frei: R22 .. R25
.LIST                                          ; verwendet: XH:XL R27:R26 fuer Schieben
;                                              ; verwendet: YH:YL R29:R28 Zeiger in SRAM
; ---------- Programmablauf ------------        ; verwendet: ZH:ZL R31:R30 in LCD-Routinen
;                                              ;
; Der PCINT am PB0 (IR-Sensor) liest den         ; ---------- SRAM ----------------------
; Stand des 16-Bit-Timers bei jeder Flanke     .DSEG ; Datensegment
; und gibt sie in Hex aus (fallende Flanken    .ORG 0x0060 ; Start SRAM
; zuerst, steigende Flanken danach. Die        Buffer:
; gemessenen Werte werden in einem SRAM-       .byte 32 ; Puffer fuer Empfangsdaten
; Puffer abgelegt.                             Bufferende:
; Ist der Schalter VonHinten Eins, werden      Letzter:
; die Werte im Puffer bei jeder eintreffen-    .byte 4 ; Letzter gemessener Wert
; den positiven Flanke um vier Positionen      LetzterEnde:
; im SRAM-Puffer nach vorne geschoben, so      ;
; dass immer die zuletzt eingetroffenen        ; ---------- Reset- und Interrupts -----
; Flanken dargestellt werden.                  .CSEG ; Code-Segment
;                                              .ORG 0 ; an den Beginn
; ---------- Schalter ------------------              rjmp Start ; Reset-Vektor, Init
.equ VonHinten = 0 ; 0: von vorne nach hinten        reti ; INT0 External Interrupt Request 0
;                  , 1: von hinten nach vorne        rjmp Pci0Isr ; PCINT0 Pin Change Int 0
;                                                    reti ; PCINT1 Pin Change Int Request 1
; ---------- Ports ---------------------             reti ; WDT Watchdog Time-out
.equ pIrIn = PINA ; IR-Detektor-Port                 reti ; TIM1_CAPT Timer/Counter1 Capture
.equ bIrIn = 0    ; IR-Detektor-Pin                  reti ; TIM1_COMPA TC1 Comp Match A
;                                                    reti ; TIM1_COMPB TC1 Compare Match B
; ---------- Timing --------------------             reti ; TIM1_OVF Timer/Counter1 Overflow
; Prozessortakt           1.000.000 Hz               reti ; TIM0_COMPA TC0 Compare Match A
; Zeit pro Takt                 1 µs                 reti ; TIM0_COMPB TC0 Compare Match B
; Vorteiler TC1                 8              .if VonHinten == 1 ; Von hinten einschieben
; Zeit pro Timer-Takt           8 µs                 rjmp Tc0Isr ; TC0_OVF, Display updaten
; TC-Stand bei 1 ms           125                    .else
;                                                    reti ; TIM0_OVF Timer/Counter0 Overflow
; ---------- Register ------------------             .endif
; verwendet: R0 von LCD, Dezimalwandlung             reti ; ANA_COMP Analog Comparator
; verwendet: R1 Dezimalwandlung                      reti ; ADC ADC Conversion Complete
; frei: R2..R                                        reti ; EE_RDY EEPROM Ready
.def rSreg = R15 ; Sicherung Status                  reti ; USI_STR USI START
.def rmp   = R16 ; Vielzweckregister                 reti ; USI_OVF USI Overflow
```

```
;
; ---------- Interrupt Service -----
;
; PCINT Interrupt
;   wird von Flanken am IR-Sensor-Eingang ausge-
;   loest
;   Bei negativen Flanken wird der 16-Bit-Timer-
;   stand gelesen (Dauer der inaktiven Pause).
;   Wenn das MSB die Maximaldauer ueberschrei-
;   tet, wird der Zeiger Y auf den Pufferanfang
;   gestellt. Der Zaehlerstand des 16-Bit
;   Timers wird an der Zeigerposition Y (MSB)
;   und Y+1 (LSB) abgelegt.
;   Bei positiven Flanken wird die Ausgabeflag-
;   ge bUpd gesetzt. In der Betriebsart VonHin-
;   ten wird auch die Schiebeflagge gesetzt.
;   Der Zaehlerstand wird an der Zeigerposition
;   Y+2 (MSB) und Y+3 (LSB) abgelegt. Ist Von-
;   Hinten nicht Eins, wird der Pufferzeiger
;   um vier erhoeht.
;   Bei beiden Flanken wird der 16-Bit-Timer
;   auf Null gestellt.
;
Pci0Isr:
        in rSreg,SREG ; Status sichern
        sbic pIrIn,bIrIn ; ueberspringe bei Null
        rjmp Pci0Isr1 ; Eins
        ; Eingang ist Low, war vorher High
        in rimp,TCNT1L ; Low Byte zuerst
        std Y+1,rimp ; LSB in SRAM Byte 1
        in rimp,TCNT1H ; High Byte danach
        st Y,rimp ; MSB in SRAM Byte 0
.if VonHinten == 0 ; nur bei vorwaerts
        cpi rimp,0x05 ; Pause feststellen
        brcs Pci0Isr2 ; keine Pause
        ldi YH,HIGH(Buffer) ; Y auf Anfang Buffer
        ldi YL,LOW(Buffer)
        in rimp,TCNT1L ; Nochmal, Low Byte zuerst
        std Y+1,rimp ; LSB in SRAM Byte 1
        in rimp,TCNT1H ; High Byte danach
        st Y,rimp ; MSB in SRAM Byte 0
        .endif
        rjmp Pci0Isr2 ; fertig
Pci0Isr1:
        ; Eingang ist High, war vorher Low
        in rimp,TCNT1L ; Low Byte zuerst
        std Y+3,rimp ; LSB in SRAM Byte 3
        in rimp,TCNT1H ; High Byte danach
        std Y+2,rimp ; MSB in SRAM Byte 2
.if VonHinten == 1 ; nur bei rueckwaerts
        sbr rFlag,1<<bShf ; setze Schiebeflagge
        .else
        adiw YL,4 ; vier Bytes weiter
        cpi YL,0x60+32 ; 32 Bytes = 8 Lo/Hi-Paare
        brcs Pci0Isr2 ; noch nicht erreicht
        sbr rFlag,1<<bUpd ; Update-Flag setzen
        ldi YH,HIGH(Bufferende) ; Ende Buffer
        ldi YL,LOW(Bufferende)
        .endif
Pci0Isr2:
        ldi rimp,0 ; Neustart Zaehler
        out TCNT1H,rimp ; Zuerst MSB schreiben
        out TCNT1L,rimp ; Danach LSB schreiben
        out SREG,rSreg ; Status wieder herstellen
        reti ; fertig
;
; TC0-Overflow Interrupt Service Routine
;
; Wird beim Ueberlauf des Zaehlers TC0 nach
; 1.024 * 256 = 0,262 s ausgeloest, wenn
; keine Flanken am IR-Eingang eintreten.
; Setzt die Update-Flagge.
;
Tc0Isr:
        in rSreg,SREG ; SREG retten
        sbr rFlag,1<<bUpd ; Update-Flagge setzen
        out SREG,rSreg ; SREG wieder herstellen
        reti
;
; ---------------- Start, Init --------------
Start:
        ldi rmp,LOW(RAMEND) ; Stapel auf Ramende
        out SPL,rmp ; in Stapelzeiger
        ; I/O initiieren
        ; LCD-Port-Ausgaenge initiieren
        ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|
(1<<bLcdCRRW)
        out pLcdCR,rmp ; Kontrollport-Ausgaenge
        clr rmp ; Ausgaenge aus
        out pLcdCO,rmp ; an Kontrollport
        ldi rmp,mLcdDRW ; Datenport-Ausgabemaske
        out pLcdDR,rmp ; auf Richtungsregister
Datenausgabeport
        ; LCD initiieren
        rcall LcdInit ; Init der LCD
        ldi ZH,HIGH(2*Codezeichen) ; Spez.zeichen
        ldi ZL,LOW(2*Codezeichen)
        rcall LcdChars ; an LCD
        ldi ZH,HIGH(2*LcdStart) ; Text ausgeben
        ldi ZL,LOW(2*LcdStart)
        rcall LcdText
        ; Startwert fuer Pufferzeiger
.if VonHinten == 1 ; rueckwaerts
        ldi YH,HIGH(Letzter) ; Hinter Pufferende
        ldi YL,LOW(Letzter)
        .else
        ldi YH,HIGH(Buffer) ; Y auf Pufferanfang
        ldi YL,LOW(Buffer)
        .endif
        ; Startwert fuer Flagge
        clr rFlag
.if VonHinten == 1 ; Nur fuer rueckwaerts
        ; Timer 0 initiieren
        ldi rmp,(1<<CS02)|(1<<CS00) ; Presc 1024
        out TCCR0B,rmp ; Timer starten
        .endif
        ; Timer 1 initiieren, freilaufend durch 8
        ldi rmp,1<<CS11 ; Prescaler durch 8
        out TCCR1B,rmp ; an Kontrollregister B
        ; PCINT0 fuer IR-Rx
        ldi rmp,1<<PCINT0 ; Pin 0 Lvl change INTs
        out PCMSK0,rmo ; in Maske 0
        ldi rmp,1<<PCIE0 ; Int PCINT0 enable
        out GIMSK,rmp
        ; Sleep Enable
        ldi rmp,1<<SE ; Sleep Mode Idle
        out MCUCR,rmp ; in MCU Kontrollregister
        ; Interrupts ermoeglichen
        sei ; I-Flagge Statusregister
Schleife:
        sleep ; schlafen legen
        nop ; aufwachen
        sbrc rFlag,bShf ; Shift-Flagge auswerten
        rcall Shift ; Shift-Flagge behandeln
        sbrc rFlag,bUpd ; Update-Flagge auswerten
        rcall Update ; Flagge behandeln
        rjmp Schleife ; schlafen legen
;
; Schiebe Buffer-Inhalt rueckwaerts
;   Schiebt den Datenpuffer um vier
;   Bytes nach niedrigeren Adressen
;
Shift:
        cbr rFlag,1<<bShf ; Loesche Flagge
        clr rmp ; TC0 ruecksetzen
        out TCNT0,rmp
        ldi rmp,1<<TOIE0 ; Interrupt einschalten
        out TIMSK0,rmp ; in Interrupt-Maske
        ldi XH,HIGH(Buffer+4) ; Buffer Herkunft
        ldi XL,LOW(Buffer+4)
        ldi ZH,HIGH(Buffer) ; Bufferzeiger Ziel
        ldi ZL,LOW(Buffer)
Shift1: ; Schiebe Puffer rueckwaerts
        ld rmp,X+ ; Lese Byte aus Herkunft
```

```
        st Z+,rmp ; schreibe Byte in Ziel
        cpi ZL,LOW(Bufferende)
        brcs Shift1
        ret
;
; Update-Flagge behandeln
;   bUpd wird gesetzt, wenn
;   a) der PCINT0 eine Flanke erkennt,
;   b) der Timer ueberlaeuft.
;   Gibt gemessene Dauer in Hex aus.
;
Update:
        cbr rFlag,1<<bUpd ; Flagge abschalten
.if VonHinten == 1 ; nur bei rueckwaerts
        clr rmp ; TC0-Interrupt abschalten
        out TIMSK0,rmp ; in Interrupt-Maske
        .endif
        ldi rmp,1 ; Loesche LCD-Anzeige
        rcall LcdC4Byte ; an LCD-Kontrollport
        ldi ZH,HIGH(Buffer) ; Z auf Buffer-Anfang
        ldi ZL,LOW(Buffer)
        clr rLine ; Zeilenzaehler auf Null
Update1:
.if VonHinten == 1 ; Nur bei rueckwaerts
        ldi rmp,'e' ; letztes Wertepaar
        cpi ZL,LOW(Bufferende-4)
        brcc Update2 ; 'e' kennzeichnet letzten
        .else
        ldi rmp,0x02 ; Pausenzeichen vorwaerts
        cpi ZL,LOW(Buffer) ; erstes Wort?
        breq Update2 ; Pausenzeichen ausgeben
        .endif
        ldi rmp,0x01 ; Low-Zeichen ausgeben
        sbrc ZL,1 ; Bit 2 Bufferadresse ist Low?
        ldi rmp,0x00 ; nein, High
Update2:
        rcall LcdD4Byte ; Spezialchar ausgeben
        ld rmp,Z+ ; Lese MSB aus Buffer
        rcall Hex2Lcd ; MSB in Hex ausgeben
        ld rmp,Z+ ; Lese LSB aus Buffer
        rcall Hex2Lcd ; LSB in Hex ausgeben
        mov rmp,ZL ; Zeilenende feststellen
        andi rmp,0x07 ; 8 Byte ausgegeben?
        brne Update1 ; nein, weiter
        inc rLine ; naechste Zeile
        cpi rLine,4 ; Ende Anzeige?
        brcc Update3 ; ja, fertig
        mov rmp,rLine ; naechste Zeile
        rcall LcdLine ; an LCD
        rjmp Update1 ; und weiter
Update3:
        ldi rmp,0x0C ; Cursor und Blink aus
        rjmp LcdC4Byte ; an LCD
;
; Byte in rmp in Hex an LCD
;   Eingabedaten: Register rmp
;   Ausgabe HH an LCD an aktueller Position
;
Hex2Lcd:
        push rmp ; Byte retten
        swap rmp ; oberes Nibble zuerst
        rcall HexNibble2Lcd
        pop rmp ; rmp wieder herstellen
; Gib Nibble in Hex auf LCD aus
HexNibble2Lcd:
        andi rmp,0x0F ; unteres Nibble maskieren
        subi rmp,-'0' ; ASCII-Null dazu addieren
        cpi rmp,'9'+1 ; A bis F?
        brcs HexNibble2Lcd1 ; nein
        subi rmp,-7 ; auf A bis F
HexNibble2Lcd1:
        rjmp LcdD4Byte ; rmp auf LCD ausgeben
;
; Starttext LCD
LcdStart:
.db "IR-Analyse ATtiny24 ",0x0D,0xFF
.db " gsc-elektronic.net ",0x0D,0xFF
.db "Ausgabe Signaldauer ",0x0D,0xFF
.db " in Hex Hi/Lo, 8 us",0xFE
;
; Spezialzeichen
Codezeichen:
.db 64,0,0,0,0,0,4,4,6,0 ; Z = 0, Low-Signal
.db 72,0,0,0,0,0,5,7,5,0 ; Z = 1, High-Signal
.db 80,0,0,0,0,0,0,0,7,0 ; Z = 2, Pausentrennung
.db 0,0 ; Ende der Tabelle
;
; LCD-Routinen einlesen
.include "Lcd4Busy.inc"
;
; Ende Quelltext
;
```

12.4.3 Endesignale

Um herauszufinden, ob IR-Signale am Ende der gesendeten Sequenz eine abweichende Formatierung aufweisen, müssen wir die Signale am Ende der Sequenz auswerten und anzeigen. Das erfordert einen gewissen Umbau des Programmes. Hier wurde ein anderer Weg gewählt. Mit einem Schalter im Programmkopf:

```
; ---------- Schalter -----------------
.equ VonHinten = 0 ; 0: von vorne nach hinten
;                  , 1: von hinten nach vorne
```

und mit .IF VonHinten = 1 werden diejenigen Programmteile assembliert, die diese Bedingung erfüllen. Assemblieren und Brennen des Hex-Codes führt jetzt zu einer geänderten Ausführung.

Das letzte Signal ist mit einem "e" gekennzeichnet. Die Reihe zeigt, dass von hinten keine geänderten Signalfolgen zu beachten sind.

12.4.4 Anzahl Signale

Eine breite Vielfalt an Varianten gibt es auch bei der Anzahl an Hi/Lo-Paaren, aus denen sich

die gesendeten Signalfolgen zusammensetzen.

Diese Software zählt einfach nur die Signale, sortiert sie nach High- und Low-Polarität und zählt die, die länger als 256*8 = 2048 µs sind.

12.4.4.1 Programm

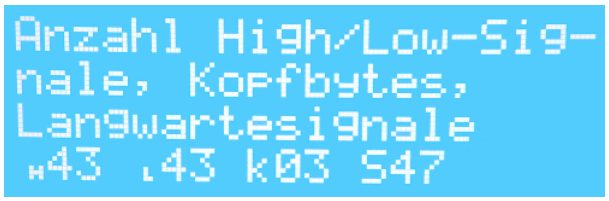
Die Software hat einen ganz ähnlichen Aufbau. Das Programm dafür ist nachfolgend aufgelistet (zum Quellcode im asm-Format geht es hier).

```
;
; ********************************************
; * IR-Empfaenger mit ATtiny24/LCD, Signalanzahl
; * (C)2016 by http://www.elektronic.net
; ********************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ---------- Programmablauf ------------
;
; Beobachtet den IR-Sensor am PA0-Eingang
; und misst mit TC1 die Dauer von Signalen.
; Die Anzahl der von der Fernsteuerung ge-
; sendeter Datensets, die Anzahl an Kopf-
; signalen und die Anzahl an aktiven Signa-
; len wird gezaehlt und 0,26 Sekunden nach
; dem letzten Signal auf der LCD in Hex
; dargestellt.
;
; ---------- Einstellungen Fernsteuerung
.equ cSet     = 0x10 ; lange Pause Signalsets
;
; ---------- Ports ---------------------
.equ pIrIn = PINA ; IR-Detektor-Port
.equ bIrIn = 0    ; IR-Detektor-Pin
;
; ---------- Timing --------------------
; Prozessortakt           1.000.000 Hz
; Zeit pro Takt                 1 µs
; Vorteiler TC1                 8
; Zeit pro Timer-Takt           8 µs
;
; ---------- Register ------------------
; verwendet: R0 von LCD, Dezimalwandlung
; verwendet: R1 Dezimalwandlung
.def rHigh = R2 ; Anzahl High-Signale
.def rLow  = R3 ; Anzahl Low-Signale
.def rKopf = R4 ; Anzahl Kopf-Signale
.def rSet  = R5 ; Anzahl Signalsets
; frei: R4..R14
.def rSreg = R15 ; Sicherung Status
.def rmp   = R16 ; Vielzweckregister
.def rmo   = R17 ; Vielzweckregister
.def rLine = R18 ; LCD-Zeilenzaehler
.def rLese = R19 ; LCD-Register
.def rimp  = R20 ; Interrupt-Vielzweck
.def rFlag = R21 ; Flaggenregister
   .equ bUpd = 0 ; nach Pause ausgeben
; frei: R23 .. R29
; verwendet: ZH:ZL R31:R30 in LCD-Routinen
;
; ---------- Reset- und Interrupts -----
.CSEG ; Code-Segment
.ORG 0 ; an den Beginn
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0 External Interrupt Request 0
  rjmp Pci0Isr ; PCINT0 Pin Chg Int Req 0
  reti ; PCINT1 Pin Chg Int Request 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT Timer/Counter1 Capture Event
  reti ; TIM1_COMPA TC1 Compare Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF Timer/Counter1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  rjmp Tc0Isr ; TIM0_OVF TC0 Overfl,Display
  reti ; ANA_COMP Analog Comparator
  reti ; ADC ADC Conversion Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; ---------- Interrupt Service -----
;
; PCINT Interrupt
; wird von allen Flanken am IR-Sensor aus-
; geloest
; Stellt fest, ob es sich um eine positive
; (aktives IR-Signal beendet) oder um eine
; negative Flanke (Pausensignal) handelt.
;
; Bei negativen Flanken (Pause) wird ge-
; prueft, ob das Signal laenger als 255 us
; war. Wenn ja, wird die Anzahl erkannter
; Kopfsignale in rKopf erhoeht. War das
; Signal laenger als 2,55 ms, wird die
; Anzahl Datensets in rSet erhoeht.
;
; Bei positiven Flanken wird die Anzahl
; an aktiven Bits in rLow erhoeht.
;
; In allen Faellen werden die Zaehler-
; staende in TC0 und TC1 rueckgesetzt
; und Interrupts des Timeout-Zaehlers
; TC0 zugelassen.
;
Pci0Isr:
  in rSreg,SREG ; Status sichern
  sbic pIrIn,bIrIn ; ueberspringe bei Null
  rjmp Pci0Isr1
  ; Eingang ist Low, war vorher High
  inc rHigh
  in rimp,TCNT1L ; Stand lesen
  in rimp,TCNT1H
  tst rimp ; MSB groesser Null
  breq Pci0Isr2
  inc rKopf ; Anzahl Kopf erhoehen
  cpi rimp,cSet ; Teste auf cSet
  brcs Pci0Isr2
  inc rSet
```

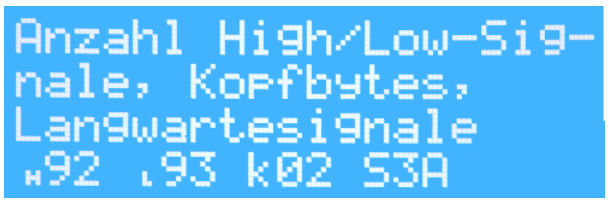
```
  rjmp Pci0Isr2
Pci0Isr1:
  inc rLow
Pci0Isr2:
  ldi rimp,0 ; Neustart Zaehler
  out TCNT1H,rimp ; Zuerst MSB schreiben
  out TCNT1L,rimp ; Danach LSB schreiben
  out TCNT0,rimp ; Zaehler ruecksetzen
  ldi rimp,1<<TOIE0 ; TimeOut starten
  out TIMSK0,rimp ; in Int-Maske
  out SREG,rSreg ; Status wieder herstellen
  reti ; fertig
;
; TC0-Overflow Interrupt Service Routine
; Wird ausgefuehrt, wenn der Timer TC0 nach
; 1.024*256 = 0,262 Sekunden ueberlaeuft.
;
; Setzt die Flagge bUpd, die zur LCD-Anzeige
; der gemessenen Signalanzahlen fuehrt.
;
Tc0Isr:
  in rSreg,SREG ; SREG retten
  sbr rFlag,1<<bUpd ; Update-Flagge setzen
  out SREG,rSreg ; SREG wieder herstellen
  reti
;
; ---------------- Start, Init --------------
Start:
  ldi rmp,LOW(RAMEND) ; Stapel auf Ramende
  out SPL,rmp ; in Stapelzeiger
  ; I/O initiieren
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Port-Ausgabemaske, Schreiben
  out pLcdDR,rmp ; Richtungsreg.Datenausgabeport
  ; LCD initiieren
  rcall LcdInit ; Init der LCD
  ldi ZH,HIGH(2*Codezeichen) ; Spezialzeichen
  ldi ZL,LOW(2*Codezeichen)
  rcall LcdChars ; an LCD
  ldi ZH,HIGH(2*LcdStart) ; Start-Text ausgeben
  ldi ZL,LOW(2*LcdStart)
  rcall LcdText
  ; Startwert fuer Flagge
  clr rFlag
  ; Alle Inhalte entleeren
  rcall Neustart ; Neustartwerte setzen
  ; Timer 0 initiieren
  ldi rmp,(1<<CS02)|(1<<CS00) ; Prescaler 1024
  out TCCR0B,rmp ; Timer starten
  ; Timer 1 initiieren, freilaufend durch 8
  ldi rmp,1<<CS11 ; Prescaler durch 8
  out TCCR1B,rmp ; an Kontrollregister B
  ; PCINT0 fuer IR-Rx
  ldi rmp,1<<PCINT0 ; Pin 0 Level change INTs
  out PCMSK0,rmo ; in Maske 0
  ldi rmp,1<<PCIE0 ; Interrupt PCINT0 enable
  out GIMSK,rmp
  ; Sleep Enable
  ldi rmp,1<<SE ; Sleep Mode Idle
  out MCUCR,rmp ; in MCU Kontrollregister
  ; Interrupts ermoeglichen
  sei ; I-Flagge Statusregister
Schleife:
  sleep ; schlafen legen
  nop ; aufwachen
  sbrc rFlag,bUpd ; Update-Flagge auswerten
  rcall Update ; Flagge behandeln
  rjmp Schleife ; schlafen legen
;
; Update-Flagge behandeln
;   bUpd wird gesetzt, wenn der Timer TC0
;   ueberlaeuft.
;   Gibt gemessene Anzahl IR-Signale in
;   Hex auf der LCD aus.
;
Update:
  cbr rFlag,1<<bUpd ; Flagge wieder abschalten
  clr rmp ; Interrupts Timer aus
  out TCNT0,rmp ; Timer auf Null
  out TIMSK0,rmp ; in Interruptmaske
  ; Display leer
  ldi rmp,0x01 ; Display loeschen
  rcall LcdC4Byte
  ; Highs anzeigen
  ldi ZH,High(2*LcdForm) ; Formvorlage
  ldi ZL,Low(2*LcdForm)
  rcall LcdText
  ldi rmp,0x01
  rcall LcdD4Byte
  mov rmp,rHigh
  rcall Hex2Lcd
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,0x00
  rcall LcdD4Byte
  mov rmp,rLow
  rcall Hex2Lcd
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,'k'
  rcall LcdD4Byte
  mov rmp,rKopf
  rcall Hex2Lcd
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,'S'
  rcall LcdD4Byte
  mov rmp,rSet
  rcall Hex2Lcd
  ldi rmp,0x0C ; Cursor und Blink aus
  rcall LcdC4Byte ; an LCD
;
; Neustart
Neustart:
  clr rKopf
  clr rHigh
  clr rLow
  ret
;
; Byte in rmp in Hex an LCD
;   Eingabedaten: Register rmp
;   Ausgabe HH an LCD an aktueller Position
;
Hex2Lcd:
  push rmp ; Byte retten
  swap rmp ; oberes Nibble zuerst
  rcall HexNibble2Lcd
  pop rmp ; rmp wieder herstellen
; Gib Nibble in Hex auf LCD aus
HexNibble2Lcd:
  andi rmp,0x0F ; unteres Nibble maskieren
  subi rmp,-'0' ; ASCII-Null dazu addieren
  cpi rmp,'9'+1 ; A bis F?
  brcs HexNibble2Lcd1 ; nein
  subi rmp,-7 ; auf A bis F
HexNibble2Lcd1:
  rjmp LcdD4Byte ; rmp auf LCD ausgeben
;
; Starttext LCD
LcdStart:
.db "IR-Analyse ATtiny24 ",0x0D,0xFF
.db " gsc-elektronic.net ",0x0D,0xFF
.db "Messungen IR-Signal ",0x0D,0xFF
.db "   Signalanzahl",0xFE
;
; Ausgabebild
LcdForm:
.db "Anzahl High/Low-Sig-",0x0D,0xFF
.db "nale, Kopfbytes,",0x0D,0xFF
.db "Langwartesignale",0x0D,0xFE
```

```
;
; Spezialzeichen
Codezeichen:
.db 64,0,0,0,0,0,4,4,6,0 ; Z = 0, Low-Signal
.db 72,0,0,0,0,0,5,7,5,0 ; Z = 1, High-Signal
.db 0,0 ; Ende der Tabelle
;
; LCD-Routinen einlesen
.include "Lcd4Busy.inc"
```

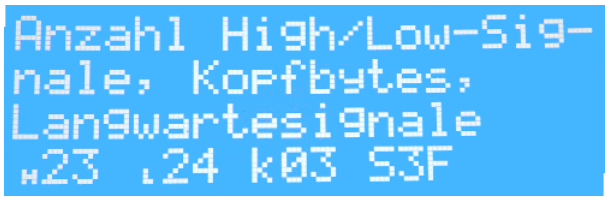
12.4.4.2 Ergebnisbeispiele



Mit dieser Software kriegt man heraus, dass die TV-Fernsteuerung 71 Hi/Lo-Wertepaare umfasst, davon drei Kopfsignale länger als 2,048 ms und 67 Datenpaare. Sie ist offensichtlich geringfügig länger als 64 Bits.

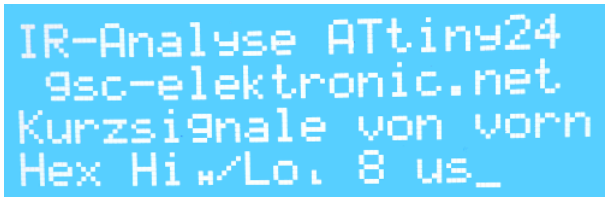


Diese hier ist ein echtes Rätsel: sie ist mit 146 Hi- und 147 Lo-Werten ein echtes Schwergewicht.



Mit ganzen neun Tasten ausgestattet, liefert die Kamerasteuerung 35 Hi- und 36 Lo-Signale. Mit 32 Bit lassen sich ganze 429 Millionen Kombinationen kodieren. Offensichtlich steht technische Rationalität beim Entwickeln von Fernsteuerungen nicht im Vordergrund.

12.4.5 Signaldauern Datenbits



Wie lange Einsen und Nullen genau definiert sind, erkennt man im Prinzip schon aus der ersten Diagnosesoftware. Um ein paar mehr Signaldauern zu sehen, können wir auf die Ausgabe des MSB verzichten, da es bei Datenbytes ohnehin Null ist. So kriegen wir 24 Bytes angezeigt. Wenn wir nur Hi- oder nur Lo-Signale anzeigen lassen und zusätzlich von vorne und von hinten anzeigen, dann haben wir schon ganz viele Datenbits im Blick.

So teilt die Software zu Beginn mit, was beim Messen im Folgenden dargestellt wird.

12.4.5.1 Programm

Das Programm ist im Folgenden aufgelistet (der Quelltext im asm-Format ist hier, zusätzlich wird die LCD-Include-Datei benötigt).

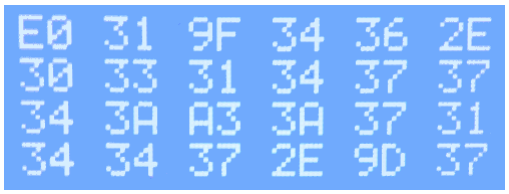
```
;
; ********************************************
; * IR-Empfaenger mit ATtiny24/LCD, Byteausgabe *
; * (C)2016 by http://www.elektronic.net       *
; ********************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ---------- Programmablauf ------------
;
; Die Dauer von IR-Signalen wird in Viel-
; fachen von 8 us gemessen und auf der
; LCD in Hex ausgegeben. Wahlweise
; - werden nur die High-Signale (Pausen)
;   oder die Low-Signale (Aktive Signale)
;   oder beide nacheinander ausgegeben
;   (Schalter cHigh und cLow),
; - die Signale zu Beginn oder am Ende
;   eines Datensets ausgegeben (Schalter
;   VonHinten).
;
; ---------- Schalter ------------------
.equ VonHinten = 0 ; 0: von vorne nach hinten
;                  ; 1: von hinten nach vorne
.equ cHigh     = 1 ; 1: High-Bytes aufzeichnen
.equ cLow      = 1 ; 1: Low-Bytes aufzeichnen
;
; ---------- Byte-Ausgabe --------------
; .bb.bb.bb.bb.bb.bb ; 6 Byte
; .bb.bb.bb.bb.bb.bb ; 12 Byte
```

```
; .bb.bb.bb.bb.bb.bb ; 18 Byte
; .bb.bb.bb.bb.bb.bb ; 24 Byte
;
; ---------- Ports --------------------
.equ pIrIn = PINA ; IR-Detektor-Port
.equ bIrIn = 0    ; IR-Detektor-Pin
;
; ---------- Timing -------------------
; Prozessortakt           1.000.000 Hz
; Zeit pro Takt                   1 us
; Vorteiler TC1                   8
; Zeit pro Timer-Takt             8 us
; Lang-Signal                30.000 us
; Langsignal-Zaehlerstand, MSB   14
.equ cLang = 14 ; MSB langes Signal
;
; ---------- Register -----------------
; verwendet: R0 von LCD, Dezimalwandlung
; verwendet: R1 Dezimalwandlung
; frei: R2..R
.def rSreg = R15 ; Sicherung Status
.def rmp   = R16 ; Vielzweckregister
.def rmo   = R17 ; Vielzweckregister
.def rLine = R18 ; LCD-Zeilenzaehler
.def rLese = R19 ; LCD-Register
.def rimp  = R20 ; Interrupt-Vielzweck
.def rFlag = R21 ; Flaggenregister
   .equ bUpd = 0 ; Update-Flagge
   .equ bShf = 1 ; Shift-Rueckwaerts
   .equ bLng = 2 ; Langes High-Signal
; frei: R22 .. R25
; verwendet: XH:XL R27:R26 fuer Schieben
; verwendet: YH:YL R29:R28 Zeiger in SRAM
; verwendet: ZH:ZL R31:R30 in LCD-Routinen
;
; ---------- SRAM ---------------------
.DSEG ; Datensegment
.ORG 0x0060 ; Start SRAM
Buffer:
.byte 24 ; Puffer fuer Empfangsdaten
Bufferende:
Letzter:
.byte 1 ; Letzter gemessener Wert
LetzterEnde:
;
; ---------- Reset- und Interrupts -----
.CSEG ; Code-Segment
.ORG 0 ; an den Beginn
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0 External Interrupt Request 0
  rjmp Pci0Isr ; PCINT0 Pin Chg Int Request 0
  reti ; PCINT1 Pin Change Interrupt Request 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT Timer/Counter1 Capture Event
  reti ; TIM1_COMPA TC1 Compare Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF Timer/Counter1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  rjmp Tc0Isr ; TIM0_OVF Display updaten
  reti ; ANA_COMP Analog Comparator
  reti ; ADC ADC Conversion Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; ---------- Interrupt Service -----
;
; PCINT-Interrupt
; wird von Flanken des IR-Sensors ausgeloest
;
; War der IR-Eingang vorher High, liegt eine
; negative Flanke vor (Dauer entspricht einer
; Pause) und es wird geprueft, ob das Signal
; laenger als 14*256*8 = 28,7 ms war. Wenn ja
; wird die bLng-Flagge gesetzt und, wenn nicht
; die Richtung VonHinten gewaehlt ist, der
; Pufferzeiger Y auf Anfang gesetzt. Wenn der
; Modus cHigh auf 1 gesetzt ist, wird die
; Dauer noch in den SRAM-Puffer geschrieben.
;
; War der IR-Eingang vorher Low, wird die
; Dauer in den SRAM-Puffer geschrieben (nur
; wenn cLow auf 1 gesetzt ist).
;
; Abschliessend wird Timer TC1 Null gesetzt.
;
Pci0Isr:
  in rSreg,SREG ; Status sichern
  sbic pIrIn,bIrIn ; ueberspringe bei Null
  rjmp Pci0Isr1 ; Eins
  ; Eingang ist Low, war vorher High
  in rimp,TCNT1L ; Low Byte zuerst
  st Y,rimp ; LSB in SRAM
  in rimp,TCNT1H ; High Byte danach
  cpi rimp,cLang ; Langsignal?
  brcs Pci0IsrNoLong ; kein Langsignal
  sbr rFlag,1<<bLng ; setze Langflagge
.if VonHinten == 0
  ldi YH,HIGH(Buffer) ; Y auf Anfang Buffer
  ldi YL,LOW(Buffer)
  .endif
  rjmp Pci0Isr2 ; Fertig
Pci0IsrNoLong:
.if cHigh == 1 ; nur bei High-Signal auswerten
  tst rimp ; teste MSB
  brne Pci0IsrMsb
.if VonHinten == 1 ; nur bei von hinten
  ; High-Signal
  sbr rFlag,1<<bShf ; Shift-Flagge setzen
  .else
  ; nur bei vorwaerts: Position erhoehen
  cpi YL,LOW(Bufferende) ; Ende Buffer?
  brcc Pci0IsrNoInc ; hinter Buffer
  adiw YL,1 ; vor Bufferende, erhoehen
Pci0IsrNoInc:
  .endif
Pci0IsrMsb: ; MSB groesser Null
  .endif
  rjmp Pci0Isr2 ; fertig
Pci0Isr1:
  ; Eingang ist High, war vorher Low
.if cLow == 1 ; Nur wenn cLow eingeschaltet
  in rimp,TCNT1L ; Low Byte zuerst
  st Y,rimp ; LSB in SRAM
  in rimp,TCNT1H ; High Byte danach
  tst rimp ; wenn Msb nicht Null ist, ignorieren
  brne Pci0IsrMsb1 ; ignorieren
.if VonHinten == 1 ; nur bei rueckwaerts
  sbr rFlag,1<<bShf ; setze Schiebeflagge
  .else
  clr rimp ; stelle Timer 0 zurueck
  out TCNT0,rimp
  cpi YL,LOW(Bufferende) ; oberhalb Buffer
  brcc Pci0IsrNoInc1 ; ja, nicht erhoehen
  adiw YL,1 ; Eingabeposition weiter
Pci0IsrNoInc1:
  .endif
Pci0IsrMsb1:
  .endif
Pci0Isr2:
  ldi rimp,0 ; Neustart Timer 1
  out TCNT1H,rimp ; Zuerst MSB schreiben
  out TCNT1L,rimp ; Danach LSB schreiben
  out SREG,rSreg ; Status wieder herstellen
  reti ; fertig
;
; TC0-Overflow Interrupt Service Routine
; wird vom Ueberlauf des Timeout-Timers TC0
; ausgeloest
;
; Setzt die bUpd-Flagge und loest die Ausgabe
; der Ergebnisse auf der LCD aus.
;
```

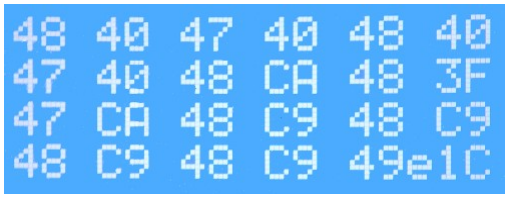
```
Tc0Isr:
  in rSreg,SREG ; SREG retten
  sbr rFlag,1<<bUpd ; Update-Flagge setzen
  out SREG,rSreg ; SREG wieder herstellen
  reti
;
; ---------------- Start, Init --------------
Start:
  ldi rmp,LOW(RAMEND) ; Stapel auf Ramende
  out SPL,rmp ; in Stapelzeiger
  ; I/O initiieren
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
Schreiben
  out pLcdDR,rmp ; auf Richtungsregister
Datenausgabeport
  ; LCD initiieren
  rcall LcdInit ; Init der LCD
  ldi ZH,HIGH(2*Codezeichen) ; Spezialzeichen
  ldi ZL,LOW(2*Codezeichen)
  rcall LcdChars ; an LCD
  ldi ZH,HIGH(2*LcdStart) ; Start-Text ausgeben
  ldi ZL,LOW(2*LcdStart)
  rcall LcdText
  ; Startwert fuer Pufferzeiger
.if VonHinten == 1 ; rueckwaerts
  ldi YH,HIGH(Letzter) ; Hinter Pufferende
  ldi YL,LOW(Letzter)
  .else
  ldi YH,HIGH(Buffer) ; Y auf Pufferanfang
  ldi YL,LOW(Buffer)
  .endif
  ; Startwert fuer Flagge
  clr rFlag
  ; Timer 0 fuer Timeout initiieren
  ldi rmp,(1<<CS02)|(1<<CS00) ; Prescaler 1024
  out TCCR0B,rmp ; Timer starten
  ; Timer 1 initiieren, freilaufend durch 8
  ldi rmp,1<<CS11 ; Prescaler durch 8
  out TCCR1B,rmp ; an Kontrollregister B
  ; PCINT0 fuer IR-Rx
  ldi rmp,1<<PCINT0 ; Pin 0 Level change INTs
  out PCMSK0,rmo ; in Maske 0
  ldi rmp,1<<PCIE0 ; Interrupt PCINT0 ermoeglichen
  out GIMSK,rmp
  ; Sleep Enable
  ldi rmp,1<<SE ; Sleep Mode Idle
  out MCUCR,rmp ; in MCU Kontrollregister
  ; Interrupts ermoeglichen
  sei ; I-Flagge Statusregister
Schleife:
  sleep ; schlafen legen
  nop ; aufwachen
  sbrc rFlag,bShf ; Shift-Flagge auswerten
  rcall Shift ; Shift-Flagge behandeln
  sbrc rFlag,bLng ; Lang-Flagge behandeln
  rcall Lang ; Lang-Flagge auswerten
  sbrc rFlag,bUpd ; Update-Flagge auswerten
  rcall Update ; Update-Flagge behandeln
  rjmp Schleife ; schlafen legen
;
; Langes Wartesignal erkannt
Lang:
  cbr rFlag,1<<bLng ; Lang-Flagge loeschen
  clr rmp ; Timer 0 ruecksetzen
  out TCNT0,rmp
  ldi rmp,1<<TOIE0 ; Overflow-Int
  out TIMSK0,rmp ; in Interrupt-Maske
  ret
;
; Schiebe letzten Wert in Buffer
; wird von gesetzter bShf-Flagge ausgeloest
;
; Schiebt den letzten gemessenen Datensatz
; in den SRAM-Puffer.
;
Shift:
  cbr rFlag,1<<bShf ; Loesche Flagge
  clr rmp ; TC0 ruecksetzen
  out TCNT0,rmp
  ldi rmp,1<<TOIE0 ; Interrupt einschalten
  out TIMSK0,rmp ; in Interrupt-Maske
  ldi XH,HIGH(Buffer) ; Bufferzeiger Anfang
  ldi XL,LOW(Buffer)
  ldi ZH,HIGH(Buffer+1) ; Bufferzeiger Ein Byte
hoeher
  ldi ZL,LOW(Buffer+1)
Shift1: ; Schiebe Bytes vorwaerts
  ld rmp,Z+ ; Lese Byte aus Herkunft
  st X+,rmp ; schreibe Byte in Ziel
  cpi ZL,LOW(LetzterEnde)
  brcs Shift1
  ret
;
; Update-Flagge behandeln
; Wird von gesetzter Update-Flagge ausgeloest
;
; Gibt die im SRAM-Puffer gespeicherten
; Dauern von High-/Low- oder von beiden
; Signalen zusammen aus.
;
Update:
  cbr rFlag,1<<bUpd ; Flagge wieder abschalten
  clr rmp ; TC0-Interrupt abschalten
  out TIMSK0,rmp ; in Interrupt-Maske
  ldi rmp,1 ; Loesche LCD-Anzeige
  rcall LcdC4Byte ; an LCD-Kontrollport
  ldi ZH,HIGH(Buffer) ; Setze Z auf Buffer-Anfang
  ldi ZL,LOW(Buffer)
Update1:
  cpi ZL,LOW(Bufferende) ; gesamter Buffer
ausgegeben?
  brcc Update5 ; Ferig
.if VonHinten == 1 ; Nur bei rueckwaerts
  ldi rmp,'e' ; letztes Wertepaar
  cpi ZL,LOW(Bufferende-1)
  brcc Update2 ; 'e' kennzeichnet letzten
  .endif
.if cHigh == 1
  ; cHigh=1
  .if cLow == 1 ; beide Signale
    ; cHigh=1, cLow=1
        ldi rmp,' ' ; Leerzeichen
        .else
        ; cHigh=1, cLow=0
        ldi rmp,0x01 ; High-Zeichen setzen
        .endif
  .else
  ; cHigh=0
  ldi rmp,0x00 ; Low-Zeichen setzen
  .endif
Update2:
  rcall LcdD4Byte ; Spezialchar ausgeben
  ld rmp,Z+ ; Lese MSB aus Buffer
  rcall Hex2Lcd ; MSB in Hex ausgeben
  cpi ZL,LOW(Buffer+6) ; Zeile 1 voll?
  brne UpDate3 ; Z=6?
  rcall LcdLine2 ; zeile 2
  rjmp Update1 ; weiter
Update3:
  cpi ZL,LOW(Buffer+12) ; Zeile 2 voll?
  brne Update4
  rcall LcdLine3 ; zeile = 3
  rjmp Update1
Update4:
  cpi ZL,LOW(Buffer+18) ; Zeile 3 voll?
  brne Update1 ; nein, weiter
  rcall LcdLine4 ; zeile = 4
  rjmp Update1
Update5:
```

```
  ldi rmp,0x0C ; Cursor und Blink aus
  rcall LcdC4Byte ; an LCD
  clr rmp ; loeschen
.if VonHinten == 1
  ; Buffer rueckwaerts loeschen
  ldi ZH,HIGH(LetzterEnde) ; auf Ende Puffer
  ldi ZL,LOW(LetzterEnde)
Lang1:
  st -Z,rmp ; loeschen
  cpi ZL,LOW(Buffer+1) ; am Anfang angekommen?
  brcc Lang1
  .else
  ; Buffer vorwaerts loeschen
  ldi ZH,HIGH(Buffer) ; auf Anfang Buffer
  ldi ZL,LOW(Buffer)
Lang1:
  st Z+,rmp ; loeschen
  cpi ZL,LOW(LetzterEnde)
  brcs Lang1
  .endif
  ret
;
; Byte in rmp in Hex an LCD
;   Eingabedaten: Register rmp
;   Ausgabe HH an LCD an aktueller Position
;
Hex2Lcd:
  push rmp ; Byte retten
  swap rmp ; oberes Nibble zuerst
  rcall HexNibble2Lcd
  pop rmp ; rmp wieder herstellen
; Gib Nibble in Hex auf LCD aus
HexNibble2Lcd:
  andi rmp,0x0F ; unteres Nibble maskieren
  subi rmp,-'0' ; ASCII-Null dazu addieren
  cpi rmp,'9'+1 ; A bis F?
  brcs HexNibble2Lcd1 ; nein
  subi rmp,-7 ; auf A bis F
HexNibble2Lcd1:
  rjmp LcdD4Byte ; rmp auf LCD ausgeben
;
; Starttext LCD
LcdStart:
.db "IR-Analyse ATtiny24 ",0x0D,0xFF
.db " gsc-elektronic.net ",0x0D,0xFF
.if VonHinten == 1
  .db "Kurzsignale am Ende ",0x0D,0xFF
  .else
  .db "Kurzsignale von vorn",0x0D,0xFF
  .endif
.if cHigh == 1
  .if cLow ==1
  .db "Hex Hi",0x01,"/Lo",0x00," 8 us",0xFE,0xFE
  .else
  .db "Hex High",0x01,", 8 us",0xFE
  .endif
  .else
  .db "Hex Low",0x00,", 8 us",0xFE,0xFE
  .endif
;
; Spezialzeichen
Codezeichen:
.db 64,0,0,0,0,0,4,4,6,0 ; Z = 0, Low-Signal
.db 72,0,0,0,0,0,5,7,5,0 ; Z = 1, High-Signal
.db 80,0,0,0,0,0,0,0,7,0 ; Z = 2, Pausentrennung
.db 0,0 ; Ende der Tabelle
;
; LCD-Routinen einlesen
.include "Lcd4Busy.inc"
;
; Ende Quelltext
;
```

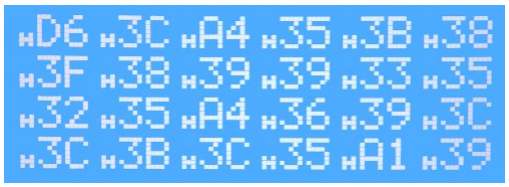
12.4.5.2 Beispiele



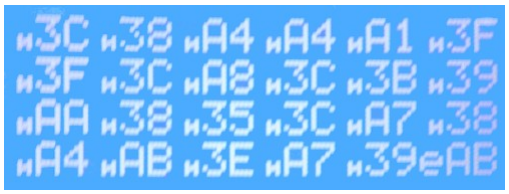
Das führt zu folgenden Ausgaben: Mit beiden Schaltern gesetzt, werden sowohl die High- als auch die Low-Dauern ausgegeben. In diesem Fall sind die Dauern mit Leerzeichen getrennt dargestellt.



Die gleiche Darstellung vom Signalende her (e signalisiert das letzte Signal).



Die Software zeigt hier nur High-Signale auf.



Und hier das Ganze vom Ende her.

12.4.6 Kodierungen

Aus den bisherigen Analysemethoden lassen sich die kodierten Tasten einer Fernsteuerung nur mühsam ermitteln (nur bis 48 Bits). Es gibt daher hier die etwas komfortablere und für mehr Bits ausgelegte Lösung.

12.4.6.1 Programm

Das Programm ist hier (zum Quellcode im asm-Format geht es hier, zusätzlich wird die LCD-Include-Datei benötigt).

```
;
; ******************************************
; * IR-Empfaenger mit ATtiny24 und LCD, Burst *
; * (C)2016 by http://www.elektronic.net      *
; ******************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ---------- Programmablauf ------------
;
; Misst die Dauer aller Eingangssignale vom
; IR-Sensor in Vielfachen von 8 us, wertet
; sie nach Kopf- und Datenbits getrennt aus
; und schreibt die Ergebnisse in Hex auf
; die LCD.
;
; ---------- Schalter ------------------
.equ cAktivHi = 1 ; 1: High-Signaldauer Bits
;                   0: Low-Signaldauer Bits
;
; ---------- Einstellungen Fernsteuerung
.equ cIRStart = 0x10 ; Pause vor Signal (MSB)
.equ cIRHigh  = 0x50 ; Null/Eins-Schwelle
;
; ---------- Ports ---------------------
.equ pIrIn = PINA ; IR-Detektor-Port
.equ bIrIn = 0    ; IR-Detektor-Pin
;
; ---------- Timing --------------------
; Prozessortakt           1.000.000 Hz
; Zeit pro Takt                   1 µs
; Vorteiler TC1                   8
; Zeit pro Timer-Takt             8 µs
;
; ---------- Register ------------------
; verwendet: R0 von LCD, Dezimalwandlung
; verwendet: R1 Dezimalwandlung
.def rShift0 = R2 ; Schieberegister, Byte 0
.def rShift1 = R3 ; dto., Byte 1
.def rShift2 = R4 ; dto., Byte 2
.def rShift3 = R5 ; dto., Byte 3
.def rShift4 = R6 ; dto., Byte 4
.def rShift5 = R7 ; dto., Byte 5
.def rShift6 = R8 ; dto., Byte 6
.def rShift7 = R9 ; dto., Byte 7
; frei: R8..R14
.def rSreg = R15 ; Sicherung Status
.def rmp   = R16 ; Vielzweckregister
.def rmo   = R17 ; Vielzweckregister
.def rLine = R18 ; LCD-Zeilenzaehler
.def rLese = R19 ; LCD-Register
.def rimp  = R20 ; Interrupt-Vielzweck
.def rFlag = R21 ; Flaggenregister
   .equ bEin = 0 ; Eingangswert vorhanden
   .equ bPol = 1 ; Polaritaet (high/low)
   .equ bUpd = 2 ; nach Pause ausgeben
   .equ bKOf = 3 ; Kopfdatenueberlauf
   .equ bDOf = 4 ; Datenbits-Ueberlauf
.def rKopf = R22 ; Kopfdatenzaehler
.def rData = R23 ; Bitzaehlerregister
; frei: R24 .. R25
; verwendet: XH:XL R27:R26 fuer Schieben
; verwendet: YH:YL R29:R28 Zeiger in SRAM
; verwendet: ZH:ZL R31:R30 in LCD-Routinen
;
; ---------- SRAM ----------------------
.DSEG ; Datensegment
.ORG 0x0060 ; Start SRAM
Kopf:
.byte 6 ; Puffer fuer Kopfdaten, MSB, LSB
Kopfende:
StatistikNull:
.byte 3 ; Summe Dauer Null,MSB/LSB/Anzahl
StatistikEins:
.byte 3 ; Summe Dauer Eins,MSB/LSB/Anzahl
StatistikPause:
.byte 3 ; Summe Dauer Pause,LSB/MSB/Anzahl
Statistikende:
;
; ---------- Reset- und Interrupts -----
.CSEG ; Code-Segment
.ORG 0 ; an den Beginn
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0 External Interrupt Request 0
  rjmp Pci0Isr ; PCINT0 Pin Change Int Req 0
  reti ; PCINT1 Pin Change Interrupt Req 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT Timer/Counter1 Capture Event
```

```
  reti ; TIM1_COMPA TC1 Compare Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF Timer/Counter1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  rjmp Tc0Isr ; TIM0_OVF TC0, Display updaten
  reti ; ANA_COMP Analog Comparator
  reti ; ADC ADC Conversion Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; ---------- Interrupt Service -----
;
; PCINT-Interrupt
; wird von allen Flanken des IR-Signals
; ausgeloest
;
; Stellt die Polaritaet des Signals fest
; und schreibt sie in die bPol-Flagge.
; Der Stand des TC1-Zaehlers wird in Y
; gelesen, der Zaehler neu gestartet
; und die bEin-Flagge gesetzt, die die
; Auswertung der gemessenen Dauer aus-
; loest.
;
Pci0Isr:
  in rSreg,SREG ; Status sichern
  cbr rFlag,1<<bPol ; Polaritaets-Flagge loeschen
  sbic pIrIn,bIrIn ; ueberspringe bei Null
  ; Eingang ist Low, war vorher High
  sbr rFlag,1<<bPol ; Polaritaets-Flagge setzen
  in YL,TCNT1L ; Low Byte zuerst
  in YH,TCNT1H ; High Byte danach
Pci0Isr2:
  ldi rimp,0 ; Neustart Zaehler
  out TCNT1H,rimp ; Zuerst MSB schreiben
  out TCNT1L,rimp ; Danach LSB schreiben
  sbr rFlag,1<<bEin ; Flagge eingegangenes Signal
  out SREG,rSreg ; Status wieder herstellen
  reti ; fertig
;
; TC0-Overflow Interrupt Service Routine
; wird 0,26 Sekunden nach dem letzten Eingangs-
; signal ausgeloest
;
; Setzt die Update-Flagge bUpd und loest die
; LCD-Ausgabe der gesammelten Daten aus.
;
Tc0Isr:
  in rSreg,SREG ; SREG retten
  sbr rFlag,1<<bUpd ; Update-Flagge setzen
  out SREG,rSreg ; SREG wieder herstellen
  reti
;
; ---------------- Start, Init --------------
Start:
  ldi rmp,LOW(RAMEND) ; Stapel auf Ramende
  out SPL,rmp ; in Stapelzeiger
  ; I/O initiieren
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske
  out pLcdDR,rmp ; auf Richtungsregister
     ; Datenausgabeport
  ; LCD initiieren
  rcall LcdInit ; Init der LCD
  ldi ZH,HIGH(2*Codezeichen) ; Spezialzeichen
  ldi ZL,LOW(2*Codezeichen)
  rcall LcdChars ; an LCD
  ldi ZH,HIGH(2*LcdStart) ; Start-Text ausgeben
  ldi ZL,LOW(2*LcdStart)
  rcall LcdText
  ; Startwert fuer Flagge
  clr rFlag
  ; Alle Inhalte entleeren
  rcall Neustart ; Neustartwerte setzen
  ; Timer 0 initiieren
  ldi rmp,(1<<CS02)|(1<<CS00) ; Prescaler 1024
  out TCCR0B,rmp ; Timer starten
  ; Timer 1 initiieren, freilaufend durch 8
  ldi rmp,1<<CS11 ; Prescaler durch 8
  out TCCR1B,rmp ; an Kontrollregister B
  ; PCINT0 fuer IR-Rx
  ldi rmp,1<<PCINT0 ; Pin 0 Level change INTs
  out PCMSK0,rmo ; in Maske 0
  ldi rmp,1<<PCIE0 ; Interrupt PCINT0 enable
  out GIMSK,rmp
  ; Sleep Enable
  ldi rmp,1<<SE ; Sleep Mode Idle
  out MCUCR,rmp ; in MCU Kontrollregister
  ; Interrupts ermoeglichen
  sei ; I-Flagge Statusregister
Schleife:
  sleep ; schlafen legen
  nop ; aufwachen
  sbrc rFlag,bEin ; Eingangs-Flagge auswerten
  rcall Eingang ; Eingangs-Flagge behandeln
  sbrc rFlag,bUpd ; Update-Flagge auswerten
  rcall Update ; Flagge behandeln
  rjmp Schleife ; schlafen legen
;
; Eingangssignal behandeln
; wird von der bEin-Flagge ausgeloest, die Sig-
; naldauer steht im Y-Doppelregister
;
; Je nach Signaldauer wird diese im SRAM-Puffer
; im Kopfbereich abgelegt (Adresse in Z) oder
; als Datenbits behandelt.
; Liegen von der Dauer her Datenbits vor, wer-
; den diese als Nullen und Einsen in ein acht
; Byte (64 Bit) breites Register eingerollt.
;
Eingang:
  cbr rFlag,bEin ; Eingangsflagge loeschen
  mov XH,YH ; Zaehlerstand kopieren
  mov XL,YL
  ; Timer ruecksetzen und starten
  ldi rmp,0 ; Timer 0 an den Anfang
  out TCNT0,rmp
  ldi rmp,1<<TOIE0 ; Timer-Ints ermoeglichen
  out TIMSK0,rmp
  tst XH ; MSB groesser Null?
  breq Eingang3 ; MSB = Null
  ; MSB ist groesser Null
  cpi XH,cIRStart ; MSB groesser Startbedingung
  brcs Eingang1
  rcall Neustart
Eingang1:
  inc rKopf ; naechstes Kopfwort
  cpi rKopf,4 ; mehr als drei Worte?
  brcs Eingang2
  sbr rFlag,1<<bKOf ; Kopf-Ueberlaufflagge
  ret
Eingang2:
  ldi ZH,HIGH(Kopf)
  ldi ZL,LOW(Kopf)
  lsl rKopf
  add ZL,rKopf
  ldi rmp,0
  adc ZH,rmp
  lsr rKopf
  st Z+,XH ; in Kopfspeicher
  st Z,XL
  ret
Eingang3:
.if cAktivHi == 1
  sbrc rFlag,bPol ; Polaritaet Signal
  .else
  sbrs rFlag,bPol ; Umgekehrte Polaritaet
  .endif
```

```
  rjmp Eingang5 ; Low-Signal/High-Signal
  ; Eingang war aktiv, Datenbits sammeln
  ldi rmp,cIRHigh ; Low-High-Schwelle
  cp rmp,XL ; kl: C clear, gr: C set
  rol rShift0 ; rolle Carry in Register
  rol rShift1
  rol rShift2
  rol rShift3
  rol rShift4
  rol rShift5
  rol rShift6
  rol rShift7
  inc rData
  cpi rData,66 ; 66 Bits zulaessig
  brcs Eingang4
  sbr rFlag,1<<bDOf ; Overflow-Flagge setzen
Eingang4:
  ; zu Summenregister addieren
  ldi ZH,HIGH(StatistikNull)
  ldi ZL,LOW(StatistikNull)
  cp rmp,XL ; noch mal Ergebnis ins Carry
  brcc Eingang6
  ldi ZH,HIGH(StatistikEins)
  ldi ZL,LOW(StatistikEins)
  rjmp Eingang6
Eingang5:
  ; Eingang ist umgekehrt polar
  ldi ZH,HIGH(Statistikpause)
  ldi ZL,LOW(Statistikpause)
Eingang6:
  ldd rmp,Z+1 ; LSB laden
  add rmp,XL ; LSB addieren
  std Z+1,rmp ; und speichern
  ld rmp,Z ; MSB laden
  adc rmp,XH ; MSB addieren
  st Z,rmp ; und speichern
  adiw ZL,2 ; auf Anzahl
  ld rmp,Z ; Anzahl lesen
  inc rmp ; erhoehen
  st Z,rmp ; und speichern
  ret
;
; Update-Flagge behandeln
Update:
  cbr rFlag,1<<bUpd ; Flagge wieder abschalten
  clr rmp ; Interrupts Timer aus
  out TIMSK0,rmp ; in Interruptmaske
  ldi rmp,0x01 ; Display loeschen
  rcall LcdC4Byte
  ldi ZH,HIGH(2*LcdMaske) ; Maske ausgeben
  ldi ZL,LOW(2*LcdMaske)
  rcall LcdText
  ; Kopfdaten anzeigen
  ldi ZH,0 ; Kopfdaten ausgeben
  ldi ZL,5
  rcall LcdPos
Update1:
  ldi ZH,HIGH(Kopf)
  ldi ZL,LOW(Kopf)
  clr R0 ; Pegel-Zeichen
Update2:
  ldi rmp,0x01 ; Pegelzeichen umkehren
  eor R0,rmp
  mov rmp,R0
  rcall LcdD4Byte ; auf LCD ausgeben
  ld rmp,Z+ ; Lese MSB
  rcall Hex2Lcd ; an LCD
  ld rmp,Z+ ; Lese LSB
  rcall Hex2Lcd ; an LCD
  cpi ZL,LOW(Kopfende)
  brcs Update2
  sbrs rFlag,bKOf
  ldi ZH,0
  ldi ZL,17
  rcall LcdPos
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,'U'
  rcall LcdD4Byte
  ldi rmp,'!'
  rcall LcdD4Byte
  ; Summen und Anzahlen ausgeben
Update3:
  ldi ZH,1 ; Zeile 2
  ldi ZL,2
  rcall LcdPos
  ldi ZH,HIGH(StatistikNull)
  ldi ZL,LOW(StatistikNull)
  ld rmp,Z+
  rcall Hex2Lcd
  ld rmp,Z+
  rcall Hex2Lcd
  ldi rmp,' '
  rcall LcdD4Byte
  ld rmp,Z+
  rcall Hex2Lcd
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,'1'
  rcall LcdD4Byte
  ldi rmp,':'
  rcall LcdD4Byte
  ld rmp,Z+
  rcall Hex2Lcd
  ld rmp,Z+
  rcall Hex2Lcd
  ldi rmp,' '
  rcall LcdD4Byte
  ld rmp,Z+
  rcall Hex2Lcd
  ldi ZH,2
  ldi ZL,2
  rcall LcdPos
  ldi ZH,HIGH(Statistikpause)
  ldi ZL,LOW(Statistikpause)
  ld rmp,Z+
  rcall Hex2Lcd
  ld rmp,Z+
  rcall Hex2Lcd
  ldi rmp,' '
  rcall LcdD4Byte
  ld rmp,Z+
  rcall Hex2Lcd
  ; Schieberegister von oben ausgeben
  ldi ZH,3
  ldi ZL,2
  rcall LcdPos
  ldi ZH,0
  ldi ZL,10
Update4:
  ld rmp,-Z ; Byte lesen
  rcall Hex2Lcd ; und in Hex ausgeben
  cpi ZL,3
  brcc Update4
  sbrs rFlag,bDOf ; Overflow?
  rjmp Update5
  ldi rmp,'U'
  rcall LcdD4Byte
  ldi rmp,'!'
  rcall LcdD4Byte
Update5:
  ldi rmp,0x0C ; Cursor und Blink aus
  rcall LcdC4Byte ; an LCD
;
; Neustart
; wird beim Init, bei langen Pausen und im
; Anschluss an die LCD-Ausgabe aufgerufen
;
; Startet die Flaggen und leert den SRAM-
; Puffer.
;
Neustart:
  ser rKopf ; auf Anfangswerte
  clr rData
```

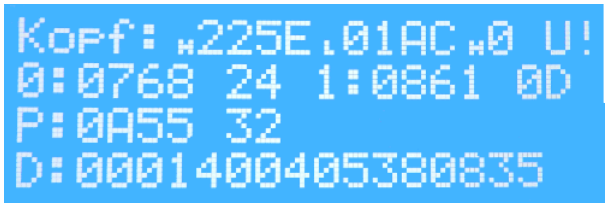
```
  cbr rFlag,(1<<bKOf)|(1<<bDOf) ; Flaggen loeschen
  ldi ZH,0 ; Register loeschen
  ldi ZL,2
Neustart1:
  st Z+,rData ; Register loeschen
  cpi ZL,10
  brcs Neustart1
  ldi ZH,HIGH(Kopf) ; SRAM loeschen
  ldi ZL,LOW(Kopf)
Neustart2:
  st Z+,rData
  cpi ZL,LOW(Statistikende)
  brcs Neustart2
  ret
;
; Byte in rmp in Hex an LCD
;   Eingabedaten: Register rmp
;   Ausgabe HH an LCD an aktueller Position
;
Hex2Lcd:
  push rmp ; Byte retten
  swap rmp ; oberes Nibble zuerst
  rcall HexNibble2Lcd
  pop rmp ; rmp wieder herstellen
; Gib Nibble in Hex auf LCD aus
HexNibble2Lcd:
  andi rmp,0x0F ; unteres Nibble maskieren
  subi rmp,-'0' ; ASCII-Null dazu addieren
  cpi rmp,'9'+1 ; A bis F?
  brcs HexNibble2Lcd1 ; nein
  subi rmp,-7 ; auf A bis F
HexNibble2Lcd1:
  rjmp LcdD4Byte ; rmp auf LCD ausgeben
;
; Starttext LCD
LcdStart:
.db "IR-Analyse ATtiny24 ",0x0D,0xFF
.db " gsc-elektronic.net ",0x0D,0xFF
.db "Messungen IR-Signal ",0x0D,0xFF
.db " in Hex mal 8 us    ",0xFE
;
; Ausgabemaske fuer Messdaten
LcdMaske:
.db "Kopf:",0x0D
;         5
.db "0:xxxx nn 1:xxxx nn ",0x0D,0xFF
;       2
.db "P:xxxx nn",0x0D
;       2
.db "D:xxxxxxxxxxxxxxxx ",0xFE
;       2
;
; Spezialzeichen
Codezeichen:
.db 64,0,0,0,0,0,4,4,6,0 ; Z = 0, Low-Signal
.db 72,0,0,0,0,0,5,7,5,0 ; Z = 1, High-Signal
.db 0,0 ; Ende der Tabelle
;
; LCD-Routinen einlesen
.include "Lcd4Busy.inc"
;
; Ende Quelltext
;
```

Neue Instruktion hier ist

- SER Register: Setzt alle Bits im Register Eins (R16 .. R31).

12.4.6.2 Beispiele

Das hier ist das Ergebnis dieser Software am Beispiel meiner TV-Fernsteuerung. Die produziert mehr als drei Kopfsignale ("U!" in der ersten Zeile rechts (aber dafür haben wir ja ein anderes Werkzeug. Aus den Summenwerten von Nullen und Einsen hinter "0:" bzw. "1:" in Kombination mit der Anzahl an ihrem Vorkommen (bei den Nullen z. B. 0x24 mal) lassen sich Durchschnitte errechnen.



Noch viel hilfreicher ist die Datenbit-Auswertung hinter "D:". Hier lassen sich die Datenbits der einzelnen Tasten der Fernbedienung in Hexadezimalform ablesen.

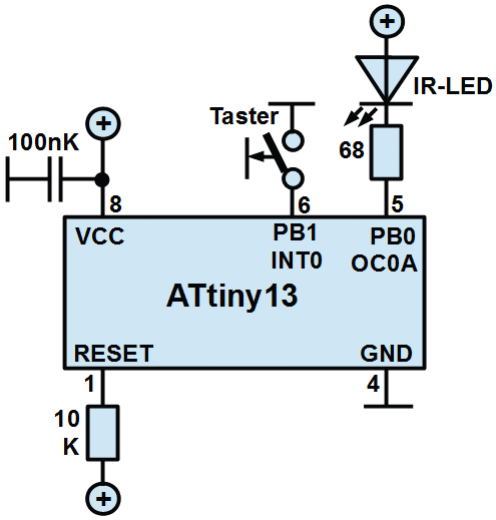
Das hier ist so eine Tabelle mit den Tastenzuordnungen der TV-Fernsteuerung. Was sich der Entwickler dabei gedacht haben mag, erschließt sich dem normalverständigen Bürger nicht so ganz.

Ich wünsche jedenfalls etwas mehr Glück beim Erkunden der eigenen schwarzen Kästchen, alles nötige Handwerkszeug ist jetzt dafür vorhanden.

TV-Gerät	
Hex	Taste
4004.0538.0835	1
4004.0538.88B5	2
4004.0538.4875	3
4004.0538.C8F5	4
4004.0538.2815	5
4004.0538.A895	6
4004.0538.6855	7
4004.0538.E8D5	8
4004.0538.1825	9
4004.0538.98A5	0
0000.E0E0.E01F	Lautstärke erhoehen
0000.E0E0.D02F	Lautstärke niedriger
4004.0520.2C09	Programm erhoehen
4004.0520.AC89	Programm niedriger
4004.0544.6322	OK

12.5 Ein IR-Sender

Damit wir nun nicht nur vorhandene Fernsteuerungen nutzen können, sondern auch mal selber Signale erzeugen können, bereichern wir die Fernsteuerwelt mit einer eigenen Steuerung und unserer eigenen Norm. Da wir aus den letzten Experimenten noch einen ATtiny13 haben, machen wir das mit dem.



12.5.1 Schaltbild zum IR-Senden

Das hier ist es schon alles, was wir brauchen: den ATtiny13, einen Taster um das Signal zu starten und eine Infrarot-LED. Wir verzichten darauf, die IR-Sendediode mit voller Leistung (mit 100 mA) anzusteuern, weil wir damit den auch für die ISP-Programmierung verwendeten Pin völlig übersteuern würden. Das senkt zwar die Reichweite, ist aber für das Experiment hinnehmbar. Wer mit voller Power senden möchte, schaltet einen PNP-Transistor als Treiberstufe dazwischen (NPN würde die Polarität umkehren, was erhebliche Konsequenzen für das Programm hätte).

12.5.2 Die IR-LED

Das hier ist eine Infrarot-LED. Sie hat eine Durchlassspannung von 1,35 V bei einem Nennstrom von 100 mA. Der Vorwiderstand bei 5 V ergibt sich bei den maximal sinnvollen 73 mA Maximalstrom, die ein ATtiny13-Ausgang in Sink-Schaltung liefert, damit zu

R [Ohm] = 1000 * (5 - 1,35 - 2 [V]) / 73 [mA] = 23 [Ohm]

Wenn man einen Transistortreiber zwischenschaltet, ergibt sich ein Vorwiderstand von

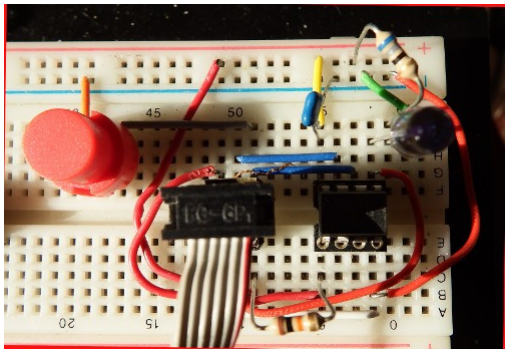
R [ohm] = 1000 * (5 - 1,35 - 0,2 [V]) / 100 [mA] = 35 [Ohm]

12.5.3 Der 68 Ohm-Widerstand

Das ist der 68 Ω-Widerstand. Mit ihm beträgt der IR-LED-Strom

I [mA] = 1000 * (5 - 1,35 - 1,0 [V]) / 68 = 39 [mA]

12.5.4 Aufbau



Das hier ist der einfache Aufbau der Hardware.

12.5.5 Das IR-Sendesignal

Für das Senden bietet sich der Timer im ATtiny13 an. Er schaltet bei Erreichen der programmierten Obergrenze die LED am OC0A-Ausgang im Takt ein und aus.

Das gesamte Timing der Software baut sich zentral auf der Erzeugung dieses 40 kHz-IR-Signals auf. Um diese mittels des 8-Bit-Timers im CTC-Modus erzeugen zu können, muss der Ausgang OC0A bei aktivem Sendesignal im 40 kHz-Takt torkeln. Der TIMO-COMPA-Interrupt kann dazu dienen, durch Zählen der erreichten Anzahl Pegelwechseltakte die Dauer der aktiven und inaktiven Signale zu kontrollieren. Da im Falle der langen Kopfsignale eine sehr große Anzahl an Durchläufen zu zählen sind, muss dies ein 16-Bit-Zähler sein.

Bei 1,2 MHz dauert ein An/Aus-Signal von 40 kHz 30 Prozessortakte, für jedes der beiden Signalteile folglich 15 Takte. Mit dem Auslösen des COMPA-Interrupts sind vier Takte erforderlich (Ablegen des Programmzählers auf dem Stapel), für den Sprung aus der Vektortabelle in die Interrupt-Service-Routine sind zwei weitere Takte erforderlich. Die weiteren Schritte in der Service-Routine wären folgende:

```
TC0CAIsr:
        in R15,SREG ; Retten Status
        sbiw R24,1 ; Zaehler abwaerts
        brne TC0CAIsrRet ; noch nicht null
        ; [... weiteres ...]
TC0CAIsrRet:
        out SREG,R15 ; Status herstellen
        reti ; fertig, zurueck
```

Das Timing dieser Routine stellt sich folgendermaßen dar:

```
TC0CAIsr: ; 4 fuer Int, 2 fuer Vektorsprung = 6
        in R15,SREG ; +1 = 7
        sbiw R24,1 ; +2 = 9
        brne TC0CAIsrRet ; +2 = 11 bei Sprung
        ; [... weiteres ...]
TC0CAIsrRet: ; 11 Takte
        out SREG,R15 ; +1 = 12
        reti ; +4 = 16
```

Damit dauert die ISR 16 Takte lang. Damit steht bereits der nächste COMPA-Interrupt an, bevor der derzeitige fertig bearbeitet ist. Für irgendetwas anderes steht weder innerhalb noch außerhalb der Service Routine Zeit zur Verfügung. Mit der Default-Taktrate des ATtiny13 funktioniert dies nicht.

Eine Taktrate von 2,4 Mhz, wie sie durch Beschreiben des CLKPR-Ports möglich ist, ist jedoch ausreichend. Selbst wenn in der Service Routine auch noch Flaggen gesetzt werden und die Torkel-Einstellung umgestellt wird (von Torkeln auf Setzen des Ausgangssignals und zurück), bleibt bis 30 Takte noch genug Luft.

12.5.6 Kontrolle der Signaldauer

Diese erfolgt sinnvollerweise mit dem Registerpaar R25:R24, da dies mit SBIW abwärts gezählt

aber nicht als Zeiger verwendet werden kann.

Jeder Takt dieses Zählers entspricht bei 40 kHz 12,5 µs. Für ein Wartesignal vor Beginn der Übertragung stehen maximal 819,2 ms zur Verfügung. Die Mikrosekunden sind mit 2 zu multiplizieren und durch 25 zu teilen, da in Assembler keine Fließkommaoperationen sinnvoll und zulässig sind.

Während der Abarbeitung des Kopfes sind im Prinzip beliebig lange Aktiv- und Inaktivdauern möglich. Damit dies möglich ist, muss eine flexible Konstruktion gewählt werden, um die festgelegte Dauer den einzelnen Bytes zuzuordnen. Damit bei acht denkbaren unterschiedlichen Kopfdauern die Abfrage, welche der Dauern jetzt dran ist, nicht allzu umfangreich wird, wird hier ein berechneter Sprung gewählt. Das ermöglicht die Instruktion IJMP. Sie bewirkt, dass die Programmadresse aus dem Z-Registerpaar geladen wird. Das ermöglicht es, Sprungadressen zu berechnen und relativ rasch an die Zieladresse zu verzweigen.

12.5.7 Programm zum Senden

Das folgende Programm kann als flexible Basis für eigene Formulierungen und Formate verwendet werden. Es lässt sich mit geringem Aufwand an nahezu jeden Bedarf anpassen. Es werden pro Tastendruck jeweils zwei Mal die Sequenz gesendet, um die korrekte Dauer der langen Pause zu Beginn des Kopfes messen zu können.

Beim ISP-Brennen ist die IR-Leuchtdiode oder der Widerstand zeitweise zu entfernen, da das Programmiergerät den hohen Strom nicht schafft und Fehler meldet.

Das Programm ist hier (zum Quelltext im asm-Format geht es hier).

```
;                                       .def rCntH = R25
; ********************************      ; benutzt: XH:XL R27:R26
; * IR-Sender 40 kHz mit ATtiny13  *     ; frei: YH:YL R29:R28
; * (C)2016 by www.gsc-elektronic.net *  ; benutzt: ZH:ZL R31:R30 Daten aus Flash
; ********************************      ;
;                                       ; -------- Konstanten und Timing ------
.NOLIST                                 .equ cTakt = 2400000 ; Prozessortakt
.INCLUDE "tn13def.inc"                  .equ cTaktNs = 1000000000 / cTakt ; Takt ns
.LIST                                   .equ cIRF = 40000 ; IR-Sendefrequenz Hz
;                                       .equ cIRNs = 1000000000 / cIRF / 2 ; IR-TX ns
; -------- Programmablauf -------       .equ cCtc = (cIRNs+cTaktNs/2) / cTaktNs - 1
;                                       ; 9,6 MHz: 120; 4,8 MHz: 60; 2,4 MHz: 30
; Sendet ueber die IR-LED eine einstellbare  ; 1,2 MHz: 15(!!), schneller als ISR!!
; Anzahl von Kopfbytes (deren Dauer ein-  ;
; stellbar ist) und danach eine einstellbare  ; -------- Aufbau IR-Signal -----------
; Anzahl von Datenbits (deren Dauer bei Eins  ; Alle Zeitangaben in us
; oder Null waehlar ist), getrennt durch  ; Anzahl Kopfbytes, 1 .. 4
; Pausen, deren Dauer ebenfalls einstellbar  .equ nKopf = 4 ; Anzahl Kopfpaare H/L, 1..4
; ist.                                  .equ cKopf = 2*nKopf-1 ; Anzahl Kopf-Signale
;                                       ; Dauer der Kopfsignale (Dauer in 2*us/25)
; -------- Register -------------       .equ cTK1 = 2*50000/25 ; Dauer High 1
; frei: R0 .. R5                        .equ cTK2 = 2*5000/25 ; Dauer Low 1
.def rData0 = R6 ; Senderegister, hoechstes  .equ cTK3 = 2*2500/25 ; Dauer High 2
.def rData1 = R7 ; dto., naechstniedriges  .equ cTK4 = 2*500/25 ; Dauer Low 2
.def rData2 = R8 ; dto., naechstniedriges  .equ cTK5 = 2*1250/25 ; Dauer High 3
.def rData3 = R9 ; dto., naechstniedriges  .equ cTK6 = 2*500/25 ; Dauer Low 3
.def rData4 = R10 ; dto., naechstniedriges  .equ cTK7 = 2*1250/25 ; Dauer High 3
.def rData5 = R11 ; dto., naechstniedriges  .equ cTK8 = 2*500/25 ; Dauer Low 3
.def rData6 = R12 ; dto., naechstniedriges  .equ cTK9 = 2*1250/25 ; Dauer High 4
.def rData7 = R13 ; Senderegister, niedrigstes  .equ cTK10 = 2*500/25 ; Dauer Low 4
.def rEor = R14 ; fuer Polaritaetsumkehr Toggle  ; Anzahl zu sendender Datenbits
.def rSreg = R15 ; Statusregister       .equ cBits = 64 ; Anzahl Bits, 1 .. 64
.def rmp = R16 ; Vielzweck              .equ cSign = cKopf + 2*cBits ; Anzahl Signale
.def rimp = R17 ; Vielzweck innerhalb Int  ; Dauer von Datenbits und Aktivperiode
.def rFlag = R18 ; Flaggenregister      .equ cKurz = 2*250/25 ; Dauer Binaer-Null
  .equ bRun = 0 ; Senden laeuft         .equ cLang = 2*750/25 ; Dauer Binaer-Eins
  .equ bSta = 1 ; Senden starten        .equ cPaus = 2*250/25 ; Dauer Aktivperiode
  .equ bTTo = 2 ; Timeout beim Senden   ;
  .equ bRst = 3 ; Zweite Sendung        ; -------- Sendeinformationen ------
.def rTOC = R19 ;  Timeout-Zaehler, zaehlt Signale; Testdaten zum Senden
; frei R20 .. R23                       .equ cWort1 = 0xAAAA ; Hoechtwertigst
.def rCntL = R24 ; Zaehlt CTC-Compares  .equ cWort2 = 0x5555
```

```
.equ cWort3 = 0xAAAA
.equ cWort4 = 0x5555 ; Niederwertigst
;
; -------- Reset- und Interruptvektoren ---
.CSEG
.ORG 0
  rjmp Start ; Programm-Init
  rjmp Int0Isr ; INT0 Tasteninterrupt
  reti ; PCINT0 Pin Change Int Request 0
  reti ; TIM0_OVF Timer/Counter Overflow
  reti ; EE_RDY EEPROM Ready
  reti ; ANA_COMP Analog Comparator
  rjmp TC0CAIsr ; TIM0_COMPA TC0 Compare A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; WDT Watchdog Time-out
  reti ; ADC Conversion Complete
; -------- Interrupt Service Routinen -----
;
; INT0-Interrupt
; wird von der fallenden Flanke des Tasten-
; drucks ausgeloest.
; Falls der Sendeablauf inaktiv ist (Flagge
; bRun ist Null), werden die Flaggen bRun
; und bSta gesetzt.
;
Int0Isr: ; Tasteninterrupt, startet Senden
  sbrc rFlag,bRun ; laeuft noch nicht
  reti ; doch, nicht neu starten
  in rSreg,SREG ; SREG retten
  sbr rFlag,(1<<bRun)|(1<<bSta) ; Flaggen
  out SREG,rSreg ; SREG wieder herstellen
  reti ; zurueck
;
; CTC-Timeout Timer: IR-LED schalten
; wird vom Compare Match A von TC0 ausgeloest
;
; Zaehlt den 16-Bit-Zaehler R25:R24 abwaerts.
; Ist Null erreicht, wird das Toggle/Clear-Bit
; des Ausganges umgedreht und die bTTO-Flagge
; gesetzt. Damit es schnell geht, erfolgt das
; Umdrehen des Toggle-Zustands statt mit be-
; dingtem Sprung mit einem Exklusiv-Oder mit
; einem Register.
; Die Toggle/Clear-Umschaltung mit den COM0A-
; Bits geht folgendermassen:
;
; Output Mode | COM0A1 | COM0A0
; ------------+--------+-------
; Toggle OC0A |   0    |   1
; Clear OC0A  |   1    |   1
;
; Ein bedingter Sprung wuerde so gehen:
;    sbic TCCR0A,COM0A1 ; 1 oder 2 Takte
;    rjmp Gesetzt ; + 2 = 3 Takte
;    sbi TCCR0A,COM0A1 ; + 2 = 4 Takte
;    rjmp Weiter ; + 2 = 6 Takte
;   Gesetzt: ; 3 Takte
;    cbi TCCR0A,COM0A1 ; + 2 = 5 Takte
;   Weiter: ; 5 oder 6 Takte
; Braeuchte also 5 oder 6 Takte.
;
; Legt man hingegen 1<<COM0A1 in ein
; Register ab (hier rEor genannt) und
; Exklusiv-Oder-t TCCR0A mit diesem Register,
; dann torkelt das COM0A1-Bit auch: aus
; Null wird Eins und umgekehrt. Das braucht
; folgende Zeiten:
;    in rimp,TCCR0A ; 1 Takt
;    eor rimp,rEor ; + 1 = 2 Takte
;    out TCCR0A,rimp ; + 1 = 3 Takte
; Das EOR spart also zwei bis drei Takte,
; kostet dafuer aber ein Register (rEor).
;
TC0CAIsr: ; 6 Takte f. Int und Vektor-rjmp
    in rSreg,SREG ; SREG retten, 7
    sbiw rCntL,1 ; CTC-Zaehler abwaerts, 9
    brne Tc0CAIsrRet ; Nicht Null, 10/11
    sbr rFlag,1<<bTTO ; Flagge setzen, 11
    in rimp,TCCR0A ; Toggle-Zustand, 12
    eor rimp,rEor ; umkehren, 13
    out TCCR0A,rimp ; neuer Toggle, 14
TC0CAIsrRet: ; 11/14
    out SREG,rSreg ; SREG herstellen, 12/15
    reti ; fertig, 16(!!)/19
;
; --------- Programmstart, Init -------
Start:
  ; Stapel einrichten
  ldi rmp,LOW(RAMEND)
  out SPL,rmp
  ; Takt auf 2,4 MHz hochsetzen
  ldi rmp,1<<CLKPCE ; CLK-Enable
  out CLKPR,rmp
  ldi rmp,1<<CLKPS1 ; Teiler = 4
  out CLKPR,rmp
  ; LED-Port initiieren
  sbi PORTB,PORTB0 ; LED aus
  sbi DDRB,DDB0 ; Pin ist Ausgang
  ; Tastenport mit Pull-Up
  sbi PORTB,PORTB1 ; Pullup an
  ; Startwerte setzen
  clr rFlag ; Flaggen loeschen
  ldi rmp,1<<COM0A1 ; Toggle-Umkehr-Bit
  mov rEor,rmp ; in Exor-Register
  ; TC0 als CTC starten
  ldi rmp,cCtc ; Compare A Wert 12,5us
  out OCR0A,rmp
  ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)
  out TCCR0A,rmp ; Ausgang high, CTC
  ldi rmp,1<<CS00 ; Vorteiler = 1
  out TCCR0B,rmp
  ; INT0 und Schlafmodus
  ldi rmp,(1<<SE)|(1<<ISC01) ; Schlafen
  out MCUCR,rmp ; und INT0
  ldi rmp,1<<INT0 ; INT0-Interrupt
  out GIMSK,rmp
  ; Interrupts
  sei ; Interrupts ermoeglichen
  ; fuer Simulation
  cbi PORTB,PORTB1 ; INT0 ausloesen
  sbi PORTB,PORTB1
;
; -------- Programmschleife ----------
Schleife:
  sleep ; schlafen legen
  nop ; nach Aufwachen
  sbrc rFlag,bTTo ; Timeout?
  rcall Timeout ; Timeout bearbeiten
  sbrc rFlag,bSta ; Startflagge
  rcall StartOut ; Senden starten
  rjmp Schleife ; weiter schlafen
;
; -------- CTC-Timeout ---------------
;
; Routine Timeout
; wird von der bTTo-Flagge ausgeloest
; (Senden beendet)
;
; Stellt zunaechst fest, ob noch Kopfdaten ge-
; sendet werden muessen oder schon Datenbits
; dran sind.
; Sind Kopfdaten zu senden (rTOC ist kleiner
; als cKopf), wird die Dauer des zu sendenden
; Kopfsignals durch Sprung in die entspechende
; Setzroutine mit IJMP geladen.
; Sind Datenbits zu senden, wird geprueft, ob
; schon alle Datenbits und Pausen gesendet sind.
; Wenn nein, wird geprueft ob zuletzt ein Daten-
; oder ein Pausenbit gesendet wurde. War es ein
; Datenbit, wird die Standard-Pausendauer cPaus
; geladen. War es eine Pause wird durch Schieben
; und Rotieren das naechste zu sendende Bit in
; das Carry-Flag geladen. Daraus wird ermittelt
; ob ein langes oder ein kurzes Signal zu senden
```

```
; ist.
; Sind alle Bits gesendet, wird der Ablauf neu
; gestartet (wenn das bRst-Bit noch nicht gesetzt
; war). Ist es gesetzt, wird alles abgeschaltet
; und auf einen Neustart durch den Tastendruck
; gewartet.
;
TimeOut: ; Flagge war gesetzt
  cbr rFlag,1<<bTTo ; Flagge loeschen, 1
  inc rTOC ; naechstes Zeichen, 2
  cpi rTOC,cKopf ; noch Kopf senden?, 3
  brcc TimeOut1 ; nein, Datenbits, 4/5
  ldi ZH,HIGH(TOK2) ; MSB Adresse Kopf, 6
  ldi ZL,LOW(TOK2) ; dto, LSB, 7
  mov rmp,rTOC ; 8
  lsl rmp ; mal zwei, 9
  add rmp,rTOC ; mal drei, 10
  add ZL,rmp ; zu Adresse, 11
  ldi rmp,0 ; Ueberlauf, 12
  adc ZH,rmp ; addieren, 13
  ijmp ; Sprung nach Z, 15
TimeOut1: ; 5
  cpi rTOC,cSign ; schon alle gesendet?, 6
  brcc TimeOutEnd ; ja, beenden, 7/8
  in rmp,TCCR0A ; Toggle/Set lesen, 8
  sbrc rmp,COM0A1 ; Toggle ein pruefen, 9/10
  rjmp TimeOut2 ; Toggle ist aus, 11
  ; Toggle ist an
  ldi rCntH,High(cPaus) ; lade Pausendauer, 11
  ldi rCntL,Low(cPaus) ; 12
  ret ; fertig, 16
TimeOut2: ; 11
  ldi rCntH,High(cKurz) ; lade kurzes Bit, 12
  ldi rCntL,Low(cKurz) ; 13
  lsl rData7 ; schiebe Bits, 14
  rol rData6 ; 15
  rol rData5 ; 16
  rol rData4 ; 17
  rol rData3 ; 18
  rol rData2 ; 19
  rol rData1 ; 20
  rol rData0 ; 21
  brcc TimeOut3 ; Bit ist Null, 22/23
  ldi rCntH,High(cLang) ; Dauer Eins, 23
  ldi rCntL,Low(cLang) ; 24
TimeOut3: ; 23/24
  ret ; fertig, 23/24
TimeOutEnd: ; 8
  sbrs rFlag,bRst ; Restart-Flagge?, 9/10
  rjmp Restart ; noch einmal beginnen, 11
  clr rmp ; Timer-Int aus, 11
  out TIMSK0,rmp ; 12
  ; Ausgang OCR0A auf Eins
  ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01) ; 13
  out TCCR0A,rmp ; 14
  cbr rFlag,(1<<bRun)|(1<<bRst) ; Flaggen
loeschen, 15
  ret ; fertig, 19
;
; Timeout Kopf, Dauer einstellen
; (Alle Routinen muessen drei Worte lang sein, da
;  die Zieladresse berechnet wird!)
;
TOK2: ; 15
  ldi rCntH,High(cTK2) ; 16
  ldi rCntL,Low(cTK2) ; 17
  ret ; 21
;
  ldi rCntH,High(cTK3)
  ldi rCntL,Low(cTK3)
  ret
;
  ldi rCntH,High(cTK4)
  ldi rCntL,Low(cTK4)
  ret
;
  ldi rCntH,High(cTK5)
  ldi rCntL,Low(cTK5)
  ret
;
  ldi rCntH,High(cTK6)
  ldi rCntL,Low(cTK6)
  ret
;
  ldi rCntH,High(cTK7)
  ldi rCntL,Low(cTK7)
  ret
;
  ldi rCntH,High(cTK8)
  ldi rCntL,Low(cTK8)
  ret
;
; Restart: Lauf noch einmal starten
; wird nach Abschluss des ersten Durchgangs
; aufgerufen und startet die Ausgabe neu
;
Restart: ; 11
  clr rmp ; Timer-Int ausschalten, 12
  out TIMSK0,rmp ; 13
  sbr rFlag,1<<bRst ; Flagge setzen, 14
  rjmp StartOut1 ; 16
;
; StartOut-Routine
; wird von der Flagge bRst ausgeloest und,
; ohne Flaggen-Loeschen, bei einem Restart
; angesprungen.
;
; Kopiert die Datentabelle aus dem Flash-
; Speicher in umgekehrter Richtung in die
; Senderegister von R14 abwaerts.
; Setzt den den Zaehler rToc auf 255 und
; laedt die Dauer der ersten Kopfpause
; (IR-LED aus) in das Doppelregister in
; R25:R24. Schaltet dann den Timer-Compare
; A Interrupt an.
;
StartOut:
  cbr rFlag,(1<<bSta)|(1<<bRst) ; Flagge loeschen
StartOut1:
  ldi ZH,High(2*Datentabelle)
  ldi ZL,Low(2*Datentabelle)
  ldi XH,0 ; in Register 13 abwaerts
  ldi XL,14
StartOut2:
  lpm rmp,Z+ ; Tabellenbyte lesen
  st -X,rmp ; in Register
  cpi XL,7 ; schon bei 6 angekommen?
  brcc StartOut2 ; nein, weiter
  ser rToC ; Zaehler auf 0xFF
  ldi rCntH,High(cTK1) ; erste Wartedauer
  ldi rCntL,Low(cTK1)
  ldi rmp,1<<OCIE0A ; Timer-Int ein
  out TIMSK0,rmp
  ret
;
; Datentabelle
Datentabelle:
.dw cWort4,cWort3,cWort2,cWort1
```

12.5.8 Analyse der gesendeten Signale

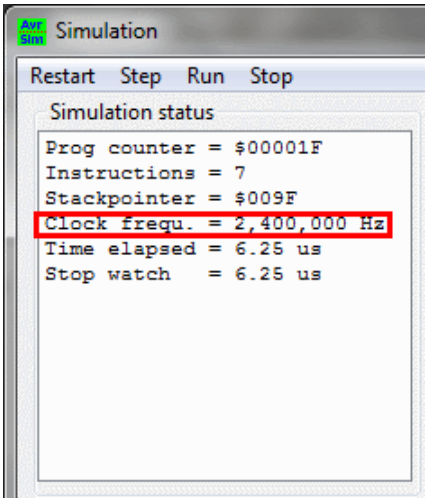
Das hier ist, was der ATtiny45 von den Signalen so mitbekommen hat. Die beiden ersten Kopfsignale sind mit 0x1901 und 0x0288 (51.208 und 5.184 µs) etwas zu lang. Dass die Anzahl der Kopfsignale bei der Senderoutine extrem groß ist, ist mit einem U! vermerkt.

Die Datensignale bei Nullen haben eine Summe von 0x036A (=874) für 32 Signale, für 32 Einsen von 0x0CC3 ergeben. Das entspricht durchschnittlich 219 µs für Nullen und 769 µs für Einsen. Kurze Nullsignale werden daher als etwas zu kurz, lange Eins-Signale und auch die sehr langen Kopfsignale als etwas zu lang gemessen.

Die Abweichungen sind aber zu verschmerzen. Die gesamte Paaranzahl wurde mit 68 gemessen.

Alle acht Datenbytes sind korrekt erkannt.

12.5.9 Simulation der Sendesignale

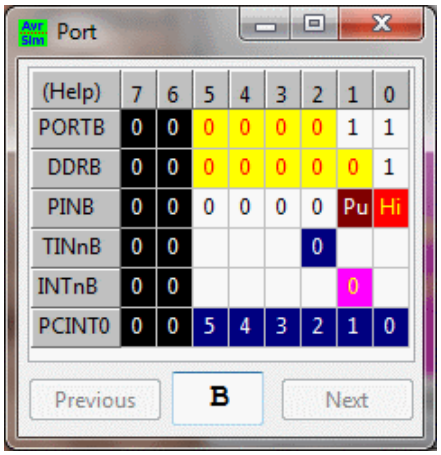


Im Folgenden werden die IR-Aussendungen simuliert mit avr_sim.

Nach dem Init des Stapelzeigers erhöhen die folgenden Instruktionen den Prozessortakt:

```
; Takt auf 2,4 MHz hochsetzen
ldi rmp,1<<CLKPCE ; CLK-Enable
out CLKPR,rmp
ldi rmp,1<<CLKPS1 ; Teiler = 4
out CLKPR,rmp
```

Die Änderung ist beim Simulator angekommen und der Prozessortakt steht jetzt bei 2,4 MHz.



Die Instruktionen

```
; LED-Port initiieren
sbi PORTB,PORTB0 ; LED aus
sbi DDRB,DDB0 ; Pin ist Ausgang
```

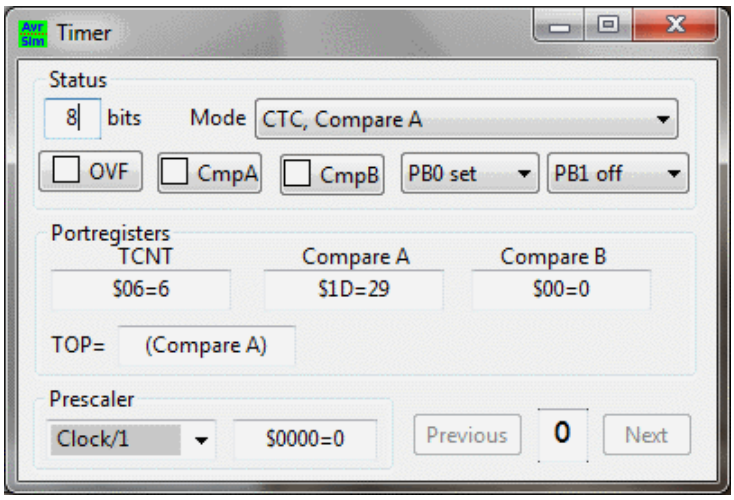
haben den IR-LED-Ausgang auf High geschaltet (LED aus) und ihn als Ausgang konfiguriert. Dann hat die Instruktion

```
; Tastenport mit Pull-Up
sbi PORTB,PORTB1 ; Pullup an
```

den Pullup-Widerstand am Tasteneingang eingeschaltet. Ferner haben die Instruktionen

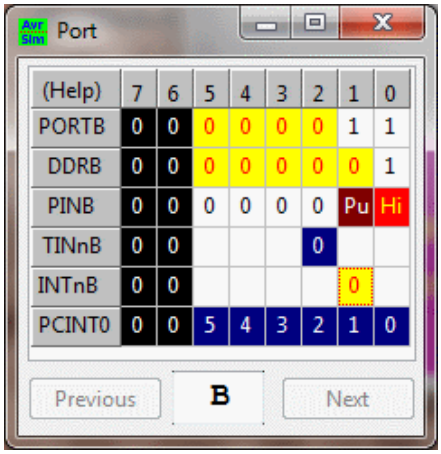
```
; INT0 und Schlafmodus
ldi rmp,(1<<SE)|(1<<ISC01) ; Schlafen
out MCUCR,rmp ; und INT0
ldi rmp,1<<INT0 ; INT0-Interrupt
out GIMSK,rmp
```

den INT0-Interrupt bei fallenden Flanken am INT0-Eingang (beim Drücken der Taste) aktiviert. Nebenbei ist noch der Schlafmodus Idle (mit Aufwachen der CPU beim Int) eingestellt.



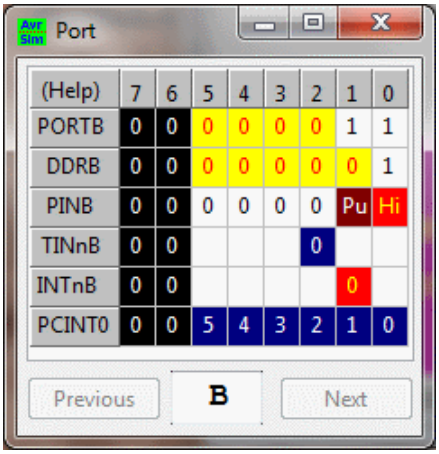
Der Timer/Counter 0 wurde als CTC (Clear TC bei Compare Match) mit 12,5µs Wiederholungszeit konfiguriert, natürlich mit einem Vorteilerwert von 1. Der PB0-Ausgang wird bei Compare Match auf Eins gesetzt, womit die IR-LED vorerst Aus bleibt. Die Befehlsfolge dazu lautete:

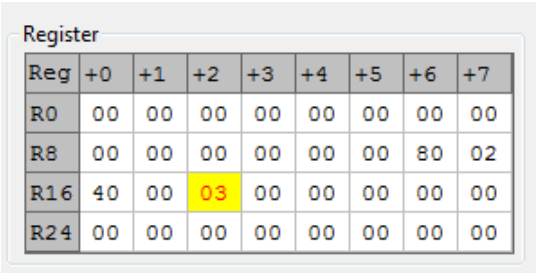
```
; TC0 als CTC starten
ldi rmp,cCtc ; Compare A =30
out OCR0A,rmp
ldi rmp,(1<<COM0A1)|
   (1<<COM0A0)|(1<<WGM01)
out TCCR0A,rmp ; Ausgang high
ldi rmp,1<<CS00 ; Vorteiler = 1
out TCCR0B,rmp
```



Durch Klicken auf INT0 in der Portdarstellung wird ein Tastendruck simuliert und der Tasteninterrupt wird angefordert.

Mit der nächsten Instruktion wird die INT0-Service-Routine ausgeführt.





Mit der Instruktion

```
sbrc rFlag,bRun ; laeuft noch nicht
reti ; doch, nicht neu starten
```

wird zunächst geprüft, ob der Ablauf schon begonnen hat. Wenn ja, kehrt der INT0 zurück.

Wenn nein, dann setzen die folgenden Instruktionen

```
in rSreg,SREG ; SREG retten
sbr rFlag,(1<<bRun)|(1<<bSta) ; Flaggen
```

nach der Rettung des Statusregisters (das nachfolgende *sbr* verändert Flaggen!) beide Flaggen, bRun und bSta, im Flaggenregister rFlag in R18 gemeinsam. Die weitere Behandlung der Flaggen erfolgt nach dem *sleep*, denn der Prozessor ist jetzt um den Schlaf gebracht.

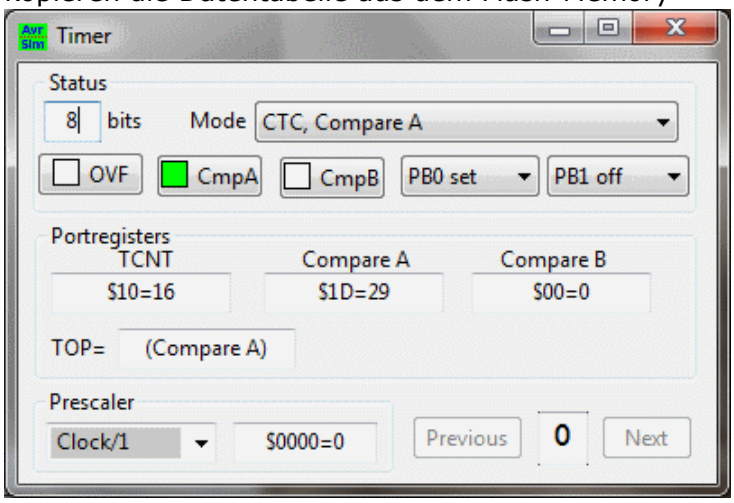
Register

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	AA	AA
R8	55	55	AA	AA	55	55	80	02
R16	AA	00	01	00	00	00	00	00
R24	00	00	06	00	00	00	1C	01

Die Instruktionen

```
        ldi ZH,High(2*Datentabelle)
        ldi ZL,Low(2*Datentabelle)
        ldi XH,0 ; in Register 13 abwaerts
        ldi XL,14
StartOut2:
        lpm rmp,Z+ ; Tabellenbyte lesen
        st -X,rmp ; in Register
        cpi XL,7 ; schon bei 6 angekommen?
        brcc StartOut2 ; nein, weiter
        ser rToC ; Zaehler auf 0xFF
```

kopieren die Datentabelle aus dem Flash-Memory




```
Datentabelle:
.dw cWort4,cWort3,cWort2,cWort1
```

rückwärts in die Register R13 bis R6.

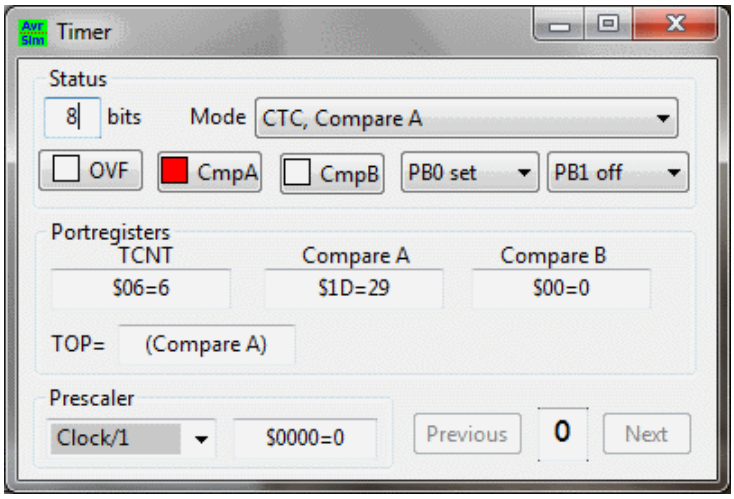
```
        ldi rCntL,Low(cTK1)
        ldi rmp,1<<OCIE0A ; Timer-Int ein
        out TIMSK0,rmp
```

hat den Compare-Match A Interrupt des Timers eingeschaltet. Noch ist aber der Ausgang PB0 inaktiv (High) und die IR-LED aus.

Der Interruptzähler wurde auf die inaktive Dauer des Kopfes gesetzt, was laut gavrasm-Symbol-Listing 4.000 ist:

```
  ldi rCntH,High(cTK1) ; erste Wartedauer
  ldi rCntL,Low(cTK1)
```

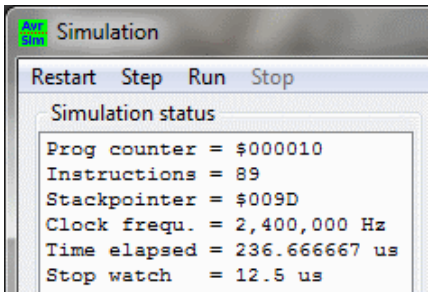
```
List of symbols:
Type nDef nUsed Decimalval  Hexvalue Name
  C     1      2         4000       FA0 CTK1
```



Das ist der Zähler, wenn der erste Interrupt ansteht. Der Zähler hat bereits weitergezählt, was durch die nötigen Schritte beim Interrupt verursacht ist (Ablegen der Ausführungsadresse auf den Stapel, Sprung zur Vektoradresse). Die Instruktionen

```
        sbiw rCntL,1 ; CTC-Zaehler
abwaerts
        brne Tc0CAIsrRet ; Nicht Null
```

zählen erst mal den Zähler in R25:R24 abwaerts und prüfen, ob der schon Null ist (Z-Flagge im SREG).



Wie vorhergeplant dauert die Zeit zwischen zwei Interrupts 12,5 µs.

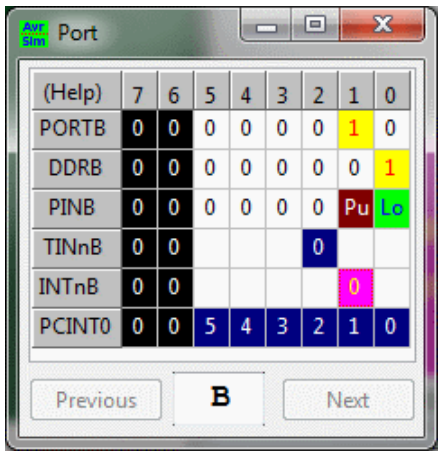
SREG

I	T	H	S	V	N	Z	C
0	0	1	0	0	0	0	1

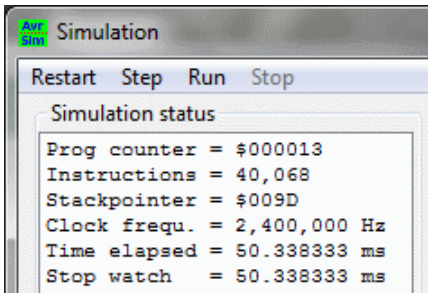
Update after every 1000 instructions

Register

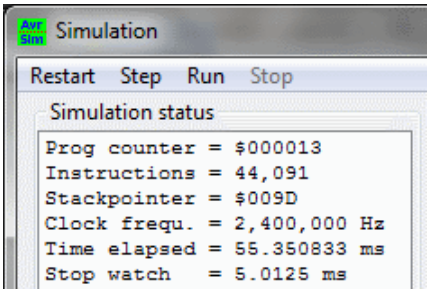
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	AA	AA
R8	55	55	AA	AA	55	55	80	35
R16	04	00	01	FF	00	00	00	00
R24	9F	0F	06	00	00	00	1C	01



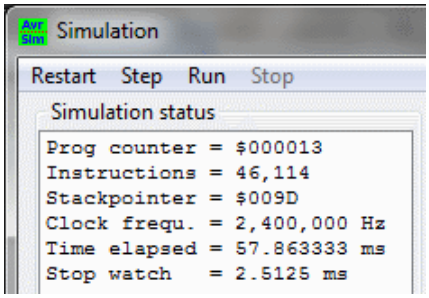
Wäre das Torkeln schon eingeschaltet, was es aber noch nicht ist, würde beim Compare-Match-A der Ausgang PB0 getorkelt und die IR-LED für 12,5 µs lang angeschaltet. So macht man mit Prozessors 40 kHz und das ohne Zählschleifen und bei 2,4 MHz Takt.



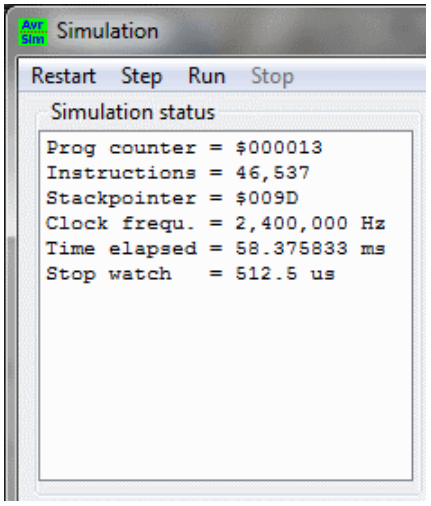
Das ist die Zeit, zu der der erste Timeout kommt: der Zähler hat Null erreicht und 50 ms sind vergangen. Die LED wäre bis dahin 2.000 mal ein- und 2.000 Mal ausgeschaltet worden, wenn denn der Ausgang aktiv geschaltet wäre.



Das ist nun die zweite Phase: eine gepulste LED für 5 ms lang (400 mal an, 400 mal aus).



In der dritten Phase bleibt die IR-LED für 2,5 ms aus.



In der vierten Phase wird die IR-LED wieder gepulst, und zwar für ca. 500 µs lang.

So geht es nun weiter bis alle Kopfbits und alle 64 Datenbits gesendet sind. Interessant ist noch, wie die einzelnen Datenbits ermittelt und gesendet werden.

Wenn alle Kopfbits gesendet sind, kommen die Datenbits dran.

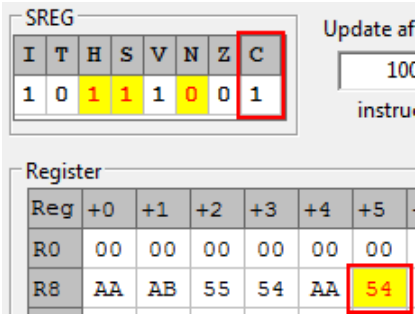
Register

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	55	54
R8	AA	AB	55	54	AA	AA	80	0D
R16	C2	C2	01	09	00	00	00	00
R24	14	00	06	00	00	00	76	00

Zunächst laden die Instruktionen

```
ldi rCntH,High(cKurz) ; lade kurz
ldi rCntL,Low(cKurz)
```

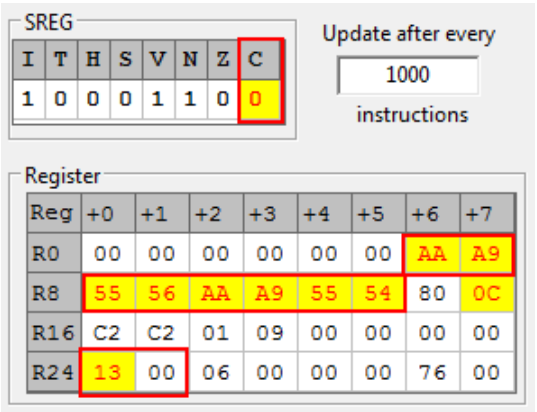
die Dauer einer kurzen Periode in den Zähler R25:R24.



Dann schiebt

```
lsl rData7 ; schiebe Bits
```

rData4 in R14 um eine Position nach links, aus 0xAA wird 0x54. Das herausgeschobene linkeste Bit, eine Eins, landet in der Carry-Flagge im SREG (in diesem Fall eine Eins).



Dann werden mit sieben Rotate-Left-Instruktionen

```
rol rData6
;       ...
rol rData0
```

jeweils alle Bytes nacheinander um eine Position nach links rotiert, wobei das Carry aus dem SREG jeweils an die Position 0 des neuen Bytes rotiert und Bit 7 des alten Bytes ins Carry wandert.

Sind alle sieben Bytes rotiert, befindet sich im Carry das niedrigste Bit aus rData0 (hier eine Null). Wäre das eine Eins, müsste der Zählerstand auf eine lange Dauer gesetzt werden (was aber bei diesem Durchlauf nicht so ist).

Man beachte, dass aus den ursprünglich im Zähler R25:R24 gesetzten 0x14 im Laufe der Schieberei schon 0x13 geworden sind, weil zwischendurch schon ein Compare-Match-A-Interrupt zugeschlagen hat.

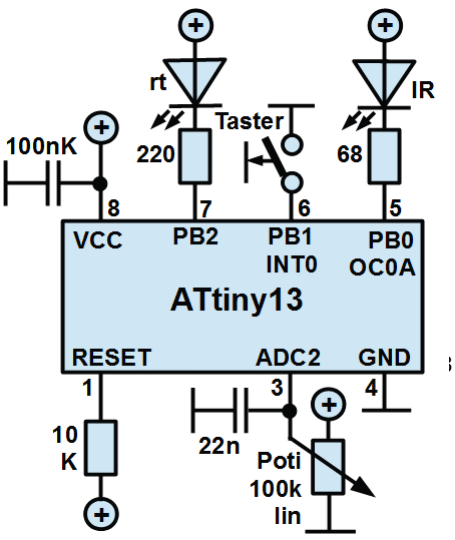
Wer das alles linear und ohne Interrupts programmieren möchte, wird schon nach kurzer Zeit mit ganz grauen Haaren aufgeben. Und nie wieder so was anfassen.

Top	Home	IR	.IF	Hardware	Messen	IR-TX	Daten-TX	3-Kanal TX/RX

12.6 Ein IR-Daten-Übertragungssystem

Das System besteht aus zwei Teilen:

1. Einem IR-Sender mit einem ATtiny13, an den ein Potentiometer angeschlossen ist. Ändert sich der eingestellte Wert oder ist ein voreinstellbarer Zeitraum abgelaufen, dann sendet der ATtiny das 10-Bit-Messergebnis des ADC in einem 16-Bit-IR-Burst.
2. Einem IR-Empfänger mit einem ATtiny24, an den eine LCD angeschlossen ist. Eingegangene IR-Bursts werden analysiert und bei korrektem Empfang das 10-Bit-Ergebnis in Prozent ausgegeben. Treten dabei Fehler auf, werden entsprechende Fehlermeldungen aus-



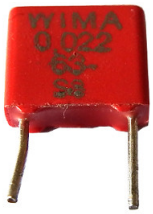
gegeben.

12.6.1 Schaltbild des IR-Datensenders

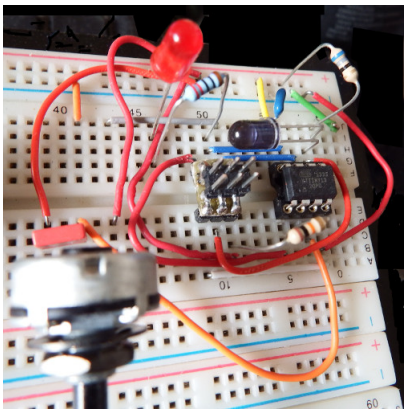
Das hier generiert die Sendesignale. Die IR-LED an OC0A produziert das Sendesignal, eine zusätzliche rote LED ist an, solange die LED sendet. Mit dem Taster lässt sich der Sendevorgang manuell auslösen. Am Potentiometer ist noch ein Folienkondensator mit 22 nF angeschlossen, weil die kleinen Spannungsschwankungen am ADC-Eingang sonst zu viele Sendevorgänge auslösen würden.

12.6.2 Folienkondensator

Das ist der Folienkondensator.



12.6.3 Aufbau



Der
Aufbau ist nicht sehr aufwändig.

12.6.4 Software

Die Software ist auf der obigen Vorlage aufgebaut, nicht benötigte Teile wurden entfernt. Sie ist im Folgenden aufgelistet (Zum Quellcode im asm-Format geht es hier). Es gibt sinnvoll nur die Zeit für die Auslösung des autarken Sendestart-Zeitraums zu verstellen (cAdc10min), diese Konstante steht jetzt auf 10 Sekunden und kann bis auf mehr als 3 Stunden erhöht werden.

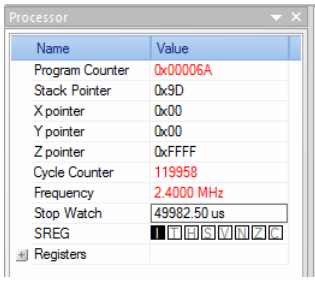
```
;
; ***************************************
; * IR-Sender 40 kHz ATtiny13 Analogwert *
; * (C)2016 by www.gsc-elektronic.net    *
; ***************************************
;
.NOLIST
.INCLUDE "tn13def.inc"
.LIST
;
; -------- Programmablauf -------
;
; Das Programm misst den Analogwert am
; Potieingang. Veraendert sich dieses
; um mehr als zwei Digits, sind mehr als
; 10 Minuten Wartezeit um oder wird die
; Taste gedrueckt, werden drei Kopfsig-
; nale und 16 Datenbitsignale ueber die
; IR-LED gesendet. Die unteren 10 Bit
; stammen aus dem AD-Wandler, die oberen
; 6 Bits stammen aus einer voreinge-
; stellten Kennung.
;
; -------- Register -------------
; frei: R0 .. R8
.def r10mL = R9 ; 10-Minuten-Zaehler
.def r10mM = R10
.def r10mH = R11
.def rData0 = R12 ; Senderegister, hoechstes
.def rData1 = R13 ; Senderegister, niedrigstes
.def rEor = R14 ; fuer Polaritaetsumkehr Toggle
.def rSreg = R15 ; Statusregister
.def rmp = R16 ; Vielzweck
.def rimp = R17 ; Vielzweck innerhalb Int
.def rFlag = R18 ; Flaggenregister
  .equ bRun = 0 ; Senden laeuft
  .equ bSta = 1 ; Senden starten
  .equ bTTo = 2 ; Timeout beim Senden
  .equ bRst = 3 ; Restart fuer zweiten Lauf
  .equ bAdc = 4 ; Analogwert fertig
.def rTOC = R19 ;  Timeout-Zaehler, zaehlt Signale
.def rAnaL = R20 ; Aktueller Analogwert
.def rAnaH = R21
.def rSntL = R22 ; letzter gesendeter Analogwert
.def rSntH = R23
.def rCntL = R24 ; Zaehlt CTC-Compares
.def rCntH = R25
; benutzt: XH:XL R27:R26
; frei: YH:YL R29:R28
; benutzt: ZH:ZL R31:R30
;
; -------- Ports und Pins -------------
.equ pOut = PORTB ; Ausgabe
.equ pDir = DDRB ; Richtung
```

```
.equ pIn  = PINB ; Eingang
.equ bIrO = PORTB0 ; IR-LED
.equ bIrD = DDB0
.equ bIrI = PINB0
.equ bLdO = PORTB2 ; Rote LED
.equ bLdD = DDB2
.equ bKyO = PORTB1 ; Taste, INT0
;
; -------- Konstanten und Timing ------
.equ cTakt = 2400000 ; Prozessortakt
.equ cTaktNs = 1000000000 / cTakt ; Takt ns
.equ cIRF = 40000 ; IR-Sendefrequenz Hz
.equ cIRNs = 1000000000 / cIRF / 2 ; IR-TX ns
.equ cCtc = (cIRNs+cTaktNs/2) / cTaktNs - 1
; 2,4 MHz: 30
;
; -------- Auto-Senden mit ADC --------
; Takt                2400000 Hz
; ADC-Prescaler           128
; ADC-Messzyklen           13
; ADC-Messfreqenz        1442 Hz
; Sekunden pro 10 Minuten 600
;.equ cAdc10min = 1442*600 ; drei Byte
.equ cAdc10min=10*1442
;
; -------- Aufbau IR-Signal -----------
; Alle Zeitangaben in us
; Anzahl Kopfbytes, 1 .. 4
.equ nKopf = 2 ; Anzahl Kopfpaare H/L
.equ cKopf = 2*nKopf-1 ; Anzahl Kopf-Signale
; Dauer der Kopfsignale (Dauer in 2*us/25)
.equ cTK1 = 2*50000/25 ; Dauer High 1
.equ cTK2 = 2*5000/25 ; Dauer Low 1
.equ cTK3 = 2*2500/25 ; Dauer High 2
.equ cTK4 = 2*500/25 ; Dauer Low 2
; Anzahl zu sendender Datenbits
.equ cBits = 16 ; Anzahl Bits
.equ cSign = cKopf + 2*cBits ; Anzahl Signale
; Dauer von Datenbits und Aktivperiode
.equ cKurz = 2*250/25 ; Dauer Binaer-Null
.equ cLang = 2*750/25 ; Dauer Binaer-Eins
.equ cPaus = 2*250/25 ; Dauer Aktivperiode
;
;  --- --- --- ---
; |TK1|TK2|TK3|TK4| Kopfsignale, ms
;  --- --- --- ---
;  Aus  An Aus  An  IR-LED
; 50,0    57,5
;     55,0    58,0
; Datensignale, alle Datenbits Eins, ms
;  --- --- --- --- ... --- --- --- ... ---
; |B16|P16|B15|P15|...|B00|P00|TK1|...|P00|
;  --- --- --- --- ... --- --- --- ... ---
;  Aus  An Aus  An ... Aus  An Aus ... Aus IR-LED
; 58,75   59,75    ...88,75   139
;     59,0     60,0       89,0     ... 178
; Duty cycle = 2*(5 + 0,5 + 16*0,25)/2 = 9,5 ms
;   6 bzw. 5%
; Stromverbr. LED:
;   I = 39 mA, E = 39*9,5/1000/3600 = 0,11uAh
;   I = 100mA, E = 100*9,5/1000/3600= 0,26uAh
;
; -------- Kennung ------
.equ cQuelle  = 0 ; Geraetenummer
.equ cZiel    = 0 ; Zielgeraet
.equ cKennung = (cQuelle<<5)|(cZiel<<3)
;
; -------- Reset- und Interruptvektoren ---
.CSEG
.ORG 0
  rjmp Start ; Programm-Init
  rjmp Int0Isr ; INT0 Tasteninterrupt
  reti ; PCINT0 Pin Change Int Request 0
  reti ; TIM0_OVF Timer/Counter Overflow
  reti ; EE_RDY EEPROM Ready
  reti ; ANA_COMP Analog Comparator
  rjmp TC0CAIsr ; TIM0_COMPA TC0 Compare A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; WDT Watchdog Time-out
  rjmp AdcIsr ; ADC Conversion Complete
;
; -------- Interrupt Service Routinen -----
;
; INT0 Interrupt
; wird von negativen Flanken am Tastereingang
; ausgeloest
;
; Falls derzeit kein Sendevorgang aktiv ist
; werden die Flaggen bRun und bSta gesetzt.
;
Int0Isr: ; Tasteninterrupt, startet Senden
  sbrc rFlag,bRun ; laeuft noch nicht
  reti ; doch, nicht neu starten
  in rSreg,SREG ; SREG retten
  sbr rFlag,(1<<bRun)|(1<<bSta) ; Flaggen
  out SREG,rSreg ; SREG wieder herstellen
  reti ; zurueck
;
; CTC-Timeout Timer, IR-LED schalten
;
; CTC-Timeout Timer, IR-LED schalten
; wird von TC0 beim Compare Match A (CTC) aus-
; geloest
;
; Zaehlt den 16-Bit-Zaehler R25:R24 abwaerts.
; Erreicht es Null, wird die bTTO-Flagge ge-
; setzt und das Toggle-Bit des TC0 umgekehrt.
;
TC0CAIsr: ; 6 Takte f. Int und Vektor-rjmp
  in rSreg,SREG ; SREG retten, 7
  sbiw rCntL,1 ; CTC-Zaehler abwaerts, 9
  brne Tc0CAIsrRet ; Nicht Null, 10/11
  sbr rFlag,1<<bTTO ; Flagge setzen, 11
  in rimp,TCCR0A ; Toggle-Zustand, 12
  eor rimp,rEor ; umkehren, 13
  out TCCR0A,rimp ; neuer Toggle, 14
TC0CAIsrRet: ; 11/14
  out SREG,rSreg ; SREG herstellen, 12/15
  reti ; fertig, 16(!!)/19
;
; ADC Ready Interrupt
; wird nach abgeschlossener AD-Wandlung ausge-
; loest
;
; Liest das Resultat in zwei Register ein, setzt
; die bAdc-Flagge. Ist das Senden nicht aktiviert
; wird der AD-Wandler neu gestartet.
;
AdcIsr:
  in rSreg,SREG ; Status retten
  in rAnaL,ADCL ; Ergebnis lesen
  in rAnaH,ADCH
  sbr rFlag,1<<bAdc ; Flagge setzen
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  sbrs rFlag,bRun ; nicht neu starten, wenn bRun
  out ADCSRA,rimp ; neu starten
  out SREG,rSreg ; Status wiederherstellen
  reti
;
; --------- Programmstart, Init -------
Start:
  ; Stapel einrichten
  ldi rmp,LOW(RAMEND)
  out SPL,rmp
  ; Takt auf 2,4 MHz hochsetzen
  ldi rmp,1<<CLKPCE ; CLK-Enable
  out CLKPR,rmp
  ldi rmp,1<<CLKPS1 ; Teiler = 4
  out CLKPR,rmp
  ; IR-LED-Port initiieren
  sbi pOut,bIrO ; LED aus
  sbi pDir,bIrD ; Pin ist Ausgang
  ; Rote LED initiieren
```

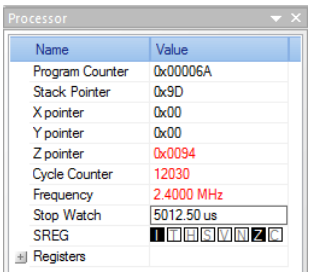
```
  sbi pOut,bLdO ; LED aus
  sbi pDir,bLdD ; Pin ist Ausgang
  ; Tastenport mit Pull-Up
  sbi pOut,bKyO ; Pullup an
  ; Startwerte setzen
  clr rFlag ; Flaggen loeschen
  ldi rmp,1<<COM0A1 ; Toggle-Umkehr-Bit
  mov rEor,rmp ; in Exor-Register
  ; TC0 als CTC starten
  ldi rmp,cCtc ; Compare A Wert 12,5us
  out OCR0A,rmp
  ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01)
  out TCCR0A,rmp ; Ausgang high, CTC
  ldi rmp,1<<CS00 ; Vorteiler = 1
  out TCCR0B,rmp
  ; ADC starten
  ldi rmp,1<<MUX1 ; ADC2 zu messender Eingang
  out ADMUX,rmp
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; neu starten
  ; INT0 und Schlafmodus
  ldi rmp,(1<<SE)|(1<<ISC01) ; Schlafen
  out MCUCR,rmp ; und INT0
  ldi rmp,1<<INT0 ; INT0-Interrupt
  out GIMSK,rmp
  ; Interrupts
  sei ; Interrupts ermoeglichen
;
; -------- Programmschleife ----------
Schleife:
  sleep ; schlafen legen
  nop ; nach Aufwachen, 20
  sbrc rFlag,bTTo ; Timeout?, 21/22
  rcall Timeout ; Timeout bearbeiten, 24
  sbrc rFlag,bSta ; Startflagge
  rcall StartOut ; Senden starten
  sbrc rFlag,bAdc ; ADC-Flagge
  rcall AdcRdy ; ADC-Wert auswerten
  rjmp Schleife ; weiter schlafen
;
; -------- ADC-Wert auswerten --------
;
; Routine AdcRdy wertet das ADC-Ergebnis aus
; wird von der bAdc-Flagge ausgeloest
;
; Unterscheidet sich der eingelesene Wert vom
; vorherigen nur um +/- 2, wird kein Sendevor-
; gang ausgeloest. In diesem Fall wird der 10-
; Minuten-Zaehler erhoeht. Erreicht er den vor-
; eingestellten Wert von 10 Minuten wird der
; Sendevorgang gestartet.
;
AdcRdy:
  cbr rFlag,1<<bAdc ; ADC-Flagge loeschen
  mov ZH,rAnaH ; aktuellen Wert laden
  mov ZL,rAnaL
  adiw ZL,2 ; Toleranz +/- 2
  sub ZL,rSntL
  sbc ZH,rSntH
  brcs AdcRdyUngl ; starte senden
  cpi ZL,5 ; oberhalb Toleranz?
  brcc AdcRdyUngl ; starte senden
  ; erhoehe Zehnminuten-Zaehler
  inc r10mL
  brne AdcRdy10m ; kein Ueberlauf
  inc r10mM
  brne AdcRdy10m ; kein Ueberlauf
  inc r10mH
AdcRdy10m:
  mov rmp,r10mL ; Zeit abgelaufen?
  cpi rmp,BYTE1(cAdc10min)
  brne AdcRdyRet
  mov rmp,r10mM
  cpi rmp,BYTE2(cAdc10min)
  brne AdcRdyRet
  mov rmp,r10mH
  cpi rmp,BYTE3(cAdc10min)
  brne AdcRdyRet
  ; 10 Minuten abgelaufen
  clr r10mL ; Zaehler wieder loeschen
  clr r10mM
  clr r10mH
AdcRdyUngl:
  ldi rmp,(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|
(1<<ADPS0)
  out ADCSRA,rmp ; ADC-Interrupts aus
  nop
  cbr rFlag,1<<bAdc ; Flagge loeschen
  sbr rFlag,(1<<bRun) ; Startflagge
  rjmp StartOut
AdcRdyRet:
  ret
;
; -------- CTC-Timeout ---------------
;
; Routine Timeout
; wird von der Flagge bTTo ausgeloest
;
; Sendet nacheinander Kopf- und Datensignale
;
TimeOut: ; Flagge war gesetzt, 24
  cbr rFlag,1<<bTTo ; Flagge loeschen, 25
  inc rTOC ; naechstes Zeichen, 26
  cpi rTOC,cKopf ; noch Kopf senden?, 27
  brcc TimeOut1 ; nein, Datenbits, 28/29
  ldi ZH,HIGH(TOK2) ; MSB Adresse Kopf, 29
  ldi ZL,LOW(TOK2) ; dto, LSB, 30
  mov rmp,rTOC ; 31
  lsl rmp ; mal zwei, 32
  add rmp,rTOC ; mal drei, 33
  add ZL,rmp ; zu Adresse, 34
  ldi rmp,0 ; Ueberlauf, 35
  adc ZH,rmp ; addieren, 36
  ijmp ; Sprung nach Z, 38
TimeOut1: ; 29
  cpi rTOC,cSign ; schon alle gesendet?, 30
  brcc TimeOutEnd ; ja, beenden, 31/32
  in rmp,TCCR0A ; Toggle/Set lesen, 32
  sbrc rmp,COM0A1 ; Toggle ein pruefen, 33/34
  rjmp TimeOut2 ; Toggle ist aus, 35
  ; Toggle ist an
  ldi rCntH,High(cPaus) ; lade Pausendauer, 35
  ldi rCntL,Low(cPaus) ; 36
  ret ; fertig, 40
TimeOut2: ; 35
  ldi rCntH,High(cKurz) ; lade kurzes Bit, 36
  ldi rCntL,Low(cKurz) ; 37
  lsl rData1 ; Bit herausschieben, 38
  rol rData0 ; 39
  brcc TimeOut3 ; Bit ist Null, 40/41
  ldi rCntH,High(cLang) ; Dauer Eins, 41
  ldi rCntL,Low(cLang) ; 42
TimeOut3: ; 41
  ret ; fertig, 45/46
TimeOutEnd: ; 32
  sbrs rFlag,bRst ; Restart-Flagge set?, 33/34
  rjmp Restart ; nein, starte neu, 35
  clr rmp ; Timer-Int aus, 35
  out TIMSK0,rmp ; 36
  ; Ausgang OCR0A auf Eins
  ldi rmp,(1<<COM0A1)|(1<<COM0A0)|(1<<WGM01) ; 37
  out TCCR0A,rmp ; 38
  ; gesendeten Wert aktualisieren
  mov rSntH,rAnaH ; nach Snt-Register kop., 39
  mov rSntL,rAnaL ; 40
  ; Run-Flagge ausschalten
  cbr rFlag,(1<<bRun)|(1<<bRst) ; Flagge loeschen,
41
  ; ADC wieder starten
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; neu starten, 43
  sbi pOut,bLdO ; rote LED aus, 45
```

```
  ret ; fertig, 49
;
; Timeout Kopf, Dauer einstellen
TOK2: ; 38
  ldi rCntH,High(cTK2) ; 39
  ldi rCntL,Low(cTK2) ; 40
  ret ; 44
;
  ldi rCntH,High(cTK3)
  ldi rCntL,Low(cTK3)
  ret
;
  ldi rCntH,High(cTK4)
  ldi rCntL,Low(cTK4)
  ret
;
; Erneuter Start, zweiter Burst
; wird nach dem Ende des Sendens ausgeloest
;
; Startet die Ausgabe erneut.
;
Restart:
  clr rmp ; Timer-Int ausschalten, 12
  out TIMSK0,rmp ; 13
  sbr rFlag,1<<bRst ; Restart-Flagge setzen, 14
  rjmp StartOut1 ; 16
;
; Senden starten
; wird von der gesetzten bSta-Flagge ausgeloest
;
; Kopiert den letzten Analogwert, verodert ihn
; mit der Kennung und schreibt ihn in die
; Senderegister rData1 und rData0.
; Setzt den rToC-Zaehler, laedt die erste in-
; aktive Pausendauer des ersten Kopfes und
; ermoeglicht den Compare-Match-A-Interrupt.
;
StartOut:
  cbr rFlag,(1<<bSta)|(1<<bRst) ; Start-Flagge
loeschen, 15
  mov rSntH,rAnaH
  mov rSntL,rAnaL
Startout1:
  mov rData1,rAnaL ; LSB Analogwert senden, 16/17
  ldi rmp,cKennung ; Kennung zu MSB ; 17/18
  or rmp,rAnaH ; Verodern, 18/19
  mov rData0,rmp ; in Senderegister, 19/20
  ser rToC ; Zaehler auf 0xFF, 20/21
  ldi rCntH,High(cTK1) ; erste Wartedauer, 21/22
  ldi rCntL,Low(cTK1) ; 22/23
  ldi rmp,1<<OCIE0A ; Timer-Int ein, 23/24
  out TIMSK0,rmp ; 24/25
  cbi pOut,bLdO ; rote LED an, 25/26
  ret ; 29/30
;
; Ende Quellcode
;
```

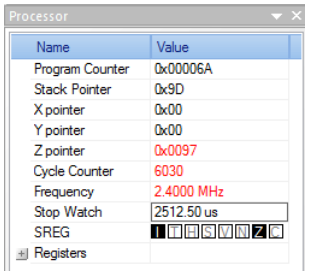
12.6.5 Simulation der Software mit dem Studio



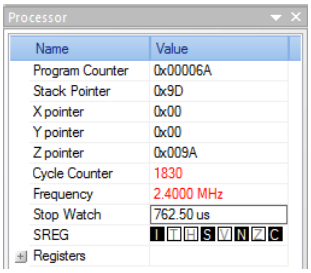

```
AdcRdyRet:
    ret
;
; -------- CTC-Timeout ---------------
TimeOut: ; Flagge war gesetzt
    cbr rFlag,1<<bTTo ; Flagge loeschen
    inc rTOC ; naechstes Zeichen
    cpi rTOC,cKopf ; noch Kopf senden?
    brcc TimeOut1 ; nein, Datenbits
    ldi ZH,HIGH(TOK2) ; MSB Adresse Kopf
    ldi ZL,LOW(TOK2) ; dto, LSB
    mov rmp,rTOC
    lsl rmp ; mal zwei
    add rmp,rTOC ; mal drei
    add ZL,rmp ; zu Adresse
    ldi rmp,0 ; Ueberlauf
    adc ZH,rmp ; addieren
```



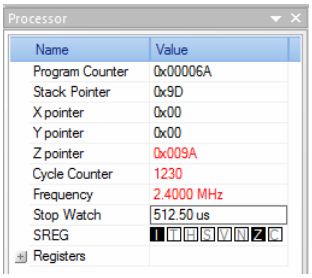
Das Senden der langen Kopfphase im Simulator dauert etwa die vorberechnete Anzahl Rechenzyklen und liegt innerhalb von Rundungsfehlern.



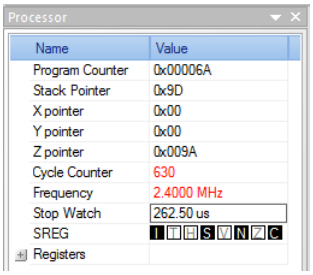
Die aktive Phase des ersten Kopfsignals liegt ebenfalls innerhalb von Rundungsfehlern, eine Phase wurde verpasst.



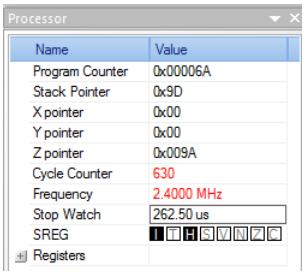
Die inaktive Phase des zweiten Kopfsignals ist ebenfalls um eine Phase länger als vorberechnet.



Wie aus der Analyse der Ausführungszeiten hervorgeht, ist auch bei diesem Signal die Dauer um eine Phase länger als berechnet. Wer es noch genauer haben will, zieht von den Kopfdaten eine Phase ab.



Das Inaktiv-Signal eines langen Einer-Bits ist ebenfalls eine Phase länger als vorgesehen.



Auch beim Null-Bit ist eine Phase mehr als berechnet.

Die Dauer des aktiven Signals ist ebenfalls geringfügig länger.

Die Abweichungen von der Solldauer sind in einem akzeptablen Bereich zumal sie deutlich kürzer sind als die Einschwingdauer von IR-Sensoren. Eine Verwechslung von Nullen und Einsen ist damit jedenfalls nicht zu befürchten.

12.6.6 Hardware des IR-Datenempfängers

Der Empfang der Daten erfolgt mit der gleichen Hardware (ATtiny24, LCD, IR-Empfängermodul) wie die bisher durchgeführten Analysen.

12.6.7 Software

Die Software (zum Quellcode im asm-Format geht es hier, zusätzlich wird die LCD-Include-Datei benötigt) ist auf die Signalfolge des Senders eingestellt. Zum Teil ergaben sich erhebliche Abweichungen der Zeiten, die das Empfangsmodul liefert, von denen in der Sendersoftware. Es wurden daher Programmteile zur Diagnose hinzugefügt, die der Analyse dienen können (Schalter "Diagnose" mit den Einstellungen "1" oder "2").

```
;                                                .INCLUDE "tn24def.inc"
; ***********************************************.LIST
; * IR-Empfaenger mit ATtiny24 und LCD, Analog-RX ;
; * (C)2016 by http://www.elektronic.net          ; ---------- Programmablauf ------------
; ***********************************************;
;                                                ; Empfaengt die vom IR-Sender gesendeten Analog-
.NOLIST                                          ; daten (Kennung und Potentiometerstellung) und
```

```
; stellt sie dezimal auf der LCD dar.
;
; In den Diagnoseeinstellungen 1 und 2 werden
; die eingehenden Rohdaten des IR-Sensors in Hex
; dargestellt.
;
; ---------- Schalter ------------------
.equ Diagnose = 0 ; 0: Keine Diagnose
;                   1: Datenbytes in hex listen
;                   2: Alle Worte in hex listen
;
; ---------- Einstellungen Fernsteuerung
.equ cLang = 20000 ; Lange Pause
.equ cKopf = 2700 ; Dauer High-Signal Kopf us
.equ cNull = 512 ; Dauer Null-Byte us
.equ cEins = 1064 ; Dauer Eins-Byte us
;
; ---------- Timing --------------------
; Prozessortakt           1.000.000 Hz
; Zeit pro Takt                   1 µs
; Vorteiler TC1                   8
; Zeit pro TC1-Timer-Takt         8 µs
; Ueberlauf TC1 bei          65.536
; Zeit bis TC1 Ueberlauf    524,288 ms
.equ cPresc = 8 ; Timer 1 Prescaler
;
; ---------- Signaldauern und Toleranzen -------
.equ nLang = cLang/cPresc ; Mindest Langpause
.equ nKopf = cKopf/cPresc ; Zaehlsignale Kopf
.equ nNull = cNull/cPresc ; Zaehlsignale Null
.equ nEins = cEins/cPresc ; Zaehlsignale Eins
.equ cToleranz = 20 ; Toleranzbreite in +/- %
.equ nKopfMin = nKopf-(nKopf*cToleranz+50)/100
.equ nKopfMax = nKopf+(nKopf*cToleranz+50)/100+1
.equ nNullMin = nNull-(nNull*cToleranz+50)/100
.equ nNullMax = nNull+(nNull*cToleranz+50)/100+1
.equ nEinsMin = nEins-(nEins*cToleranz+50)/100
.equ nEinsMax = nEins+(nEins*cToleranz+50)/100+1
;
; Alle Datenkonstanten ueberpruefen
.if nNullMin > 255
  .error "nNullMin zu gross!"
  .endif
.if nNullMax > 255
  .error "nNullMax zu gross!"
  .endif
.if nEinsMin > 255
  .error "nEinsMin zu gross!"
  .endif
.if nEinsMax > 255
  .error "nEinsMax zu gross!"
  .endif
;
; ---------- Ports ---------------------
.equ pIrIn = PINA ; IR-Detektor-Port
.equ bIrIn = 0    ; IR-Detektor-Pin
;
; ---------- Register ------------------
; verwendet: R0 von LCD, Dezimalwandlung
; verwendet: R1 Dezimalwandlung
; frei: R2..R5
.def rErrL = R6
.def rErrH = R7
.def rMul1 = R8 ; Multiplikation
.def rMul2 = R9
.def rMul3 = R10
.def rMulH = R11
.def rFlags = R12 ; Flaggenspeicher
.def rDataL = R13 ; Empfangsdaten
.def rDataH = R14
.def rSreg = R15 ; Sicherung Status
.def rmp   = R16 ; Vielzweckregister
.def rmo   = R17 ; Vielzweckregister
.def rLine = R18 ; LCD-Zeilenzaehler
.def rLese = R19 ; LCD-Register
.def rimp  = R20 ; Interrupt-Vielzweck
.def rFlag = R21 ; Flaggenregister
  .equ bSta = 0 ; Startflagge
  .equ bKpf = 1 ; Kopf korrekt
  .equ bKKu = 2 ; Kopf zu kurz
  .equ bKKl = 3 ; Kopf zu lang
  .equ bDKu = 4 ; Datenbit zu kurz
  .equ bDMi = 5 ; Datenbit mittellang
  .equ bDLa = 6 ; Datenbit zu lang
  .equ bDOv = 7 ; Anzahl Datenbits falsch
.def rDCtr = R22 ; Datenbitzaehler
; frei: R23 .. R25
; verwendet: XH:XL R27:R26 diverse Verwendungen
; verwendet: YH:YL R29:R28 Zaehler Signaldauer
; verwendet: ZH:ZL R31:R30 diverse Verwendungen
;
; ---------- Diagnose-SRAM-Puffer ------
.DSEG
.ORG 0x0060
.if Diagnose == 1
  Buffer:
  .byte 16
  Bufferende:
  .endif
.if Diagnose == 2
  Buffer:
  .byte 36
  Bufferende:
  .endif
;
; ---------- Reset- und Interrupts -----
.CSEG ; Code-Segment
.ORG 0 ; an den Beginn
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0 External Interrupt Request 0
  rjmp Pci0Isr ; PCINT0 Pin Change Int Req 0
  reti ; PCINT1 Pin Change Interrupt Req 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT Timer/Counter1 Capture Event
  reti ; TIM1_COMPA TC1 Compare Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  rjmp Tc1Isr ; TIM1_OVF TC1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; TIM0_OVF TC0 Overflow
  reti ; ANA_COMP Analog Comparator
  reti ; ADC ADC Conversion Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; ---------- Interrupt Service -----
;
; PCINT Interrupt
;   wird von allen Pegelwechseln des IR-Sensors
;   ausgeloest
;
; Liest bei aktiven IR-Signalen den Zaehlerstand
; des TC1 aus und setzt diesen auf Null zurueck.
; Ueberschreitet die Dauer eine Obergrenze, wird
; die Flagge bSta gesetzt und dadurch ein Neu-
; start ausgeloest (langes Kopfsignal).
; Ist ein Kopfsignal eingegangen und ist das
; Signal
; a) kuerzer als die Bitdauer fuer die Unter-
;    grenze von Null (nNullMin), wird die bDKu-
;    Flagge gesetzt,
; b) laenger als die Maximalgrenze fuer Nullen
;    (nNullMax), aber kuerzer als die Minimal-
;    grenze fuer Einsen (nEinsMin), wird die
;    bDMi-Flagge gesetzt,
; c) laenger als die Maximalgrenze fuer Einsen
;    nEinsMax), wird die bDLa-Flagge gesetzt,
; d) innerhalb der Grenzen fuer eine Null, wird
;    eine Null in das Ergebnis einrotiert,
; e) innerhalb der Grenzen fuer eine Eins wird
;    eine Eins in das Ergebnis einrotiert.
; Sind mehr als 16 Datenbits empfangen, wird die
; bDOv-Flagge gesetzt.
```

```
; Sind Diagnosemodi gewaehlt, werden die Daten
; im SRAM-Puffer abgelegt.
;
Pci0Isr:
  sbis pIrIn,bIrIn ; ueberspringe bei Eins, 1/2
  reti ; 5
  ; Eingang ist Low, war vorher High
  in YL,TCNT1L ; Zaehlerstand lesen,3
  in YH,TCNT1H ; 4
  ldi rimp,0 ; Zaehlerstand ruecksetzen, 5
  out TCNT1H,rimp ; 6
  out TCNT1L,rimp ; 7
  ldi rimp,1<<TOIE1 ; Zaehler-Int starten
  out TIMSK1,rimp
  in rSreg,SREG ; Status sichern
  cpi YH,High(nLang) ; Lange Pause?
  brcs Pci0Isr1 ; Signal war kuerzer
  ldi rFlag,1<<bSta ; Startflagge setzen
.if diagnose != 0
  ldi XH,High(Buffer) , Neustart Puffer
  ldi XL,Low(Buffer)
  .if diagnose == 2
    st X+,YH ; in SRAM
    st X+,YL
    .endif
  .endif
  rjmp Pci0IsrRet
Pci0Isr1: ; Signal nicht langer Kopf
  sbrs rFlag,bSta ; Startflagge?
  rjmp Pci0IsrRet ; Warten bis Start
  sbrc rFlag,bKpf ; Kopfflagge?
  rjmp Pci0Isr2 ; Schon korrekt
.if diagnose == 2
  st X+,YH ; in SRAM
  st X+,YL
  .endif
  subi YL,LOW(nKopfMin) ; - min Kopfdauer
  sbci YH,HIGH(nKopfMin)
  brcs Pci0IsrErrKk ; Kopf zu kurz
  subi YL,LOW(nKopfMax) ; - max Kopfdauer
  sbci YH,HIGH(nKopfMax)
  brcc Pci0IsrErrKl ; Kopf zu lang
  sbr rFlag,1<<bKpf ; Kopf ist ok
  clr rDCtr ; Datenbitzaehler starten
  clr rDataL ; Daten leer
  clr rDataH
  rjmp Pci0IsrRet
Pci0Isr2: ; Kopf war korrekt, pruefe Bits
.if Diagnose == 1
  tst YH ; MSB Null?
  breq Pci0Isr2d ; ja
  ldi YL,0xFF ; signalisiere Ueberlauf
Pci0Isr2d:
  st X+,YL ; in SRAM
  rjmp Pci0IsrRet
  .endif
.if Diagnose == 2
  st X+,YH ; MSB und
  st X+,YL ; LSB in SRAM
  rjmp Pci0IsrRet
  .endif
  cpi YL,LOW(nNullMin) ; min Nulldauer
  brcs Pci0IsrErrDKu ; kuerzer als Null
  cpi YL,LOW(nNullMax) ; max Nulldauer
  brcc Pci0Isr3 ; auf Eins pruefen
  clc ; Carry Null
  rjmp Pci0Isr4 ; in Ergebnis schieben
Pci0Isr3:
  cpi YL,Low(nEinsMin) ; min Einsdauer
  brcs Pci0IsrErrDMi ; mittellang
  cpi YL,Low(nEinsMax) ; max Einsdauer
  brcc Pci0IsrErrDLa ; zu lang
Pci0Isr4:
  rol rDataL ; in Ergebnis rollen
  rol rDataH
  inc rDCtr ; Anzahl Bits
  cpi rDCtr,17 ; Datenbits ok?
  brcs Pci0IsrRet ; ja
  sbr rFlag,1<<bDOv ; Zu viele Bits
  rjmp Pci0IsrErrData ; Fehler
Pci0IsrErrKk:
  sbr rFlag,(1<<bKpf)|(1<<bKKu) ; Kopf zu kurz
  ldi rmp,Low(nKopfMin) ; Original herstellen
  add YL,rmp
  ldi rmp,High(nKopfMin)
  adc YH,rmp
  rjmp Pci0IsrErrData ; Fehler
Pci0IsrErrKl:
  sbr rFlag,(1<<bKpf)|(1<<bKKl) ; Kopf zu lang
  ldi rmp,Low(nKopfMin+nKopfMax) ; Original
  add YL,rmp ; wieder herstellen
  ldi rmp,High(nKopfMin+nKopfMax)
  adc YH,rmp
  rjmp Pci0IsrErrData ; Fehler
Pci0IsrErrDKu:
  sbr rFlag,1<<bDKu ; Datenbit zu kurz
  rjmp Pci0IsrErrData ; Fehler
Pci0IsrErrDMi:
  sbr rFlag,1<<bDMi ; Datenbit mittellang
  rjmp Pci0IsrErrData ; Fehler
Pci0IsrErrDLa:
  sbr rFlag,1<<bDLa ; Datenbit zu lang
Pci0IsrErrData:
  lsl YL ; N mal 8
  rol YH
  lsl YL
  rol YH
  lsl YL
  rol YH
  mov rErrL,YL ; in Fehlerregister
  mov rErrH,YH
Pci0IsrRet:
  out SREG,rSreg ; Status wieder herstellen
  reti ; fertig
;
; TC1-Overflow Interrupt Service Routine
;   wird von Ueberlaeufen des TC1 ausge-
;   loest
;
; Ueberlauf erfolgt nach 8*65.536 = 0,52
; Sekunden nach dem letzten aktiven IR-
; Sensor-Signal.
; Falls ein aktives Kopfsignal empfangen
; wurde, wird die Ausgabeflagge T im SREG
; gesetzt.
;
Tc1Isr:
  sbrs rFlag,bSta ; pruefe Startbit
  rjmp Tc1Isr1
  sbrc rFlag,bKpf ; pruefe Kopfbit
  set ; setze Ausgabeflagge
Tc1Isr1:
  reti
;
; ---------------- Start, Init --------------
Start:
  ldi rmp,LOW(RAMEND) ; Stapel auf Ramende
  out SPL,rmp ; in Stapelzeiger
  ; I/O initiieren
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
Schreiben
  out pLcdDR,rmp ; auf Richtungsregister
Datenausgabeport
  ; LCD initiieren
  rcall LcdInit ; Init der LCD
  ldi ZH,HIGH(2*LcdStart) ; Start-Text ausgeben
  ldi ZL,LOW(2*LcdStart)
  rcall LcdText
  ; Startwert fuer Flaggen
```

```
  clr rFlag
  clt
  ; Timer 1 initiieren, freilaufend durch 8
  ldi rmp,1<<CS11 ; Prescaler durch 8
  out TCCR1B,rmp ; an Kontrollregister B
  ; PCINT0 fuer IR-Rx
  ldi rmp,1<<PCINT0 ; Pin 0 Level change INTs
  out PCMSK0,rmo ; in Maske 0
  ldi rmp,1<<PCIE0 ; Interrupt PCINT0  enable
  out GIMSK,rmp
  ; Sleep Enable
  ldi rmp,1<<SE ; Sleep Mode Idle
  out MCUCR,rmp ; in MCU Kontrollregister
  ; Interrupts ermoeglichen
  sei ; I-Flagge Statusregister
Schleife:
  sleep ; schlafen legen
  nop ; aufwachen
  brtc Schleife ; Update-Flagge auswerten
  rcall Update ; Flagge behandeln
  rjmp Schleife ; schlafen legen
;
; Routine UpDate
;   wird von einem gesetzten T-Flag im
;   SREG ausgeloest und gibt die Ergeb-
;   nisse und eventuelle Fehlermeldun-
;   gen auf der LCD aus. Bei den Diag-
;   semodi werden die im SRAM gespei-
;   cherten Empfangsdaten ausgegeben.
;
Update:
  clt ; Flagge wieder abschalten
  clr rmp ; Interrupts Timer aus
  out TIMSK1,rmp ; in Interruptmaske
  mov rmp,rFlag ; Fehlerflaggen
  andi rmp,0xFC ; isolieren
  mov rFlags,rmp ; zwischenspeichern
  ldi rFlag,0 ; Flaggen fuer Neustart
.if Diagnose == 1
  rjmp Diag ; gib Diagnose aus
  .endif
.if Diagnose == 2
  rjmp DiagW ; gib Diagnose aus
  .endif
  brne UpdateErr ; Fehlerflaggen
  cpi rDCtr,16 ; 16 Bit empfangen?
  breq UpdateKorrekt ; ja, Ausgabe
  rjmp UpdateZuWenigBits ; Ausgabe Fehler
UpdateKorrekt:
  ldi ZH,2 ; Position LCD setzen
  clr ZL
  rcall LcdPos
  ldi rmp,'R' ; Ausgabe R:
  rcall LcdD4Byte
  ldi rmp,':'
  rcall LcdD4Byte
  ldi ZH,2 ; Position LCD setzen
  ldi ZL,2
  rcall LcdPos
  rcall Multi ; Ergebnis multiplizieren
  rjmp Prozent ; als Prozentzahl ausgeben
UpdateErr:
  rcall LcdLine3 ; ab Zeile 3 loeschen
  ldi ZH,HIGH(2*LcdAusgabe)
  ldi ZL,LOW(2*LcdAusgabe)
  rcall LcdTextC
  rcall LcdLine3
  ldi rmp,'E' ; E: ausgeben
  rcall LcdD4Byte
  ldi rmp,':'
  rcall LcdD4Byte
  ldi ZH,HIGH(2*ErrKKu) ; Fehlerausgaben
  ldi ZL,LOW(2*ErrKKu)
  mov rmp,rFlags ; Fehlerflaggen
  sbrc rmp,bKKu
  rjmp Update1
  ldi ZH,HIGH(2*ErrKLa)
  ldi ZL,LOW(2*ErrKLa)
  sbrc rmp,bKKl
  rjmp Update1
  ldi ZH,HIGH(2*ErrDKu)
  ldi ZL,LOW(2*ErrDKu)
  sbrc rmp,bDKu
  rjmp Update1
  ldi ZH,HIGH(2*ErrDMi)
  ldi ZL,LOW(2*ErrDMi)
  sbrc rmp,bDMi
  rjmp Update1
  ldi ZH,HIGH(2*ErrDLa)
  ldi ZL,LOW(2*ErrDLa)
  sbrc rmp,bDLa
  rjmp Update1
  ldi ZH,HIGH(2*ErrDBi)
  ldi ZL,LOW(2*ErrDBi)
Update1:
  rcall LcdTextC ; Fehlertext ausgeben
  ldi rmp,'='
  rcall LcdD4Byte
  rjmp DezimalY ; Y dezimal ausgeben
;
UpdateZuWenigBits: ; Anzahl Bits ausgeben
  rcall LcdLine4
  ldi rmp,'B' ; B: ausgeben
  rcall LcdD4Byte
  ldi rmp,':'
  rcall LcdD4Byte
  rcall DezimalAus ; Anzahl Bits dezimal
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,'b'
  rcall LcdD4Byte
  ldi rmp,'i'
  rcall LcdD4Byte
  ldi rmp,'t'
  rjmp LcdD4Byte
;
; Zahl in rDataH:rDataL in % umwandeln
.equ cMulti = 1000*256/1023 ; mal 1000/1023
;
; Multi nimmt die empfangene 10-Bit-Zahl mit
; 256 * 1000 / 1023 = 250 mal.
;
Multi:
  mov rmp,rDataH ; oberste Bits loeschen
  andi rmp,0x03 ; Ziel/Quelle loeschen
  mov rDataH,rmp ; und zurueckschreiben
  clr rMul1 ; Ergebnis loeschen
  clr rMul2
  clr rMul3
  clr rMulH ; Hilfsregister loeschen
  ldi rmp,cMulti ; Multiplikator in rmp
Multi1:
  lsr rmp ; rmp rechts schieben
  brcc Multi2 ; kein Carry, nicht addieren
  add rMul1,rDataL ; addieren
  adc rMul2,rDataH
  adc rMul3,rMulH
Multi2:
  lsl rDataL ; Multiplikator links schieben
  rol rDataH
  rol rMulH
  tst rmp ; Ende Multiplikation?
  brne Multi1 ; weiter multiplizieren
  ret
;
; Zahl in rMul3:rMul2 in Dezimal auf LCD ausgeben
; Prozent wandelt das Ergebnis in Dezimal-
; format um und gibt es im Format nnn,n% auf
; der LCD aus.
;
Prozent:
  ldi ZH,HIGH(2*Dezimal) ; Dezimaltabelle
  ldi ZL,LOW(2*Dezimal)
  clt ; fuehrende Nullen unterdruecken
```

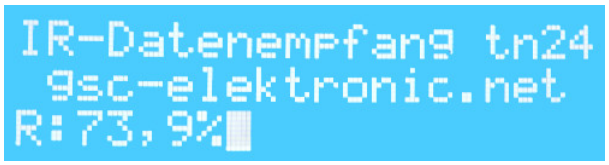
```
Prozent1:
  lpm XL,Z+ ; Dezimalzahl laden
  lpm XH,Z+
  tst XL ; Ende Tabelle feststellen
  breq Prozent5 ; letzte Stelle
  clr rmp ; rmp ist Zaehler
Prozent2:
  sub rMul2,XL ; Dezimalzahl abziehen
  sbc rMul3,XH
  brcs Prozent3 ; Carry aufgetreten
  inc rmp ; hoch zaehlen
  rjmp Prozent2 ; weiter abziehen
Prozent3:
  add rMul2,XL ; Abziehen rueckgaengig
  adc rMul3,XH
  tst rmp ; Null?
  brne Prozent4 ; nein, ausgeben
  brts Prozent4 ; Fuehrende Nullen aus
  cpi XL,10 ; Dezimaltrennstelle?
  brne Prozent1 ; nein, weiter
  set ; Fuehrende Nullen ausgeben
Prozent4:
  subi rmp,-'0' ; in Dezimal ASCII
  rcall LcdD4Byte
  cpi XL,10 ; Dezimaltrennstelle?
  brne Prozent1 ; nein
  ldi rmp,',' ; Dezimaltrennstelle
  rcall LcdD4Byte
  rjmp Prozent1 ; weiter
Prozent5:
  clt ; T-Flagge loeschen
  mov rmp,rMul2 ; letzte Dezimalstelle
  subi rmp,-'0' ; in ASCII
  rcall LcdD4Byte
  ldi rmp,'%' ; Prozentzeichen
  rcall LcdD4Byte
  ldi rmp,' '
  rjmp Lcd4Byte
;
; Gibt die Anzahl empfangener Datenbits rDCtr
; als Byte in dezimal aus
;
DezimalAus:
  ldi ZH,HIGH(2*Dezimal100) ; Hunderter
  ldi ZL,LOW(2*Dezimal100)
  clt ; Fuehrende Nullen unterdruecken
DezimalAus1:
  lpm XL,Z+ ; Dezimalzahl laden
  lpm XH,Z+
  tst XL ; Ende der Tabelle?
  breq DezimalAus5 ; ja, letzte Stelle
  clr rmp ; rmp ist Zaehler
DezimalAus2:
  sub rDCtr,XL ; Dezimalzahl abziehen
  brcs DezimalAus3 ; carry, fertig
  inc rmp ; naechste Subtraktion
  rjmp DezimalAus2 ; und weiter abziehen
DezimalAus3:
  add rDCtr,XL ; letzte Abziehen rueckgaengig
  tst rmp ; Fuehrende Null?
  brne DezimalAus4 ; nein, ausgeben
  brts DezimalAus4 ; fuehrende Null ist aus
  rjmp DezimalAus1 ; weiter umwandeln
DezimalAus4:
  subi rmp,-'0' ; in Dezimal-ASCII
  rcall LcdD4Byte
  set ; Fuehrende Nullen ausgeben
  rjmp DezimalAus1 ; und weiter
DezimalAus5:
  clt ; T-Flagge loeschen
  mov rmp,rDCtr ; Letzte Stelle
  subi rmp,-'0' ; in ASCII
  rjmp LcdD4Byte
;
; Gibt die Dauer des letzten fehlerhaft empfan-
; genen Datenbits in rErrH:rErrL in Mikrosekun-
; den dezimal aus
;
DezimalY:
  ldi ZH,HIGH(2*Dezimal) ; Z auf Tabelle
  ldi ZL,LOW(2*Dezimal)
  clt ; Fuehrende Nullen unterdruecken
DezimalY2:
  lpm XL,Z+ ; Dezimalzahl aus Tab holen
  lpm XH,Z+
  tst XL ; Ende Tabelle?
  breq DezimalY6 ; ja, letzte Stelle
  clr rmp ; rmp ist Zaehler
DezimalY3:
  sub rErrL,XL ; Error-Y subtrahieren
  sbc rErrH,XH
  brcs DezimalY4 ; carry, fertig
  inc rmp ; noch mal subtrahieren
  rjmp DezimalY3 ; weiter subtrahieren
DezimalY4:
  add rErrL,XL ; Subtraktion rueckgaengig
  adc rErrH,XH
  tst rmp ; Ergebnis Null?
  brne DezimalY5 ; nein
  brts DezimalY5 ; fuehrende Nullen aus?
  rjmp DezimalY2 ; weiter
DezimalY5:
  set ; fuehrende Nullen nicht unterdruecken
  subi rmp,-'0' ; in ASCII
  rcall LcdD4Byte
  rjmp DezimalY2 ; weiter umwandeln
DezimalY6:
  clt ; Loesche T-Flagge
  mov rmp,rErrL ; letzte Dezimalstelle
  subi rmp,-'0' ; in ASCII
  rcall LcdD4Byte
  ldi rmp,' '
  rjmp LcdD4Byte
;
; Dezimaltabelle
Dezimal:
.dw 10000
.dw 1000
Dezimal100:
.dw 100
.dw 10
.dw 0
;
; Starttext LCD
LcdStart:
.db "IR-Datenempfang tn24",0x0D,0xFF
.db " gsc-elektronic.net ",0x0D,0xFF
LcdAusgabe:
.db "                    ",0x0D,0xFF
.db "                    ",0xFE,0xFF
;
; Fehlertexte
ErrKKu:
.db "Kopf zu kurz",0xFE,0xFE
ErrKLa:
.db "Kopf zu lang",0xFE,0xFE
ErrDKu:
.db "DBit zu kurz",0xFE,0xFE
ErrDMi:
.db "DBit Mittel",0xFE
ErrDLa:
.db "DBit zu lang",0xFE,0xFE
ErrDBi:
.db "DBits > 16",0xFE,0xFE
;
.if Diagnose == 1 ; Datenbyte Ausgabe
Diag:
  ldi rmp,0x01 ; Display loeschen
  rcall LcdC4Byte
  ldi XH,High(Buffer) ; Anfang Puffer
  ldi XL,Low(Buffer)
Diag1:
  ld rmp,X+ ; Pufferbyte
  rcall HexOut ; in Hex ausgeben
```

```
  ldi rmp,' '
  rcall LcdD4Byte
  cpi XL,Low(Buffer+6) ; Zeile 2?
  brne Diag1a
  rcall LcdLine2 ; Zeile 2
  rjmp Diag2
Diag1a:
  cpi XL,Low(Buffer+12) ; Zeile 3?
  brne Diag2
  rcall LcdLine3 ; Zeile 3
Diag2:
  cpi XL,Low(Bufferende) ; fertig?
  brne Diag1 ; nein, weiter
  ret
  .endif
;
.if Diagnose != 0 ; Hexziffern ausgeben
HexOut:
  push rmp ; rmp retten
  swap rmp ; oberes und unteres Nibble
  rcall HexOutNibble ; Nibble ausgeben
  pop rmp ; rmp wieder herstellen
HexOutNibble:
  andi rmp,0x0F ; unteres Nibble isolieren
  subi rmp,-'0' ; in ASCII
  cpi rmp,'9'+1 ; A..F?
  brcs HexOutNibble1 ; nein
  subi rmp,-7 ; in A..F
HexOutNibble1:
  rjmp LcdD4Byte ; Zeichen ausgeben
  .endif
;
.if Diagnose == 2 ; Woerter ausgeben
DiagW:
  ldi rmp,0x01 ; LCD loeschen
  rcall LcdC4Byte
  ldi XH,HIGH(Buffer) ; X auf Puffer
  ldi XL,LOW(Buffer)
  ldi ZH,0 ; Zeilenzaehler
  ldi ZL,0 ; Spaltenzaehler
DiagW1:
  ld rmp,X+ ; MSB lesen
  tst rmp ; MSB Null?
  brne DiagWW ; nein, als Wort
  cpi ZL,19 ; noch Platz in der Zeile?
  brcs DiagW2 ; ja, gib aus
  rcall NextLine ; naechste Zeile
DiagW2:
  ld rmp,X+ ; lese LSB
  rcall HexOut ; gib in Hex aus
  subi ZL,-2 ; zwei Zeichen addieren
  rjmp DiagW3 ; weiter
DiagWW:
  cpi ZL,16 ; noch Platz in der Zeile?
  brcs DiagWW1 ; ja
  rcall NextLine ; naechste Zeile
DiagWW1:
  rcall HexOut ; gib MSB aus
  ld rmp,X+ ; lese LSB
  rcall HexOut ; gib LSB aus
  subi ZL,-4 ; vier Zeichen addieren
DiagW3:
  cpi ZL,20 ; Ende der Zeile?
  brcc DiagW4 ; ja
  ldi rmp,' ' ; Leerzeichen ausgeben
  rcall LcdD4Byte
  subi ZL,-1 ; ein Zeichen addieren
DiagW4:
  cpi XL,Low(Bufferende) ; Ende Puffer?
  brcs DiagW1 ; nein, weiter
  ret
;
NextLine: ; Zeilenvorschub LCD
  push rmp ; rmp erhalten
  inc ZH ; naechste Zeile
  clr ZL ; Zeilenfang
  rcall LcdPos
  pop rmp ; rmp wieder herstellen
  ret
  .endif
;
; LCD-Routinen einlesen
.include "Lcd4Busy.inc"
;
; Ende Quelltext
;
```

Neue Instruktion hier ist:

- SBCI Register,Konstante: zieht die Konstante plus das Übertragsbit vom Register ab.

12.6.8 Anwendung



So sieht der Empfang aus. Und das ohne Fließkomma-Monsterbibliothek, einfach nur mit Grips.

Top	Home	IR	.IF	Hardware	Messen	IR-TX	Daten-TX	3-Kanal TX/RX

12.7 Ein Dreikanal-IR-Empfänger mit Schaltern

12.7.1 Aufgabe

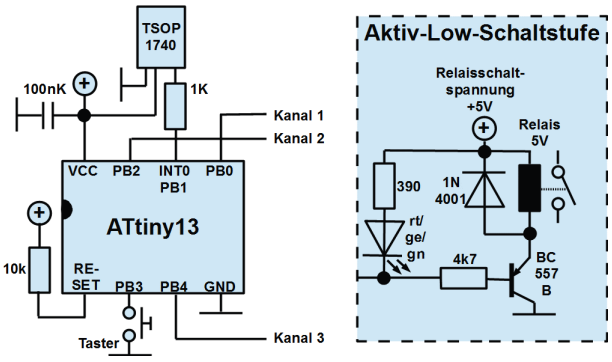
Wer wünscht sich das nicht: mit der IR-Fernsteuerung Geräte ein- und ausschalten? Die Zeit des Spezialistentums, wo der Experte sich erst mal tagelang mit den Codes seiner Fernsteuerung beschäftigt, sind vorbei. Diese folgende Fernsteuerung lernt selbstständig, welche Codes zum Schalten verwendet werden sollen. Dazu wird sie einmal trainiert, speichert ihre gelernten

Codes im internen EEPROM und ruft sie daraus ab. Bis zur nächsten Umprogrammierung.

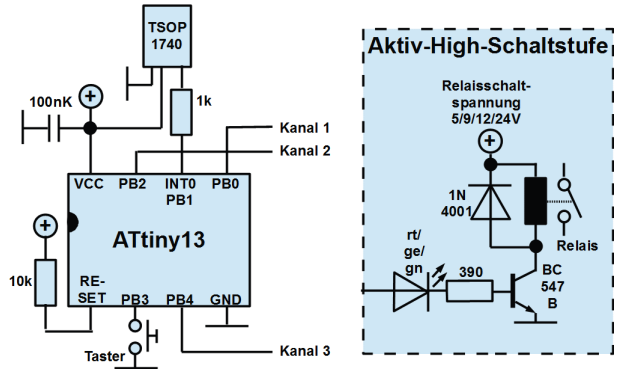
12.7.2 Hardware

Die Hardware gibt es in zwei Versionen:

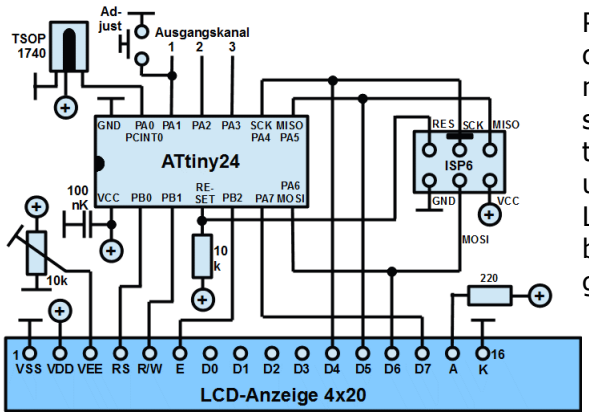
- Einfach, anwenderfreundlich und sparsam mit einem ATtiny13, und
- in der Komfortversion mit LCD-Anzeige für den Forscher und Entwickler, der herausfinden möchte, warum seine besondere Fernsteuerung vom ATtiny13 nicht erkannt wird.



Zunächst die Version mit dem ATtiny13 mit 5V-Relais-Schaltstufen. Verbraucht das Relais weniger als 50 mA Schaltstrom, kann auch auf den Treibertransistor verzichtet werden. Der 1k-Widerstand dient dazu, den TSOP vom Programmierpin der ISP-Schnittstelle zu entkoppeln. Der Taster an PB3 dient beim Einschalten dazu, den Lern- oder Adjust-Modus anzuwerfen.



Selbige Schaltung, nun mit Aktiv-High-Ausgangstreiber. Der kann mit beliebigen Relaisspannungen arbeiten.



Die Platzierung der Ausgangskanäle ist erkennbar. PA1 dient beim Programmstart auch zur Abfrage des Tasters, ob die Neuprogrammierung der Erkennungswerte erfolgen soll. Beim Design der Treiberstufe für diesen Kanal ist zu beachten, dass der interne Pull-Up nur etwa 50 kΩ beträgt und die LED und der Basisstrom den Eingang nicht zu stark auf Low aussteuern dürfen, weil sonst die Schaltung beim Starten immer in den Lern- oder Adjust-Modus gehen würde.

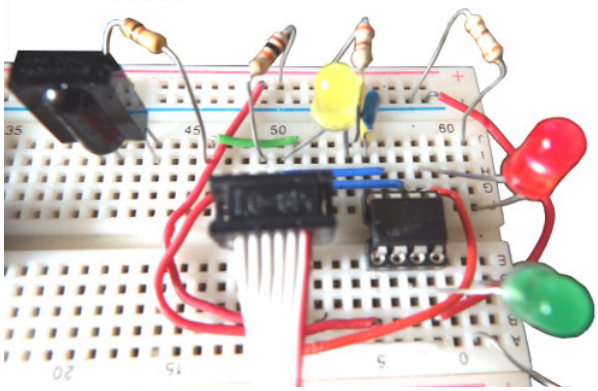
12.7.3 Bauteile

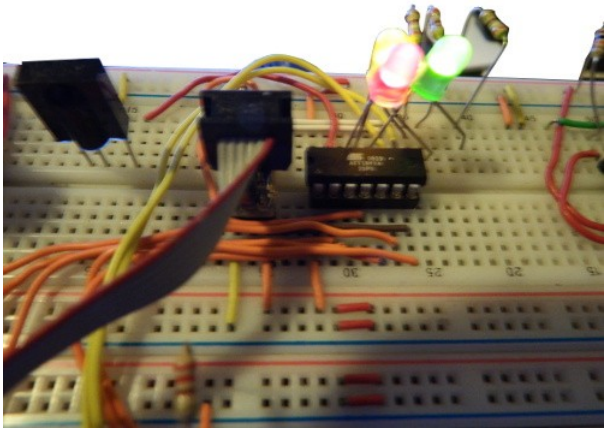
Der TSOP-IR-Empfänger wurde bereits oben beschrieben. Der 1k-Widerstand hat die Ringe braun-schwarz-rot-gold.

Die Bauteile der Treiberstufen sind nicht in der Hardwareliste angegeben, da sie zu eng mit dem Verwendungszweck des Geräts zu tun haben.

12.7.4 Aufbau

Das ist der Aufbau der Schaltung mit dem ATtiny13. Angeschlossen sind drei farbige LED, um den Erfolg zu kontrollieren.





Das gleiche mit dem ATtiny24.

12.7.5 Besonderheiten und Struktur des Programms

Das Programm tut weitgehend dasselbe, egal ob es auf einem ATtiny13 oder einem ATtiny24 läuft. Auf einem Atiny24 mit angeschlossener LCD wurde die entsprechenden Ausgaberoutinen daher mit einem Schalter eingefügt. Auch die Reset- und Interruptvektoren sind natürlich typspezifisch. Da der ATtiny13 keinen 16-Bit-Zähler hat, wird die Ermittlung der Signaldauer mit dem 8-Bit-Timer TC0 durchgeführt. Für lange Signale wurde noch ein überlaufgesteuerter MSB-Zähler hinzugefügt.

Im Unterschied zu den anderen Experimenten in dieser Lektion arbeitet der Zähler in dieser Version mit einem Vorteiler durch 64, so dass jeder Timer-Tick beim Tiny13 53, beim Tiny24 64 µs entspricht. Diese Auflösung erwies sich als hinreichend geeignet.

Um die Genauigkeit bei der Erkennung von Nullen und Einsen noch zu erhöhen, wird die Dauer von jeweils 16 Signalen gemittelt. Dazu werden diese in einem Puffer im SRAM gespeichert, nach Ende des Bursts aufsummiert und durch 16 geteilt. Der jeweils letzte ermittelte Wert wird verwendet und zusammen mit den erkannten Tastencodes gespeichert.

Die Erkennung der Tasten erfolgt mittels der letzten übertragenen 16 Bits. Das hat sich ebenfalls als hinreichend erwiesen.

Die Messung und Erkennung des IR-Signals erfolgt nahezu vollständig in der INT0- (ATtiny13) bzw. der PCINT0- (ATtiny24) Interrupt-Service-Routine. Die ist bis auf zwei Instruktionen zu Beginn für beide gleich. Der Ablauf der ISR ist wie folgt:

- Handelt es sich um ein langes Signal (Kopfsignal), wird ein Neustart eingeleitet (Rücksetzen der Anzahl empfangener Bits und der beiden Schieberegister).
- Ist das Signal kurz, wird es mit dem Vergleichswert (Null/Eins-Schwelle) verglichen und jeweils eine Null oder eine Eins in die Schieberegister eingeschoben.
- In allen beiden Fällen wird der Timer neu gestartet.

Die Ermittlung, wann alle Datenbits übertragen sind, erfolgt nach einem einstellbaren Zeitraum, nachdem der Timer nicht mehr von übertragenen IR-Signalen zurückgesetzt wird und das MSB des Timers den eingestellten Wert überschreitet. Erst dann wird eine Flagge gesetzt.

Die weitere Verarbeitung der empfangenen Bitkombinationen erfolgt im Hauptprogramm. Ist der Lernmodus aktiv, dann

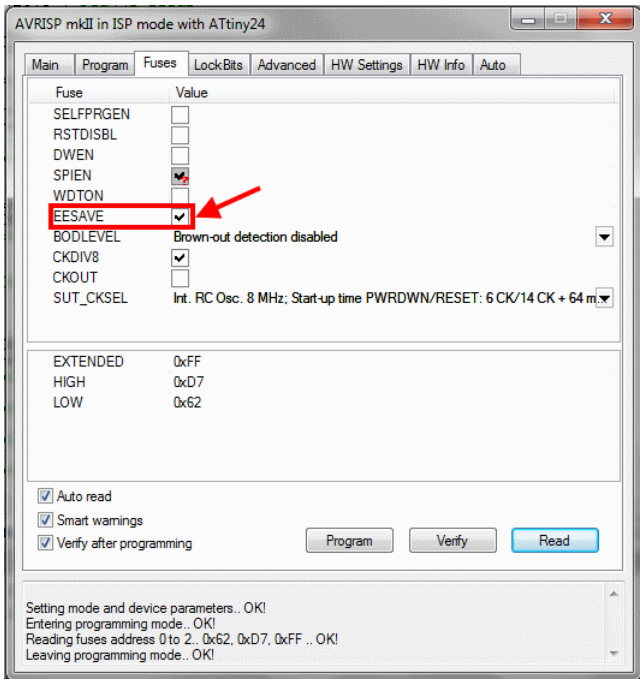
- blinkt diejenige LED, die als nächstes mit der Tastenzuordnung dran ist, im Sekundentakt,
- die jeweils beiden letzten eingegangenen Bitkombinationen werden auf Gleichheit überprüft,
- bei vorliegender Gleichheit wird geprüft, ob die betreffende Bitkombination bereits für eine andere Taste gespeichert ist, ist das der Fall, wird weiter geblinkt,
- ist das nicht der Fall, wird diese Bitkombination an der zutreffenden Position im SRAM gespeichert und die nächste Taste angesteuert. Zum Quittieren bleibt die LED im programmierten Kanal für fünf Sekunden an.
- Zuerst werden die drei Kanäle mit denjenigen Tasten programmiert, die zum Einschalten führen sollen. Danach folgen die drei Ausschalttasten.
- Sind alle sechs Tasten programmiert, werden die Tastenzuordnungen im EEPROM abgelegt und der Lernmodus wird verlassen.

Im normalen Betriebsmodus werden eingehende Bitkombinationen auf Übereinstimmung mit den gespeicherten Kombinationen verglichen und, bei Übereinstimmung, der betreffende Kanal ein- oder ausgeschaltet.

Folgende Bedingungen führen dazu, dass das Programm zu Beginn in den Lernmodus geht:

- Die dafür vorgesehene Taste ist beim Start gedrückt, oder
- beim Assemblieren wurde der Schalter cInput auf 1 gesetzt, oder
- beim Lesen der gespeicherten Bitkombinationen aus dem EEPROM liefert die Null/Eins-Schwelle eine Null oder 0xFF (natives EEPROM).

Beim EEPROM ist noch zu beachten, dass durch das Erase beim Programmieren des Maschinencodes auch das EEPROM geleert (mit 0xFF überschrieben) wird.



Dieses Verhalten kann man ändern, indem man die EESave-Fuse setzt. Dann wird beim Erase nicht der Inhalt des EEPROM überschrieben.

12.7.6 Gemessene IR-Fernsteuercodes

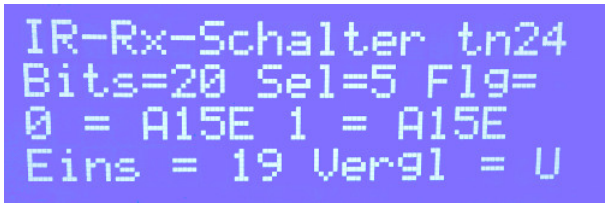
Der Inhalt des EEPROMs kann mit den Programmiertools des Studios jederzeit ausgelesen werden. Man erhält dann Dateien mit der Endung .hex, die folgendermaßen aussehen:

```
:100000001108F5C8B5881528754895A80DFF7FFF1C
:10001000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0
:10002000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0
:10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
:10004000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC0
:10005000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFB0
:10006000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA0
:10007000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF90
:00000001FF
```

Die Bytes des EEPROM beginnen mit "1108...", das letzte Byte ("1C") ist eine Prüfsumme. In Klarschrift ergibt sich daraus (jeweils in hex):

Gerät	Rot		Gelb		Grün		Null/ Eins
	Ein	Aus	Ein	Aus	Ein	Aus	
HDR	0805	C8C5	8885	2825	4528	A8A5	11
TV	20DF	10EF	A05F	906F	609F	50AF	12
Kamera	41BE	619E	817E	E11E	C13E	A15E	19

12.7.7 Diagnosen mit der ATtiny24-Version mit LCD

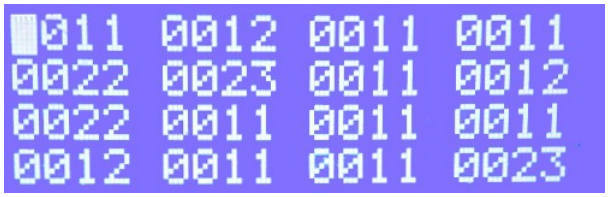


Dargestellt werden in Hexadezimal-Format:

- die Anzahl empfangener Bits,
- die Nummer der gerade programmierten oder erkannten Taste (gerade: Einschalten, ungerade: Ausschalten),
- Flaggen (A: Lernmodus, D: Doppel, E: Error),
- 0: letzte empfangene Bitkombination, 1: vorletzte empfangene Bitkombination,
- Null/Eins-Schwelle in Timerticks,
- Vergleichsergebnis (nur im Lernmodus).

Mit den Schaltern am Kopf des Quellcodes können zwei weitere Diagnosen durchgeführt werden.

Mit dem Schalter cDataB werden 16 gespeicherte Datenbytes sowie die daraus ermittelte Null/Eins-Schwelle dargestellt.



Mit dem Schalter cDataW kann man sich das Ergebnis wortweise ausgeben lassen.

12.7.8 Programm

Das Programm für diesen lernenden Fernsteuerempfänger im im Folgenden gelistet (den Quellcode im asm-Format gibt es hier, wenn der ATtiny24 mit angeschlossener LCD verwendet werden soll, muss noch die LCD-Include-Datei dazu).

Vor dem Assemblieren müssen die Schalter im Kopf dem eigenen Bedarf angepasst werden.

```
;
; ************************************
; * IR-Empfaenger 3-Kanal mit tn13/24 *
; * (C)2016 by www.gsc-elektronic.net *
; ************************************
;
; -------- Programmablauf -------
;
; Das Programm laeuft wahlweise auf ei-
; nem ATtiny13 oder auf einem ATtiny24
; (mit angeschlossener 4-zeiliger LCD,
; gibt die gewuenschten Empfangsdaten
; auf LCD aus).
;
; Schaltet drei Kanaele ein und aus,
; wenn im EEPROM gespeicherte Fernsteuer-
; signale ueber den IR-Sensor eintreffen.
; Die Fernsteuercodes koennen im Selbst-
; lernmodus einmalig eingestellt werden.
;
; Startet im Selbstlernmodus, wenn
; a) beim Programmstart der Jumper ge-
;    setzt ist, oder wenn
; b) die Daten im EEPROM fehlerhaft sind.
;
; Im Selbstlernmodus wartet das Programm
; auf IR-Signalfolgen. Sind zwei Signal-
; folgen eingegangen, deren letzte 16 Bits
; gleich waren, wird dies als Einschalt-
; code fuer den Kanal gespeichert. Die
; folgende korrekte Signalfolge wird als
; Ausschaltcode fuer diesen Kanal gespei-
; chert und dann der naechste Kanal ein-
; gestellt. Sind alle drei Kanaele mit
; ihren Ein- und Ausschaltcodes fertig
; eingestellt, werden die Daten ins EEPROM
; geschrieben und der Selbstlernmodus be-
; endet.
;
; Im Schaltmodus wartet das Programm auf
; eingehende Signalfolgen und vergleicht
; deren letzte 16 Bits mit den gespeicher-
; ten sechs Kanalkennungen. Bei Gleichheit
; mit einer der Kennungen wird die zuge-
; hoerige Aktion (an- oder ausschalten)
; ausgefuehrt.
;
; -------- Schalter -------------
.equ cAvrTyp = 13 ; Zieltyp, ATtn13 oder 24
.equ cAktivH = 1 ; 1 = Ausgaenge Aktiv High
.equ cLcd    = 0 ; 1 = LCD angeschlossen
.equ cInput  = 0 ; 1 = Eingabe erzwingen
.equ cDataB  = 0 ; 1 = Anzeige Bytes Rohdaten
.equ cDataW  = 0 ; 1 = Anzeige Words Rohdaten
;
.if (cAvrTyp != 13) && (cAvrTyp != 24)
  .error "Falscher Typ"
  .endif
.if (cLcd == 1) && (cAvrTyp == 13)
  .error "LCD passt nicht zum Typ"
  .endif
;
.NOLIST
.if cAvrTyp == 13
  .INCLUDE "tn13def.inc"
  .else
  .INCLUDE "tn24def.inc"
  .endif
.LIST
;
.if cAvrTyp == 13
; Hardware: ATtiny13
;                  _________
;                 /         |
;   +5V/10k o--|Reset  VCC|--o +5V
;               |          |
;    Taster o--|PB3    PB2|--o Ausgang ge
;               |          |
;Ausgang gn o--|PB4    PB1|--o TSOP1740
;               |          |
;        0V o--|GND    PB0|--o Ausgang rt
;               |__________|
;
  .else
; Hardware: ATtiny24
;                  _________
;                 /         |
;       +5V o--|VCC    GND|--o 0V
;               |          |
;    LCD-RS o--|PB0    PA0|--o TSOP1740
;               |          |
;   LCD-R/W o--|PB1    PA1|--o LED rt
;               |          |
;     RESET o--|RES    PA2|--o LED ge
;               |          |
;     LCD-E o--|PB2    PA3|--o LED gn
;               |          |
;    LCD-D7 o--|PA7    PA4|--o LCD-D4/SCK
;               |          |
;MOSI/LCD-D6o--|PA6    PA5|--o LCD-D5/MISO
;               |__________|
;
  .endif
;
; -------- Ports, Portpins ----------
.if cAvrTyp == 13
  ; Ports
  .equ pOut = PORTB ; Ausgabeport tn13
```

```
  .equ pDir = DDRB ; Richtungsport
  .equ pIn  = PINB ; Eingabeport
  ; Portbits
  .equ bIrO = PORTB1 ; IR-Empfaenger Ausg.
  .equ bIrD = DDB1 ; IR-Empfaenger Richtung
  .equ bIrI = PINB1 ; IR-Empfaenger Eingang
  .equ bRtO = PORTB0 ; Schaltausgang rot
  .equ bRtD = DDB0 ; Richtung rot
  .equ bGeO = PORTB2 ; Schaltausgang gelb
  .equ bGeD = DDB2 ; Richtung gelb
  .equ bGnO = PORTB4 ; Schaltausgang gruen
  .equ bGnD = DDB4 ; Richtung gruen
  .equ bTaO = PORTB3 ; Taste Ausg. Pullup
  .equ bTaI = PINB3 ; Taste Eingang
  .else
  ; Ports ATtiny24
  .equ pOut = PORTA ; Ausgabeport tn24
  .equ pDir = DDRA ; Richtungsport
  .equ pIn  = PINA ; Eingabeport
  ; Portpins
  .equ bIrO = PORTA0 ; IR-Empfaenger Ausg.
  .equ bIrD = DDA0 ; IR-Empfaenger Richtung
  .equ bIrI = PINA0 ; IR-Empfaenger Eingang
  .equ bRtO = PORTA1 ; Schaltausgang rot
  .equ bRtD = DDA1 ; Richtung rot
  .equ bGeO = PORTA2 ; Schaltausgang gelb
  .equ bGeD = DDA2 ; Richtung gelb
  .equ bGnO = PORTA3 ; Schaltausgang gruen
  .equ bGnD = DDA3 ; Richtung gruen
  .equ bTaO = PORTA1 ; Taste Ausg. Pullup
  .equ bTaD = DDA1 ; Taste Richtung
  .equ bTaI = PINA1 ; Taste Eingang
  .endif
;
; -------- Timing, IR-Signale -------
.if cAvrTyp == 13
  ; Takt:                1200000 Hz
  ; TC0-Prescaler             64
  ; TC0-Tick             18.750 Hz
  ;                          53,33 us
  ; TC0-Overflow         13.563 us
  .equ cTcTick = 53
  ;
  .else ; ATtiny24
  ; Takt:                1000000 Hz
  ; TC0-Prescaler             64
  ; TC0-Tick              15.625 Hz
  ;                           64 us
  ; TC0-Overflow          16.384 us
  ; Pause bis Auswertung  30.000 us
  ; Mittelwert Nullen        448 us
  ; Mittelwert Einsen       1.288 us
  ; Null/Eins-Schwelle       868 us
  .equ cTcTick = 64
  .endif
; Null/Eins-Schwelle:
.equ cEins = 868/cTcTick ; tn13:16; tn24:13
; Kopf/Daten-Schwelle
.equ cKopf = 4*cEins ; tn13:48; tn24:39
; Auswertepause nach letztem Bit
.equ cAusw = 1
; Halbe Sekunde, MSB
.equ cSekH = 500000/cTcTick / 256 ; 36/30
;
; -------- Register -----------------
; benutzt: R0 von LCD-Routine (nur LCD)
.def rBits= R5 ; Empfangene Bits
.def rBitsO=R6 ; Empfangene Bits Anzeige
.def rEins= R7 ; Null/Eins-Schwelle
.def rI1L = R8 ; vorletzter Bitburst, LSB
.def rI1H = R9 ; dto., MSB
.def rI0L = R10 ; letzter Bitburst, LSB
.def rI0H = R11 ; dto., MSB
.def rIRL = R12 ; aktueller Bitburst, LSB
.def rIRH = R13 ; dto., MSB
.def rCntH= R14 ; MSB TC0-Zaehler
.def rSreg= R15 ; SREG Status
.def rmp  = R16 ; Vielzweckregister
.def rimp = R17 ; dto., Interrupts
.def rFlag= R18 ; Flaggen
  .equ bOvf = 0 ; Ueberlauf, Signal ausw.
  .equ bAdj = 1 ; Lernmodus
  .equ bSek = 2 ; Halbe Sekunde zu Ende
  .equ bLng = 3 ; Lange Bestaetigung
  .equ bZwo = 4 ; Zweite Messung
  .equ bEqu = 5 ; Gleichheit Burst 1 und 2
  .equ bDop = 6 ; Doppel empfangen
  .equ bErr = 7 ; Keine Uebereinstimmung
.def rSel = R19 ; 0: Rt Ein, 1: Rt Aus
; 2: Ge Ein, 3: Ge Aus, 4: Gn Ein, 5: Gn Aus
.def rCtr = R20 ; Zaehler EEPROM/Input
.def rmo = R21 ; fuer LCD und LED-Steuerung
.if cLcd == 1
  .def rLine = R22
  .def rLese = R23
  .endif
; frei: R24 .. R25
; verwendet: X fuer Zeigeroperationen
; verwendet: Y fuer Speichern im SRAM
; verwendet: Z fuer diverse Zwecke
;
; -------- SRAM -------------
.DSEG
.ORG 0x0060
sBuffer: ; Datenspeicher fuer Mittelwert-
.Byte 16 ; ermittlung Nullen und Einsen
sBufferEnde:
;
sCodes:
.Byte 4 ; Erkennungscodes Rot, Ein/Aus
.Byte 4 ; dto., Gelb, Ein/Aus
.Byte 4 ; dto., Gruen, Ein/Aus
sCodesTastenEnde:
sCodesEins:
.Byte 1 ; Null/Eins-Schwelle
sCodesEnde:
;
; -------- Reset- und Int-Vektoren ---
.CSEG
.ORG 0x0000
.if cAvrTyp == 13
  rjmp Start ; RESET-Vektor
  rjmp Int0Isr ; INT0 Ext. Int Request 0
  reti ; PCINT0 Pin Change Int Request 0
  rjmp TC0OIsr ; TIM0_OVF TC0 Overflow
  reti ; EE_RDY EEPROM Ready
  reti ; ANA_COMP Analog Comparator
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; WDT Watchdog Time-out
  reti ; ADC ADC Conversion Complete
  .else
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0 External Int Request 0
  rjmp Pci0Isr ; PCINT0 Pin Change Int 0
  reti ; PCINT1 Pin Change Int Request 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT TC1 Capture
  reti ; TIM1_COMPA TC1 Comp Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF Timer/Counter1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  rjmp Tc0OIsr ; TC0_OVF, MSB Timer
  reti ; ANA_COMP Analog Comparator
  reti ; ADC ADC Conversion Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
  .endif
;
; -------- Int Service ------
;
; INT0 (ATtiny13) bzw. PCINT (ATtiny24) Interrupt
```

```
;   wird aufgerufen bei
;   a) Low-High Pegelwechseln beim ATtiny13, oder
;   b) jedem Pegelwechsel beim ATtiny24.
;
; Aktive Pulslaenge vom IR-Empfaenger auswerten.
;
.if cAvrTyp == 24
Pci0Isr: ; Bei ATtiny24 PCINT0-Interrupt
  sbic pIn,bIRI ; ignoriere die
  reti ; Low-Phasen vom IR-Empfaenger
  .else
Int0Isr: ; Bei ATtiny13 INT0-Interrupt
  .endif
  in rSreg,SREG ; SREG sichern
.if cDataW == 1
    ;
    ; wird ausgefuehrt wenn die Roh-
    ; daten auf LCD ausgegeben werden
    ; sollen, schreibt aktive Pulsdauer
    ; in SRAM-Puffer
    ;
  st Y+,rCntH
  in rimp,TCNT0
  st Y+,rimp
  clr rimp
  out TCNT0,rimp
  mov rCntH,rimp
  cpi YL,Low(sBufferEnde)
  brcs Int0IsrWRet
  sbr rFlag,1<<bOvf
  ldi YH,High(sBuffer)
  ldi YL,Low(sBuffer)
Int0IsrWRet:
  out SREG,rSreg
  reti
  .endif
  tst rCntH ; Langes Signal
  brne Int0IsrLang
  in rimp,TCNT0
  cpi rimp,cKopf ; laenger als Kopf
  brcs Int0IsrKurz
  ; Kopfsignal empfangen, Neustart
Int0IsrLang:
  clr rimp ; starte Zaehler neu
  out TCNT0,rimp
  clr rCntH ; loesche MSB Zaehlerbyte
  mov rBitsO,rBits
  clr rBits ; Anzahl empfangene Bits auf 0
  clr rIRL ; loesche Bitsammel-Register
  clr rIRH
  out SREG,rSreg ; fertig
  reti
Int0IsrKurz:
  ; Kurzes Datensignal
  in rimp,TCNT0 ; lese Zaehlerstand
  st Y+,rimp ; speichere im SRAM
  cp rEins,rimp ; vergleiche Dauer Null/Eins
  rol rIRL ; schiebe Bit in Sammelregister
  rol rIRH
  inc rBits ; erhoehe Anzahl Bits
  cpi YL,Low(sBufferende) ; Zeigerende?
  brne Int0IsrRet ; nicht oberhalb
  ldi YH,High(sBuffer) ; Neubeginn,
  ldi YL,Low(sBuffer) ; wieder an Anfang
Int0IsrRet:
  ; Rueckkehr kurzes Signal
  clr rimp ; loesche Zaehler
  out TCNT0,rimp
  out SREG,rSreg ; SREG herstellen
  reti
;
; Zaehler-Ueberlaeufe auswerten
TC0Isr:
  in rSreg,SREG ; SREG sichern
  inc rCntH ; Ueberlaeufe zaehlen
  mov rimp,rCntH ; laengere Pause?
  cpi rimp,cSekH ; halbe Sekunde um?
  brne TC0Isr1
  sbr rFlag,1<<bSek ; Sekundenflagge
  clr rCntH
  rjmp TC0IsrRet
TC0Isr1:
  cpi rimp,cAusw ; Auswerten?
  brne TC0IsrRet
  tst rBits
  breq TC0IsrRet
;.if cDataW != 1
  sbr rFlag,1<<bOvf ; Auswerteflagge
;       .endif
  mov rI0H,rIRH ; empfangene Bits
  mov rI0L,rIRL ; in Null-Speicher
  mov rBitsO,rBits ; Kop. Anzahl Bits
  clr rBits
TC0IsrRet:
  out SREG,rSreg ; SREG herstellen
  reti
;
; -------- Start, Init ------
Start:
  ; Stapel initiieren
  ldi rmp,LOW(RAMEND)
  out SPL,rmp
  ; Init Port-Ausgaenge Richtung
  sbi pDir,bRtD ; Ausgaenge, Richtung
  sbi pDir,bGeD
  sbi pDir,bGnD
  ; Port-Ausgaenge inaktiv
.if cAktivH == 1
  cbi pOut,bRtO ; Ausgaenge, Startwert
  cbi pOut,bGeO
  cbi pOut,bGnO
  .else
  sbi pOut,bRtO
  sbi pOut,bGeO
  sbi pOut,bGnO
  .endif
  ; Init Taste und IR-Emfaengereingang
  sbi pOut,bIrO ; Pullup IR-Empfaenger
  ; Flaggen loeschen
  clr rFlag
  ; Voreinstellung Null/Eins-Erkennung
  ldi rmp,cEins
  mov rEins,rmp
  ; Bufferzeiger fuer Werteaufzeichnung
  ldi YH,High(sBuffer)
  ldi YL,Low(sBuffer)
  ; Lesen Erkennungswerte aus EEPROM
  rcall LeseCodes
  lds rEins,sCodesEins
  ; Wenn Taste aktiv dann Adjust
.if cAvrTyp == 24
  cbi pDir,bTaD ; Zeitweise als Eingang
  sbi pOut,bTaO ; Pullup einschalten
  .else
  sbi pOut,bTaO ; Pullup einschalten
  .endif
  nop ; etwas abwarten
  nop
  sbis pIn,bTaI ; wenn Tasteneingang
  sbr rFlag,1<<bAdj ; in Lern-Modus
.if cAvrTyp == 24
  sbi pDir,bTaD ; Wieder als Ausgang
  .endif
  ; Wenn Force-Input alles loeschen
.if cInput == 1
  rcall StartInput ; Input vorbereiten
  .else
  sbrc rFlag,bAdj ; Lern-Modus?
  rcall StartInput ; Input vorbereiten
  .endif
  ; Wenn LCD angeschlossen, Init LCD
.if cLcd == 1
    ; LCD-Kontroll-Ports initiieren
  cbi pLcdCO,bLcdCOE ; Ausgang E Null
```

```
  cbi pLcdCO,bLcdCORS ; Ausgang RS Null
  cbi pLcdCO,bLcdCORW ; Ausgang RW Null
  sbi pLcdCR,bLcdCRE ; Ausgang E Richtung
  sbi pLcdCR,bLcdCRRS ; Ausgang RS Richtung
  sbi pLcdCR,bLcdCRRW ; Ausgang RW Richtung
  ; Datenausgangsport LCD
  rcall LcdStart ; Init und Textausgabe
  .endif
  ; Starten Timer TC0
  ldi rmp,(1<<CS01)|(1<<CS00) ; Prescaler auf 64
  out TCCR0B,rmp
  ldi rmp,1<<TOIE0 ; Overflow Int
  out TIMSK0,rmp
  ; Sleep mode idle, INT0/PCINT0 IR-Signale
.if cAvrTyp == 13
  ldi rmp,(1<<SE)|(1<<ISC01)
  out MCUCR,rmp
  ldi rmp,1<<INT0 ; INT0-Interrupts
  out GIMSK,rmp
  .else
  ldi rmp,1<<SE ; Sleep-Mode Idle
  out MCUCR,rmp
  ldi rmp,1<<PCINT0 ; Pin 0 Lvl chg INTs
  out PCMSK0,rmp ; in Maske 0
  ldi rmp,1<<PCIE0 ; Int PCINT0 enable
  out GIMSK,rmp ; in Int-Maske
  .endif
  ; Interrupts enable
  sei
Schleife:
  sleep ; Schlafen
  nop ; Wecken
  sbrc rFlag,bOvf ; Auswerten?
  rcall Overflow ; ja
  sbrc rFlag,bSek ; Sekundentakt?
  rcall Sekunde ; ja
  rjmp Schleife
;
; Routine Sekunde
;   wird ausgeloest von der Flagge bSek
;   zu jeder halben Sekunde
;
; Stellt fest, ob der Zustand Lernen aktiv
; ist. Wenn ja, prueft die Routine ob ein
; IR-Burst eingegangen ist. Wenn nein,
; wird die LED an und aus geschaltet. Wenn
; ja, wird geprueft ob schon 5 Sekunden
; vorbei sind. Wenn nein wird weiter ge-
; wartet. Wenn ja, wird die aktuelle LED
; ausgeschaltet und der aktuelle Eingabeka-
; nal erhoeht. Sind alle Kanaele fertig
; eingestellt, werden die Codes in das
; EEPROM geschrieben und der Lernmodus
; beendet.
;
Sekunde:
  ; Halbe Sekunde um
  cbr rFlag,1<<bSek
  ; Nur im Adjust-Modus
  sbrs rFlag,bAdj
  ret ; nicht adjust
  ; Lange An-Phase?
  sbrs rFlag,bLng ; Bestaetigung?
  rjmp LedBlink ; Nein
  ; Lange An-Phase zu Ende?
  dec rCtr ; Fuenf-Sekunden zaehlen
  brne SekundeRet ; Nein
  ; Lange An-Phase zu Ende
  cbr rFlag,1<<bLng ; Flagge clear
  ; Naechste Eingabeposition
  rcall LedOff ; aktuelle LED aus
  subi rSel,-2 ; zwei dazu
  cpi rSel,6 ; bis rSel = 6
  brcs LedBlink ; noch nicht
  brne SekundeEnde ; alle eingestellt
  ldi rSel,1 ; Weiter mit Aus-Codes
  rjmp LedBlink ; blinken
SekundeEnde:
  ; alle Kanaele eingestellt
  rcall SchreibeCodes ; EEPROM-Write
  clr rSel ; Neustart mit Kanal Null
  cbr rFlag,1<<bAdj ; Adjust clear
  ; alle Ausgaenge Aus
.if cAktivH == 1
  cbi pOut,bRtO ; Ausgang Null
  cbi pOut,bGeO
  cbi pOut,bGnO
  .else
  sbi pOut,bRtO ; Ausgang High
  sbi pOut,bGeO
  sbi pOut,bGnO
  .endif
SekundeRet:
  ret
;
; Routine LedBlink
;   wird bei aktivem Einlesen in jeder
;   halben Sekunde aufgerufen und schaltet
;   die LED im aktuellen Eingabekanal
;   in rSel an und aus
;
LedBlink:
  rcall GetOutPin ; Aktiv-Pin in rmp
  in rmo,pOut ; Pins lesen
  eor rmp,rmo ; Polaritaet umkehren
  out pOut,rmp ; Pins schreiben
  ret
;
; Routine LedOn
;   schaltet die LED des aktuellen Eingabe-
;   kanals rSel an
;
; Wertet aus, ob die LED aktiv high oder low
; angeschlossen ist
;
LedOn:
  rcall GetOutPin ; Aktiv-Pin in rmp
  in rmo,pOut ; Pins lesen
.if cAktivH == 1
  or rmp,rmo ; Pin aktivieren
  .else
  com rmp ; umkehren
  and rmp,rmo ; Pin auf Null
  .endif
  out pOut,rmp ; Pins schreiben
  ret
;
; Routine LedOff
;   schaltet die LED des aktuellen Eingabe-
;   kanals rSel aus
;
; Wertet aus, ob die LED aktiv high oder low
; angeschlossen ist
;
LedOff:
  rcall GetOutPin ; Aktiv-Pin in rmp
  in rmo,pOut ; Pins lesen
.if cAktivH == 1
  com rmp ; umkehren
  and rmp,rmo ; Pin auf Null
  .else
  or rmp,rmo ; Pin auf Eins
  .endif
  out pOut,rmp ; Pins schreiben
  ret
;
; Routine GetOutPin
;   Holt aktuellen Outputpin in das Register
;   rmp
;
; Input: rSel gibt aktuellen Eingabekanal an
; Output: rmp gibt Maske des zugehoerigen
;         LED-Pins zurueck
;
```

```
GetOutPin:
  mov rmp,rSel ; aktuelle Position
  lsr rmp ; Ein/Aus-Bit weg schieben
  cpi rmp,1 ; LED gelb aktiv?
  brcs GetOutPinRt ; nein, rot
  brne GetOutPinGn ; nein, gruen
  ldi rmp,1<<bGeO ; gelb
  ret
GetOutPinRt:
  ldi rmp,1<<bRtO ; rot
  ret
GetOutPinGn:
  ldi rmp,1<<bGnO ; gruen
  ret
;
; Routine Overflow
;   ausgeloest durch Ueberlauf des Timers
;   nach Empfang des letzten IR-Signals
;   Ergebnis der empfangenen Signale wird
;   ausgewertet
;
Overflow:
  cbr rFlag,1<<bOvf ; Flagge clear
.if cLcd == 1
  .if cDataW == 1
  ldi rmp,0x01 ; LCD loeschen
  rcall LcdC4Byte
  ldi XH,High(sBuffer) ; X Zeiger
  ldi XL,Low(sBuffer) ; auf Buffer
  clr rLine ; in Zeile 0
  RohdatenW1:
  ld rmp,X+
  rcall LcdHex
  ld rmp,X+
  rcall LcdHex
  ldi rmp,' '
  rcall LcdD4Byte
  mov rmp,XL
  andi rmp,0x07
  brne RohdatenW1
  inc rLine
  cpi rline,4
  brcc RohdatenWRet
  mov ZH,rLine
  clr ZL
  rcall LcdPos
  rjmp RohdatenW1
  ldi YH,High(sBuffer)
  ldi YL,Low(sBuffer)
  ldi rmp,1<<PCIE0
;       out GIMSK,rmp
  RohdatenWRet:
  ret
  .endif
  .endif
  ; pruefen ob Lernmodus
  sbrs rFlag,bAdj
  rjmp OverflowCheck ; nein
  ; Ab hier Lernmodus
OverflowCalc:
  clr rmp ; Ints aus
  out GIMSK,rmp
  out TIMSK0,rmp
  ; Durchschnittsdauer der letzten
  ; 16 empfangenen Datenbits
  ; berechnen
  ldi ZH,High(sBuffer) ; Anfang
  ldi ZL,Low(sBuffer) ; Buffer
  clr XH ; X ist Summe
  clr XL
OverflowSum:
  ld rmp,Z+ ; Byte aus Buffer
  add XL,rmp ; addieren
  ldi rmp,0 ; Ueberlauf behandeln
  adc XH,rmp
  cpi ZL,LOW(sBufferende) ; Ende?
  brne OverflowSum ; nein, weiter
  lsr XH ; durch 2
  ror XL
  lsr XH ; durch 4
  ror XL
  lsr XH ; durch 8
  ror XL
  lsr XH ; durch 16
  ror XL
  mov rEins,XL ; in Erkennungsreg.
  sts sCodesEins,XL ; in SRAM
  ; Zwei Messungen vergleichen
  sbrc rFlag,bZwo ; Zweiter Burst?
  rjmp OverflowZwei ; beide vergl.
  sbr rFlag,1<<bZwo ; Flagge setzen
  mov rI1L,rI0L ; Messwert kopieren
  mov rI1H,rI0H
  rjmp OverflowNormal
OverflowZwei:
  ; Zweiten Burst vergleichen
  cbr rFlag,(1<<bZwo)|(1<<bEqu)
  cp rI1L,rI0L ; LSB vergleichen
  brne OverflowNormal ; ungleich
  cp rI1H,rI0H ; MSB vergleichen
  brne OverflowNormal ; ungleich
  ; Gleichheit, pruefe auf Doppel
  sbr rFlag,1<<bEqu ; Gleich-Flagge
  ldi XH,High(sCodes) ; X auf Codes
  ldi XL,Low(sCodes) ; im SRAM
OverflowDoppel:
  cpi XL,Low(sCodesTastenEnde)
  breq OverflowTasteOk ; Ende Vergl.
  ld rmp,X+ ; Lese LSB
  cp rmp,rI1L ; vergleiche LSB
  ld rmp,X+ ; Lese MSB
  brne OverflowDoppel ; Doppel gef.
  cp rmp,rI1H ; vergleiche MSB
  brne OverflowDoppel ;
  sbr rFlag,1<<bDop ; Doppelflagge
  rjmp OverflowNormal
  ; speichere Code, starte lange Pause
OverflowTasteOk:
  ldi XH,High(sCodes) ; Code Start
  ldi XL,Low(sCodes)
  mov rmp,rSel ; aktive Taste
  lsl rmp ; mal zwei
  add XL,rmp ; Zu Zeiger addieren
  ldi rmp,0 ; Ueberlauf behandeln
  adc XH,rmp
  st X+,rI1L ; Wert ablegen
  st X,rI1H
  rcall LedOn ; LED einschalten
  ldi rCtr,5 ; lange Pause
  sbr rFlag,(1<<bEqu)|(1<<bLng)
  rjmp OverflowNormal
OverflowCheck:
  ; Empfangenen Datensatz auswerten
  ldi XH,High(sCodes) ; Zeiger Codes
  ldi XL,Low(sCodes)
  ser rSel ; mit 0xFF beginnen
OverflowCheck1:
  inc rSel ; naechster Wert
  cpi rSel,6 ; Ende erreicht?
  brcc OverflowNf ; ja
  ld rmp,X+ ; Wert aus SRAM lesen
  cp rmp,rI0L ; LSB vergleichen
  ld rmp,X+ ; MSB lesen
  brne OverflowCheck1 ; ungleich
  cp rmp,rI0H ; MSB vergleichen
  brne OverflowCheck1 ; ungleich
  ; Korrekte Taste erkannt
  cbr rFlag,(1<<bErr)|(1<<bDop)
  sbrs rSel,0 ; Pin Einschalten?
  rcall LedOn ; Pin einschalten
  sbrc rSel,0 ; Pin ausschalten?
  rcall LedOff ; Pin ausschalten
  mov rI1H,rI0H ; Wert kopieren
  mov rI1L,rI0L
```

```
  rjmp OverflowNormal
OverflowNf:
  ; Keine Uebereinstimmung gefunden
  clr rSel ; Null setzen
  sbr rFlag,1<<bErr ; Flagge setzen
OverflowNormal:
; Wenn LCD angeschlossen: Ausgabe
.if cLcd == 1
  ; wenn Rohdatenausgabe eingestellt
  .if cDataB == 1
    ldi rmp,0x01 ; LCD loeschen
    rcall LcdC4Byte
    ldi XH,High(sBuffer) ; X Zeiger
    ldi XL,Low(sBuffer) ; auf Buffer
    clr rLine ; in Zeile 0
  Rohdaten1:
    ld rmp,X+ ; Byte lesen
    rcall LcdHex ; in Hex ausgeben
    ldi rmp,' ' ; Leerzeichen
    rcall LcdD4Byte
    cpi XL,Low(sBufferende) ; fertig?
    brcc RohdatenEnde
    mov rmp,XL ; vier Bytes pro Zeile
    andi rmp,0x03 ; naechste Zeile?
    brne Rohdaten1 ; nein
    inc rLine ; naechste Zeile
    mov ZH,rLine
    clr ZL
    rcall LcdPos
    rjmp Rohdaten1 ; weiter
  RohdatenEnde:
    mov rmp,rEins ; Null/Eins-
    rcall LcdHex ; Schwelle ausgeben
    .else
    ; LCD angeschlossen, diverse Daten
    ldi ZH,1 ; Anzahl Bits in Zeile 2
    ldi ZL,5
    rcall LcdPos
    mov rmp,rBitsO
    rcall LcdHex
    ldi ZH,1 ; rSel in Zeile 2
    ldi ZL,12
    rcall LcdPos
    mov rmp,rSel
    andi rmp,0x07
    subi rmp,-'0'
    rcall LcdD4Byte
    ldi ZH,1 ; Flaggen in Zeile 2
    ldi ZL,18
    rcall LcdPos
    ldi rmp,' '
    sbrc rFlag,bAdj
    ldi rmp,'A'
    sbrc rFlag,bErr
    ldi rmp,'E'
    rcall LcdD4Byte
    ldi ZH,2 ; Letzte zwei Messwerte
    ldi ZL,4 ; in Zeile 3
    rcall LcdPos
    mov rmp,rI0H
    rcall LcdHex
    mov rmp,rI0L
    rcall LcdHex
    ldi ZH,2
    ldi ZL,13
    rcall LcdPos
    mov rmp,rI1H
    rcall LcdHex
    mov rmp,rI1L
    rcall LcdHex
    ldi ZH,3 ; Null/Eins-Schwelle
    ldi ZL,7 ; in Zeile 3
    rcall LcdPos
    mov rmp,rEins
    rcall LcdHex
    ldi ZH,3 ; Gleich/Ungleich/Doppel-
    ldi ZL,18 ; Flagge in Zeile 4
    rcall LcdPos
    ldi rmp,'U' ; Ungleich
    sbrc rFlag,bEqu ; Gleich
    ldi rmp,'G'
    sbrc rFlag,bDop ; Doppel?
    ldi rmp,'D'
    sbrc rFlag,bErr ; Error?
    ldi rmp,'E'
    rcall LcdD4Byte
    .endif
  .endif
  ; Interrupts wieder einschalten
  clr rCntH
  ldi rmp,1<<TOIE0 ; Timer Int
  out TIMSK0,rmp
.if cAvrTyp == 13
  ldi rmp,1<<INT0 ; INT0-Interrupts
  out GIMSK,rmp
  .else
  ldi rmp,1<<PCIE0 ; Int PCINT0 enable
  out GIMSK,rmp ; in Int-Maske
  .endif
  ret
;
; Lese Codes aus dem EEPROM
;   liest den Inhalt des EEPROMs in das SRAM
;   wird aufgerufen beim Init
;
; Anzahl zu lesender Bytes: 13
; Prueft, ob die Null/Eins-Schwelle auf 0xFF
; steht. Falls ja (EEPROM ist leer), wird der
; Lernmodus gestartet.
;
LeseCodes:
  ldi ZH,0 ; Zeiger auf EEPROM-Adresse
  ldi ZL,0
  ldi XH,High(sCodes) ; Zeiger auf SRAM
  ldi XL,Low(sCodes)
  ldi rmp,sCodesEnde-sCodes ; Anzahl
  mov rCtr,rmp
LeseCodes1:
  sbic EECR,EEPE ; warte bis bereit
  rjmp LeseCodes1
  out EEARL,ZL ; Adresszaehler
  sbi EECR,EERE ; Read enable
  in rmp,EEDR ; Byte lesen
  st X+,rmp ; in SRAM speichern
  adiw ZL,1 ; naechste Adresse
  dec rCtr ; Zaehler
  brne LeseCodes1 ; weiter lesen
  cpi rmp,0x00 ; Null/Eins auf Null?
  breq LeseCodes2 ; ja, Daten korrupt
  cpi rmp,0xFF ; Null/Eins auf FF?
  brne LeseCodes3 ; ja, Daten korrupt
LeseCodes2:
  rjmp StartInput ; Daten korrupt, neu
LeseCodes3: ; Daten ok
  ret
;
; Schreibe Codes ins EEPROM
;   schreibt alle gelernten Codes in das EEPROM
;   wird aufgerufen wenn alle Codes eingelesen
;   sind
;
; Anzahl zu schreibender Bytes: 13
;
SchreibeCodes:
  ldi ZH,0 ; Zeiger auf EEPROM-Adresse
  ldi ZL,0
  ldi XH,High(sCodes) ; Zeiger auf SRAM
  ldi XL,Low(sCodes)
  ldi rmp,sCodesEnde-sCodes
  mov rCtr,rmp
SchreibeCodes1:
  sbic EECR,EEPE ; Warte bis bereit
  rjmp SchreibeCodes1 ; weiter warten
  clr rmp ; Erase und Schreiben
```

```
  out EECR,rmp
  out EEARL,ZL ; Adresse ausgeben
  ld rmp,X+ ; Byte aus SRAM lesen
  out EEDR,rmp ; in Datenregister
  cli ; disable ints wg. Timeout
  sbi EECR, EEMPE ; Master Program
  sbi EECR, EEPE ; Program enable
  sei ; enable Interrupts
  adiw ZL,1 ; Adresse erhoehen
  dec rCtr ; Bytezaehler
  brne SchreibeCodes1 ; weiter
  ret
;
; Routine StartInput
;   initiiert einen Neustart des Codelernens,
;   wird aufgerufen
;   a) beim Init, wenn noch keine Codes ge-
;      setzt sind,
;   b) beim Init, wenn der Jumper geschlos-
;      sen ist,
;   c) beim Init wenn beim EEPROM-Lesen Feh-
;      ler festgestellt werden
;
; Fuert folgende Schritte aus:
; 1. Loeschen aller Codes durch Ueberschrei-
;    ben mit Nullen
; 2. Setzen eines Default Null/Eins-Schwel-
;    lenwertes
; 3. Trainingskanal auf Null
; 4. Setzen der bAdj-Flagge
;
StartInput:
  ldi ZH,High(sCodes) ; Zeiger
  ldi ZL,Low(sCodes)
  clr rmp ; Nullen
StartInput1:
  st Z+,rmp ; Codes loeschen
  cpi ZL,Low(sCodesEnde) ; Ende?
  brne StartInput1 ; weiter
  ldi rmp,cEins ; Def Null/Eins
  sts sCodesEins,rmp ; in SRAM
  mov rEins,rmp ; und in Register
  clr rSel ; Neustart
  sbr rFlag,1<<bAdj ; Lernmodus
  ret
;
.if cLcd == 1
  ; LCD-Include laden
  .include "Lcd4Busy.inc"
  ; LCD starten
  LcdStart:
    rcall LcdInit ; Init LCD
    ldi ZH,High(2*Ausgabetext)
    ldi ZL,Low(2*Ausgabetext)
    rjmp LcdText ; Text ausgeben
  ; LCD-Ausgabetext
  Ausgabetext:
  .db "IR-Rx-Schalter tn24",0x0D
  .db "Bits=xx Sel=x Flg=x",0x0D
  ;   5      12    18
  .db "0 = xxxx 1 = xxxx  ",0x0D
  ;        4        13
  .db "Eins = xx Vergl = x",0xFE
  ;           7          18
    ;
    ; Routine LcdHex
    ; gibt ein Byte in rmp in Hex auf LCD aus
    ; wird von der Ausgabe der Rohdaten aufge-
    ; rufen
    ;
  LcdHex:
    push rmp ; Byte retten
    swap rmp ; oberes Nibble zuerst
    rcall LcdHexN ; Nibble ausgeben
    pop rmp ; rmp wieder herstellen
    ; Gib Nibble in Hex auf LCD aus
  LcdHexN:
    andi rmp,0x0F ; unteres Nibble maskieren
    subi rmp,-'0' ; ASCII-Null dazu addieren
    cpi rmp,'9'+1 ; A bis F?
    brcs LcdHexN1 ; nein
    subi rmp,-7 ; auf A bis F
  LcdHexN1:
    rjmp LcdD4Byte ; rmp auf LCD ausgeben
  .endif
;
; Vorbelegung von Tasten
; (Beispieleinstellung TV-Fernsteuerung)
;
; Wird nur benoetigt, wenn man
; a) seine Fernbedienung schon genau kennt, und
; b) die Einstellungen nicht durch Training son-
;    dern durch die Voreinstellung einstellen
;    moechte.
;
.ESEG
.ORG 0x00
EeAnf:
 ; rot, Taste 1 ein, Taste 4 aus
  .DW 0x0835,0xC8F5
 ; gelb, Taste 2 ein, Taste 5 aus
  .DW 0x88B5,0x2815
 ; gruen, Taste 3 ein, Taste 6 aus
  .DW 0x4875,0xA895
  ; Null/Eins-Schwelle
  .DB cEins
EeEnde:
;
; Ende Quellcode
;
```

Der Code belegt beim ATtiny13 57% des Flash-Memories und 45% des SRAMs.

```
ATtiny13 memory use summary [bytes]:
Segment   Begin    End      Code   Data   Used    Size   Use%
----------------------------------------------------------------
[.cseg] 0x000000 0x000248    584      0    584    1024  57.0%
[.dseg] 0x000060 0x00007d      0     29     29      64  45.3%
[.eseg] 0x000000 0x00000d      0     13     13      64  20.3%
```

Es ist also noch jede Menge Platz frei für Weiteres.

Lektion 13: Frequenzzähler und Induktivitätsmessgerät

Und noch mal was richtig Praktisches: ein Frequenzzähler, mit Umwandlung von 24-Bit- und 40-Bit-Binärzahlen in ihre Dezimalentsprechung. Und obendrein ein praktisches Induktivitätsmessgerät, mit Quadrieren und Dividieren großer Zahlen in Assembler.

13.0 Übersicht

- 13.1 Einführung in die Frequenzmessung
- 13.2 Einführung in die Dezimalumwandlung (24- und 32-Bit)
- 13.3 Digitalsignale messen mit PCINT
- 13.4 Analogsignalmessung mit Analogvergleicher
- 13.5 Induktivitätsmessung mit PCINT

13.1 Einführung in die Frequenzmessung

13.1.1 Wahl der Torzeit

Die Messung von Frequenzen ist eigentlich trivial: man zählt einfach die Anzahl Sinuskurven des Eingangssignals über eine Sekunde und zeigt diese dann auf der LCD-Anzeige an. Nun ist eine Sekunde etwas lang. Wie wäre es mit 0,5 Sekunden oder gar 0,25 Sekunden?

Das Erkennen von Rechtecksignalen kann man mit dem Eingang INT0 oder, mit dem PCINT0 oder PCINT1, an jedem anderen Eingang vornehmen. Verwendet man INT0, kann man die zu erkennende Flanke (aufwärts, abwärts, beide) vorwählen. Beim PCINT tritt der Interrupt bei beiden Flanken ein, also pro Schwingung zweimal.

Die großartige Aufgabe in Assembler ist dann die Multiplikation der gezählten Impulse mit zwei oder vier. Der C-Programmierer wirft jetzt seine großartige Multiplikationsbibliothek an (und steigt erst mal auf einen größeren AVR-Typ um, weil ihm das Flash dadurch schon arg knapp wird), während der Assemblerkundige mit zweimaligem Links-Shiften und viermaligem Links-Rotieren in ganzen sechs Takten schon fertig ist, also z. B. so:

```
; Zaehlergebnis ueber 0,25 Sekunden in R3:R2:R1
        lsl R1 ; mal zwei
        rol R2
        rol R3
        lsl R1 ; mal vier
        rol R2
        rol R3
```

Wieso eigentlich drei Register? Mit denen lässt sich bis 256*256*256-1 = 16.777.215 zählen, also in einer Viertelsekunde bis 66,8 MHz. Also weit oberhalb dessen, was normale Elektronikbastler so an Schwingungen machen und auch weit oberhalb dessen, was so ein AVR normal zählen kann. Der ist mit 1 MHz Takt schon bei unter 100 kHz voll mit Zählen ausgelastet und geht in die Sättigung. Mit der folgenden Zähl-Interrupt-Serviceroutine kriegen wir nämlich folgende Ausführungszeiten:

```
; Interrupt starten: 4 Takte; Vektorsprung: 2 Takte
CntIsr:
        in R15,SREG ; SREG sichern, +1 = 7
        inc R1 ; Einer aufwaerts zaehlen, +1 = 8
        brne CntIsrRet ; +1/2 = 9/10
        inc R2 ; 256-er aufwaerts
        brne CntIsrRet
        inc R3 ; 65536-er aufwaerts
CntIsrRet: ; Minimum 10 Takte
        out SREG,R15 ; SREG wieder herstellen, +1 = 11
```

```
        reti ; +4 = 15
```

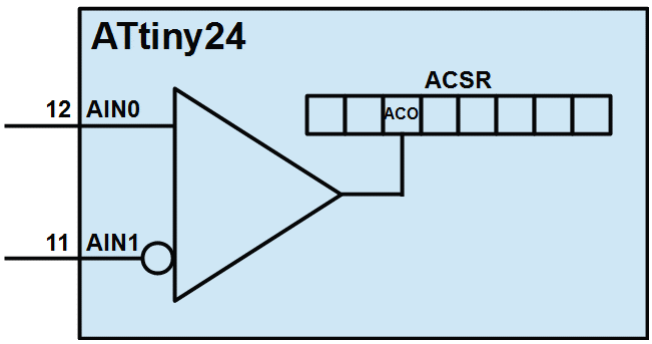
Mit 15 Takten sind bei 1 MHz Systemtakt 15 µs vergangen und bei 1.000.000 / 15 = 66.667 Hz ist es aus mit Zählen. Mit einem Takt von 8 MHz kommen wir immerhin noch bis 533 kHz Zählrate. Verwendet man zum Zählen den eingebauten Timer, indem man statt des Vorteilers den externen Eingang T1 (beim ATtiny24 PA4 am Pin 9) zählen lässt, kommt man etwa auf 250 kHz (1 MHz Takt) oder 2 MHz (8 MHz Takt). Allerdings hängt an diesem Pin in unserer Schaltung schon der Datenport D4 der LCD und wir müssten uns zusätzlich was einfallen lassen, wie wir wechselweise das Messsignal und den LCD-Datenport, lesend und schreibend, an diesen Pin bringen. Mit einem anderen AVR hätten wir dieses Problem möglicherweise nicht.

Wer noch mehr zählen will, taktet den AVR mit 20 MHz (maximal 1,3 MHz Zählrate) oder schaltet einen binären Vorteiler zwischen Messsignal und AVR, allerdings unter Einbußen bei der Auflösung.

Umgekehrt sind bei ganz niedrigen Frequenzen von weniger als 10 Hz (kommt selten vor) entweder längere Torzeiten angebracht. Oder man wechselt die Messmethode: man zählt einfach die Zeit t zwischen zwei Signalen, die Frequenz ist dann 1 / t. Die wären dann z. B. dadurch messbar, indem der Timer bei 1 MHz Takt mit einem Vorteiler von 1 zählt, wie lange es von Schwingung zu Schwingung braucht. Dann kriegt man direkt die Mikrosekunden. Durch Teilen von 1.000.000 durch diese Zahl kriegt man die Frequenz in mHz. Wie man solche großen Zahlen binär teilt (ohne wie der C-Programmierer die ganze riesige Fließkommabibliothek zu bemühen und dann doch auf einen 96-pinnigen ATxmega umzusteigen), kriegen wir etwas später in dieser Lektion.

13.1.2 Analogsignale auswerten

Oft tun uns zu messende Signale nicht den Gefallen, schon als steilflankige Rechteckschwingung mit 5 V Amplitude vorzuliegen. Die Signale von einem Mikrofon, einem Lautsprecher oder aus anderen Quellen sind Sinusse mit irgendeiner Amplitude von 5 mV (dynamisches Mikrofon) bis 2 V (Lautsprecher, 1 Watt an 4 Ohm). Der Elektronikbastler verstärkt jetzt diese Signale so viel, dass da schon ein Rechteck rauskommt und schickt dieses schon steile Signal noch mal durch einen Schmitt-Trigger. Jede Menge externe Elektronik jedenfalls.



Nicht so der AVR-Kundige. Der weiß, dass jeder uralte popelige AVR diese ganze Mimik schon an Bord hat und nur darauf harrt, eingeschaltet zu werden. Die Mimik nennt sich Analogvergleicher (Analog Comparator). Sie besteht aus einem Operationsverstärker, dessen positiver Eingang am Pin AIN0 (PA1) und dessen negativer Eingang am Pin AIN1 (PA2) angeschlossen ist. Das Vergleichsergebnis ist im Bit ACO im Analogstatus- und Kontrollregister ACSR ablesbar. Ist das Bit ACIE (Bit 3) in diesem Register gesetzt, dann wird bei jedem Wechsel des Bits der Analogvergleicher-Interrupt ausgelöst und die Ausführung verzweigt an dessen Vektor-Adresse (beim ATtiny24 0x000C). Das kann man sich auch zum Frequenzmessen zunutze machen.

Damit ist fast alles schon vorhanden, um Analogsignale niedrigster Amplitude auf Polaritätswechsel hin zu untersuchen: die ausgelösten Interrupts müssen einfach nur gezählt werden. Es sind doppelt so viele wie die Frequenz, weil jeder Sinusdurchgang zweimal die Polarität wechselt. Wir können uns bei einer Messzeit von 250 ms also einmal Linksschieben und zweimal Linksrotieren des Zählergebnisses sparen.

13.1.3 Induktivitäten messen

Induktivitäten sind Spulen. Die Größe der Induktivität wird in Henry (H) angegeben. Ihre Größe lässt sich anhand des (Schein-)Widerstands messen, den sie einer Wechselspannung mit

der Frequenz F entgegensetzen. Dieser Scheinwiderstand Z ergibt sich aus der Formel

$$Z_L \text{ (Ohm)} = 2 * \Pi * F \text{ (Hz)} * L \text{ (H)}$$

Darin wird der Teil "2 * Π * F" auch als Kreisfrequenz bezeichnet und mit ω abgekürzt.

Nun lässt sich der Widerstand nur etwas aufwändig messen, besonders, da es sich um Wechselspannung handeln muss. Eleganter, und viel einfacher, ist es, die Spule mit einem Kondensator zu einem Schwingkreis zu verschalten. Schwingkreise lassen sich bekanntlich zum Schwingen anregen. Sie schwingen dann auf der Frequenz, bei der der Scheinwiderstand der Spule gleich groß ist wie der des Kondensators. Dessen Scheinwiderstand ist umgekehrt zur Frequenz (je größer die Frequenz, desto kleiner der Widerstand), also

$$Z_C = 1 / (\omega * C \text{ (Farad)})$$

Die Gleichheit der beiden Scheinwiderstände $Z_L = Z_C$ zeitigt die Schwingungsgleichung:

$$Z_L = Z_C \text{ oder } \omega * L = 1 / (\omega * C) \text{ oder } \omega^2 = 1 / (L * C) \text{ oder}$$
$$\omega = \sqrt{(1/(L * C))} \text{ oder } F = 1 / (2 * \Pi) * \sqrt{(1/(L * C))}$$

Regt man den Schwingkreis mit der Spule und einem bekannten Kondensator zum Schwingen an, kann man aus der gemessenen Frequenz mit der Formel

$$L(H) = 1 / (4 * \Pi^2 * C(F) * F(Hz)^2)$$

direkt die Induktivität ermitteln.

Da der Assemblerprogrammierer solche komplizierten Formeln etwas scheut (der C-Programmierer nicht, der nimmt einfach seine Riesen-Fließkommabibliothek und steigt auf Mega um), müsste sich das mit etwas Intelligenz umformulieren lassen. Da sowohl 4 als auch Π^2 als auch C sich nicht ändern, kann man 1 / (4 * Π^2 * C) einmal ausrechnen und dann nur durch F^2 teilen. Nehmen wir das Ganze noch mit 1.000.000 mal, kriegen wir die Induktivität in µH. Für einen Kondensator von 50 nF resultiert daraus die Zahl 506.605.918.079, oder hexadezimal 75.F4.10.D7.7F, eine 40-bittige Zahl.

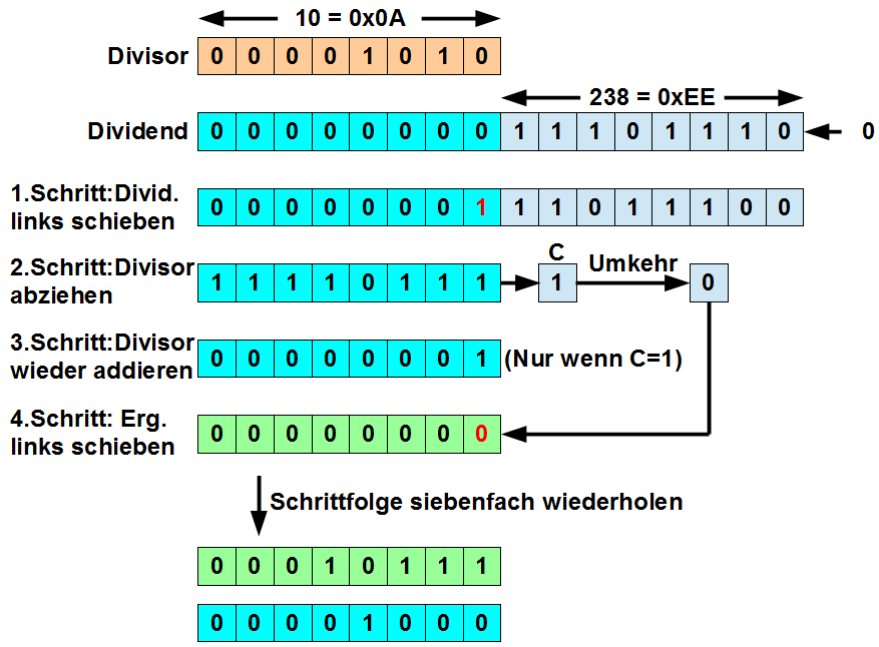
Diese ist durch F^2 zu teilen. Statt nun wieder die großartige Fließkomma-Bibliothek zu bemühen nimmt der Assemblerprogrammierer die gemessene Frequenz, einfach mit sich selbst mal. Für die Frequenzmessung kommt eine 24-bittige Zahl zum Einsatz (siehe oben), eigentlich liefert die beim Malnehmen ein 48-bittiges Ergebnis. Auf die oberen 8 Bit können wir verzichten.

Max Frequenz:	Hex	Dezimal	Dimension
F²=	7FFFFFFFFF	549.755.813.887	Hz²
F=	0B504F	741.455	Hz
Min Frequenz:	**Hex**	**Dezimal**	**Dimension**
L=	003B9AC9FF	999.999.999	µH
F²=	00000001FB	507	Hz²
F=	000017	23	Hz

Begrenzen wir F^2 auf 40 Bits, darf F maximal 741,455 kHz werden, was bei 1 MHz Taktfrequenz ohnehin nicht mehr messbar ist.

Die Untergrenze ergibt sich daraus, dass die Ausgabe von Induktivitäten von über 999 H unnötig ist. Es kommen daher nur Frequenzen von 23 und mehr Hz für die Ausgabe in Frage.

13.1.3.1 Division 8-Bit durch 8-Bit



Die einfachste Division geht mit 8 Bits folgendermaßen. Zuerst wird das höchste Bit des Dividenten in den noch leeren Divisionsbereich hineingeschoben. Von diesen 8 Bits wird der Divisor abgezogen. Tritt dabei ein Überlauf auf, wird die Subtraktion wieder zurück genommen und eine Null in das Ergebnisregister geschoben. Trat kein Überlauf auf, wird eine Eins in das Ergebnis geschoben.

Diese Abfolge wird weitere sieben Male wiederholt und die Division ist fertig.

In Assembler sieht das so aus.

```
;
; Division 8 Bit durch 8 Bit
;
        ldi R16,0xEE ; Dividend
        ldi R17,10 ; Divisor
        ldi R18,8 ; Anzahl Bits
        clr R19 ; Dividend laufend
        clr R20 ; Ergebnis
Shift:
        lsl R16 ; Oberstes Bit Dividend
        rol R20 ; in laufenden Dividend
        sub R20,R17 ; Divisor abziehen
        brcc Eins ; Carry clear = Eins
        add R20,R17 ; Divisor wieder addieren
        clc ; Ergebnisbit = 0
        rjmp Resultat ; in Ergebnis
Eins:
        sec ; loesche Ergebnisbit
Resultat:
        rol R19 ; in Ergebnis rollen
        dec R18 ; abwaerts zaehlen
        brne Shift ; noch weiter
        nop ; fertig
```

Das Studio sagt, dass die Division 92 µs benötigt.

Eigentlich ist binäre Division sogar einfacher als dezimale, da es ja immer nur entweder Abziehen (Eins) oder nicht Abziehen (Null) gibt.

13.1.3.2 Division 16-Bit durch 8-Bit

Bei 16 Bits kommen jeweils zwei Register zum Einsatz, bei denen Überträge mit addiert und subtrahiert werden.

```
;
; Division 16 Bit durch 8 Bit
;
        ldi R31,High(50000) ; Dividend
        ldi R30,Low(50000)
        ldi R16,75 ; Divisor, LSB
        clr R17 ; MSB
        clr R27 ; Ergebnis, MSB
        clr R26 ; LSB
        clr R18 ; Dividend, aktuell, LSB
```

```
        clr R19 ; MSB
        ldi R20,16 ; 16 Bits, Zaehler
Shift:
        lsl R30 ; Dividend links schieben
        rol R31 ; MSB
        rol R18 ; in aktuellen rollen, LSB
        rol R19 ; MSB
        sub R18,R16 ; subtrahiere Divisor, LSB
        sbc R19,R17 ; MSB
        brcc Eins ; kein Carry, schiebe 1
        add R18,R16 ; Carry, subtrahieren rueckgaengig
        adc R19,R17 ; MSB
        clc ; schiebe 0
        rjmp Ergebnis ; in das Ergebnis
Eins:
        sec ; schiebe 1
Ergebnis:
        rol R26
        rol R27
        dec R20
        brne Shift
        nop
```

Auch das ist nicht gerade riesiger Aufwand. Das Ganze braucht nun 265 µs.

13.1.3.3 Division 40-Bit durch 40-Bit

Nachdem das Prinzip nunmehr verstanden ist, ist die 40-Bit-Division eigentlich kein Problem mehr. Das Dividieren der 40-bittigen Zahl durch eine maximal 40-bittige Zahl (F^2) mit einem potenziell 40-bittigen Resultat erfordert aber insgesamt 160 Bits oder 20 Register, weil für den Dividenten 80 Bits gebraucht werden. Bei 20 Registern zum Dividieren bleibt nicht viel für Anderes übrig. Die Lösung für diesen Engpass ist die Verlegung des Dividenten in das SRAM. Mit fünfmaligem

```
        ld R16,Z ; Z zeigt auf Tabelle im SRAM
        rol R16
        st Z+,R16
```

wird dann das jeweils höchstwertige Bit des verbleibenden Dividenten in das Carry geschoben und von da aus in den aktuellen Dividenden. Das braucht etwas länger, aber auch das ist kein Riesendrum.

Die folgende Tabelle vollzieht den Ablauf im Programm für eine gemessene Frequenz von 1.000 Hz (=0x0003E8, F^2 = 1.000.000 = 0x0F4240).

Dez.	Hex	Ergebnis hex	Nach Subtr.	Subtr	Nach Rollen	Rollen	SRAM, Hex	SRAM, Dez
40	28	0000000000	0000000000	0	0000000000	0	75F410D77F	506.605.918.079
39	27	0000000000	0000000001	0	0000000001	1	EBE821AEFE	1.013.211.836.158
38	26	0000000000	0000000003	0	0000000003	1	D7D0435DFC	926.912.044.540
37	25	0000000000	0000000007	0	0000000007	1	AFA086BBF8	754.312.461.304
36	24	0000000000	000000000E	0	000000000E	0	5F410D77F0	409.113.294.832
35	23	0000000000	000000001D	0	000000001D	1	BE821AEFE0	818.226.589.664
34	22	0000000000	000000003A	0	000000003A	0	7D0435DFC0	536.941.551.552
33	21	0000000000	0000000075	0	0000000075	1	FA086BBF80	1.073.883.103.104
32	20	0000000000	00000000EB	0	00000000EB	1	F410D77F00	1.048.254.578.432
31	1F	0000000000	00000001D7	0	00000001D7	1	E821AEFE00	996.997.529.088
30	1E	0000000000	00000003AF	0	00000003AF	1	D0435DFC00	894.483.430.400
29	1D	0000000000	000000075F	0	000000075F	1	A086BBF800	689.455.233.024
28	1C	0000000000	0000000EBE	0	0000000EBE	0	410D77F000	279.398.838.272
27	1B	0000000000	0000001D7D	0	0000001D7D	1	821AEFE000	558.797.676.544
26	1A	0000000000	0000003AFA	0	0000003AFA	0	0435DFC000	18.083.725.312
25	19	0000000000	00000075F4	0	00000075F4	0	086BBF8000	36.167.450.624
24	18	0000000000	000000EBE8	0	000000EBE8	0	10D77F0000	72.334.901.248

Dez.	Hex	Ergebnis hex	Nach Subtr.	Subtr	Nach Rollen	Rollen	SRAM, Hex	SRAM, Dez
23	17	0000000000	000001D7D0	0	000001D7D0	0	21AEFE0000	144.669.802.496
22	16	0000000000	000003AFA0	0	000003AFA0	0	435DFC0000	289.339.604.992
21	15	0000000000	0000075F41	0	0000075F41	1	86BBF80000	578.679.209.984
20	14	0000000000	00000EBE82	0	00000EBE82	0	0D77F00000	57.846.792.192
19	13	0000000001	00000E3AC4	1	00001D7D04	0	1AEFE00000	115.693.584.384
18	12	0000000003	00000D3348	1	00001C7588	0	35DFC00000	231.387.168.768
17	11	0000000007	00000B2450	1	00001A6690	0	6BBF800000	462.774.337.536
16	10	000000000F	0000070661	1	00001648A1	1	D77F000000	925.548.675.072
15	0F	000000001E	00000E0CC3	0	00000E0CC3	1	AEFE000000	751.585.722.368
14	0E	000000003D	00000CD746	1	00001C1986	0	5DFC000000	403.659.816.960
13	0D	000000007B	00000A6C4D	1	000019AE8D	1	BBF8000000	807.319.633.920
12	0C	00000000F7	000005965A	1	000014D89A	0	77F0000000	515.127.640.064
11	0B	00000001EE	00000B2CB5	0	00000B2CB5	1	EFE0000000	1.030.255.280.128
10	0A	00000003DD	000007172B	1	000016596B	1	DFC0000000	960.998.932.480
9	09	00000007BA	00000E2E57	0	00000E2E57	1	BF80000000	822.486.237.184
8	08	0000000F75	00000D1A6E	1	00001C5CAE	0	7F00000000	545.460.846.592
7	07	0000001EEB	00000AF29D	1	00001A34DD	1	FE00000000	1.090.921.693.184
6	06	0000003DD7	000006A2FB	1	000015E53B	1	FC00000000	1.082.331.758.592
5	05	0000007BAE	00000D45F7	0	00000D45F7	1	F800000000	1.065.151.889.408
4	04	000000F75D	00000B49AF	1	00001A8BEF	1	F000000000	1.030.792.151.040
3	03	000001EEBB	000007511F	1	000016935F	1	E000000000	962.072.674.304
2	02	000003DD76	00000EA23F	0	00000EA23F	1	C000000000	824.633.720.832
1	01	000007BAED	00000E023F	1	00001D447F	1	8000000000	549.755.813.888
0	Rdg	000007BAEE	00000CC23E	1	00001C047E	0	0000000000	0

Das Ergebnis der Division nach der Rundung, 0x07BAEE oder 506.606 µH, stimmt mit der Berechnung überein. Die Berechnung dauert laut Studio 277 µs für die Multiplikation, 2.919 µs für die Division und 487 µs für die Dezimalumwandlung, insgesamt 3.232 µs für alles. Nicht so arg lang für solche Riesenzahlen. Und schon gar kein Grund auf einen 3 GHz-AMD-Prozessor umzusteigen, weil der binäre Division schon hardwaremäßig integriert hat. So ein popeliger 8-Bit-Tiny kann das auch, wie man sieht.

Damit steht der intelligenten Berechnung der Induktivität nichts mehr im Wege.

Top | Home | Frequenzmessung | Dezimalumwandlung | Digital | Analog | Induktivität

13.2 Einführung in die Dezimalumwandlung (24- und 32-Bit)

Schon in der Lektion 11 mit dem EEPROM-Einschaltzähler hatten wir 8- und 16-Bit-Zahlen von Binär in Dezimalzahlen verwandelt und dabei führende Nullen unterdrückt. Bei den Infrarot-Experimenten haben wir es uns einfach gemacht und die Zahlen aus gutem Grund lieber hexadezimal ausgegeben. Nun haben wir es mit Monsterzahlen von 24 und 32 Bit Länge zu tun. Ich bin mir nicht sicher, was der C-Programmierer jetzt macht, auf jeden Fall ist er auf eine riesige Fließkomma-Bibliothek angewiesen und steigt mindestens auf einen Mega, wenn nicht auf einen Xmega um. Was bloß das Gerücht, Assembler sei schwer zu lernen, aus ansonsten ganz cleveren Menschen machen kann.

Dabei ist die Erweiterung von 16-Bit-Dezimalumwandlung auf 24 oder 32 Bit lange Zahlen eigentlich ganz easy, wenn man das Prinzip mal verstanden hat: wiederholtes Abziehen der Dezimalzahlen, von der größten angefangen immer zehnmal weniger und nacheinander bis herunter zur 10. Entsprechend hatten wir bei 8 Bit mit dezimal 100 angefangen und bei 16 Bit mit

dezimal 10.000. Bei 24 Bit (256 hoch 3 minus 1 = 16.777.215) beginnen wir mit 10 Millionen, bei 32 Bit (256 hoch 4 minus 1 = 4.294.967.295) eben einfach mit einer Milliarde. Spätestens bei 40 Bit stoßen wir allerdings auf eine Grenze, die mit den Eigenheiten von Assemblerprogrammen zu tun hat. Die verwalten nämlich Ganzzahlen manchmal als vorzeichenbehaftete 32-Bit-Zahlen und können daher eigentlich nur 31 Bit Länge und keinesfalls 40 Bit. Größere Zahlen muss man in Assembler auf andere Weise handhaben. In dieser Lektion werden wir es mit einer 40-Bit-Zahl zu tun kriegen.

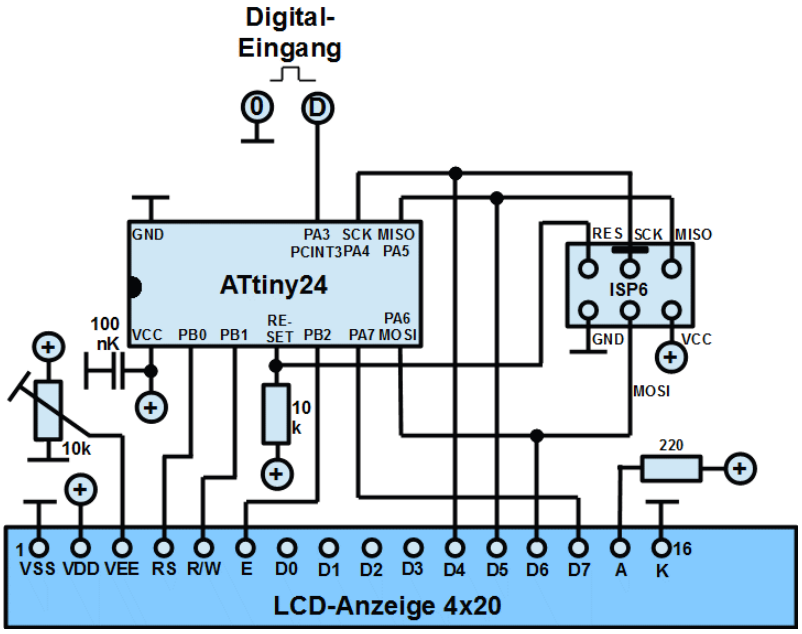
Mehr dazu später, wenn wir es brauchen.

13.3 Digitalsignale messen mit PCINT

13.3.1 Aufgabenstellung

In der ersten Aufgabe sind digitale Rechtecksignale zu messen.

13.3.2 Hardware, Aufbau

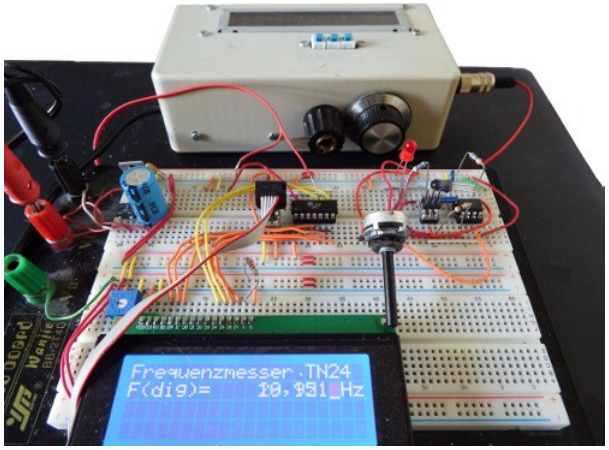


Der Hardwareaufwand bei dieser Aufgabe ist Null. Die Signalquelle wird einfach an den Eingang PA3 angeschlossen. Sie muss dazu aber ein digitales Signal mit Null und 5 V Amplitude liefern.

Alternativer Aufbau

Wer dieses oder die nächsten Experimente lieber auf kompaktere Weise aufbauen möchte und die vielen Zuleitungen zur LCD fest verdrahtet sehen möchte, kann sich das Board hier als gedruckte Platine bauen. Alle Programmbeispiele dieser und der nachfolgenden Lektion laufen darauf ohne Änderungen.

So kann das Ganze aussehen.



13.3.3 Programm

Das Programm ist im Folgenden gelistet (der Quellcode in asm-Format ist hier, zusätzlich wird die LCD-Include-Datei benötigt).

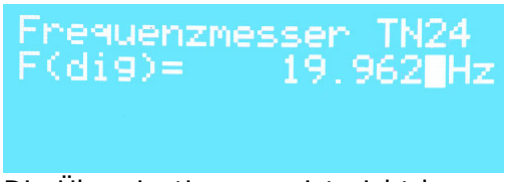
```
;
; ************************************
; * Frequenzmessg Digital ATtiny24/LCD *
; * (C)2016 by www.gsc-elektronic.net  *
; ************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----------- Programmablauf -----------
;
; Misst die Anzahl Pegelwechsel an PA0 mittels
; PCINT ueber eine Viertelsekunde lang, nimmt
; diese mit Zwei mal, wandelt das Ergebnis in
; eine Dezimalzahl um und gibt sie auf der LCD
; aus.
;
; ----------- Hardware -----------------
; Digitaler Frequenzzaehler Eingang an
;   PCINT3/PA3
;
; ----------- Timing -------------------
; Messzeit                 250 ms
; Prozessor-Takt     8.000.000 Hz
; TC1-Prescaler             64
; TC1-Takt             125.000 Hz
; TC1-Takte in 250 ms   31.250
.equ cTc1CmpA = 31249
;
; ----------- Messwertmittelung --------
; Aktueller Messwert        / 2 plus
; Vorheriger Messwert       / 4 plus
; Vorvorheriger Messwert    / 8 plus
; Vorvorvorheriger Messwert / 8 =
; Aktuelle Anzeige          = F
;
; ----------- Ports, Pins --------------
.equ pOut = PORTA ; Ausgabeport
.equ pDir = DDRA ; Richtungsport
.equ bIO = PORTB3 ; Pin-Out Digital
.equ bID = DDA3 ; Pin-Richtung Digital
;
; ----------- Register -----------------
; benutzt: R0, R1 fuer LCD
.def rM0L = R2 ; aktueller Messwert, LSB
.def rM0M = R3 ; dto., MSB
.def rM0H = R4 ; dto., HSB
; frei: R5 .. R14
.def rSreg = R15 ; Statusregister
.def rmp = R16 ; Vielzweckregister
.def rmo   = R17 ; Vielzweckregister
.def rLine = R18 ; LCD-Zeilenzaehler
.def rLese = R19 ; LCD-Register
.def rimp = R20 ; Vielzweck, Interrupts
.def rFlag = R21 ; Flaggen
  .equ bTO = 0 ; Timeout vom Timer
.def rHilf = R22 ; Hilfsregister Dezimal
; frei: R22 .. R25
; benutzt: R27:R26 X ; fuer diverse Zwecke
; frei: R29:R28 Y
; benutzt: R31:R30 Z ; fuer LCD
;
; ----------- SRAM ---------------------
.DSEG
.ORG 0x0060
sM: ; Messwertspeicher, vier Werte:
;  aktuell/2, letzter/4, vorletzter/8,
;    vorvorletzter/8
;  jeweils: L(+0), M(+1), H(+2)
.Byte 12
sMEnd:
;
; ---- Reset- und Int-Vektoren ---------
.CSEG
.ORG 0x0000
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0 External Int 0
  rjmp Pci0Isr ; PCI Request 0
  reti ; PCINT1 PCI Request 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT TC1 Capture Event
  rjmp Tc1Isr ; TIM1_COMPA TC1 Cmp Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF TC1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; TIM0_OVF TC0 Overflow
  reti ; ANA_COMP Analog Comparator
  reti ; ADC ADC Conversion Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routinen -----
;
; PCINT Interrupt
;   wird von jedem Pegelwechsel am PCINT3-
;   Eingang ausgeloest
;
; Erhoeht den 24-Bit-Zaehler rM0H:rM0M:rM0L
;
Pci0Isr: ; Impulse am Digitaleingang zaehlen
  in rSreg,SREG ; Status retten
  inc rM0L ; zaehlen
  brne Pci0IsrRet
  inc rM0M ; MSB erhoehen
  brne Pci0IsrRet
  inc rM0H
Pci0IsrRet:
  out SREG,rSreg ; Status herstellen
  reti
;
; TC1 Compare Match A Interrupt
;   wird nach Ablauf jedes CTC-Zyklusses
;   ausgeloest
;
; Ausloesung nach 64 * (31.249 + 1) / 8 =
;   250 ms.
;
; Stoppt den PCINT-Interrupt und setzt
; die bTO-Flagge die die Ergebnisaus-
; gabe ausloest
;
Tc1Isr: ; Timeout Zaehler
  in rSreg,SREG ; Status retten
  ldi rimp,0 ; Zaehlen anhalten
  out GIMSK,rimp ; Int-Disable
  sbr rFlag,1<<bTO ; Timeout-Flagge
  out SREG,rSreg ; Status herstellen
  reti
;
; ----------- Hauptprogramm-Init ------
Start:
  ; Stapel einrichten
  ldi rmp,LOW(RAMEND) ; RAM-Ende
  out SPL,rmp ; in Stapelzeiger
  ; Auf 8 MHz Takt umstellen
```

```
  ldi rmp,1<<CLKPCE ; Change Enable
  out CLKPR,rmp ; in Clock Prescaler
  ldi rmp,0 ; Precaler / 1
  out CLKPR,rmp
  ; Port initialisieren
  sbi pOut,bIO ; Eingabepin Pullup
  cbi pDir,bID ; Eingabepin Input
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport, Schreiben
  out pLcdDR,rmp ; auf Richtung Datenport
  ; LCD Init
  rcall LcdInit ; starten LCD
  ldi ZH,High(2*LcdTextOut) ; Z auf Text
  ldi ZL,Low(2*LcdTextOut)
  rcall LcdText ; Gib Text aus
  ; Timer Init
  ldi rmp,High(cTc1CmpA) ; Compare auf
  out OCR1AH,rmp ; Messzyklusdauer
  ldi rmp,Low(cTc1CmpA)
  out OCR1AL,rmp
  clr rmp ; TC1 Normal operation
  out TCCR1A,rmp
  ldi rmp,(1<<CS11)|(1<<CS10) ; Presc 64
  out TCCR1B,rmp
  ldi rmp,1<<OCIE1A ; Compare Match Int
  out TIMSK1,rmp
  ; PCINT3 aktivieren
  ldi rmp,1<<PCINT3 ; Pin change PA3
  out PCMSK0,rmp ; maskieren
  ldi rmp,1<<PCIE0 ; PCINT0 Interrupt
  out GIMSK,rmp ; in Int-Maske
  ; Sleep Mode
  ldi rmp,1<<SE ; Sleep enable
  out MCUCR,rmp ; in Kontrollregister
  ; Interrupts enablen
  sei ; Ints zulassen
Schleife:
  sleep ; schlafen legen
  nop ; nach Aufwachen
  sbrc rFlag,bTO ; Timeout-Flagge?
  rcall Auswerten ; gesetzt, Auswerten
  rjmp Schleife
;
; Routine Auswerten der Zaehlergebnisse
;   wird von der bTO-Flagge ausgeloest
;
; Stoppt den Compare-Match-Int von TC1,
; nimmt den Messwert mit zwei Mal, mit-
; telt die aktuelle und die letzten drei
; Messungen (Durchschnitt ueber eine Se-
; kunde), gibt den Durchschnitt dezimal
; auf der LCD aus und startet die naech-
; ste Messung.
;
Auswerten:
  cbr rFlag,1<<bTO ; Flagge ruecksetzen
  clr rmp ; Compare Match Int aus
  out TIMSK1,rmp
  ; Messwerte verschieben+dividieren
  ldi ZH,High(sMEnd) ; Ziel
  ldi ZL,Low(sMEnd)
  ldi XH,High(sMEnd-3) ; Quelle
  ldi XL,Low(sMEnd-3)
  ld rmp,-X ; vorletzter nach vorvorletzter
  st -Z,rmp ; kopieren
  ld rmp,-X
  st -Z,rmp
  ld rmp,-X
  st -Z,rmp
  ld rmp,-X ; letzter nach vorl.+Division
  lsr rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X ; neuester nach letzt.+Division
  lsr rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  st -Z,rM0H ; neuester ablegen
  st -Z,rM0M
  st -Z,rM0L
  adiw ZL,3 ; Zeiger auf letzten
  ldi rmp,4
  mov R0,rmp ; R0 ist Zaehler
Auswerten1:
  ld rmp,Z+ ; lese LSB
  add rM0L,rmp ; Zu Ergebnis addieren
  ld rmp,Z+ ; lese MSB
  adc rM0M,rmp ; addieren mit Uebertrag
  ld rmp,Z+ ; lese HSB
  adc rM0H,rmp ; addieren mit Uebertrag
  dec R0
  brne Auswerten1 ; Weitere Werte addieren
  ldi ZH,1 ; LCD auf Ergebnisposition
  ldi ZL,8
  rcall LcdPos
  rcall DezimalAus ; gib dezimal aus
;
; Routine Neustart
;   wird beim Init und nach Ausgabe des aktuellen
;   Messwertes aufgerufen
;
; Loescht die Zaehlergebnisse und den TC1-Zaehler
; und ermoeglicht PCINT- und TC1-Compare-Match-A-
; Interrupts
;
Neustart:
  clr rM0L ; Zaehler loeschen
  clr rM0M
  clr rM0H
  ldi rmp,1<<PCIE0 ; PCINT0 Interrupt
  out GIMSK,rmp ; in Int-Maske
  clr rmp
  out TCNT1H,rmp ; Zaehler ruecksetzen
  out TCNT1L,rmp
  ldi rmp,1<<OCIE1A ; Compare Match Int
  out TIMSK1,rmp
  ret
;
; 3 Byte-Zahl in rM0H:rM0M:rM0L in dezimal
;   auf der LCD ausgeben
;
; Wird von der Auswerte-Routine aufgerufen
; Verwendet die 24-Bit-Dezimalwert-Tabelle
; bis 9,99 MHz (maximal bis 0x98.96.7F)
; Unterdrueckt fuehrende Nullen und Dezimal-
; trennzeichen
;
DezimalAus:
  ldi ZH,High(2*DezimalTab)
  ldi ZL,Low(2*DezimalTab)
  clt ; Fuehrende Nullen
DezimalAus1:
  lpm XL,Z+ ; lese Dezimalzahl
  lpm XH,Z+
  lpm rHilf,Z+
  clr rmp ; Teilerzaehler
  cp XL,rmp
  brne DezimalAus2
  cp XH,rmp
  brne DezimalAus2
```

```
  cp rHilf,rmp
  breq DezimalAusEnd
DezimalAus2:
  sub rM0L,XL ; abziehen
  sbc rM0M,XH
  sbc rM0H,rHilf
  brcs DezimalAus3 ; Ueberlauf
  inc rmp
  rjmp DezimalAus2 ; weiter subtrahieren
DezimalAus3:
  add rM0L,XL ; Ruecknahme Subtraktion
  adc rM0M,XH
  adc rM0H,rHilf
  tst rmp ; Null?
  brne DezimalAus4 ; Nicht Null
  brts DezimalAus5 ; keine Nullen unterdr.
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,' '
  rjmp DezimalAusKomma
DezimalAus4:
  set ; keine fuehrenden Nullen unterdr.
DezimalAus5:
  subi rmp,-'0'
  rcall LcdD4Byte
  ldi rmp,'.'
DezimalAusKomma:
  cpi XL,Byte1(1000000)
  breq DezimalAusKomma1
  cpi XL,Byte1(1000)
  breq DezimalAusKomma1
  rjmp DezimalAus1
DezimalAusKomma1:
  rcall LcdD4Byte
  rjmp DezimalAus1
DezimalAusEnd:
  ldi rmp,'0' ; letzte Ziffer
  add rmp,rM0L ; addieren
  rjmp LcdD4Byte
;
DezimalTab:
.db Byte1(1000000),Byte2(1000000)
.db Byte3(1000000),Byte1(100000)
.db Byte2(100000),Byte3(100000)
.db Byte1(10000),Byte2(10000)
.db Byte3(10000),Byte1(1000)
.db Byte2(1000),Byte3(1000)
.db Byte1(100),Byte2(100)
.db Byte3(100),Byte1(10)
.db Byte2(10),Byte3(10)
.db 0,0,0,0
;
; LCD Starttext
LcdTextOut:
.db "Frequenzmesser tn24 ",0x0D,0xFF
.db "F(dig)= x,xxx,xxx Hz",0x0D,0xFF
;            8
.db "                    ",0x0D,0xFF
.db "                    ",0xFE,0xFE
;
; LCD-Include
.include "Lcd4Busy.inc"
;
; Ende Quellcode
;
```

13.3.4 Messbeispiel

Mit einem quarzgetriebenen Signalgenerator liefert die Messung folgendes Ergebnis:



Die Übereinstimmung ist nicht berauschend, was an dem recht ungenauen RC-Generator im ATtiny24 liegt. Der ist bei 3 V Betriebsspannung kalibriert. Wer es genauer möchte, kann das Oscillator Calibration Byte verändern. Wie das gemacht wird, steht im Handbuch. Oder kann das Messergebnis per Multiplikation etwas genauer gestalten. Die entsprechenden Grundlagen für die Multiplikation sind alle hier beschrieben.

Top	Home	Frequenzmessung	Dezimalumwandlung	Digital	Analog	Induktivität

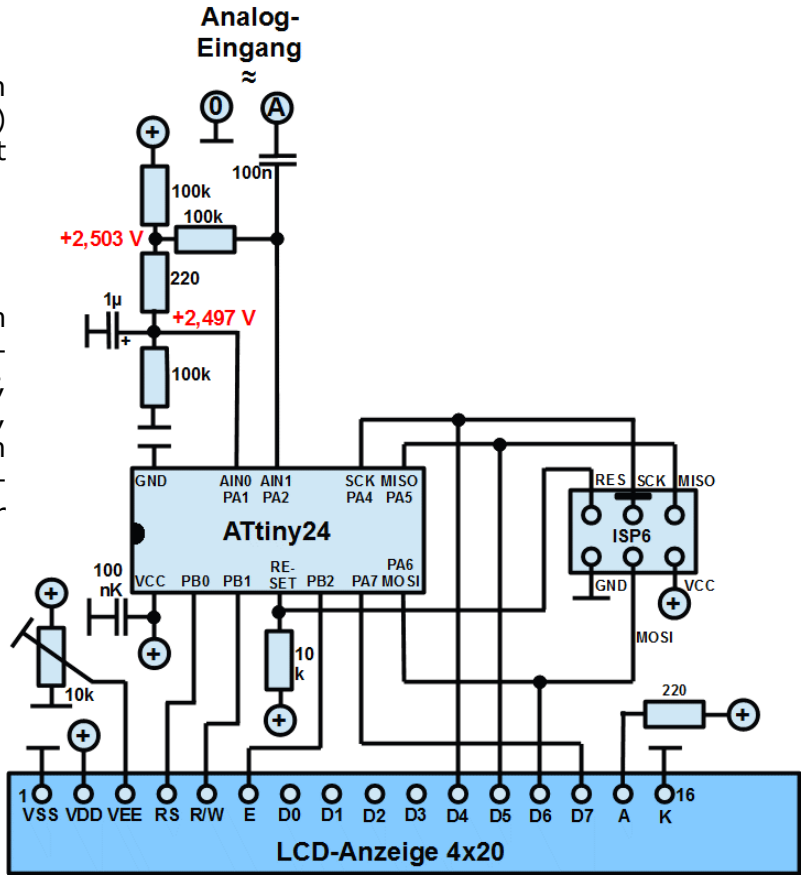
13.4 Analogsignalmessung mit Analogvergleicher

13.4.1 Aufgabe

Die Frequenz von sinusförmigen Wechselspannungen ab 5 mV(eff) sollen gemessen und angezeigt werden.

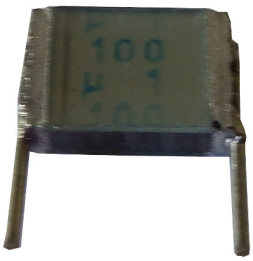
13.4.2 Schaltbild

Das ist die nötige Hardware zum Messen. Sie besteht im Wesentlichen aus einem Spannungsteiler, der für eine gewisse Ruhe sorgt, solange kein Signal angeschlossen ist. Die zu messende Wechselspannung wird über einen Kondensator zugeführt.



13.4.3 Bauteile

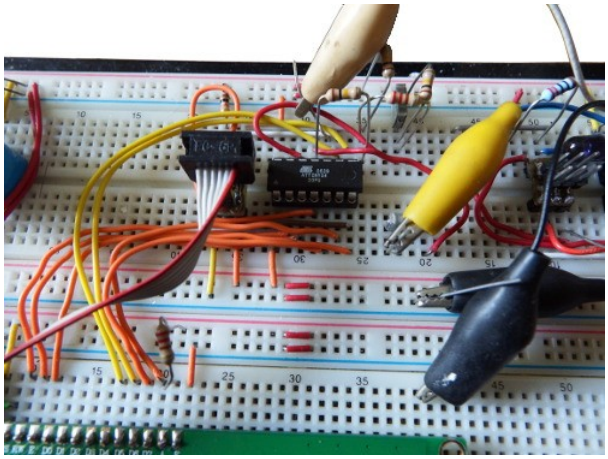
Das ist der 1 µF-Elko, der die Referenzspannung konstant hält. Das längere Bein ist wie immer der Pluspol.



Das ist eine mögliche Bauform des Folienkondensators von 100 nF.

Das sind die beiden Widerstände von 100 k, aus denen der Spannungsteiler aufgebaut ist. Den 220 Ω hatten wir bereits früher.

13.4.4 Aufbau



So oder ähnlich sieht der Aufbau der Schaltung aus. Wer die Anschlussdrähte der Widerstände etwas kürzt und die Schaltung kompakter aufbaut als hier gezeigt, kriegt weniger störende Streusignale in den Eingang.

13.4.5 Programm

Das Programm ist im Folgenden gelistet (hier geht es zum Quellcode im asm-Format, zusätzlich wird die LCD-Include-Datei benötigt). Eine Besonderheit und Unterschied zu allen bisherigen Formulierungen ist noch wichtig: der Betrieb des Analogvergleichers im Interruptmodus ist mit dem Schlafmodus inkompatibel. Manchmal, und unvorhersehbar, setzt im Schlafmodus das Aufwachen nach Interrupts völlig aus, d. h. weder Analogvergleicher- noch Timer-Interrupts wecken die CPU auf. Der Chip versinkt in den Dauerschlaf und die Messungen setzen aus. Nur ein Reset haucht der CPU wieder Leben ein, aber nur für eine gewisse Zeit lang. ATMEL beschreibt diesen Fehler im Handbuch korrekt. Diese Software arbeitet daher nicht im Schlafmodus.

Die Software misst nacheinander sowohl das Analogsignal wie auch das Signal am Digitaleingang. Beide Ereignisse werden von der gleichen Interrupt-Service-Routine gezählt, nur sind nacheinander der PCINT vom Digitaleingang und der Analogvergleicher-Int eingeschaltet. Der Softwareteil zur Mittelung arbeitet daher mit zwei SRAM-Puffern.

In dieser Version wird der prozessorinterne Takt auf 8 MHz heraufgesetzt, nur um zu zeigen, wie das geht.

```
;
; ************************************
; * Frequenzmessg Analog ATtiny24/LCD *
; * (C)2016 by www.gsc-elektronic.net *
; ************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----------- Programmablauf -----------
;
; Nacheinander werden
; a) die Anzahl Pegelwechsel am PCINT3-
;    Eingang,
; b) die Anzahl Pegelwechsel an den Analog-
;    vergleicher-Eingaengen
; ueber 250 ms lang gezaehlt, gemittelt und
; in Hz auf der LCD angezeigt.
;
; ----------- Hardware -----------------
; Analoger Frequenzzaehler am Analog-
;   vergleicher AIN0/PA1 und AIN1/PA2
;
; ----------- Timing -------------------
; Messzeit                250 ms
; Prozessor-Takt     8.000.000 Hz
; TC1-Prescaler             64
; TC1-Takt             125.000 Hz
; TC1-Takte in 250 ms   31.250
.equ cTc1CmpA = 31249
;
; ----------- Messwertmittelung --------
; Aktueller Messwert        / 2 plus
; Vorheriger Messwert       / 4 plus
; Vorvorheriger Messwert    / 8 plus
; Vorvorvorheriger Messwert / 8 =
; Aktuelle Anzeige          = F
;
; ----------- Ports, Pins --------------
.equ pOut = PORTA ; Ausgabeport
.equ pDir = DDRA ; Richtungsport
.equ bIO = PORTB3 ; Pin-Out Digital
.equ bID = DDA3 ; Pin-Richtung Digital
;
; ----------- Register -----------------
; benutzt: R0, R1 fuer LCD
.def rM0L = R2 ; aktueller Messwert, LSB
.def rM0M = R3 ; dto., MSB
.def rM0H = R4 ; dto., HSB
; frei: R5 .. R14
.def rSreg = R15 ; Statusregister
.def rmp = R16 ; Vielzweckregister
.def rmo   = R17 ; Vielzweckregister
.def rLine = R18 ; LCD-Zeilenzaehler
.def rLese = R19 ; LCD-Register
```

```
.def rimp = R20 ; Vielzweck, Interrupts
.def rFlag = R21 ; Flaggen
  .equ bTO = 0 ; Timeout vom Timer
  .equ bAn = 1 ; Analogvergleicher aktiv
.def rHilf = R22 ; Hilfsregister Dezimal
; frei: R22 .. R25
; benutzt: R27:R26 X ; fuer diverse Zwecke
; frei: R29:R28 Y
; benutzt: R31:R30 Z ; fuer LCD
;
; ----------- SRAM ---------------------
.DSEG
.ORG 0x0060
sMD: ; Digitalwerte
.Byte 12
sMDEnd:
sMA: ; Analogwerte
.Byte 12
sMAEnd:
;
; ---- Reset- und Int-Vektoren ---------
.CSEG
.ORG 0x0000
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0 External Int 0
  rjmp CntIsr ; PCI Request 0
  reti ; PCINT1 PCI Request 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT TC1 Capture Event
  rjmp Tc1Isr ; TIM1_COMPA TC1 Cmp Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF TC1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; TIM0_OVF TC0 Overflow
  rjmp CntIsr ; ANA_COMP Analog Comparator
  reti ; ADC ADC Conversion Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routinen -----
;
; CntIsr Interrupt
;   wird wechselweise vom PCINT-Interrupt und
;   vom Analogvergleicher-Interrupt bei jedem
;   Pegelwechsel ausgeloest
;
; Zaehlt die Anzahl Pegelwechsel in
;   rM0H:rM0M:rM0L
;
CntIsr: ; Impulse am Analogvergleicher zaehlen
  in rSreg,SREG ; Status retten
  inc rM0L ; zaehlen
  brne CntIsrRet
  inc rM0M ; MSB erhoehen
  brne CntIsrRet
  inc rM0H
CntIsrRet:
  out SREG,rSreg ; Status herstellen
  reti
;
; TC1 Compare Match A Interrupt
;   wird nach 250 ms vom Compare Match A
;   ausgeloest
;
; Schaltet die Comparator- und PCINT-
;   Interrupts ab und setzt die bTO-Flagge.
;
Tc1Isr: ; Timeout Zaehler
  ldi rimp,0
  out ACSR,rimp ; Disable Int Comparator
  out GIMSK,rimp ; Disable PCInt
  in rSreg,SREG ; Status retten
  sbr rFlag,1<<bTO ; Timeout-Flagge
  out SREG,rSreg ; Status herstellen
  reti
;
; ----------- Hauptprogramm-Init ------
Start:
  ; Stapel einrichten
  ldi rmp,LOW(RAMEND) ; RAM-Ende
  out SPL,rmp ; in Stapelzeiger
  ; Auf 8 MHz Takt umstellen
  ldi rmp,1<<CLKPCE ; Change Enable
  out CLKPR,rmp ; in Clock Prescaler
  ldi rmp,0 ; Precaler / 1
  out CLKPR,rmp
  ; Port initialisieren
  sbi pOut,bIO ; Eingabepin Pullup
  cbi pDir,bID ; Eingabepin Input
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabe, Schreiben
  out pLcdDR,rmp ; auf Richtung Datenausgabeport
  ; LCD Init
  rcall LcdInit ; starten LCD
  ldi ZH,High(2*LcdTextOut) ; Z auf Text
  ldi ZL,Low(2*LcdTextOut)
  rcall LcdText ; Gib Text aus
  ldi rmp,0x0C ; Cursor und Blink aus
  rcall LcdC4Byte
  ; Timer Init
  ldi rmp,High(cTc1CmpA) ; Compare auf
  out OCR1AH,rmp ; Messzyklusdauer
  ldi rmp,Low(cTc1CmpA)
  out OCR1AL,rmp
  clr rmp ; TC1 Normal operation
  out TCCR1A,rmp
  ldi rmp,(1<<CS11)|(1<<CS10) ; Presc 64
  out TCCR1B,rmp
  ldi rmp,1<<OCIE1A ; Compare Match Int
  out TIMSK1,rmp
  ; Analogvergleicher deaktivieren
  ldi rmp,0 ; Disable Int Comparator
  out ACSR,rmp
  ldi rmp,(1<<ADC2D)|(1<<ADC1D) ; Input disable
  out DIDR0,rmp
  ; PCINT3 aktivieren
  ldi rmp,1<<PCINT3 ; Pin change PA3
  out PCMSK0,rmp ; maskieren
  ldi rmp,1<<PCIE0 ; PCINT0 Interrupt
  out GIMSK,rmp ; in Int-Maske
  ; Kein Sleep Mode wegen Analogcomparator!
  ldi rmp,0 ; Sleep enable
  out MCUCR,rmp ; in Kontrollregister
  ; Interrupts enablen
  sei ; Ints zulassen
Schleife:
  sbrc rFlag,bTO ; Timeout-Flagge?
  rcall Auswerten ; gesetzt, Auswerten
  rjmp Schleife
;
; Rountine Auswerten der Zaehlergebnisse
;   wird von der bTO-Flagge ausgeloest
;
; Berechnet abhaengig von der Flagge bAn fuer
;   PCINT- und Analogvergleicherergebnisse
;   den Durchschnitt aus der aktuellen und
;   den drei letzten Messwerten und gibt ihn
;   dezimal auf der LCD aus.
;
Auswerten:
  cbr rFlag,1<<bTO ; Flagge ruecksetzen
  clr rmp ; Compare Match Int aus
  out TIMSK1,rmp
  ; Messwerte im SRAM verschieben/Dividieren
  sbrc rFlag,bAn ; Digitalwerte auswerten?
  rjmp AuswertenAnalog
  ldi ZH,High(sMDEnd) ; Ziel
  ldi ZL,Low(sMDEnd)
```

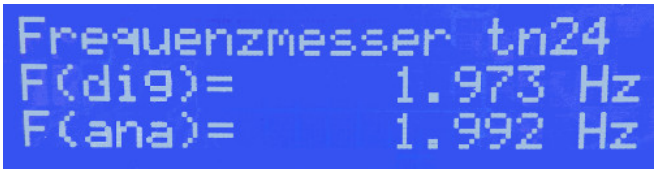
```
  ldi XH,High(sMDEnd-3) ; Quelle
  ldi XL,Low(sMDEnd-3)
  rjmp AuswertenShift
AuswertenAnalog:
  ldi ZH,High(sMAEnd) ; Ziel
  ldi ZL,Low(sMAEnd)
  ldi XH,High(sMAEnd-3) ; Quelle
  ldi XL,Low(sMAEnd-3)
AuswertenShift:
  ; Messwerte im SRAM verschieb.+dividieren
  ld rmp,-X ; vorletzter nach vorvorletzter
  st -Z,rmp ; kopieren
  ld rmp,-X
  st -Z,rmp
  ld rmp,-X
  st -Z,rmp
  ld rmp,-X ; letzter nach vorletzt.+Div
  lsr rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X ; neuester nach letzter+Div
  lsr rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  st -Z,rM0H ; neuester ablegen
  st -Z,rM0M
  st -Z,rM0L
  adiw ZL,3 ; Zeiger auf letzten
  ldi rmp,4
  mov R0,rmp ; R0 ist Zaehler
Auswerten1:
  ld rmp,Z+ ; lese LSB
  add rM0L,rmp ; Zu aktuellem Ergeb. add.
  ld rmp,Z+ ; lese MSB
  adc rM0M,rmp ; addieren mit Uebertrag
  ld rmp,Z+ ; lese HSB
  adc rM0H,rmp ; addieren mit Uebertrag
  dec R0
  brne Auswerten1 ; Weitere Werte addieren
  ldi ZH,1 ; Digitalergebnis ausgeben
  sbrc rFlag,bAn ; Analogflagge gesetzt?
  ldi ZH,2 ; ja, Analogposition
  ldi ZL,8
  rcall LcdPos
  rcall DezimalAus ; gib dezimal aus
;
; Routine Neustart
;   wird vom Init und nach Beenden der Ausgabe
;   aufgerufen
;
; Startet abhaengig von der Flagge bAn einen
; Messzyklus am digitalen oder am analogen
; Eingang.
;
Neustart:
  clr rM0L ; Letzte Dezimale loeschen
  ldi rmp,1<<bAn ; Analog-Flagge umkehren
  eor rFlag,rmp ; Invertiert bAn-Flagge
  sbrc rFlag,bAn ; Analogflagge gesetzt?
  rjmp NeustartAnalog ; ja
  ; Digital messen, PCINT3 aktivieren
  ldi rmp,1<<PCINT3 ; Pin change PA3
  out PCMSK0,rmp ; maskieren
  ldi rmp,1<<PCIE0 ; PCINT0 Interrupt
  out GIMSK,rmp ; in Int-Maske
  rjmp Neustart1
NeustartAnalog:
  ldi rmp,1<<ACIE ; Enable Int Comparator
  out ACSR,rmp
Neustart1:
  clr rmp
  out TCNT1H,rmp ; Zaehler ruecksetzen
  out TCNT1L,rmp
  ldi rmp,1<<OCIE1A ; Compare Match Int
  out TIMSK1,rmp
  ret
;
; 3 Byte-Zahl in rM0H:rM0M:rM0L in dezimal
; auf der LCD ausgeben
;   wird von der Auswerten-Routine aufgerufen
;
; Wandelt rM0H:rM0M:rM0L in Dezimalzahl um
; und gibt sie an der aktuellen LCD-Position
; aus, unterdrueckt fuehrende Nullen und De-
; zimaltrennzeichen
;
DezimalAus:
  ldi ZH,High(2*DezimalTab)
  ldi ZL,Low(2*DezimalTab)
  clt ; Fuehrende Nullen
DezimalAus1:
  lpm XL,Z+ ; lese Dezimalzahl
  lpm XH,Z+
  lpm rHilf,Z+
  clr rmp ; Teilerzaehler
  cp XL,rmp
  brne DezimalAus2
  cp XH,rmp
  brne DezimalAus2
  cp rHilf,rmp
  breq DezimalAusEnd
DezimalAus2:
  sub rM0L,XL ; abziehen
  sbc rM0M,XH
  sbc rM0H,rHilf
  brcs DezimalAus3 ; Ueberlauf
  inc rmp
  rjmp DezimalAus2 ; weiter subtrahieren
DezimalAus3:
  add rM0L,XL ; Ruecknahme Subtraktion
  adc rM0M,XH
  adc rM0H,rHilf
  tst rmp ; Null?
  brne DezimalAus4 ; Nicht Null
  brts DezimalAus5 ; keine Nullen unterdr.
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,' '
  rjmp DezimalAusKomma
DezimalAus4:
  set ; keine fuehrenden Nullen unterdr.
DezimalAus5:
  subi rmp,-'0'
  rcall LcdD4Byte
  ldi rmp,'.'
DezimalAusKomma:
  cpi XL,Byte1(1000000)
  breq DezimalAusKomma1
  cpi XL,Byte1(1000)
  breq DezimalAusKomma1
  rjmp DezimalAus1
DezimalAusKomma1:
  rcall LcdD4Byte
  rjmp DezimalAus1
DezimalAusEnd:
  ldi rmp,'0' ; letzte Ziffer
  add rmp,rM0L ; addieren
  rjmp LcdD4Byte
;
DezimalTab:
.db Byte1(1000000),Byte2(1000000)
.db Byte3(1000000),Byte1(100000)
.db Byte2(100000),Byte3(100000)
.db Byte1(10000),Byte2(10000)
```

```
.db Byte3(10000),Byte1(1000)
.db Byte2(1000),Byte3(1000)
.db Byte1(100),Byte2(100)
.db Byte3(100),Byte1(10)
.db Byte2(10),Byte3(10)
.db 0,0,0,0
;
; LCD Starttext
LcdTextOut:
.db "Frequenzmesser tn24 ",0x0D,0xFF
.db "F(dig)= x.xxx.xxx Hz",0x0D,0xFF
;           8
.db "F(ana)= x.xxx.xxx Hz",0x0D,0xFF
;           8
.db "                    ",0xFE,0xFE
;
; LCD-Include
.include "Lcd4Busy.inc"
;
; Ende Quellcode
;
```

13.4.6 Messbeispiele

Der analoge Messeingang ist sehr empfindlich, bei höheren Frequenzen reichen 2 mV(eff) für eine stabile Messung aus. Bei niedrigen Frequenzen macht sich der hohe kapazitive Scheinwiderstand des 100 nF-Kondensators etwas bemerkbar, so dass höhere Amplituden erforderlich sind. Bei offenem Eingang machen sich Signale am Digitaleingang störend bemerkbar.

Das gleiche Signal, einmal analog (mit kleiner Amplitude) und gleichzeitig digital eingespeist, zeigt nicht immer das gleiche Ergebnis. Ursache dafür dürfte Übersprechen auf den Analogeingang sein.



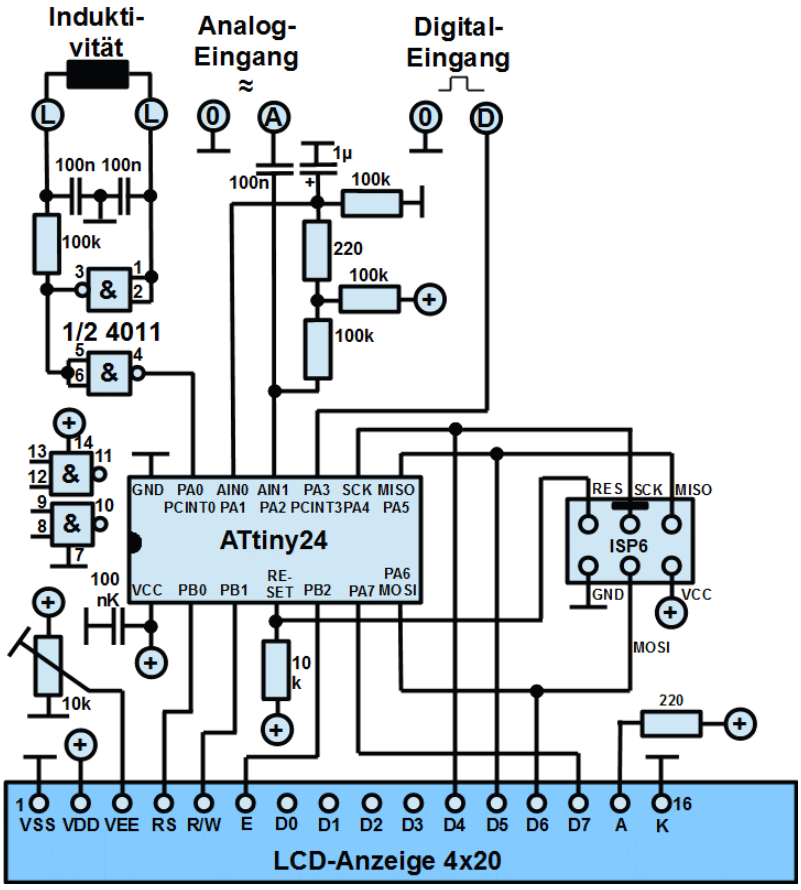
13.5 Induktivitätsmessung mit PCINT

13.5.1 Aufgabe

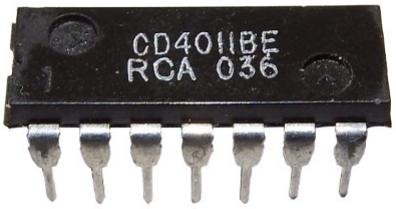
Die Induktivität von Spulen ist zu messen. Sie soll einen weiten Messbereich von 1 mH bis über 10 H umfassen.

13.5.2 Schaltbild

Dies ist das komplette Schaltbild zur gleichzeitigen Messung von Digital- und Analogfrequenzen sowie zur Bestimmung der Induktivität. Der induktive Teil ist mit einem CMOS-NAND-Gatter realisiert, das die besten Schwingeigenschaften über den weiten Messbereich aufweist. Die beiden Kondensatoren von 100 nF bilden mit der Induktivität den Schwingkreis, sie sind in Serie geschaltet (wirksame Kapazität 50 nF). Die Rückkopplung ist über 100 k recht niedrig angekoppelt, reicht aber über den gesamten Messbereich gut aus. Das zweite NAND-Gatter dient der Signalauskopplung, zwei weitere Gatter in der Packung sind unbenutzt.

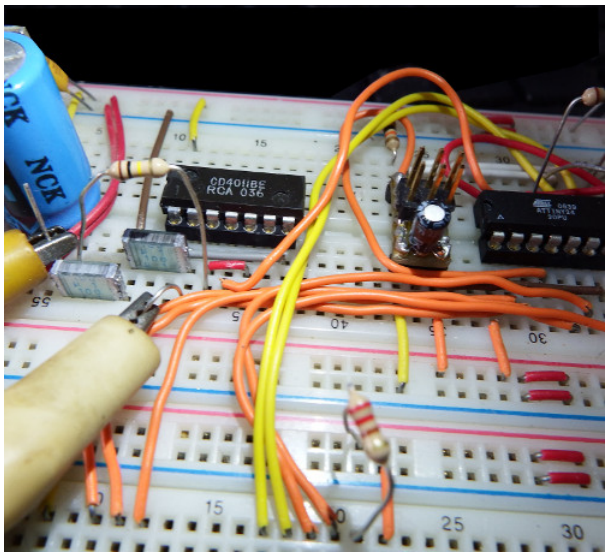


13.5.3 Bauteile



Das ist das Vierfach- NAND. Es eignen sich auch andere invertierende CMOS-Gatter.

13.5.4 Aufbau



Das ist der Aufbau. Die Spule ist mit Krokodilklemmen angekoppelt.

13.5.5 Programm

Das hier ist das Programm (zum Quellcode im asm-Format geht es hier, zusätzlich wird die

LCD-Include-Datei benötigt). Es misst nacheinander an allen drei Eingängen und stellt die Messergebnisse auf drei LCD-Zeilen dar. Auf den Schlafmodus wurde wieder verzichtet, weil der Analogvergleicher das Aufwecken blockiert.

```
;
; ************************************
; * Frequenz- und Induktivitaetsmessg *
; * (C)2016 by www.gsc-elektronic.net *
; ************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; ----------- Programmablauf -----------
;
; Misst nacheinander Pegelwechsel
; a) am PCINT3-Eingang,
; b) an den Analogvergleichereingaengen,
; c) am LC-Oszillator
; ueber 250 ms lang, rechnet diese in
; Frequenzen (a, b) oder Induktivitaet (c)
; um und stellt das Ergebnis dezimal auf
; der LCD dar.
;
; ----------- Hardware -----------------
; Digitaler Frequenzzaehler am Input PA3
; und
; Analoger Frequenzzaehler am Analog-
;   vergleicher AIN0/PA1 und AIN1/PA2
; und
; Induktivitaetsmessung aus Frequenz am
;   Input PA0 (4011-LC-Oszillator)
;
; ----------- Timing -------------------
; Messzeit                 250 ms
; Prozessor-Takt     1.000.000 Hz
; TC1-Prescaler             64
; TC1-Takt              15.625 Hz
; TC1-Takte in 250 ms    3.906
.equ cTc1CmpA = 3905
;
; ----------- Messwertmittelung --------
; Aktueller Messwert        / 2 plus
; Vorheriger Messwert       / 4 plus
; Vorvorheriger Messwert    / 8 plus
; Vorvorvorheriger Messwert / 8 =
; Aktuelle Anzeige          = F
;
; ----------- Ports, Pins --------------
.equ pOut = PORTA ; Ausgabeport
.equ pDir = DDRA ; Richtungsport
.equ bIO = PORTB3 ; Pin-Out Digital
.equ bID = DDA3 ; Pin-Richtung Digital
;
; ----------- Register -----------------
; benutzt: R0, R1 fuer LCD
.def rM0L = R2 ; aktueller Messwert, LSB
.def rM0M = R3 ; dto., MSB
.def rM0H = R4 ; dto., HSB
.def rMH0 = R5 ; 5-Byte Hilfsregister
.def rMH1 = R6 ;   fuer Multiplikation
.def rMH2 = R7 ;   und Division
.def rMH3 = R8
.def rMH4 = R9
.def rME0 = R10 ; 5-Byte Ergebnisregister
.def rME1 = R11 ;   fuer Multiplikation
.def rME2 = R12 ;   und Division
.def rME3 = R13
.def rME4 = R14
; frei: R5 .. R14
.def rSreg = R15 ; Statusregister
.def rmp = R16 ; Vielzweckregister
.def rmo   = R17 ; Vielzweckregister
.def rLine = R18 ; LCD-Zeilenzaehler
.def rLese = R19 ; LCD-Register
.def rimp = R20 ; Vielzweck, Interrupts
.def rFlag = R21 ; Flaggen
  .equ bAn = 0 ; Analogvergleicher aktiv
  .equ bIp = 1 ; Induktivitaetsmessung
  .equ bTO = 2 ; Timeout vom Timer
.def rHilf = R22 ; Hilfsregister Dezimal
; frei: R22 .. R25
; benutzt: R27:R26 X ; fuer diverse Zwecke
; frei: R29:R28 Y
; benutzt: R31:R30 Z ; fuer LCD
;
; ----------- SRAM ---------------------
.DSEG
.ORG 0x0060
sMD: ; Digitalwerte, 4*(HSB:MSB:LSB)
.Byte 12
sMDEnd:
sMA: ; Analogwerte
.Byte 12
sMAEnd:
sMI: ; Induktivitaetswerte
.Byte 12
sMIEnd:
sDividend: ; fuer Division
.Byte 5
sDividendEnde:
;
; ---- Reset- und Int-Vektoren ---------
.CSEG
.ORG 0x0000
  rjmp Start ; Reset-Vektor, Init
  reti ; INT0 External Int 0
  rjmp CntIsr ; PCI Request 0
  reti ; PCINT1 PCI Request 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT TC1 Capture Event
  rjmp Tc1Isr ; TIM1_COMPA TC1 Compare Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF TC1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; TIM0_OVF TC0 Overflow
  rjmp CntIsr ; ANA_COMP Analog Comparator
  reti ; ADC ADC Conversion Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; ----- Interrupt Service Routinen -----
;
; CntIsr wird nacheinander von PCINT- und
;   Analogvergleicher-Interrupts ausgeloest
;
; Zaehlt die Impulse in rM0H:rM0M:rM0L
;
CntIsr: ; Impulse zaehlen
  in rSreg,SREG ; Status retten
  inc rM0L ; zaehlen
  brne CntIsrRet
  inc rM0M ; MSB erhoehen
  brne CntIsrRet
  inc rM0H
CntIsrRet:
  out SREG,rSreg ; Status herstellen
  reti
;
; TC1 Compare Match A Interrupt
;   wird nach 250 ms vom Compare Match aus-
```

```
;   geloest
;
; Schaltet Analogvergleicher-, PCINT- und
; TC1-Interrupts aus und setzt die bTO-Flagge.
;
Tc1Isr: ; Timeout Zaehler
  ldi rimp,0
  out ACSR,rimp ; Disable Int Comparator
  out GIMSK,rimp ; Disable PCInt
  out TIMSK1,rimp ; Diable Timer-Int
  in rSreg,SREG ; Status retten
  sbr rFlag,1<<bTO ; Timeout-Flagge
  out SREG,rSreg ; Status herstellen
  reti
;
; ----------- Hauptprogramm-Init ------
Start:
  ; Stapel einrichten
  ldi rmp,LOW(RAMEND) ; RAM-Ende
  out SPL,rmp ; in Stapelzeiger
  ; Port initialisieren
  sbi pOut,bIO ; Eingabepin Pullup
  cbi pDir,bID ; Eingabepin Input
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
Schreiben
  out pLcdDR,rmp ; auf Richtungsregister
Datenausgabeport
  ; LCD Init
  rcall LcdInit ; starten LCD
  ldi ZH,High(2*LcdTextOut) ; Z auf Text
  ldi ZL,Low(2*LcdTextOut)
  rcall LcdText ; Gib Text aus
  ldi rmp,0x0C ; Cursor und Blink aus
  rcall LcdC4Byte
  ; Timer Init
  ldi rmp,High(cTc1CmpA) ; Compare auf
  out OCR1AH,rmp ; Messzyklusdauer
  ldi rmp,Low(cTc1CmpA)
  out OCR1AL,rmp
  clr rmp ; TC1 Normal operation
  out TCCR1A,rmp
  ldi rmp,(1<<CS11)|(1<<CS10) ; Presc 64
  out TCCR1B,rmp
  ldi rmp,1<<OCIE1A ; Compare Match Int
  out TIMSK1,rmp
  ; Analogvergleicher deaktivieren
  ldi rmp,0 ; Disable Int Comparator
  out ACSR,rmp
  ldi rmp,(1<<ADC2D)|(1<<ADC1D) ; Input disable
  out DIDR0,rmp
  ; PCINT3 aktivieren
  ldi rmp,1<<PCINT3 ; Pin change PA3
  out PCMSK0,rmp ; maskieren
  ldi rmp,1<<PCIE0 ; PCINT0 Interrupt
  out GIMSK,rmp ; in Int-Maske
  ; Sleep Mode, kein Sleep wegen Analogcomparator
  clr rmp ; Sleep mode disable
  out MCUCR,rmp ; in Kontrollregister
  ; Interrupts enablen
  sei ; Ints zulassen
Schleife:
  ; kein Schlafen wegen Analog-Comparator
  sbrc rFlag,bTO ; Timeout-Flagge?
  rcall Auswerten ; gesetzt, Auswerten
  rjmp Schleife
;
; Routine Auswerten der Zaehlergebnisse
;   wird von gesetzter bTO-Flagge ausge-
;   loest
;
; Wertet Digital-, Analog- und Induktions-
; messwerte aus (Flaggenbits 0 und 1:
; 0 = Digital, 1 = Analog, 2 = Indukt.)
; Mittelt den aktuellen und die drei
; vorherigen Messwert und gibt sie als
; Frequenz in Hz (0 und 1) oder als
; Induktivitaet in uH (2) aus.
;
Auswerten:
  cbr rFlag,1<<bTO ; Flagge ruecksetzen
  clr rmp ; Compare Match Int aus
  out TIMSK1,rmp
  ; Messwerte im SRAM verschieben/Dividieren
  cpi rFlag,0x01 ; Induktivitaetswerte?
  breq AuswertenAnalog ; Analog
  brcs AuswertenDigital ; Digital
  ; Induktivitaet auswerten
  ldi ZH,High(sMIEnd) ; Ziel
  ldi ZL,Low(sMIEnd)
  ldi XH,High(sMIEnd-3) ; Quelle
  ldi XL,Low(sMIEnd-3)
  rjmp AuswertenShift
AuswertenDigital:
  ldi ZH,High(sMDEnd) ; Ziel
  ldi ZL,Low(sMDEnd)
  ldi XH,High(sMDEnd-3) ; Quelle
  ldi XL,Low(sMDEnd-3)
  rjmp AuswertenShift
AuswertenAnalog:
  ldi ZH,High(sMAEnd) ; Ziel
  ldi ZL,Low(sMAEnd)
  ldi XH,High(sMAEnd-3) ; Quelle
  ldi XL,Low(sMAEnd-3)
AuswertenShift:
  ; Messwerte im SRAM verschieben+dividieren
  ld rmp,-X ; vorletzter nach vorvorletzter
  st -Z,rmp ; kopieren
  ld rmp,-X
  st -Z,rmp
  ld rmp,-X
  st -Z,rmp
  ld rmp,-X ; letzter nach vorletzter+Division
  lsr rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X ; neuester nach letzter+Division
  lsr rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  ld rmp,-X
  ror rmp
  st -Z,rmp
  st -Z,rM0H ; neuester ablegen
  st -Z,rM0M
  st -Z,rM0L
  adiw ZL,3 ; Zeiger auf letzten
  ldi rmp,4
  mov R0,rmp ; R0 ist Zaehler
Auswerten1:
  ld rmp,Z+ ; lese LSB
  add rM0L,rmp ; Zu aktuellem Ergebnis addieren
  ld rmp,Z+ ; lese MSB
  adc rM0M,rmp ; addieren mit Uebertrag
  ld rmp,Z+ ; lese HSB
  adc rM0H,rmp ; addieren mit Uebertrag
  dec R0
  brne Auswerten1 ; Weitere Werte addieren
  cpi rFlag,0x02 ; Induktivitaet?
  brcs Auswerten2
  rcall Indukt ; Induktivitaet ausgeben
  rjmp Neustart
Auswerten2:
```

```
  mov ZH,rFlag ; Position Digital/Analog-Ausgabe
  inc ZH
  ldi ZL,8
  rcall LcdPos
;
; 3 Byte-Zahl in rM0H:rM0M:rM0L in dezimal
; auf der LCD ausgeben
;
; Wandelt Zahl bis 9,99 Mio. in Dezimalformat um
; und gib sie auf LCD aus, mit Unterdrueckung
; fuehrender Nullen und Dezimaltrennzeichen
;
Dezimal3Aus:
  ldi ZH,High(2*Dezimal3Tab)
  ldi ZL,Low(2*Dezimal3Tab)
  clt ; Fuehrende Nullen
Dezimal3Aus1:
  lpm XL,Z+ ; lese Dezimalzahl
  lpm XH,Z+
  lpm rHilf,Z+
  clr rmp ; Teilerzaehler
  cp XL,rmp
  brne Dezimal3Aus2
  cp XH,rmp
  brne Dezimal3Aus2
  cp rHilf,rmp
  breq Dezimal3AusEnd
Dezimal3Aus2:
  sub rM0L,XL ; abziehen
  sbc rM0M,XH
  sbc rM0H,rHilf
  brcs Dezimal3Aus3 ; Ueberlauf
  inc rmp
  rjmp Dezimal3Aus2 ; weiter subtrahieren
Dezimal3Aus3:
  add rM0L,XL ; Ruecknahme Subtraktion
  adc rM0M,XH
  adc rM0H,rHilf
  tst rmp ; Null?
  brne Dezimal3Aus4 ; Nicht Null
  brts Dezimal3Aus5 ; keine Nullen unterdr.
  ldi rmp,' '
  rcall LcdD4Byte
  ldi rmp,' '
  rjmp Dezimal3AusKomma
Dezimal3Aus4:
  set ; keine fuehrenden Nullen unterdr.
Dezimal3Aus5:
  subi rmp,-'0'
  rcall LcdD4Byte
  ldi rmp,'.'
Dezimal3AusKomma:
  cpi XL,Byte1(1000000)
  breq Dezimal3AusKomma1
  cpi XL,Byte1(1000)
  breq Dezimal3AusKomma1
  rjmp Dezimal3Aus1
Dezimal3AusKomma1:
  rcall LcdD4Byte
  rjmp Dezimal3Aus1
Dezimal3AusEnd:
  ldi rmp,'0' ; letzte Ziffer
  add rmp,rM0L ; addieren
  rcall LcdD4Byte
  rjmp Neustart
;
Dezimal3Tab:
.db Byte1(1000000),Byte2(1000000)
.db Byte3(1000000),Byte1(100000)
.db Byte2(100000),Byte3(100000)
.db Byte1(10000),Byte2(10000)
.db Byte3(10000),Byte1(1000)
.db Byte2(1000),Byte3(1000)
.db Byte1(100),Byte2(100)
.db Byte3(100),Byte1(10)
.db Byte2(10),Byte3(10)
.db 0,0,0,0
```

```
;
; Routine Indukt
;   wird von Auswerten aufgerufen wenn Mess-
;   zyklus Induktivitaet gemessen hat
;
; Rechnet Induktivitaet aus und gibt sie aus.
; Prueft, ob Frequenz kleiner 23 oder groe-
; sser 0x0B5050 ist und gibt Fehlermeldung
; aus.
; Multipliziert F mit sich selbst und teilt
; Konstante in Dividendtabelle durch F hoch
; 2. Gibt das 32-Bit-Ergebnis auf der LCD
; aus.
;
Indukt:
  ldi ZH,3
  ldi ZL,6
  rcall LcdPos
  tst rM0M
  brne InduktN2
  tst rM0H
  brne InduktN2
  mov rmp,rM0L ; LSB Frequenz in rmp
  cpi rmp,2 ; Frequenz 0 oder 1?
  brcc InduktN1
  ; F = 0 oder 1, Zeile leeren und Null
  ldi XL,10
InduktN0:
  ldi rmp,' '
  rcall LcdD4Byte
  dec XL
  brne InduktN0
  ldi rmp,'0'
  rjmp LcdD4Byte
;
InduktN1: ; F groesser Null
  cpi rmp,23 ; F zwischen 2 und 22?
  brcc InduktN2
  ldi ZH,High(2*Underflow22)
  ldi ZL,Low(2*Underflow22)
  rjmp LcdTextC
Underflow22:
.db "(F < 23 Hz) ",0xFE,0xFF
InduktN2:
  ldi rmp,0x50
  cp rM0L,rmp
  cpc rM0M,rmp
  ldi rmp,0x0B
  cpc rM0H,rmp
  brcs InduktN3
  ldi ZH,High(2*Overflow)
  ldi ZL,Low(2*Overflow)
  rjmp LcdTextC
Overflow:
.db " (F > Max)  ",0xFE,0xFF
InduktN3:
  ; rM0H:rM0M:rM0L mit sich selbst multipliz
  mov rMH0,rM0L ; Zahl kopieren
  mov rMH1,rM0M
  mov rMH2,rM0H
  clr rMH3
  clr rMH4
  clr rME0 ; Ergebnis leeren
  clr rME1
  clr rME2
  clr rME3
  clr rME4
Indukt1:
  lsr rM0H ; niedrigstes Bit herausschieben
  ror rM0M
  ror rM0L
  brcc Indukt2
  add rME0,rMH0 ; zum Ergebnis addieren
  adc rME1,rMH1
  adc rME2,rMH2
  adc rME3,rMH3
  adc rME4,rMH4
```

```
Indukt2:
  tst rM0L
  brne Indukt3
  tst rM0M
  brne Indukt3
  tst rM0H
  breq Indukt4
Indukt3:
  lsl rMH0 ; mit 2 multiplizieren
  rol rMH1
  rol rMH2
  rol rMH3
  rol rMH4
  rjmp Indukt1
Indukt4:
  ; Division, Dividend in SRAM laden
  ldi ZH,High(2*Dividendtabelle)
  ldi ZL,Low(2*Dividendtabelle)
  ldi XH,High(sDividend)
  ldi XL,Low(sDividend)
Indukt5:
  lpm rmp,Z+ ; Dividend aus Tabelle
  st X+,rmp ; in SRAM
  cpi XL,Low(sDividendEnde)
  brcs Indukt5
  ; Dividend leeren
  clr rMH0
  clr rMH1
  clr rMH2
  clr rMH3
  clr rMH4
  ; Ergebnis leeren
  clr rM0L
  clr rM0M
  clr rM0H
  clr ZL
  clr ZH
  ; Dividend in SRAM rechts schieben
  ldi rmp,8*(sDividendEnde-sDividend)+1
  mov R0,rmp
Indukt6:
  ; Dividieren
  ldi XH,High(sDividend)
  ldi XL,Low(sDividend)
  ldi rmp,sDividendEnde-sDividend
  mov R1,rmp
  clc ; Carry loeschen
Indukt7:
  ld rmp,X ; Byte aus SRAM laden
  rol rmp ; hoechstes Bit in Carry
  st X+,rmp ; Byte in SRAM speichern
  dec R1
  brne Indukt7
  ; Carry in Hilfsregister schieben
  rol rMH0
  rol rMH1
  rol rMH2
  rol rMH3
  rol rMH4
  sub rMH0,rME0 ; Abziehen
  sbc rMH1,rME1
  sbc rMH2,rME2
  sbc rMH3,rME3
  sbc rMH4,rME4
  brcc Indukt8 ; Abziehen korrekt
  add rMH0,rME0 ; wieder addieren
  adc rMH1,rME1
  adc rMH2,rME2
  adc rMH3,rME3
  adc rMH4,rME4
  clc ; Carry clr
  rjmp Indukt9
Indukt8:
  sec ; Carry auf Eins
Indukt9:
  dec R0
  breq Indukt10 ; weiter dividieren
  rol rM0L ; in Ergebnis rollen
  rol rM0M
  rol rM0H
  rol ZL
  rol ZH
  rjmp Indukt6
  ; Ergebnis runden
Indukt10:
  ldi rmp,0
  adc rM0L,rmp
  adc rM0M,rmp
  adc rM0H,rmp
  adc ZL,rmp
  adc ZH,rmp
  mov rMH0,ZL
  mov rMH1,ZH
;
; 4 Byte-Zahl in rMH0:rM0H:rM0M:rM0L in dezimal
; auf der LCD ausgeben
;
; Wandelt in dezimal um, unterdrueckt fuehrende
; Nullen und Dezimaltrennzeichen
;
Dezimal4Aus:
  ldi ZH,High(2*Dezimal4Tab)
  ldi ZL,Low(2*Dezimal4Tab)
  clt ; Fuehrende Nullen
Dezimal4Aus1:
  lpm rME0,Z+ ; Lese Dezimalzahl
  lpm rME1,Z+
  lpm rME2,Z+
  lpm rME3,Z+
  clr rmp
  or rmp,rME0 ; Ende der Tabelle?
  or rmp,rME1
  or rmp,rME2
  or rmp,rME3
  breq Dezimal4AusEnd
  clr rmp
Dezimal4Aus2:
  sub rM0L,rME0 ; abziehen
  sbc rM0M,rME1
  sbc rM0H,rME2
  sbc rMH0,rME3
  brcs Dezimal4Aus3
  inc rmp
  rjmp Dezimal4Aus2
Dezimal4Aus3:
  add rM0L,rME0 ; abziehen rueckgaengig
  adc rM0M,rME1
  adc rM0H,rME2
  adc rMH0,rME3
  tst rmp
  brne Dezimal4Aus4
  brts Dezimal4Aus5
  ldi rmp,' '
  ldi rmp,' '
  rjmp Dezimal4Aus6
Dezimal4Aus4:
  set ; Kein fuehrenden Nullen
Dezimal4Aus5:
  subi rmp,-'0'
  ldi rmp,'.'
Dezimal4Aus6:
  cpi ZL,Low(2*Dezimal4Tab1Mio)
  breq Dezimal4Aus7
  cpi ZL,Low(2*Dezimal4Tab1000)
  brne Dezimal4Aus1
Dezimal4Aus7:
  rjmp Dezimal4Aus1
Dezimal4AusEnd:
  ldi rmp,'0'
  add rmp,rM0L
  rcall LcdD4Byte
  rjmp Neustart
;
Dividendtabelle:
```

```
.db 0x7F,0xD7,0x10,0xF4,0x75,0x00
;
Dezimal4Tab:
.dw LWRD(100000000),HWRD(100000000)
.dw LWRD(10000000),HWRD(10000000)
.dw LWRD(1000000),HWRD(1000000)
Dezimal4Tab1Mio:
.dw LWRD(100000),HWRD(100000)
.dw LWRD(10000),HWRD(10000)
.dw LWRD(1000),HWRD(1000)
Dezimal4Tab1000:
.dw LWRD(100),HWRD(100)
.dw LWRD(10),HWRD(10)
.dw 0,0
;
Neustart:
  clr rM0L ; Letztes Ergebnis loeschen
  clr rM0M
  clr rM0H
  inc rFlag ; naechster Messmodus
  cpi rFlag,3
  brcs NeuStart1
  clr rFlag ; erster Messmodus
NeuStart1:
  ; rFlag=0:digital, =1:analog, =2:L
  cpi rFlag,0x01
  brcs NeustartDigital
  breq NeustartAnalog
  ; Neustart Induktivitaetsmessung
  ldi rmp,1<<PCINT0 ; Pin Change PA0
  rjmp NeuStartPcInt
  ; Digital messen, PCINT3 aktivieren
NeuStartDigital:
  ldi rmp,1<<PCINT3 ; Pin change PA3
NeuStartPcInt:
  out PCMSK0,rmp ; maskieren
  ldi rmp,1<<PCIE0 ; PCINT0 Interrupt
  out GIMSK,rmp ; in Int-Maske
  rjmp Neustart2
NeustartAnalog:
  clr rmp
  out GIMSK,rmp
  ldi rmp,1<<ACIE ; Enable Int Comparator
  out ACSR,rmp
Neustart2:
  ldi rmp,1<<TSM ; Prescaler Sync Mode
  out GTCCR,rmp
  ldi rmp,(1<<TSM)|(1<<PSR10) ; Reset Presc1
  out GTCCR,rmp
  clr rmp
  out GTCCR,rmp ; Prescaler Count Mode
  out TCNT1H,rmp ; 16-Bit-Zaehler ruecksetzen
  out TCNT1L,rmp
  ldi rmp,1<<OCIE1A
  out TIMSK1,rmp ; in Timer-Int-Maske
  ret
;
; LCD Starttext
LcdTextOut:
.db "Frequenzmesser tn24 ",0x0D,0xFF
.db "F(dig)= x.xxx.xxx Hz",0x0D,0xFF
;            8
.db "F(ana)= x.xxx.xxx Hz",0x0D,0xFF
;            8
.db "L =   xxx.xxx.xxx ",0xE4,"H",0xFE,0xFE
;          6
; LCD-Include
.include "Lcd4Busy.inc"
;
; Ende Quellcode
;
```

13.5.6 Simulation der Programmausführung

Die Simulation wird wieder mit avr_sim vorgenommen. Alle Aufrufe von LCD-Routinen werden dazu mit ";" auskommentiert und durch Schreiboperationen in das SRAM ersetzt, um die Ergebnisse anschauen zu können. Das Registerpaar Y kann für diese Schreiboperationen verwendet werden.

Nützlich ist es, die 24-Bit-Dezimalumwandlung und die Induktivitätsberechnung zu simulieren.

13.5.6.1 Simulation der 24-Bit-Binär-Dezimal-Umwandlung

Register

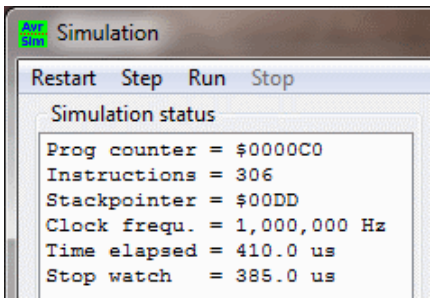
Reg	+0	+1	+2	+3	+4
R0	00	00	87	D6	12
R8	00	00	00	00	00
R16	87	00	00	00	00
R24	00	00	00	00	60

Um die Dezimalumwandlung zu erproben, wird die Dezimalzahl 1.234.567 in binärem Format in die Register rM0H:rM0M:rM0L geladen und damit die Umwandlungsroutine aufgerufen.

Das Umwandlungsergebnis, statt auf die LCD ins SRAM geschrieben, ist korrekt.

SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	31	2C	32	33	34	2C	35	36	37	FF	FF	FF	FF	FF	FF	FF	1,234,567..
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	.....



Die Zeit, die für die Umwandlung benötigt wird, ist deutlich unterhalb von einer Millisekunde. Das stört die parallel ablaufende nächste Frequenzmessung nicht im Geringsten, selbst wenn wir noch die Schreiboperationen zur LCD (ca. 40 µs pro Zeichen) dazu zählen.

13.5.6.2 Simulation der Induktiviätsberechnung

Register

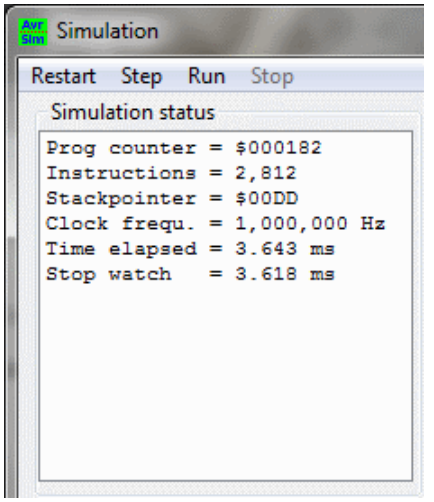
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	E8	03	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	E8	00	00	00	00	00	00	00
R24	00	00	00	00	90	00	00	00

Als Erstes simulieren wir die Induktivitätsberechnung mit einer Frequenz von 1.000 Hz. Dieser Wert wird in die Register R4:R3:R2 geschrieben.

Das Resultat ist mit 506,606 mH korrekt (wie die obige Beispielrechnung ausweist).

SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	00	00	00	00	00	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	20	20	20	20	35	30	36	2C	36	30	36	FF	FF	FF	FF	FF	506,606. . . .



3,6 ms sind vergangen. Die Division dauert halt etwas länger als die Multiplikation, insbsondere wenn man die zu dividierende Zahl mühsam Bit für Bit aus dem SRAM in das Divisionsregister schieben muss.

Das wäre das Ergebnis, wenn der LC-Oszillator mit gigantischen 10.000 Hz daherkäme: 5,6 mH.

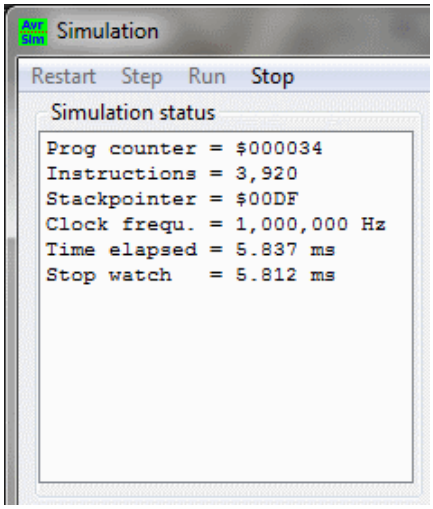
SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	00	00	00	00	00	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	20	20	20	20	20	20	35	2C	30	36	36	FF	FF	FF	FF	FF	5,066. . . .

Das andere Extrem: Simulation von 50 Hz Oszillatorfrequenz resultiert in einer Induktivität von 202 H, was eine ziemlich große Spule wäre.

SRAM

	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	00	00	00	00	00	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	32	30	32	2C	36	34	32	2C	33	36	37	FF	FF	FF	FF	FF	202,642,367. . . . .



Der Zeitbedarf bei der Berechnung von 50 Hz ist ein bisschen länger, dauert aber nicht zu lang.

Simulation hilft dabei, komplizierte Berechnungen zu entwickeln und zu überprüfen, ob sie unter allen realistischen Bedingungen korrekt rechnen. Damit können Unterprogramme mit einer klar definierten Aufgabenstellung mit Eingangszahlen (in Registern oder im SRAM) gefüttert werden und es kann ziemlich einfach gecheckt werden, ob sie zu korrekten Resultaten kommen.

13.5.7 Messbeispiele

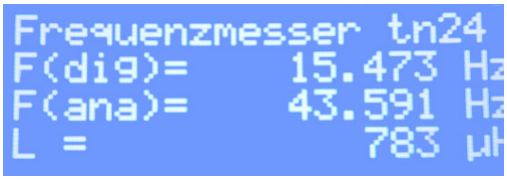
Bei den beiden folgenden Messungen waren der Digital- und der Analog-Eingang unbeschaltet, die dargestellten Messwerte in Zeile 2 und 3 der LCD sind Einstrahlungen aus den Messobjekten und kapazitive Einstreuungen vom Aufbau.

13.5.6.1 Große Spule

Das ist die Messung mit einer relativ großen Spule. Die Messung stimmt relativ gut mit der auf andere Weise ermittelten Induktivität überein. Wieder ist die angezeigte Induktivität etwas zu groß, die gemessene Frequenz also etwas zu niedrig.

13.5.6.2 Lautsprecherspule

Auch die Lautsprecherspule, die wir in einem anderen Experiment verwendet haben, hat eine Induktivität. Sie ist mit knapp 800 µH doch recht bescheiden.



Lektion 14: Spannungs-, Strom- und Temperaturmessgerät

Jetzt wird es ganz praktisch: wir messen Spannungen (mit höherem Messbereich wie bei einem Digitalmultimeter), relativ hohe Ströme (und machen uns dabei mit der im ATtiny24 vorhandenen Möglichkeit vertraut, Differenzspannungen mit hoher Verstärkung zu messen) und wir werfen den im ATtiny24 eingebauten Temperatursensor an. Alles Zeugs also, das der gemeine Basic- und C-Programmierer so hasst, weil er es entweder gar nicht zu Gesicht bekommt (die Differenzverstärker oder den Temperatursensor beim Basic) oder es schon nur deswegen nicht nutzt, weil es nicht in allen AVR in immer gleicher Weise vorhanden und eingebaut ist (C-Programmierer hassen Hardware, die es nur in wenigen AVR gibt und die daher in keiner Bibliothek auftauchen und insbesondere solche, die sich nicht auf beliebige andere Prozessoren übertragen lässt). Der gemeine Assemblerprogrammierer hat kein Verständnis für solche Art der Selbstkasteiung und nutzt ohne jeden Skrupel, was er mit ein paar Zeilen Quellcode einschalten kann.

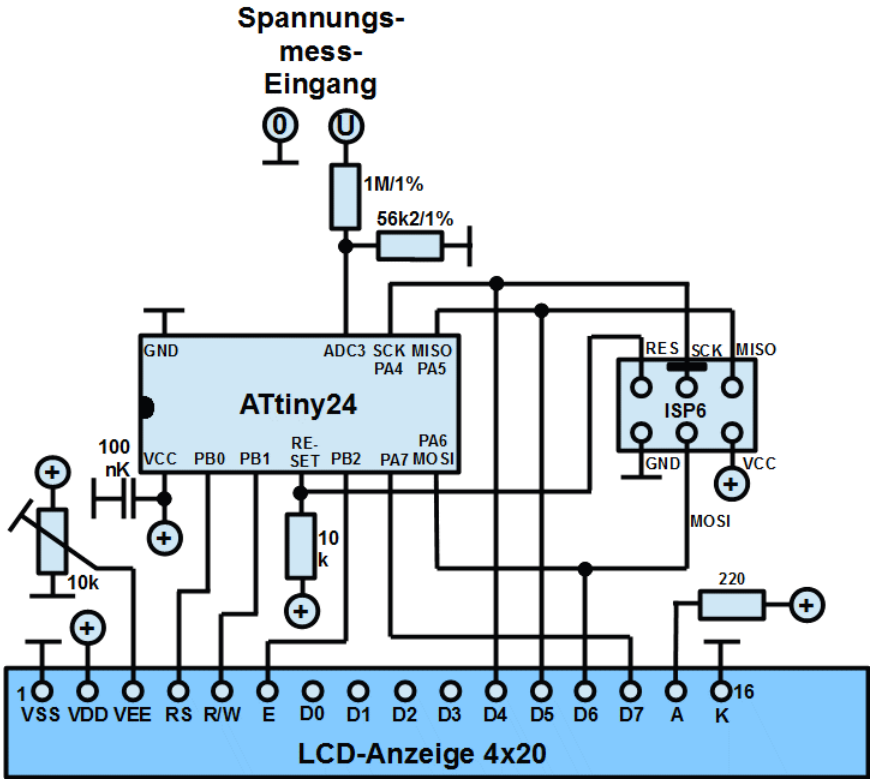
14.0 Übersicht

- 14.1 Spannungen messen, umrechnen und darstellen
- 14.2 Ströme messen, umrechnen und darstellen
- 14.3 Temperaturen messen, umrechnen und darstellen

14.1 Spannungen messen, umrechnen und darstellen

14.1.1 Schaltbild

Der Bauteilebedarf für die Spannungsmessung ist minimal: ein Spannungsteiler, der die Messspannung herabsetzt und die geteilte Spannung an den ADC3-Eingang heranführt. In dieser Konfiguration können bis etwa 20,5 V gemessen werden. Die Größe des 56k2 ergibt sich daraus, dass für das Sampling des ADC keine übermäßige Extrazeit erforderlich wird (bei zu hohem Eingangswiderstand wird der Sampling-Zeitraum von 1,5 Takten auf 2 oder mehr verlängert).



14.1.2 Bauteile

Links ist der Widerstand von 56,2 k abgebildet, rechts der 1M. Die exakte Größe der Widerstände ist unkritisch, weil sie im Kopf der Software beliebig verstellt werden können.

Alternativer Aufbau

Wer diese Experimente lieber auf kompaktere Weise aufbauen möchte und die vielen Zuleitungen zur LCD fest verdrahtet sehen möchte, kann sich das Board hier als gedruckte Platine bauen. Alle Programmbeispiele dieser Lektion laufen darauf ohne Änderungen.

14.1.3 Messbereich, Mess-, Umrechnungs- und Darstellungsvorgang

14.1.3.1 Messbereich

Bei Vollausschlag erreicht der ADC-Eingang die Referenzspannung 1,1 V. Die Spannung an ADC3 ist $U_{ADC3} = U_{Ein} * 56,2 / (1000 + 56,2) = 1,1$ [V]. Also ist $U_{Ein} = 1,1 * (1000 + 56,2) / 56,2 = 20,673$ [V]. Pro ADC-Bit beträgt die Auflösung 0,02 [V].

14.1.3.2 Messvorgang

Bei der Messung kommen folgende Bedingungen zum Einsatz:

- Als Quelle wird beim Init der ADC3 im MUX-Port der ADC eingestellt. Da keine anderen Kanäle gemessen werden, bleibt es dauerhaft dabei.
- Als Referenz für den ADC wird die interne Referenzspannung von 1,1 V verwendet. Das macht die Messergebnisse unabhängig von der Betriebsspannung.
- Die Messungen werden über 64 Einzelmessungen gemittelt. Das ergibt stabile Ergebnisse und eine Messfreuenz von 119 Hz.

14.1.3.3 Umrechnung

In die Umrechnung der Zählergebnisse des ADC gehen folgende Parameter ein:

- Das Verhältnis der beiden Widerstände mit $U = (1M+56k2)/56k2 * U_{mess}$.
- Bei einer Spannung von 1,1 V liefert der ADC als Ergebnis 1024, also ist $N_{ADC} = U_{mess} * 1024 / 1,1$.
- Durch das Aufsummieren von 64 Messergebnissen liefert der ADC $64 * N_{ADC}$ als Summenwert.
- Als Darstellungsauflösung werden Hundertstel Volt (0,01 V) gewählt. Bei Vollausschlag von 20 V entspricht das der erzielbaren Genauigkeit der Messung mit dem 10-Bit-ADC.

Daraus ergibt sich die anzeigende Spannung (in 0,01 V) zu

$$U (0,01V) = (1M+56k2)/56k2 * 1,1 / 1024 * 100 / 64 * N_{Sum-ADC}$$

Die vordersten Parameter, mit denen die ermittelte Summe zu multiplizieren ist, beträgt

$$U (0,01V) = 0,03154442 * N_{Sum-ADC}$$

Damit wir keine Fließkommabibliothek brauchen, multiplizieren wir den Faktor mit 65536 (und streichen die untersten 16 Bits des Multiplikationsergebnisses einfach weg), also etwa so:

$$U (0,01V) = 2067 * N_{Sum-ADC} / 65536$$

Damit ist die Umrechnung eine 16-Bit-mal-16-Bit-Aufgabe, mit einem bis zu 32 Bit langen Resultat.

Durch eine geänderte Konstante lassen sich alle beliebigen Teilerverältnisse von Widerständen leicht abbilden.

14.1.3.4 Darstellungsvorgang

Das Ergebnis der Multiplikation und das Teilen durch 65536 liefern Werte bis knapp über 2.000. Für die Konvertierung in ASCII-Ziffern reicht es also aus, maximal die Tausender zu ermitteln. Die Unterdrückung führender Nullen darf sich nur auf die Tausender auswirken, danach nicht mehr. Das Dezimalkomma ist nach den Hunderten zu setzen.

14.1.4 Programm

Das Programm ist recht straight forward (zum Quelltext im asm-Format geht es hier, es wird noch die Include-Datei benötigt).

```
;
; ************************************
; * Spannungen messen mit dem ATtiny24 *
; * (C)2016 by www.gsc-elektronbic.net *
; ************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; -------- Programmablauf --------------
;
; Die analoge Spannung am ADC3-Eingang wird
; fortlaufend gemessen und ueber 64 Messungen
; gemittelt. Das gemittelte Messergebnis wird
; mit dem Widerstandsverhaeltnis multipli-
; ziert und auf der LCD dezimal ausgegeben.
;
; -------- Schalter --------------------
.equ debug = 0
.equ debugN = 1000
;
; -------- Hardware --------------------
;   ATtiny24          _______
;              1 /        |14
;       VCC o--|VCC   GND|--o GND
;              2|         |13
;    LCD-RS o--|PB0      |--o NC
;              3|         |12
;    LCD-RW o--|PB1      |--o NC
;              4|         |11
;   VCC-10k o--|RES      |--o NC
;              5|         |10  1 M  - Ein
;     LCD-E o--|PB2 ADC3 |--o 56k2 - GND
;              6|         |9
;    LCD-D7 o--|PA7   PA4|--o LCD-D4
;              7|         |8
;    LCD-D6 o--|PA6   PA5|--o LCD-D5
;               |_________|
;
; -------- Timing ----------------------
; Taktrate          1.000.000 Hz
; ADC-Vorteiler           128
; ADC-Takt              7.812,5 Hz
; ADC-Takte pro Messg.     14,5
; Messfrequenz            538,8 Hz
; Messzyklus, 64 Messungen  8,42 Hz
;
; -------- Konstanten ------------------
.equ cRE =   1000000 ; Ohm
.equ cRG  =     56200 ; Ohm
.equ cMult = ((cRE+cRG)*110+cRG/2)/cRG
;
; -------- Register --------------------
; frei: R0 .. R4
.def rAL= R5
.def rAH = R6
.def rM0 = R7
.def rM1 = R8
.def rM2 = R9
.def rM3 = R10
.def rE0 = R11
.def rE1 = R12
.def rE2 = R13
.def rE3 = R14
.def rSreg = R15 ; SREG-Sicherung
.def rmp = R16 ; Vielzweckregister
.def rimp = R17 ; Interrupt Vielzweck
.def rFlag = R18 ; Flaggenregister
        .equ bRdy = 0 ; Ready-Flagge
.def rCtr = R19 ; Zaehler Messungen
.def rLese = R20 ; LCD
.def rLine = R21 ; LCD
.def rmo = R22 ; LCD
; frei: R23 .. R29
; verwendet: Z = R31:R30 fuer LCD
;
; -------- Reset- und Interrupt-Vektoren
  rjmp Start ; RESET-Vektor
  reti ; INT0 External Int Request 0
  reti ; PCINT0 Pin Change Int Request 0
  reti ; PCINT1 Pin Change Int Request 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT TC1 Capture Event
  reti ; TIM1_COMPA TC1 Compare Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF TC1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; TIM0_OVF TC0 Overflow
  reti ; ANA_COMP Analog Comparator
  rjmp AdcIsr ; ADC ADC Conv Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; -------- Interrupt Service Routinen --
;
; ADC Ready Interrupt
;   wird nach dem Abschluss der AD-Wandlung
;   ausgeloest
;
; Summiert das Wandlungsergebnis auf und
; zaehlt den Zaehler rCtr abwaerts. Ist
; dieser Null wird die bRdy-Flagge gesetzt.
; Die nachste Wandlung wird angestossen.
;
AdcIsr:
```

```
  in rSreg,SREG ; Status retten
  in rimp,ADCL ; Addiere Ergebnis
  add rAL,rimp
  in rimp,ADCH
  adc rAH,rimp ; mit Uebertrag
  dec rCtr ; Zaehler Messwerte
  brne AdcIsrRet ; nicht Null
  sbr rFlag,1<<bRdy
  ldi rCtr,64 ; Neustart Zaehler
  mov rM0,rAL ; Transfer Messwertsumme
  mov rM1,rAH
  clr rAL ; Messwertsumme loeschen
  clr rAH
AdcIsrRet:
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rimp ; Starte naechste Messung
  out SREG,rSreg
  reti
;
; --------- Reset-Vektor, Programmstart
Start:
  ; Stapel anlegen
  ldi rmp,LOW(RAMEND) ; Ende SRAM
  out SPL,rmp ; in Stapelzeiger
.if debug == 1
  ldi rmp,Low(debugN*64)
  mov rM0,rmp
  ldi rmp,High(debugN*64)
  mov rM1,rmp
  rjmp Ausgabe
  .else
  ; LCD Init
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
Schreiben
  out pLcdDR,rmp ; auf Richtungsregister
Datenausgabeport
  ; LCD Init
  rcall LcdInit ; starten LCD
  ldi ZH,High(2*LcdTextOut) ; Z auf Text
  ldi ZL,Low(2*LcdTextOut)
  rcall LcdText ; Gib Text aus
  ldi rmp,0x0C ; Cursor und Blink aus
  rcall LcdC4Byte
  ; ADC starten
  ldi rmp,(1<<REFS1)|(1<<MUX1)|(1<<MUX0) ;
Int.Ref, Kanal3
  out ADMUX,rmp
  clr rAL ; Summe loeschen
  clr rAH
  ldi rCtr,64 ; 64 Messungen
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; Starte erste Messung
  .endif
  ; Schlafmodus
  ldi rmp,1<<SE ; Sleep Mode Idle
  out MCUCR,rmp
  ; Interrupts
  sei ; setze Int-Flagge
  ;
; Hauptprogrammschleife
Schleife:
  sleep ; Schlafen
  nop ; Aufwachen
  sbrc rFlag,bRdy ; ueberspringe wenn noch nicht
fertig
  rcall Ausgabe
  rjmp Schleife
;
; Ergebnis ausrechnen
Ausgabe:
  cbr rFlag,1<<bRdy ; loesche Flagge
  ; mit Konstante cMult multiplizieren
  clr rM2 ; Bytes 3 und 4 loeschen
  clr rM3
  clr rE0 ; Ergebnis loeschen
  clr rE1
  clr rE2
  clr rE3
  ldi ZH,High(cMult) ; Konstante in Z
  ldi ZL,Low(cMult)
Ausgabe1:
  lsr ZH ; Bit aus MSB in LSB
  ror ZL ; niedrigstes Bit in Carry
  brcc Ausgabe2
  add rE0,rM0 ; Multiplikator addieren
  adc rE1,rM1
  adc rE2,rM2
  adc rE3,rM3
Ausgabe2:
  mov rmp,ZL
  or rmp,ZH
  breq Ausgabe3
  lsl rM0 ; Multiplikator mal zwei
  rol rM1
  rol rM2
  rol rM3
  rjmp Ausgabe1 ; weiter multipliz.
Ausgabe3:
  ; Runden, Ergebnis in rE3:rE2
  ldi rmp,0x7F
  add rE0,rmp
  adc rE1,rmp
  ldi rmp,0
  adc rE2,rmp
  adc rE3,rmp
  ; Auf LCD ausgeben
  ldi ZH,1
  ldi ZL,4
  rcall LcdPos
  clt ; Fuehrende Nullen
  ldi ZH,High(2*DezimalTab)
  ldi ZL,Low(2*DezimalTab)
Ausgabe4:
  lpm rE0,Z+ ; lese Dezimalzahl
  lpm rE1,Z+
  tst rE0 ; LSB Null?
  breq Ausgabe8
  clr rmp
Ausgabe5:
    ; Dezimalzahl abziehen
  sub rE2,rE0
  sbc rE3,rE1
  brcs Ausgabe6 ; fertig
  inc rmp ; weiter abziehen
  rjmp Ausgabe5
Ausgabe6:
  add rE2,rE0 ; wieder addieren
  adc rE3,rE1
  tst rmp ; Null?
  brne Ausgabe7
  brts Ausgabe7
  ldi rmp,' '
  rcall LcdD4Byte
  set
  rjmp Ausgabe4
Ausgabe7:
  set
  subi rmp,-'0'
  rcall LcdD4Byte
  cpi ZL,LOW(2*DezimalTab10)
  brne Ausgabe4
  ldi rmp,',' ; Kommastelle aus
  rcall LcdD4Byte
  rjmp Ausgabe4
Ausgabe8:
  ldi rmp,'0'
  add rmp,rE3
```

```
  rjmp LcdD4Byte
;
; Dezimaltabelle Tausender abwaerts
DezimalTab:
.dw 1000
.dw 100
DezimalTab10:
.dw 10
.dw 0
;
; --------- LCD-Routinen ---------------
```

```
LcdTextOut:
.db "  Spannungsmessung  ",0x0D,0xFF
.db "U = xx,xx V",0xFE
;        4

.include "Lcd4Busy.inc"
;
;
; Ende Quellcode
;
```

14.1.5 Messbeispiel

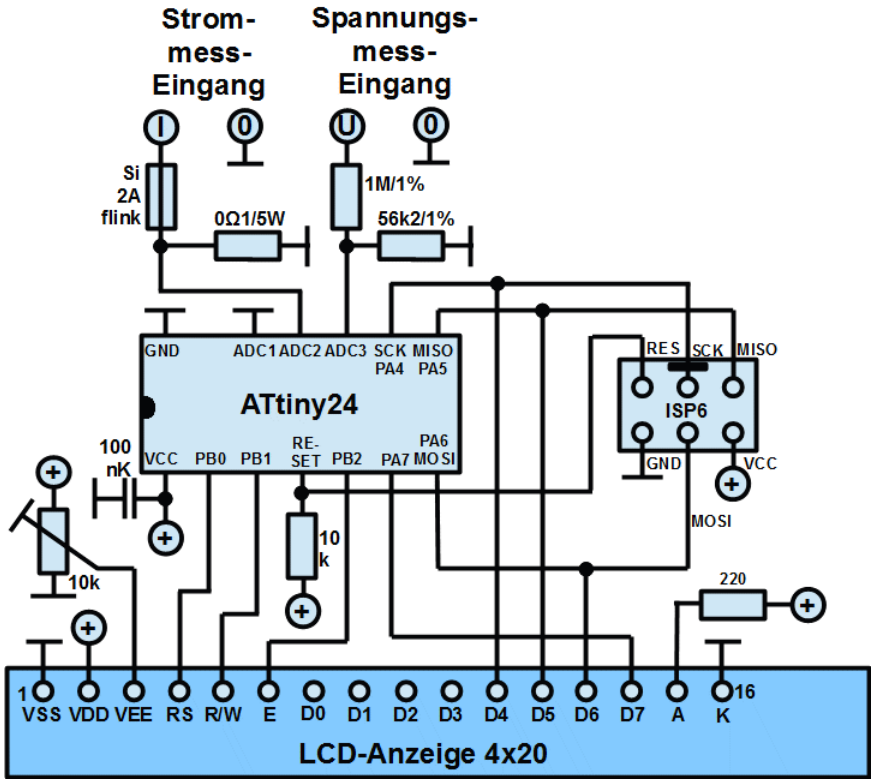
So sieht die Anzeige beim Messen der eigenen Betriebsspannung aus.

Top	Home	Spannung messen	Strommessen	Temperaturmessen

14.2 Ströme messen, umrechnen und darstellen

Beim Messen von Strömen kommt es im Gegensatz zur Spanungsmessung darauf an, einen möglichst niedrigen Eingangswiderstand zu haben, um das Messobekt durch die Messschaltung nicht übermässig zu beeinflussen.

14.2.1 Schaltbild

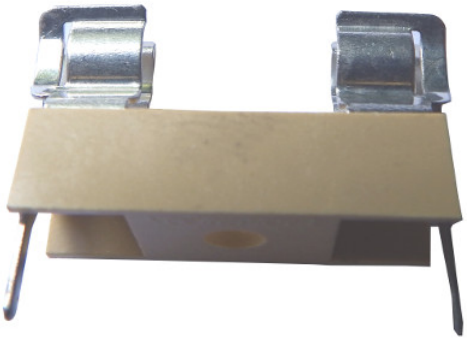


Gemessen wird der Strom über den Spannungsabfall an dem Widerstand von 0,1 Ω. Die vorgeschaltete Sicherung von 2 A schützt auch den Prozessoreingang gegen Überlastung. Der Eingang ADC1 dient als Differential-eingang, um die Spannung am Messeingang ADC2 um das 20-Fache zu verstärken.

14.2.2 Bauteile

Das hier ist ein handelsüblicher 0,1 Ω-Widerstand. Eigentlich ist die Leistung von 5 W, die dieser Widerstand abkann, bei Weitem überkandidelt. Es gibt aber keine 0,25- oder 1 W-Typen zu kaufen, also nehmen wir dies hier.

So sieht die 2A-Sicherung aus.Es tut auch jeder andere Typ oberhalb 550 mA.



Das ist ein entsprechender Sicherheitshalter, noch ohne angelötete Drahtstücke für das Breadboard. Und dazu gehört so eine Plastikhaube. Eigentlich hier nicht erforderlich, da wir ja nur im Niederspannungsbereich messen.

14.2.2 Messbereich, Mess-, Umrechnungs- und Darstellungsvorgang

14.2.2.1 Messbereich

Bei Vollausschlag erreicht der ADC-Eingang die Referenzspannung 1,1 V, geteilt durch die gain von 20 also 0,055 V. Bei einem Widerstand von 0,1 [Ω] entspricht das einem Strom von I = U / R = 0,055 / 0,1 = 0,55 [A] oder 550 [mA]. Pro ADC-Bit ergibt sich eine Auflösung von 550/1024 = 0,53 [mA].

14.2.2.2 Messung

Wie bei der Spannungsmessung werden auch am Stromeingang 64 Einzelmesswerte aufsummiert und erst dann ausgewertet.

Bei der Messung wird der Diffentialeingang ADC2 - ADC1 verwendet. ADC1 liegt auf 0 V, als Verstärkung der Differenz von (ADC2 - ADC1) wird eine gain von 20 eingestellt.

Der maximal messbare Strom beträgt in dieser Konfiguration 0,55 A, bei einer Auflösung von 0,5 mA/Digit. Will man höhere Ströme messen und anzeigen, dann stellt man den AD-Wandler auf eine Gain von 1 um. Bei gleicher Referenzspannung von 1,1 V kann man dann bis zu 11 A messen, allerdings würde das dann die maximale Leistung des 0,1 Ω-Widerstands mit 12,1 W überschreiten. Unter Einhaltung dieser maximalen Last wären maximal 7 A messbar, als Dauerlast und bei normaler Wärmeabfuhr des Widerstands ca. 3,5 A. Die Messauflösung bei dieser Konfiguration betrüge 11 mA pro Digit, die Umrechnung und Werteausgabe wäre entsprechend anzupassen. Natürlich wäre auch die Absicherung gegen Überschreitung des zulässigen Maximalstroms (Sicherung) entsprechend von 2 A auf 8 A zu ändern.

14.2.2.3 Umrechnung

Die Messspannung am ADC2-Eingang ist U_{mess} [V] = R [Ω] * I [A], folglich ist I [A] = U_{mess} [V] / R [Ω]. Mit einer Differentialverstärkung von 20, die per Software eingestellt wird, ist I [A] = 20 * U_{mess} [V] / R [Ω]. Bei einer Referenzspannung von 1,1 V ist N_{mess} = 20 * U_{mess} * 1024 / 1,1. Also ist 20 * U_{mess} = N_{mess} * 1,1 / 1024 und folglich I [A] = N_{mess} * 1,1 / 1024 / R. Für 64 aufsummierte Messungen ist I [A] = $N_{Sum\text{-}mess}$ * 1,1 / 1024 / R / 64 / 20. Für eine Auflösung von 0,0001 A ist I [0,0001 A] = 10000 * $N_{Sum\text{-}mess}$ * 1,1 / 1024 / R / 64 / 20 oder 0,0083923 * $N_{Sum\text{-}mess}$ / R. Multipliziert mit 65536 ergibt sich ein Multiplikationsfaktor von 550. Mit 0,1 Ω wird daraus ein Multiplikationsfaktor von 5.500, also I [0,0001 A] = 5.500 * $N_{Sum\text{-}mess}$ / 65536.

14.2.2.4 Darstellungsvorgang

Die ermittelte Messwertsumme (16-Bit) wird mit der 16-Bit-Zahl 5.500 multipliziert und die beiden untersten Bytes des Ergebnisses werden zum Runden verwendet. Zur Konvertierung in das Anzeigeformat 123,4 mA werden zunächst die Tausender, dann die Hunderter und Zehner verwendet. Nach den Hundertern werden führende Leerzeichen nicht mehr unterdrückt. Vor der Einer-Ausgabe kommt noch das Komma.

14.2.3 Programm

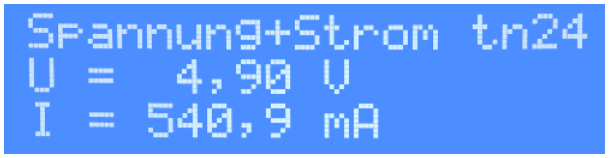
Das Programm ist hier (zum Quellcode geht es hier, es wird noch die LCD-Include-Datei benötigt).

```
;                                       ; Messzyklus, 64 Messungen  8,42 Hz
; ************************************  ; Zwei Messzyklen, 128 M.   4,21 Hz
; * Spannungen und Stroeme messen     *  ;
; * (C)2016 by www.gsc-elektronbic.net *  ; -------- Konstanten ------------------
; ************************************  ; Spannungsmessung
;                                       .equ cRE =   1000000 ; Ohm
.NOLIST                                 .equ cRG  =     56200 ; Ohm
.INCLUDE "tn24def.inc"                  .equ cMultU = ((cRE+cRG)*110+cRG/2)/cRG
.LIST                                   ; Strommessung
;                                       .equ cRI = 100 ; Milliohm
; -------- Programmablauf ---------------.equ cMultI = (550000+cRI/2) / cRI
;                                       ; ADC-MUX-Konstanten
; Nacheinander werden die Spannungen an .equ cMuxU = (1<<REFS1)|(1<<MUX1)|(1<<MUX0)
; den AD-Eingaengen                     .equ cMuxI = (1<<REFS1)|0b101101
; a) ADC3 mit dem Spannungsteiler, und  ;
; b) ADC2 und ADC1 als Differenzwert mit; -------- Register --------------------
;    dem Stromshunt                     ; frei: R0 .. R4
; je 64 mal gemessen und gemittelt und  .def rAL= R5
; danach in eine Spannung (a) oder einen.def rAH = R6
; Stromwert (b) umgerechnet und auf der .def rM0 = R7
; LCD ausgegeben.                       .def rM1 = R8
;                                       .def rM2 = R9
; -------- Hardware --------------------.def rM3 = R10
;   ATtiny24      _______                 .def rE0 = R11
;             1 /       |14             .def rE1 = R12
;       VCC o--|VCC  GND|--o GND         .def rE2 = R13
;             2|        |13             .def rE3 = R14
;    LCD-RS o--|PB0     |--o NC         .def rSreg = R15 ; SREG-Sicherung
;             3|        |12             .def rmp = R16 ; Vielzweckregister
;    LCD-RW o--|PB1 ADC1|--o GND        .def rimp = R17 ; Interrupt Vielzweck
;             4|        |11             .def rFlag = R18 ; Flaggenregister
;   VCC-10k o--|RES ADC2|--o 0,1 Ohm      .equ bRdyU = 0 ; Ready-Flagge Spannung
;             5|        |10  1 M  - Ein   .equ bRdyI = 1 ; Ready-Flagge Strom
;     LCD-E o--|PB2 ADC3|--o 56k2 - GND .def rCtr = R19 ; Zaehler Messungen
;             6|        |9              .def rLese = R20 ; LCD
;    LCD-D7 o--|PA7  PA4|--o LCD-D4     .def rLine = R21 ; LCD
;             7|        |8              .def rmo = R22 ; LCD
;    LCD-D6 o--|PA6  PA5|--o LCD-D5     ; frei: R23 .. R29
;              |________|                ; verwendet: Z = R31:R30 fuer LCD
;                                       ;
; -------- Timing ----------------------; -------- Reset- und Interrupt-Vektoren
; Taktrate          1.000.000 Hz          rjmp Start ; RESET-Vektor
; ADC-Vorteiler           128             reti ; INT0 External Int Request 0
; ADC-Takt              7.812,5 Hz        reti ; PCINT0 Pin Change Int Request 0
; ADC-Takte pro Messg.     14,5           reti ; PCINT1 Pin Change Int Request 1
; Messfrequenz            538,8 Hz        reti ; WDT Watchdog Time-out
```

```
  reti ; TIM1_CAPT TC1 Capture Event
  reti ; TIM1_COMPA TC1 Compare Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF TC1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; TIM0_OVF TC0 Overflow
  reti ; ANA_COMP Analog Comparator
  rjmp AdcIsr ; ADC ADC Conv Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; -------- Interrupt Service Routinen --
;
; ADC Ready Interrupt
;   wird nach jeder erfolgten AD-Wandlung
;   ausgeloest
;
; Addiert den Messwert zur Summe, zaehlt
; den rCtr abwaerts. Erreicht er Null,
; wird er Null wird der gegenteilige
; Messkanal im Multiplexer eingestellt.
; Wurde zuletzt U gemessen wird der
; Summenwert in die U-Zwischenregister
; kopiert und die bRdyU-Flagge gesetzt.
; Wurde I gemessen, wird der Summenwert
; in das I-Zwischenregister kopiert und
; die Flagge bRdyI gesetzt.
; In allen Faellen wird die naechste
; Wandlung gestartet.
;
AdcIsr:
  in rSreg,SREG ; Status retten
  in rimp,ADCL ; Addiere Ergebnis
  add rAL,rimp
  in rimp,ADCH
  adc rAH,rimp ; mit Uebertrag
  dec rCtr ; Zaehler Messwerte
  brne AdcIsrRet ; nicht Null
  in rimp,ADMUX ; Mux lesen
  cpi rimp,cMuxU ; Wurde U gemessen?
  breq AdcIsrI ; ja, starte I-Messung
  sbr rFlag,1<<bRdyI
  ldi rimp,cMuxU ; nein, starte U-Messung
  out ADMUX,rimp
  rjmp AdcIsrCont
AdcIsrI:
  sbr rFlag,1<<bRdyU
  ldi rimp,cMuxI ; starte I-Messung
  out ADMUX,rimp
AdcIsrCont:
  ldi rCtr,64 ; Neustart Zaehler
  mov rM0,rAL ; Transfer Messwertsumme
  mov rM1,rAH
  clr rAL ; Messwertsumme loeschen
  clr rAH
AdcIsrRet:
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rimp ; Starte naechste Messung
  out SREG,rSreg
  reti
;
; --------- Reset-Vektor, Programmstart
Start:
  ; Stapel anlegen
  ldi rmp,LOW(RAMEND) ; Ende SRAM
  out SPL,rmp ; in Stapelzeiger
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabemaske,
Schreiben
  out pLcdDR,rmp ; auf Richtungsregister
Datenausgabeport
  ; LCD Init
  rcall LcdInit ; starten LCD
  ldi ZH,High(2*LcdTextOut) ; Z auf Text
  ldi ZL,Low(2*LcdTextOut)
  rcall LcdText ; Gib Text aus
  ldi rmp,0x0C ; Cursor und Blink aus
  rcall LcdC4Byte
  ; ADC starten
  ldi rmp,cMuxU ; U-Messung starten
  out ADMUX,rmp
  clr rAL ; Summe loeschen
  clr rAH
  ldi rCtr,64 ; 64 Messungen
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; Starte erste Messung
  ; Schlafmodus
  ldi rmp,1<<SE ; Sleep Mode Idle
  out MCUCR,rmp
  ; Interrupts
  sei ; setze Int-Flagge
  ;
; Hauptprogrammschleife
Schleife:
  sleep ; Schlafen
  nop ; Aufwachen
  sbrc rFlag,bRdyU ; ueberspringe wenn noch nicht
fertig
  rcall AusgabeU
  sbrc rFlag,bRdyI ; I-Messung fertig?
  rcall AusgabeI
  rjmp Schleife
;
; Routine AusgabeU
;   wird von der Flagge bRdyU ausgeloest
;
; Rechnet die Spannung am Spannungseingang
; aus und schreibt sie auf die LCD.
;
AusgabeU:
  cbr rFlag,1<<bRdyU ; loesche Flagge
  ; mit Konstante cMultU multiplizieren
  ldi ZH,High(cMultU) ; Konstante in Z
  ldi ZL,Low(cMultU)
  rcall Multiplikation ; rM1:rM2 * ZH:ZL
  ; Runden, Ergebnis in rE3:rE2
  ldi rmp,0x7F
  add rE0,rmp
  adc rE1,rmp
  ldi rmp,0
  adc rE2,rmp
  adc rE3,rmp
  ; Auf LCD ausgeben
  ldi ZH,1
  ldi ZL,4
  rcall LcdPos
  clt ; Fuehrende Nullen
  ldi ZH,High(2*DezimalTab)
  ldi ZL,Low(2*DezimalTab)
AusgabeU1:
  lpm rE0,Z+ ; lese Dezimalzahl
  lpm rE1,Z+
  tst rE0 ; LSB Null?
  breq AusgabeU5
  clr rmp
AusgabeU2:
    ; Dezimalzahl abziehen
  sub rE2,rE0
  sbc rE3,rE1
  brcs AusgabeU3 ; fertig
  inc rmp ; weiter abziehen
  rjmp AusgabeU2
AusgabeU3:
  add rE2,rE0 ; wieder addieren
  adc rE3,rE1
  tst rmp ; Null?
```

```
  brne AusgabeU4
  brts AusgabeU4
  ldi rmp,' '
  rcall LcdD4Byte
  set
  rjmp AusgabeU1
AusgabeU4:
  set
  subi rmp,-'0'
  rcall LcdD4Byte
  cpi ZL,LOW(2*DezimalTab10)
  brne AusgabeU1
  ldi rmp,',' ; Kommastelle aus
  rcall LcdD4Byte
  rjmp AusgabeU1
AusgabeU5:
  ldi rmp,'0'
  add rmp,rE3
  rjmp LcdD4Byte
;
; Dezimaltabelle Tausender abwaerts
DezimalTab:
.dw 1000
.dw 100
DezimalTab10:
.dw 10
.dw 0
;
; Routine AusgabeI
;   wird von der Flagge bRdyI ausgeloest
;
; Berechnet den Strom und gibt ihn auf der
; LCD aus
;
AusgabeI:
  cbr rFlag,1<<bRdyI ; loesche Flagge
  ; mit Konstante cMultI multiplizieren
  ldi ZH,2
  ldi ZL,4
  rcall LcdPos
  ldi ZH,High(cMultI) ; Konstante in Z
  ldi ZL,Low(cMultI)
  rcall Multiplikation ; rM1:rM2 * ZH:ZL
  ; Runden, Ergebnis in rE3:rE2:rE1
  ldi rmp,0x7F
  add rE0,rmp
  adc rE1,rmp
  ldi rmp,0
  adc rE2,rmp
  adc rE3,rmp
  ; Auf LCD ausgeben
  clt ; Fuehrende Nullen
  ldi ZH,High(2*DezimalTab)
  ldi ZL,Low(2*DezimalTab)
AusgabeI1:
  lpm rM2,Z+ ; lese Dezimalzahl
  lpm rM3,Z+
  tst rM2 ; LSB Null?
  breq AusgabeI7
  clr rmp
AusgabeI2:
    ; Dezimalzahl abziehen
  sub rE2,rM2
  sbc rE3,rM3
  brcs AusgabeI3 ; fertig
  inc rmp ; weiter abziehen
  rjmp AusgabeI2
AusgabeI3:
  add rE2,rM2 ; wieder addieren
  adc rE3,rM3
  tst rmp ; Null?
  brne AusgabeI4
  brts AusgabeI5
  cpi ZL,Low(2*Dezimaltab10)
  breq AusgabeI4
  ldi rmp,' '
  rcall LcdD4Byte
  rjmp AusgabeI1
AusgabeI4:
  set
AusgabeI5:
  subi rmp,-'0'
  rcall LcdD4Byte
  rjmp AusgabeI1
AusgabeI7:
  ldi rmp,',' ; Kommastelle aus
  rcall LcdD4Byte
  ldi rmp,'0'
  add rmp,rE2
  rjmp LcdD4Byte
;
; 16- * 16-Bit Multiplikation
Multiplikation: ; rM1:rM2 * ZH:ZL
; Ergebnis in rE3:rE2:rE1:rE0
;
  clr rM2 ; Bytes 3 und 4 loeschen
  clr rM3
  clr rE0 ; Ergebnis loeschen
  clr rE1
  clr rE2
  clr rE3
Multiplikation1:
  lsr ZH ; Bit aus MSB in LSB
  ror ZL ; niedrigstes Bit in Carry
  brcc Multiplikation2
  add rE0,rM0 ; Multiplikator addieren
  adc rE1,rM1
  adc rE2,rM2
  adc rE3,rM3
Multiplikation2:
  mov rmp,ZL
  or rmp,ZH
  breq Multiplikation3
  lsl rM0 ; Multiplikator mal zwei
  rol rM1
  rol rM2
  rol rM3
  rjmp Multiplikation1 ; weiter multipliz.
Multiplikation3:
  ret
;
; --------- LCD-Routinen ---------------
LcdTextOut:
.db "Spannung+Strom tn24",0x0D
.db "U = xx,xx V",0x0D
.db "I = xxx,x mA",0xFE,0xFF
;        4
;
; LCD-Include-Routinen
.include "Lcd4Busy.inc"
;
; Ende Quellcode
;
```

14.2.4 Messbeispiel



Das ist ein solches Messbeispiel.

Top	Home	Spannung messen	Strommessen	Temperaturmessen

14.3 Temperaturen messen, umrechnen und darstellen

14.3.1 Hardware

Für die Messung der Temperatur sind keine externen Bauelemente erforderlich.

14.3.2 Messbereich, Mess-, Umrechnungs- und Darstellungsvorgang

14.3.2.1 Messbereich

ATMEL gibt für folgende Temperaturen typische ADC-Messergebnisse an:

t [°C]	ADC
-40	230
25	300
85	370

14.3.2.2 Messvorgang

Die Temperaturmessung wird durch Umschaltung des Multiplexers auf Kanal 8 eingeschaltet. Der entsprechende Umschaltbefehl ist im Handbuch angegeben und im Quelltext verwendet.

13.3.2.3 Umrechnung

Aus den tabellierten Messwerten ergibt sich für N_{mess} = 1,1194 * t [°C] + 273,88. Umgekehrt ergibt sich daraus für t [°C] = 0,89286 * N_{mess} – 244,52). Für 64 Messungen und multipliziert mit 65536 ergibt sich

t [°C] = 65536 / 64 * 0,89286 * $N_{Sum\text{-}mess}$ / 65536 - 245 = 914 * $N_{Sum\text{-}mess}$ / 65536 - 245.

Faktisch ist der Parameter 245 ungenau und muss justiert werden. Um diese Zahl praktisch zu bestimmen, ist im Programm die Ausgabe der gemessenen Temperatur in hex auf dem Display vorgesehen. Diese ergab z.B. bei 21°C 0x013A, also 35,8. Die Temperatur war also um 15°C zu hoch und der Parameter 245 ist folglich um 15 zu erhöhen.

Wer es noch genauer haben möchte, muss auch die Steigung durch Messung bei zwei verschiedenen Temperaturen neu bestimmen und den Parameter cMultT im Programm entsprechend anpassen.

Wer die Temperatur auf 0,1 °C genau ausgeben will, muss den Multiplikationsfaktor und den abzuziehenden Betrag ändern. So kommt z.B. eine Multiplikation der Messwertsumme mit 9.143 bei einer Subtraktion von 2.445 infrage, um Zehntelgrad zu erhalten (unter Division des Multiplikationsergebnisses mit 65.536). Da die physikalische Auflösung der Messung aber nur bei 0,89 °C liegt, sind die dann angezeigten Zehntelgrad-Werte arg ungenaue Schätzungen. Jedes moderne Hightech-Handy ist da genauer, sofern es nicht gerade eben erst aus der Hosentasche gezogen wurde und eine mehr oder weniger gelungene Mischung aus Körper- und

Umgebungstemperatur anzeigt.

14.3.2.4 Darstellung

Die dargestellte Temperatur ist ein ganzzahliger Wert, genauer ist die Messung nicht. Es können positive und negative Temperaturen vorkommen.

14.3.3 Programm

Das Programm ist hier (zum Quellcode im asm-Format geht es hier, es wird noch die LCD-Include-Routine benötigt.

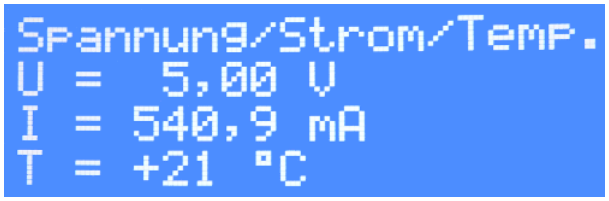
```
;
; ***************************************
; * Spannung, Strom und Temperatur messen *
; * (C)2016 by www.gsc-elektronic.net     *
; ***************************************
;
.NOLIST
.INCLUDE "tn24def.inc"
.LIST
;
; -------- Programmablauf --------------
;
; Misst nacheinander die Spannung
; a) am ADC3-Eingang (Widerstandsteiler),
; b) am Differenzeingang ADC2 und ADC0 mit
;    interner Verstaerkung,
; c) am internen Temperatursensor
; fuer je 64 Male, summiert sie auf, rechnet
; sie in Spannung (a), Strom (b) und Tempe-
; ratur um und gibt sie auf der LCD aus.
;
; -------- Schalter --------------------
.equ debugDisplay = 0 ; zeigt Temperatur
;                       messung in hex an
;
; -------- Hardware --------------------
;   ATtiny24      _______
;             1 /       |14
;       VCC o--|VCC  GND|--o GND
;             2|        |13
;    LCD-RS o--|PB0     |--o NC
;             3|        |12
;    LCD-RW o--|PB1 ADC1|--o GND
;             4|        |11
;   VCC-10k o--|RES ADC2|--o 0,1 Ohm
;             5|        |10  1 M  - Ein
;     LCD-E o--|PB2 ADC3|--o 56k2 - GND
;             6|        |9
;    LCD-D7 o--|PA7  PA4|--o LCD-D4
;             7|        |8
;    LCD-D6 o--|PA6  PA5|--o LCD-D5
;              |________|
;
; -------- Timing ----------------------
; Taktrate          1.000.000 Hz
; ADC-Vorteiler           128
; ADC-Takt              7.812,5 Hz
; ADC-Takte pro Messg.     14,5
; Messfrequenz            538,8 Hz
; Messzyklus, 64 Messungen  8,42 Hz
; Drei Messzyklen, 192 M.   2,81 Hz
;
; -------- Konstanten ------------------
; Spannungsmessung
.equ cRE =   1000000 ; Ohm
.equ cRG  =     56200 ; Ohm
.equ cMultU = ((cRE+cRG)*110+cRG/2)/cRG
; Strommessung
.equ cRI = 100 ; Milliohm
.equ cMultI = (550000+cRI/2) / cRI
; Temperaturmessung
.equ cMultT = 914
.equ cMinusT = 245+15 ; Temperaturkorrektur
; ADC-MUX-Konstanten
.equ cMuxU = (1<<REFS1)|(1<<MUX1)|(1<<MUX0)
.equ cMuxI = (1<<REFS1)|0b101101
.equ cMuxT = (1<<REFS1)|0b100010
;
; -------- Register --------------------
; frei: R0 .. R4
.def rAL= R5
.def rAH = R6
.def rM0 = R7
.def rM1 = R8
.def rM2 = R9
.def rM3 = R10
.def rE0 = R11
.def rE1 = R12
.def rE2 = R13
.def rE3 = R14
.def rSreg = R15 ; SREG-Sicherung
.def rmp = R16 ; Vielzweckregister
.def rimp = R17 ; Interrupt Vielzweck
.def rFlag = R18 ; Flaggenregister
  .equ bRdyU = 0 ; Ready-Flagge Spannung
  .equ bRdyI = 1 ; Ready-Flagge Strom
  .equ bRdyT = 2 ; Ready-Flagge Temperatur
.def rCtr = R19 ; Zaehler Messungen
.def rLese = R20 ; LCD
.def rLine = R21 ; LCD
.def rmo = R22 ; LCD
; frei: R23 .. R29
; verwendet: Z = R31:R30 fuer LCD
;
; -------- Reset- und Interrupt-Vektoren
  rjmp Start ; RESET-Vektor
  reti ; INT0 External Int Request 0
  reti ; PCINT0 Pin Change Int Request 0
  reti ; PCINT1 Pin Change Int Request 1
  reti ; WDT Watchdog Time-out
  reti ; TIM1_CAPT TC1 Capture Event
  reti ; TIM1_COMPA TC1 Compare Match A
  reti ; TIM1_COMPB TC1 Compare Match B
  reti ; TIM1_OVF TC1 Overflow
  reti ; TIM0_COMPA TC0 Compare Match A
  reti ; TIM0_COMPB TC0 Compare Match B
  reti ; TIM0_OVF TC0 Overflow
  reti ; ANA_COMP Analog Comparator
  rjmp AdcIsr ; ADC ADC Conv Complete
  reti ; EE_RDY EEPROM Ready
  reti ; USI_STR USI START
  reti ; USI_OVF USI Overflow
;
; -------- Interrupt Service Routinen --
;
; ADC Ready Interrupt
;   wird bei jeder abgeschlossenen Wandlung
;   ausgeloest
;
; Liest das Messergebnis und summiert es auf.
; Zaehlt den Zaehler rCtr abwaerts. Wenn er
; Null erreicht, wird die Summe in einen
```

```
; Zwischenspeicher kopiert und der naechste
; Messkanal in die MUX geschrieben. Je nach-
; dem welcher Kanal zuletzt gemessen wurde,
; werden die Flaggen bRdyU, bRdyI und bRdyT
; gesetzt.
; In allen Faellen wird die naechste Wandlung
; gestartet.
;
AdcIsr:
  in rSreg,SREG ; Status retten
  in rimp,ADCL ; Addiere Ergebnis
  add rAL,rimp
  in rimp,ADCH
  adc rAH,rimp ; mit Uebertrag
  dec rCtr ; Zaehler Messwerte
  brne AdcIsrRet ; nicht Null
  in rimp,ADMUX ; Mux lesen
  cpi rimp,cMuxU ; Wurde U gemessen?
  breq AdcIsrI ; ja, starte I-Messung
  cpi rimp,cMuxI ; Wurde I gemessen?
  breq AdcIsrT ; ja, starte T-Messung
  sbr rFlag,1<<bRdyT ; starte U-Messung
  ldi rimp,cMuxU
  out ADMUX,rimp
  rjmp AdcIsrCont
AdcIsrI:
  sbr rFlag,1<<bRdyU
  ldi rimp,cMuxI ; starte I-Messung
  out ADMUX,rimp
  rjmp AdcIsrCont
AdcIsrT:
  sbr rFlag,1<<bRdyI
  ldi rimp,cMuxT ; starte T-Messung
  out ADMUX,rimp
AdcIsrCont:
  ldi rCtr,64 ; Neustart Zaehler
  mov rM0,rAL ; Transfer Messwertsumme
  mov rM1,rAH
  clr rAL ; Messwertsumme loeschen
  clr rAH
AdcIsrRet:
  ldi rimp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rimp ; Starte naechste Messung
  out SREG,rSreg
  reti
;
; --------- Reset-Vektor, Programmstart
Start:
  ; Stapel anlegen
  ldi rmp,LOW(RAMEND) ; Ende SRAM
  out SPL,rmp ; in Stapelzeiger
  ; LCD-Port-Ausgaenge initiieren
  ldi rmp,(1<<bLcdCRE)|(1<<bLcdCRRS)|(1<<
bLcdCRRW)
  out pLcdCR,rmp ; Kontrollport-Ausgaenge
  clr rmp ; Ausgaenge aus
  out pLcdCO,rmp ; an Kontrollport
  ldi rmp,mLcdDRW ; Datenport-Ausgabe, Schreiben
  out pLcdDR,rmp ; auf Richtungsregister
Datenausgabeport
  ; LCD Init
  rcall LcdInit ; starten LCD
  ldi ZH,High(2*LcdDefChar)
  ldi ZL,Low(2*LcdDefChar)
  rcall LcdChars
  ldi ZH,High(2*LcdTextOut) ; Z auf Text
  ldi ZL,Low(2*LcdTextOut)
  rcall LcdText ; Gib Text aus
  ldi rmp,0x0C ; Cursor und Blink aus
  rcall LcdC4Byte
  ; ADC starten
  ldi rmp,cMuxU ; U-Messung starten
  out ADMUX,rmp
  clr rAL ; Summe loeschen
  clr rAH
  ldi rCtr,64 ; 64 Messungen
  ldi rmp,(1<<ADEN)|(1<<ADSC)|(1<<ADIE)|
(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0)
  out ADCSRA,rmp ; Starte erste Messung
  ; Schlafmodus
  ldi rmp,1<<SE ; Sleep Mode Idle
  out MCUCR,rmp
  ; Interrupts
  sei ; setze Int-Flagge
  ;
; Hauptprogrammschleife
Schleife:
  sleep ; Schlafen
  nop ; Aufwachen
  sbrc rFlag,bRdyU ; ueberspringe nicht fertig
  rcall AusgabeU
  sbrc rFlag,bRdyI ; I-Messung fertig?
  rcall AusgabeI
  sbrc rFlag,bRdyT ; T-Messung fertig?
  rcall AusgabeT
  rjmp Schleife
;
; Routine AusgabeU
;   wird von der Flagge bRdyU ausgeloest
;
; Rechnet das Messergebnis in eine Spannung um
; und gibt sie auf der LCD aus.
;
AusgabeU:
  cbr rFlag,1<<bRdyU ; loesche Flagge
  ; mit Konstante cMultU multiplizieren
  ldi ZH,High(cMultU) ; Konstante in Z
  ldi ZL,Low(cMultU)
  rcall Multiplikation ; rM1:rM2 * ZH:ZL
  ; Runden, Ergebnis in rE3:rE2
  ldi rmp,0x7F
  add rE0,rmp
  adc rE1,rmp
  ldi rmp,0
  adc rE2,rmp
  adc rE3,rmp
  ; Auf LCD ausgeben
  ldi ZH,1
  ldi ZL,4
  rcall LcdPos
  clt ; Fuehrende Nullen
  ldi ZH,High(2*DezimalTab)
  ldi ZL,Low(2*DezimalTab)
AusgabeU1:
  lpm rE0,Z+ ; lese Dezimalzahl
  lpm rE1,Z+
  tst rE0 ; LSB Null?
  breq AusgabeU5
  clr rmp
AusgabeU2:
  ; Dezimalzahl abziehen
  sub rE2,rE0
  sbc rE3,rE1
  brcs AusgabeU3 ; fertig
  inc rmp ; weiter abziehen
  rjmp AusgabeU2
AusgabeU3:
  add rE2,rE0 ; wieder addieren
  adc rE3,rE1
  tst rmp ; Null?
  brne AusgabeU4
  brts AusgabeU4
  ldi rmp,' '
  rcall LcdD4Byte
  set
  rjmp AusgabeU1
AusgabeU4:
  set
  subi rmp,-'0'
  rcall LcdD4Byte
  cpi ZL,LOW(2*DezimalTab10)
  brne AusgabeU1
  ldi rmp,',' ; Kommastelle aus
```

```
  rcall LcdD4Byte
  rjmp AusgabeU1
AusgabeU5:
  ldi rmp,'0'
  add rmp,rE3
  rjmp LcdD4Byte
;
; Dezimaltabelle Tausender abwaerts
DezimalTab:
.dw 1000
.dw 100
DezimalTab10:
.dw 10
.dw 0
;
; Routine AusgabeI
;   ausgeloest von der Flagge bRdyI
;
; Ergebnis in einen Strom umwandeln und auf
; der LCD ausgeben
;
AusgabeI:
  cbr rFlag,1<<bRdyI ; loesche Flagge
  ; mit Konstante cMultI multiplizieren
  ldi ZH,2
  ldi ZL,4
  rcall LcdPos
  ldi ZH,High(cMultI) ; Konstante in Z
  ldi ZL,Low(cMultI)
  rcall Multiplikation ; rM1:rM2 * ZH:ZL
  ; Runden, Ergebnis in rE3:rE2:rE1
  ldi rmp,0x7F
  add rE0,rmp
  adc rE1,rmp
  ldi rmp,0
  adc rE2,rmp
  adc rE3,rmp
  ; Auf LCD ausgeben
  clt ; Fuehrende Nullen
  ldi ZH,High(2*DezimalTab)
  ldi ZL,Low(2*DezimalTab)
AusgabeI1:
  lpm rM2,Z+ ; lese Dezimalzahl
  lpm rM3,Z+
  tst rM2 ; LSB Null?
  breq AusgabeI7
  clr rmp
AusgabeI2:
  ; Dezimalzahl abziehen
  sub rE2,rM2
  sbc rE3,rM3
  brcs AusgabeI3 ; fertig
  inc rmp ; weiter abziehen
  rjmp AusgabeI2
AusgabeI3:
  add rE2,rM2 ; wieder addieren
  adc rE3,rM3
  tst rmp ; Null?
  brne AusgabeI4
  brts AusgabeI5
  cpi ZL,Low(2*Dezimaltab10)
  breq AusgabeI4
  ldi rmp,' '
  rcall LcdD4Byte
  rjmp AusgabeI1
AusgabeI4:
  set
AusgabeI5:
  subi rmp,-'0'
  rcall LcdD4Byte
  rjmp AusgabeI1
AusgabeI7:
  ldi rmp,',' ; Kommastelle aus
  rcall LcdD4Byte
  ldi rmp,'0'
  add rmp,rE2
  rjmp LcdD4Byte
;
; Routine AusgabeT
;   wird von der Flagge bRdyT ausgeloest
;
; Rechnet das Ergebnis in eine Temperatur
; um und gibt sie auf der LCD aus
;
AusgabeT:
  cbr rFlag,1<<bRdyT ; loesche Flagge
.if debugDisplay == 1
  lsr rM1 ; / 2
  ror rM0
  lsr rM1 ; / 4
  ror rM0
  lsr rM1 ; / 8
  ror rM0
  lsr rM1 ; / 16
  ror rM0
  lsr rM1 ; / 32
  ror rM0
  lsr rM1 ; / 64
  ror rM0
  ldi ZH,3
  ldi ZL,11
  rcall LcdPos
  mov rmp,rM1
  rcall LcdHex
  mov rmp,rM0
  rjmp LcdHex
LcdHex:
  push rmp
  swap rmp
  rcall LcdHexN
  pop rmp
LcdHexN:
  andi rmp,0x0F
  subi rmp,-'0'
  cpi rmp,'9'+1
  brcs LcdHexN1
  subi rmp,-7
LcdHexN1:
  rjmp LcdD4Byte
  .endif
  ; mit Konstante cMult multiplizieren
  ldi ZH,High(cMultT) ; Konstante in Z
  ldi ZL,Low(cMultT)
  rcall Multiplikation ; rM1:rM0 * ZH:ZL
  ; Runden, Ergebnis in rE3:rE2
  ldi rmp,0x7F
  add rE0,rmp
  adc rE1,rmp
  ldi rmp,0
  adc rE2,rmp
  adc rE3,rmp
  ; Konstante abziehen
  ldi rmp,Low(cMinusT)
  sub rE2,rmp
  ldi rmp,High(cMinusT)
  sbc rE3,rmp
  ; Auf LCD ausgeben
  ldi ZH,3
  ldi ZL,4
  rcall LcdPos
  ldi rmp,'+'
  tst rE2
  brpl AusgabeT1
  ldi rmp,'-'
  neg rE2
AusgabeT1:
  rcall LcdD4Byte
  ldi ZL,10
  clr rmp
AusgabeT2:
  sub rE2,ZL
  brcs AusgabeT3
  inc rmp
  rjmp AusgabeT2
```

```
AusgabeT3:
  add rE2,ZL
  tst rmp
  brne AusgabeT4
  ldi rmp,' '
  rjmp AusgabeT5
AusgabeT4:
  subi rmp,-'0'
AusgabeT5:
  rcall LcdD4Byte
  ldi rmp,'0'
  add rmp,rE2
  rjmp LcdD4Byte
;
; 16- * 16-Bit Multiplikation
Multiplikation: ; rM1:rM2 * ZH:ZL
; Ergebnis in rE3:rE2:rE1:rE0
;
  clr rM2 ; Bytes 3 und 4 loeschen
  clr rM3
  clr rE0 ; Ergebnis loeschen
  clr rE1
  clr rE2
  clr rE3
Multiplikation1:
  lsr ZH ; Bit aus MSB in LSB
  ror ZL ; niedrigstes Bit in Carry
  brcc Multiplikation2
  add rE0,rM0 ; Multiplikator addieren
  adc rE1,rM1
  adc rE2,rM2
  adc rE3,rM3
Multiplikation2:
  mov rmp,ZL
  or rmp,ZH
  breq Multiplikation3
  lsl rM0 ; Multiplikator mal zwei
  rol rM1
  rol rM2
  rol rM3
  rjmp Multiplikation1 ; weiter multipliz.
Multiplikation3:
  ret
;
; --------- LCD-Routinen ---------------
LcdTextOut:
.db "Spannung/Strom/Temp.",0x0D,0xFF
.db "U = xx,xx V",0x0D
.db "I = xxx,x mA",0x0D,0xFF
.db "T = +xx ",0x00,"C",0xFE,0xFF
;        4
; Gradzeichen
LcdDefChar:
.db 64,0,14,10,14,0,0,0,0,0 ; Z = 0, Grad
.db 0,0 ; Ende der Tabelle

; LCD-Include-Routinen
.include "Lcd4Busy.inc"
;
; Ende Quellcode
;
```

14.3.4 Messbeispiel



Das ist ein Beispiel für eine Temperaturmessung.

Top	Home	Spannung messen	Strommessen	Temperaturmessen

Fazit aus den hier dargestellten Lektionen

Prozessorhardware

Die AVR bieten eine Menge eingebauter Hardware zur Nutzung an. Die kann mit ein paar wenigen Konfigurationsbits in den richtigen Ports mühelos zur Mitarbeit bewegt werden. Das Einschalten ist mit ein paar SBI- oder CBI-Instruktionen auf die richtigen Portbits einfach zu bewerkstelligen. Wer einmal kapiert hat, wie der Timer funktioniert und welche Bits sein Verhalten wie beeinflussen, braucht sich nicht mehr mit den Eigenheiten einer rudimentären Hochsprache wie Basic herumschlagen. Die ganze Welt des Timers steht dann offen und nicht nur das, was der Basic-Compiler gerade so anzubieten geruhen will. Man kann erstaunlich viel mit so einem Timer anfangen, wenn man sich nur traut, wie die entsprechenden Lektionen demonstriert haben. Und die paar SBI und CBI sind nicht gerade ganz große Programmierkunst oder undurchschaubares Hexenwerk.

Nur ein paar wenige Hardware-Features sind in den 14 Lektionen bislang nicht vorgekommen, was aber zu verschmerzen sein wird. Zum Einen ist das assynchrone (z. B. UART) und synchrone (z. B. I2C) Übertragungshardware. Wer zu Hause was Funktionierendes basteln will, braucht diesen Quatsch aber meistens auch gar nicht, es sein denn er möchte unbedingt eine ganze Farm von Prozessoren miteinander verkabeln (wie das Autohersteller oder voll ausgeflippte Informatiker gerne tun) oder man möchte Messwerte gerne dem PC mitteilen. Braucht also kaum ein Mensch, der eine maßgeschneiderte Lösung für sein Hardwareproblem haben will.

Selbige Klientel verwenden auch sehr gerne ein weiteres Hardware-Feature, das hier aber voll zu kurz kommt: den Bootmodus. Wer viele Tausend AVR immer mit den neuesten Ergüssen seiner Programmierkünste versehen möchte, braucht das unbedingt. Heute dies, morgen das, und übermorgen wieder alles anders. Das macht die Unausgereiftheit des Erdachten zum Konzept, wie das heute bei der Herstellung von Software so Sitte und ganz üblich ist. An so was kann nur jemand Freude haben, der sich unbedingt mehr Probleme wünscht als er lösen kann, um so mehr Daseinsberechtigung zu erringen. Natürlich braucht man das unbedingt, wenn man mehrere Millionen Dieselautos von Betrugssoftware in den Ehrlich-Modus zurückstellen muss. Normale Menschen brauchen das aber nicht, weil sie nie auf die Idee kämen, Selbstbetrugssoftware zu schreiben.

Externe Hardware

Das ist ein Punkt, der dem Programmierer immer Kopfzerbrechen machen sollte. Eine fehlerhaft konzipierte externe Hardware ist immer von Übel. Sie macht naheliegende Softwarekonzepte zunichte und zwingt im ungünstigen Fall zu immer neuen Kompromissen in Bezug auf die Funktionssicherheit. Wer das Prellen von Schaltern und Tasten vergessen hat, wird mit seiner Software nicht sehr glücklich werden. Meine HighTech-ATmega-Weckuhr zwingt mich morgens dazu, den WeckAus-Taster drei bis vier Mal zu drücken, bis der Alarm endlich aus ist. Das Prellen hat der Taster erst nach zwei Jahren Betrieb angefangen, wahrscheinlich hat sich die Korrosion Zeit gelassen, daher fiel das noch nicht beim Design der Software auf.

Da das Design der externen Hardware immer auch Konsequenzen für die Typauswahl und das Programm haben, können sich geänderte externe Beschaltungen bis hin zu einer völligen Neukonzeption des Assemblerprogramms auswirken. Hier frühzeitig Grips zu investieren, ist reiner Selbstschutz.

Da zu behaupten, man sei halt mit Elefantendesign auf der sicheren Seite, ist reine Theorie. Wozu soll man in einem Modellflugzeug 96-polige Chips einsetzen, von denen man aber nur

ganze sechs Pins tatsächlich braucht (einen als Impulseingang, einen als PWM-Ausgang für die Motorleistung und je zwei für Richtung und Intensität für Höhen- und Seitenruder)? Es scheint, die Arduino-sozialisierten Programmierer hatten noch nie wirkliche Probleme zu lösen, deshalb konnten sie sich lange in ihrem selbsterrichteten Arduino-Elfenbeinturm einmauern und können sich eine Problemlösungswelt ohne USB-Schnittstelle schon mal gar nicht vorstellen. Sie kennen den Wert einer LED fürs Debugging überhaupt nicht, weil sie schon immer viel mächtigere Werkzeuge ihr Eigen nennen. Die aber halt bei einer Rudermaschine mit einem putzig kleinen ATtiny10 aber rein gar nix bringen, weil in den allenfalls Bruchteile ihrer Megabibliothek reinpassen und kein Platz mehr ist für echten Code.

Selbstbeschränkung bleibt Selbstbeschränkung, auch wenn sie sich mit modernen Schlagworten ein knallbuntes Mäntelchen überstreift.

Programmdesign

Besonders für Anfänger im Mikroprozessor-Programmiergeschäft dürfte dieser Punkt der abschreckendste sein. Die elendig langen Listings, die spätestens mit Lektion 6 hier Einzug hielten, dürften den Eindruck erwecken, als sei das alles viel zu kompliziert um es zu kapieren. Deshalb hier folgende Ratschläge als Lese-, Analyse- und Verständnishilfe:

1. Interruptgesteuerte Programme haben immer diesselbe Grundstruktur, denselben Aufbau. Finde heraus, was die einzelnen Interrupts tun und welche Flaggen diese unter welchen Bedingungen setzen. Das ist schon die halbe Miete beim Verständnis der Abläufe.
2. Die Pozessorhardware wird immer zuerst initialisiert. Welche interne Hardware wird hier für welche Zwecke wie eingeschaltet? In welchen Modi laufen Timer, ADCs und andere interne Hardware?
3. Versuche aus der Dokumentation auf Standardabläufe zu schließen. Eine Multiplikationsroutine sieht immer gleich aus, die 18 bis 20 Instruktionen dafür gehören zusammen, ein genaues Verständnis jeder einzelnen Codezeile ist dann überflüssig, wenn der Sinn und Zweck dieses Codepackens einmal klar ist.
4. Versuche die Ablaufalgorithmen aufzumalen und die entsprechenden Bedingungen zu klären.

Die Grundentscheidung, ob für die Lösung eines Problems bestehende und verfügbare Softwareschnipsel geeignet sind oder ob deren Anpassung an den geänderten Bedarf mehr Aufwand erfordert als das Neudesign dürfte für den Anfänger sehr schwer zu entscheiden sein. Mit zunehmender Erfahrung in Design und Programmierung dürfte das Neudesign zunehmend die Oberhand gewinnen.

Anhang 1: Bevorzugte Registerverwendungen

#	8 Bit, bevorzugte Verwendungen	Name	16 Bit Registerpaare	Namen
0	Lesen aus dem Flashspeicher (LPM)		Multiplikationsergebnis	R1:R0
1	Rechnen Speichern SRAM-Zugriffe			
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15	SREG-Zwischenspeicher	rSreg		
16	Vielzweckregister (Temp)	rmp		
17	Vielzweckregister ISR (iTemp)	rimp		
18	Flaggenregister	rFlag		
19	Zähler mit LDI/ORI/ANDI/SUBI/SBR			
20				
21				
22				
23				
24			16-Bit-Zähler	rCntH:rCntL
25				
26			Zeiger	X, XH:XL
27				
28			Zeiger mit Displacement	Y, YH:YL
29				
30			Lesen aus dem Flashspeicher	Z, ZH:ZL
31				

Anhang 2: Instruktionsliste AVR-Assembler

(Markierte Mnemonics verzweigen beim Anklicken zu einer Erläuterung, Kürzel siehe Abkürzungen)

Instruktionsliste								
Mnem.	**P1**	**P2**	**Beschreibung (englisch)**	**Aktion**	**Flaggen**	**Clk**	**Einschränkungen**	**Worte**
Arithmetische und logische Operationen								
ADD	Rx	Ry	Addiere (Add)	Rx ← Rx + Ry + C	Z,C,N,V,S,H	1		1
ADC	Rx	Ry	Addiere mit Überlauf (Add with Carry)	Rx ← Rx + Ry + C	Z,C,N,V,S,H	1		1
ADIW	RdL	K	Addiere Konstante zu Wort (Add Immediate Word)	RdH:RdL ← RdH:RdL + K	Z,C,N,V,S	2	RdL=24/26/28/30, K: 0 bis 63	1
SUB	Rx	Ry	Subtrahiere (Subtract)	Rx ← Rx - Ry	Z,C,N,V,S,H	1		1
SUBI	Rh	K	Subtrahiere Konstante (Subtract Immediate)	Rh ← Rh - K	Z,C,N,V,S,H	1	R: 16 bis 31	1
SBC	Rx	Ry	Subtrahiere Register mit Überlauf (Subtract with Carry)	Rx ← Rx - Ry - C	Z,C,N,V,S,H	1		1
SBCI	Rh	K	Subtrahiere Konstante mit Überlauf (Subtract with Carry Immediate)	Rh ← Rh - K - C	Z,C,N,V,S,H	1	R: 16 bis 31	1
CP	Rx	Ry	Vergleiche (Compare)	Rx - Ry	Z,C,N,V,S,H	1		1
CPC	Rx	Ry	Vergleiche mit Überlauf (Compare with Carry)	Rx - Ry - C	Z,C,N,V,S,H	1		1
CPI	Rh	K	Vergleiche mit Konstante (Compare Immediate)	Rx - K	Z,C,N,V,S,H	1	R: 16 bis 31, K: 0 bis 255	1
SBIW	RdL	K	Subtrahiere Konstante von Wort (Subtract Immediate Word)	RdH:RdL ← RdH:RdL - K	Z,C,N,V,S	2	RdL=24/26/28/30, K: 0 bis 63	1
AND	Rx	Ry	Binäres UND (And)	Rx ← Rx UND Ry	Z,N,V,S	1		1
ANDI	Rh	K	Binäres UND mit Konstante (And Immediate)	Rh ← Rh UND K	Z,N,V,S	1	R: 16 bis 31, K: 0 bis 255	1
OR	Rx	Ry	Binäres ODER (Or)	Rx ← Rx ODER Ry	Z,N,V,S	1		1
ORI	Rh	K	Binäres ODER mit Konstante (Or Immediate)	Rh ← Rh ODER K	Z,N,V,S	1	R: 16 bis 31, K: 0 bis 255	1
EOR	Rx	Ry	Exklusiv-ODER (Exclusive Or)	Rx ← Rx EXOR Ry	Z,N,V,S	1		1
COM	Rx		Einerkomplement (Complement)	Rx ← 255 - Rx	Z,C,N,V,S	1		1
NEG	Rx		Zweierkomplement (Two's complement)	Rx ← 256 - Rx	Z,C,N,V,S,H	1		1
SBR	Rh	K	Setze Bits in Konstante K (Set bits in register)	Rh ← Rh ODER K	Z,N,V,S	1	R: 16 bis 31, K: 0 bis 255	1
CBR	Rh	K	Lösche Bits in Konstante K (Clear bits in register)	Rh ← Rh UND (NEG K)	Z,N,V,S	1	R: 16 bis 31, K: 0 bis 255	1
INC	Rx		Erhöhe um Eins (Increase)	Rx ← Rx + 1	Z,N,V,S	1		1
DEC	Rx		Vermindere um Eins (Decrease)	Rx ← Rx - 1	Z,N,V,S	1		1
TST	Rx		Prüfe auf Null (Test zero)	Rx ← Rx ODER Rx	Z,N,V,S	1		1
CLR	Rx		Lösche alle Bits (Clear)	Rx ← 0	Z,N,V,S	1		1
SER	Rh		Setze alle Bits (Set register)	Rh ← 255	-	1	R: 16 bis 31	1
MUL	Rx	Ry	Multipliziere 8 Bits (Multiply)	R1:R0 ← Rx * Ry	Z,C	2		1
MULS	Rx	Ry	Multipliziere mit Vorzeichen (Multiply signed)	R1:R0 ← Rx * Ry	Z,C	2	Rx,Ry: 16 bis 31	1
MULSU	Rx	Ry	Multipliziere ohne und mit Vorzeichen (Multiply signed/unsigned)	R1:R0 ← Rx * Ry	Z,C	2	Rx,Ry: 16 bis 31	1
FMUL	Rx	Ry	Fließkomma-Multiplikation (Float multiply)	R1:R0 ← Rx * Ry	Z,C	2	Rx,Ry: 16 bis 23	1
FMULS	Rx	Ry	Fließkomma-Multiplikation mit Vorz. (Float multiply signed)	R1:R0 ← Rx * Ry	Z,C	2	Rx,Ry: 16 bis 23	1
FMULSU	Rx	Ry	Fließkomma-Multiplikation ohne und mit Vorzeichen (Float multiply signed/unsigned)	R1:R0 ← Rx * Ry	Z,C	2	Rx,Ry: 16 bis 23	1
DES	K		Datenver- und Entschlüsselung (Data encryption standard)	(R7:R0, R15:R8)	-	1/2	(nur MEGA/XMEGA), K<16	1
Sprunginstruktionen								
RJMP	K		Relativer Sprung (Relative jump)	(PC) ← (PC) +/- K	-	2	K: -2048 bis 2047	1
IJMP			Indirekter Sprung (Indirect jump)	(PC) ← Z	-	2		1

Instruktionsliste								
Mnem.	**P1**	**P2**	**Beschreibung (englisch)**	**Aktion**	**Flaggen**	**Clk**	**Einschränkungen**	**Worte**
EIJMP			Erweiterter Indirekter Sprung (Extended indirect jump)	(PC) ← EIND + Z	-	2	(nur XMEGA)	1
JMP	K		Weitsprung (Wide jump)	(PC) ← K	-	3	K: 0 bis 65535	2
RCALL	K		Relativer Aufruf (Relative call)	(Stack) ← (PC), (PC) ← (PC) +/- K	-	2/3/4	K: -2048 bis 2047	1
ICALL			Indirekter Aufruf (Indirect call)	(Stack) ← (PC), (PC) ← Z	-	2/3/4		1
EICALL			Erweiterter Indirekter Aufruf (Extended indirect call)	(Stack) ← (PC), (PC) ← EIND+Z	-	3/4		1
CALL	K		Weitaufruf (Wide call)	(Stack) ← (PC), (PC) ← K	-	3/4/5		2
RET			Rückkehr vom Aufruf (Return)	(PC) ← (Stack)	-	4		1
RETI			Rückkehr Interrupt Service (Return from interrupt)	(PC) ← (Stack), I ← 1	-	4		1
CPSE	Rx	Ry	Überspringe bei Gleich (Compare and skip if equal)	Rx=Ry: (PC) ← (PC + 1)	-	2/3		1
SBRC	Rx	B	Überspringe bei Registerbit Null (Skip if bit in register is clear)	(Bit)=0: (PC) ← (PC+1)	-	2/3		1
SBRS	Rx	B	Überspringe bei Registerbit Eins (Skip if bit in register is set)	(Bit)=1: (PC) ← (PC+1)	-	2/3		1
SBIC	PL	B	Überspringe bei Portbit Null (Skip if I/O bit is clear)	(Bit)=0: (PC) ← (PC+1)	-	2/3	PL: 0 bis 31	1
SBIS	PL	B	Überspringe bei Portbit Eins (Skip if I/O bit is set)	(Bit)=1: (PC) ← (PC+1)	-	2/3	PL: 0 bis 31	1
BRBS	K	B	Springe relativ wenn Bit im SREG Eins (Branch if bit in status register is set)	(SREG-Bit=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis +64	1
BRBC	K	B	Springe relativ wenn Bit im SREG Null (Branch if bit in status register is clear)	(SREG-Bit=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BREQ	K		Springe relativ wenn Z-Bit im SREG Eins (Branch if eqal)	(SREG-Z=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRNE	K		Springe relativ wenn Z-Bit im SREG Null (Branch if not equal)	(SREG-Z=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRCS	K		Springe relativ wenn C-Bit im SREG Eins (Branch if carry set)	(SREG-C=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRCC	K		Springe relativ wenn C-Bit im SREG Null (Branch if carry clear)	(SREG-C=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRSH	K		Springe relativ wenn C-Bit im SREG Null (Branch if same or higher)	(SREG-C=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRLO	K		Springe relativ wenn C-Bit im SREG Eins (Branch if lower)	(SREG-C=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRMI	K		Springe relativ wenn N-Bit im SREG Eins (Branch if minus)	(SREG-N=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRPL	K		Springe relativ wenn N-Bit im SREG Null (Branch if plus)	(SREG-N=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRGE	K		Springe relativ wenn S-Bit im SREG Null (Branch if greater or equal)	(SREG-S=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRLT	K		Springe relativ wenn S-Bit im SREG Eins (Branch if less than)	(SREG-S=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRHS	K		Springe relativ wenn H-Bit im SREG Eins (Branch if half carry set)	(SREG-H=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRHC	K		Springe relativ wenn H-Bit im SREG Null (Branch if half carry clear)	(SREG-H=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRTS	K		Springe relativ wenn T-Bit im SREG Eins (Branch if T flag set)	(SREG-T=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRTC	K		Springe relativ wenn T-Bit im SREG Null (Branch if T flag clear)	(SREG-T=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRVS	K		Springe relativ wenn V-Bit im SREG Eins (Branch if overflow set)	(SREG-V=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRVC	K		Springe relativ wenn V-Bit im SREG Null (Branch if overflow clear)	(SREG-V=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
BRIE	K		Springe relativ wenn I-Bit im SREG Eins (Branch if I flag enabled)	(SREG-I=1): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1

Instruktionsliste								
Mnem.	**P1**	**P2**	**Beschreibung (englisch)**	**Aktion**	**Flaggen**	**Clk**	**Einschränkungen**	**Worte**
BRID	K		Springe relativ wenn I-Bit im SREG Null (Branch if I flag disabled)	(SREG-I=0): (PC) ← (PC) +/- K	-	1/2	K: -63 bis + 64	1
Datenkopier-Instruktionen								
MOV	Rx	Ry	Register Kopieren (Move register)	Rx ← Ry	-	1		1
MOVW	Rx	Ry	Registerpaar kopieren (Move word)	Rx+1:Rx ← Ry+1:Ry	-	1	Rx, Ry: gerade	1
LDI	Rh	K	Konstante in Register (Load immediate)	Rh ← K	-	1	R: 16 bis 31, K: 0 bis 255	1
LDS	Rh	A	SRAM in Register (Load direct from data space)	Rx ← (SRAM-A)	-	2/3/4	R: 16 bis 31	2
LD	Rx	X	SRAM Adresse X in Register (Load indirect from address X)	Rx ← (X)	-	2/3/4		1
LD	Rx	X+	SRAM Adresse X in Register mit Inkrement (Load indirerct from address X and increment)	Rx ← (X), X = X + 1	-	2/3		1
LD	Rx	-X	Dekrement X und SRAM X in Register (Decrement and load from address X)	X = X - 1, Rx ← (X)	-	2/3/4		1
LD	Rx	Y	SRAM Adresse Y in Register (Load indirect from address Y)	Rx ← (Y)	-	2/3/4		1
LD	Rx	Y+	SRAM Adresse Y in Register mit Inkrement (Load indirect from address Y and increment)	Rx ← (Y), Y = Y + 1	-	2/3		1
LD	Rx	-Y	Dekrement Y und SRAM Y in Register (Decrement and load from address Y)	Y = Y - 1, Rx ← (Y)	-	2/3/4		1
LDD	Rx	Y+K	SRAM Adresse (Y+K) in Register (Load indirect from address Y with displacement)	Rx ← (Y+K)	-	2/3	K: 0 bis 63	1
LD	Rx	Z	SRAM Adresse Z in Register (Load indirect from address Z)	Rx ← (Z)	-	2/3/4		1
LD	Rx	Z+	SRAM Adresse Z in Register mit Inkrement (Load indirect from address Z and increment)	Rx ← (Z), Z = Z + 1	-	2/3		1
LD	Rx	-Z	Dekrement Z und SRAM Z in Register (Decrement and load indirect from address Z)	Z = Z - 1, Rx ← (Z)	-	2/3/4		1
LDD	Rx	Z+K	SRAM Adresse Z+K in Register (Load indirect from address Z with displcement)	Rx ← (Z+K)	-	2/3	K: 0 bis 63	1
STS	A	Rh	Register in SRAM (Store direct in data space)	(SRAM-A) ← Rx	-	2/3/4	R: 16 bis 31	2
ST	X	Rx	Register in SRAM Adresse X (Store indircet at address X)	(X) ← Rx	-	2/3/4		1
ST	X+	Rx	Register in SRAM Adresse X mit Inkrement (Store indirect at address X and increment)	(X) ← Rx, X = X + 1	-	2/3		1
ST	-X	Rx	Dekrement X und Register in SRAM X (Decrement and store indirect at address X)	X = X - 1, (X) ← Rx	-	2/3/4		1
ST	Y	Rx	Register in SRAM Adresse Y (Store indirect at address Y)	(Y) ← Rx	-	2/3/4		1
ST	Rx	Y+	Register in SRAM Adresse Y mit Inkrement (Store indirect at address Y and increment)	(Y) ← Rx, Y = Y + 1	-	2/3		1
ST	Rx	-Y	Dekrement Y und Register in SRAM Y (Decrement and store indirect at address Y	Y = Y - 1, (Y) ← Rx	-	2/3/4		1
STD	Y+K	Rx	Register in SRAM Adresse Y+K (Store indirect at address Y with displacement)	(Y+K) ← Rx	-	2/3	K: 0 bis 63	1
ST	Z	Rx	Register in SRAM Adresse Z (Store indirect at address Z)	(Z) ← Rx	-	2/3/4		1
ST	Z+	Rx	Register in SRAM Adresse Z mit Inkrement (Store indirect at address Z and increment)	(Z) ← Rx, Z = Z + 1	-	2/3		1
ST	-Z	Rx	Dekrement Z und Register in SRAM	Z = Z - 1, (Z) ← Rx	-	2/3/		1

Instruktionsliste								
Mnem.	**P1**	**P2**	**Beschreibung (englisch)**	**Aktion**	**Flaggen**	**Clk**	**Einschränkungen**	**Worte**
			Z (Decrement and store indirect at address Z)			4		
STD	Z+K	Rx	Register in SRAM Adresse Z+K (Store indirect at address Z with displacement)	(Z+K) ← Rx	-	2/3	K: 0 bis 63	1
LPM			Flashspeicher Adresse (Z) in Register R0 (Load from program memory)	R0 ← (Flash Z)	-	3		1
LPM	Rx	Z	Flashspeicher Adresse (Z) in Register (Load register from program memory)	Rx ← (Flash Z)	-	3		1
LPM	Rx	Z+	Flashspeicher Adresse (Z) in Register, Inkrement (Load register from program memory and increment)	Rx ← (Flash Z), Z = Z + 1	-	3		1
ELPM			Erweiterte Flash-Adresse (Z) in Register R0 (Extended load from program memory)	R0 ← (Flash Z)	-	3		1
ELPM	Rx	Z	Erweiterte Flash-Adresse (Z) in Register (Extended load register from program memory)	Rx ← (Flash Z)	-	3		1
ELPM	Rx	Z+	Erw. Flash-Adresse (Z) in Register, Inkrement (Extended load register from program memory and increment)	Rx ← (Flash Z), Z = Z + 1	-	3		1
SPM			Wort R1:R0 in Flashspeicher Z (Store in program memory)	(Flash Z) ← R1:R0	-	N		1
SPM	Z+		Wort R1:R0 in Flashspeicher (Z), Inkrement (Store in program memory at address Z and increment)	(Flash Z) ← R1:R0, Z = Z+ 1	-	N		1
IN	Rx	P	Port in Register (Load register from port)	Rx ← P	-	1	P: 0 bis 63	1
OUT	P	Rx	Register in Port (Store register in port)	P ← Rx	-	1	P: 0 bis 63	1
PUSH	Rx		Register auf Stapel (Push register to stack)	(Stack) ← Rx, SP = SP - 1	-	2		1
POP	Rx		Stapel in Register (Pop register from stack)	Rx ← (Stack), SP = SP + 1	-	2		1
XCH	Z	Rx	Tausche Inhalt Register und SRAM Z (Exchange register and data storage at Z)	Rx ↔ (Z)	-	1		1
LAS	Z	Rx	ODER Register mit SRAM (Z) und tausche (Load and set)	Rx ← Rx ODER (Z), (Z) ↔ Rx	-	1		1
LAC	Z	Rx	UND Komplement Register mit SRAM (Z) nach SRAM (Z) (Load and clear)	Rx ← (255-Rx) UND (Z), (Z) ↔ Rx	-	1		1
LAT	Z,	Rd	XOR Register mit SRAM (Z) und tausche (Load and toggle)	Rx EXOR (Z), Rx ↔ (Z)	-	1		1
Bit-Operationen								
LSL	Rx		Logisches Linksschieben (Logical shift left)	Rx ← Rx * 2	Z,C,N,V,H	1		1
LSR	Rx		Logisches Rechtsschieben (Logical shift right)	Rx ← Rx / 2	Z,C,N,V	1		1
ROL	Rx		Binäres Linksrotieren über C (Rotate left)	Rx ← Rx * 2 mit Bit 0 = C/C = Bit 7	Z,C,N,V,H	1		1
ROR	Rx		Binäres Rechtsrotieren über C (Rotate right)	Rx ← Rx / 2 mit Bit 7 = C/C = Bit 0	Z,C,N,V	1		1
ASR	Rx		Arithmetisches Rechtsschieben (Arithmetical shift right)	Rx ← Rx(6:0) / 2, Bit 6 = 0	Z,C,N,V	1		1
SWAP	Rx		Vertausche oberes und unteres Nibble (Swap nibbles)	Rx ← (7:4) ↔ (3:0)	-	1		1
BSET	B		Setze Bit im SREG (Set bit in status register)	SREG ← SREG ODER (1<<B)	-	1	B: 0 bis 7	1
BCLR	B		Lösche Bit im SREG (Clear bit in register)	SREG ← SREG UND (255-(1<<B))	-	1	B: 0 bis 7	1
SBI	PL	B	Setze Bit im Port (Set bit I/O)	PL ← PL ODER (1<<B)	-	2	PL: 0 bis 31	1

Instruktionsliste								
Mnem.	**P1**	**P2**	**Beschreibung (englisch)**	**Aktion**	**Flaggen**	**Clk**	**Einschränkungen**	**Worte**
CBI	PL	B	Lösche Bit im Port (Clear bit I/O)	PL ← PL UND (255-(1<<B))	-	2	PL: 0 bis 31	1
BST	Rx	B	Registerbit in T (Bit store in T flag)	SREG-T ← Rx-Bit B	T	1	B: 0 bis 7	1
BLD	Rx	B	T in Registerbit (Bit load from T flag)	Rx-Bit B ← T	-	1	B: 0 bis 7	1
SEC			SREG C setzen (Set flag C in status register)	SREG-Bit C ← 1	C	1		1
CLC			SREG C löschen (Clear flag C in status register)	SREG-Bit C ← 0	C	1		1
SEN			SREG N setzen (Set flag N in status register)	SREG-Bit N ← 1	N	1		1
CLN			SREG N löschen (Clear flag N in status register)	SREG-Bit N ← 0	N	1		1
SEZ			SREG Z setzen (Set flag Z in status register)	SREG-Bit Z ← 1	Z	1		1
CLZ			SREG Z löschen (Clear flag Z in status register)	SREG-Bit Z ← 0	Z	1		1
SEI			SREG I setzen (Set flag I in status register)	SREG-Bit I ← 1	I	1		1
CLI			SREG I löschen (Clear flag I in status register)	SREG-Bit I ← 0	I	1		1
SES			SREG S setzen (Set flag S in status register)	SREG-Bit S ← 1	S	1		1
CLS			SREG S löschen (Clear flag S in status register)	SREG-Bit S ← 0	S	1		1
SEV			SREG V setzen (Set flag V in status register)	SREG-Bit V ← 1	V	1		1
CLV			SREG V löschen (Clear flag V in status register)	SREG-Bit V ← 0	V	1		1
SET			SREG T setzen (Set flag T in status register)	SREG-Bit T ← 1	T	1		1
CLT			SREG T löschen (Clear flag T in status register)	SREG-Bit T ← 0	T	1		1
SEH			SREG H setzen (Set flag H in status register)	SREG-Bit H ← 1	H	1		1
CLH			SREG H löschen (Clear flag H in status register)	SREG-Bit H ← 0	H	1		1
Controller-Instruktionen								
BREAK			Ausführung stoppen, Kontrolle an Debugger (Break execution)		-	1		1
NOP			Nichts tun (No operation)		-	1		1
SLEEP			Schlafen (Sleep)		-	1		1
WDR			Watchdog rücksetzen (Watch dog reset)	WDR-Zähler ← 0	-	1		1

Abkürzungen:

A	Adresse 0 .. 65.535
B	Bit 0 .. 7
K	Konstante (Wertebereiche siehe Spalte Beschränkungen)
P	Portregister, 0 .. 63
PL	Niedriges Portregister, 0 .. 31
Rh	Oberes Register R16 .. R31
Rx	Allgemeines oder Zielregister (Parameter 1) R0 .. R31
Ry	Quellregister R0 .. R31
Rdl, Rdh	Doppelregister, LSB und MSB, R25:R24, R27:R26 (X), R29:28 (Y), R31:R30 (Z)

Anhang 3: Direktiven und Ausdrücke in AVR-Assembler

Direktive	Parameter	Beschreibung
Listing		
.LIST		Schaltet die Ausgabe im Listing ein
.NOLIST		Schaltet die Ausgabe im Listing aus
Quelltext-Herkunft		
.INCLUDE	„(Dateiname)"	Assembliert den Code in der Zieldatei, Dateiname mit relativem oder absolutem Pfad
.INCLUDE	„...def.inc"	Einfügen der Headerdatei mit den typspezifischen Konstantendefinitionen des AVR-Typs
Ziel-Segmente		
.CSEG		Assembliert in das Code-Segment, Adresszähler Program Counter PC
.ESEG		Assembliert in das EEPROM-Segment, nur Label und .ORG-/.DB-/.DW- Direktiven sind zulässig, eigener EEPROM-Adresszähler
.DSEG		Assembliert in das SRAM-Segment, nur Label und .ORG-/.BYTE-/.WORD- Direktiven sind zulässig, eigener SRAM-Adresszähler
Adressenmanipulation		
.ORG	Adresse	Setzt den im Segment verwendeten Adresszähler vorwärts auf die angegebene Adresse
.BYTE	N	Reserviert N Bytes und erhöht den Adresszähler um N
.WORD	N	Reserviert N Worte und erhöht den Adresszähler um 2*N Bytes
Tabellen anlegen		
.DB	b1,b2,..bn "(Text)"	Fügt die Bytes b1 bis bn bzw. die ASCII-Werte des Textes in das Code- oder EEPROM-Segment ein, im Flashspeicher nur wortweise (gerade Anzahl)
.DW	w1,w2,..wn	Fügt die 2-Byte-Werte w1 bis wn in das Code-oder EEPROM-Segment ein
Symbole, Namen		
.DEF	Name = Rn	Ordnet dem Register Rn den Namen zu
.EQU	Name = Wert	Setzt eine Konstante mit dem Namen auf den angegebenen Wert, keine nachträgliche Änderung zulässig
.SET	Name = Wert	Setzt eine Konstante mit dem Namen auf den angegebenen Wert, nachträgliche Änderung(en) zulässig
.UNDEF	Name	Beendet die Namenszuordnung
Makros		
.MACRO	Name, Parameter	Beginnt ein Makro mit dem Namen und den Parametern
.ENDMACRO .ENDM		Beendet das Makro
Typdefinition		
.DEVICE	"Typname"	Schaltet die Instruktionsprüfung für den angegebenen AVR-Typ ein (in def.inc enthalten)
Meldungen, Fehler		
.MESSAGE	"TEXT"	Gibt den angegebenen Text aus
.ERROR	"TEXT"	Provoziert einen Fehler und gibt die angegebene Fehlermeldung aus
Bedingte Assemblierung		
.EXIT		Beendet die Assemblierung, der Rest des Quellcodes wird ignoriert
.IF	Bedingung	Assembliert wenn die Bedingung erfüllt ist
.IFDEF	Symbol	Assembliert wenn das Symbol definiert ist
.IFNDEF	Symbol	Assembliert wenn das Symbol nicht definiert ist
.ELSE		Assembliert wenn bei .IF die Bedingung in .IF nicht erfüllt, wenn bei .IFDEF das Symbol nicht definiert ist oder bei .IFNDEF das Symbol definiert ist
.ELIF	Bedingung	Assembliert, wenn die Bedingung in .IF nicht eingehalten und die Bedingung in .ELIF eingehalten ist
.ENDIF		Beendet .IF, .IFDEF, .ELSE und .ELIF
gavrasm-Besonderheiten		
.DB	%YEAR%, %MONTH%, %DAY%, %HOUR%, %MINUTE%, %SOURCE%	Fügt aktuelle Datumsinformationen und den Dateinamen ein
.IFDEVICE	"Devicename"	Bedingte Assemblierung wenn es sich um den angegebenen AVR-Typ handelt

Wichtig! Auf den Namen der Direktive muss ein Leerzeichen oder Tabulator folgen, sonst wird diese nicht erkannt!

Liste der Ausdrücke in AVR-Assembler

Zahlenarten

An Zahlenarten sind zulässig:

Zahlenart	Kennzeichen	Beispiel
Dezimalzahl	(kein)	.equ Zahl = 123
Binärzahl	0b	.equ Zahl = 0b1111011
Hexadezimalzahl	0x	.equ Zahl = 0x7B

Groß- und Kleinschreibung werden in Assembler nicht unterschieden. Das spielt nur bei Tabellen mit ASCII-Zeichen eine Rolle, z. B. .db "abcd" ist nicht identisch mit .db "ABCD".

Ausdrücke

Ausdrücke erlauben es in Assembler, Zahlen zu berechnen oder logische Entscheidungen zu fällen.

Wichtig ist zu beachten, dass nur der Assembler selbst diese Berechnungen ausführt und nur dann das das Ergebnis im ausführbaren Code landet, wenn es mit LDI-Instruktionen oder als Tabelle mit .db- oder .dw-Direktiven in den Code übernommen wird.

In Assembler werden alle Operationen mit Ganzzahlen ausgeführt. Das heißt, alle verwendeten Zahlen sind ganzzahlig und alle Operationen führen nur zu ganzzahligen Ergebnissen. Beim Dividieren wird daher das Ergebnis grundsätzlich abgerundet. Soll das Ergebnis aufgerundet werden, kann dies folgendermaßen erfolgen:

```
.equ N1 = 2345
.equ N2 = 56
.equ Erg1 = N1 / N2 ; Ergebnis ist 41
.equ Erg = (N1 + N2 / 2) / N2 ; Ergebnis ist 42
```

In Ausdrücken können Leer- und Tabulatorzeichen enthalten sein, diese werden beim Auswerten ignoriert. Das Einfügen solcher Trennzeichen ist aber nicht erforderlich. Es ist unzulässig, in zusammengehörigen Symbolen Trennzeichen einzufügen (z. B. bei ! = oder < =).

Ausdrucks-typ	Symbol Ausdruck	Beschreibung	Beispiel
Logisch	!	Nicht, Umkehr des Wahrheitswerts	.equ f2 = ! f1 ; ergibt wahr (1) wenn f1 falsch (0)
	&&	Verknüpft linken und rechten Ausdruck mit logischem UND	.equ f = 1 && 0 ; ergibt falsch (0)
	||	Verknüpft linken und rechten Ausdruck mit logischem ODER	.equ f = 0 || 1 ; ergibt wahr (1)
Vergleich	<	Vergleich ob linker Ausdruck kleiner als rechter Ausdruck	.equ f = 234 < 567 ; ergibt falsch (0)
	<=	Vergleich ob linker Ausdruck kleiner oder gleich rechter Ausdruck	.equ f = 234 /lt;= 234 ; ergibt wahr (1)
	>	Vergleich ob linker Ausdruck größer als rechter Ausdruck	.equ f = 234 > 567 ; ergibt falsch (0)
	>=	Vergleich ob linker Ausdruck größer oder gleich rechter Ausdruck	.equ f = 234 >= 234 ; ergibt wahr (1)
	==	Vergleich ob linker Ausdruck gleich rechter Ausdruck	.equ f = 234 == 234 ; ergibt wahr (1)
	!=	Vergleich ob linker Ausdruck ungleich rechter Ausdruck	.equ f = 234 != 234 ; ergibt falsch (0)
Binär	~	Bitweise Nicht	.equ 0b1010 ~ 0011 ; ergibt 0b1001
	&	Bitweise UND	.equ f = 0b1010 & 0b0010 ; ergibt 0b0010
	|	Bitweises ODER	.equ f = 0b1010 | 0b0011 ; ergibt 0b1011
	^	Bitweises Exklusiv-Oder	.equ f = 0b1010 | 0b0011 ; ergibt 0b1001
	<<	Linksschieben (Multiplikation mit 2^n)	.equ f = 1 << 7 ; ergibt 0b1000000
	>>	Rechtsschieben (Division durch 2^n)	.equ f = 0b11000000 >> 3 ; ergibt 0b00110000
Arithmetik	+	Plus, Addieren	.equ f = 123 + 24 ; ergibt 147

	-	Subtrahieren, Minus	.equ f = 123 - 24 ; ergibt 99
	*	Malnehmen, Multiplizieren	.equ f = 123 * 24 ; ergibt 2952
	/	Teilen, Dividieren	.equ f = 123 / 24 ; ergibt 5!
	%	Modulo, Divisionsrest	.equ f = 123 % 24 ; ergibt 3

Funktionen

Die folgenden Funktionen können für Berechnungen verwendet werden.

Funktion	liefert	Beispiel
LOW	die niedrigsten 8 Bits	.equ f = Low(0x1234) ; ergibt 0x34
HIGH	die oberen 8 Bits	.equ f = High(0x1234) ; ergibt 0x12
BYTE2	die oberen 8 Bits	.equ f = Byte2(0x1234) ; ergibt 0x12
BYTE3	die Bits 16 bis 23	.equ f = Byte3(0x123456) ; ergibt 0x12
BYTE4	die Bits 24 bis 31	.equ f = Byte4(0x12345678) ; ergibt 0x12
LWRD	die untersten 16 Bits	.equ f = Lwrd(0x12345678) ; ergibt 0x5678
HWRD	die oberen 16 Bits	.equ f = Hwrd(0x12345678) ; ergibt 0x1234
EXP2	zwei hoch Zahl	.equ f = Exp2(16) - 1 ; ergibt 65535
LOG2	Zweierlogarithmus der Zahl	.equ f = Log2(65535) ; ergibt 15 (!)

Prioritätsregeln

Je höher der Ausdruck in der obigen Tabelle steht, desto prioritärer wird er bei der Berechnung ausgewertet. Die Bearbeitung der Formeln erfolgt bei gleicher Priorität von links nach rechts. Mit Klammern kann die Prioritätsregel außer Kraft gesetzt werden (Klammerausdrücke werden immer zuerst ausgewertet).

Anhang 4: Einführung in Binär- und Hexadezimalzahlen

In diesem Kurs wird oft mit binären und mit hexadezimalen Zahlen umgegangen. Diese Kurzdarstellung zeigt einige Grundlagen auf, deren Kenntnis dabei hilfreich sein könnte.

Binärzahlen

Bei Controllern gibt es nur Nullen und Einsen. Damit wird alles gemacht. Nicht nur reine Zahlen werden damit dargestellt, sondern einzelne Bits in Registern, I/O-Ports oder Portregistern können nur Null oder Eins sein. Irgendwas dazwischen gibt es bei Controllern sehr selten (z. B. bei Analog-Digitalwandlern wird aus einer analogen Spannung eine digitale Zahl angefertigt).

Bei Zahlensystemen basieren diese immer auf einer Grundgröße, der Basis. Im Zehnersystem ist es die zehn, im Binärsystem halt die zwei. Entsprechend gibt es im Dezimalsystem 10 verschiedene Ziffern (von 0 bis 9), im Binärsystem nur zwei (0 und 1).

Wird eine Zahl größer als mit nur einer Ziffer darstellbar ist, wird nach links hin angebaut. Zwölf sind daher eine links angebaute 1, aber mit zehn malgenommen, plus zwei. Allgemein kriegt man so jede beliebigt große Zahl in Griff. Fünfstellige Zahlen im Dezimalsystem gehen dann so:

$$12345 = 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0$$

wobei jede Ziffer (ausser der Null, aber die kommt hier nicht vor) hoch 0 gleich Eins ist.

Bei Binärzahlen ist das ganz ähnlich. Hier ist dezimal 12345 in binär

$$11.0000.0011.1001 = 1 * 2^{13} + 1 * 2^{12} + 0 * 2^{11} + 0 * 2^{10} + 0 * 2^9 + 0 * 2^8 + 0 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

Das ist noch einfacher zu rechnen als Dezimal: alle Bits mit Null davor zählen nix, alle Bits mit Eins soviel wie die Zweierpotenz. Damit vereinfacht sich die Zahl zu

$$1 * 2^{13} + 1 * 2^{12} + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^0$$

Die dezimalen Werte der Zweierpotenzen sind 1, 2, 4, 8, 16, 32, 64, 128, usw. und die müssen wir bei Einsen zusammenzählen um wieder dezimal 12345 herauszukriegen.

Es ist klar, dass große Dezimalzahlen, hier eine fünfstellige, in binär noch mehr Ziffern haben (hier: 14).

Aber Obacht: kein Mensch muss jetzt mit 40-stelligen Binärzahlen (das ist die größte, die in diesem Kurs vorkommt) umgehen und sich den Mund mit Nullen und Einsen fusselig reden, weil es auch viel einfacher geht (hexadezimal) und weil es dafür Werkzeug gibt, ohne die Zweierpotenz von 40 ausrechnen zu müssen.

Das täglich Brot von Assembler-Programmierern hört bei 8 Binärziffern schon auf, mehr passt sowieso nicht in ein AVR-Register. Daher sollte man die Zweierpotenzen bis 128 schon im Kopf haben, mehr aber braucht kein Mensch (der keine 64-Bit-CPU programmieren muss).

Im Gegenteil haben es die ganz kleinen Binärzahlen bei Controllern oft viel dicker hinter den Ohren. So reicht ein einzelnes Bit in einem Portregister dazu aus, um eine LED ein- oder auszuknipsen. Drei Bits (CS02, CS01, CS00) reichen bei einem Timer dazu aus, um aus acht verschiedenen Taktquellen des Timers auszuwählen. Und so weiter und so fort. Auch das sind gewichtige Zahlen, obwohl sie so klein sind.

Um dem Assembler mitzuteilen, dass man mit 111 keine Dezimalzahl meint sondern eine binäre, schreibt man "0b" davor, also 0b111 sind von hinten nach vorne, dezimal 1 + 2 + 4 = 7. Ein gewaltiger Unterschied zu dezimal 111.

Um dem Assembler beizubringen, dass man das linkeste Bit eines Registers (mit acht Bits) bit-

te schön gesetzt haben möchte, schreibt man

```
ldi R16,1<<7
```

Das *ldi* meint "load immediate" oder "lade Konstante in Register". Die zwei Pfeile nach links sagen, er möge eine 1 sieben Mal nach links schieben (und von rechts immer Nullen dazu schieben), und schon haben wir dezimal 128 schön unleserlich in das Register gezaubert. Und damit gleich gelernt, dass das Malnehmen mit 2 bei Assemblers einfach mit einmal Linksschieben der Bits geht.

Und um es ganz verrückt zu machen, könnten wir den Assembler dazu veranlassen die Zahl 12345 nicht nur mit ".equ Zahl = 12345", sondern auch mit

```
.equ Zahl = (1<<13) | (1<<12) | (1<<5) | (1<<4) | (1<<3) | (1<<0)
```

mit dem Namen "Zahl" in einen internen Symbolspeicher zu packen und den Wert 12345 dazu zu schreiben. Auf dass wir später im Quellcode darauf zugreifen können. Das | steht dabei für ein binäres ODER und setzt das 12345 aus den jeweils einzelnen gesetzten Einsen zusammen. Wer jetzt im Quellcode nach der Zeichenfolge "12345" sucht, der wird nicht mehr fündig werden.

Hexadezimalzahlen

Um die Zahlenhuberei bei Binärzahlen erheblich zu vereinfachen, werden jeweils vier Binärzahlen zu einer Ziffer zusammengefasst und als Hexadezimalzahl bezeichnet.

Das Umwandeln der Binärzahlen 0b0000 bis 0b1001 ist dabei noch einfach: die Dezimalziffern 0 bis 9 reichen dafür noch aus. Ab 0b1010 wird es aber schwieriger, die Ziffern 0b1010 bis 0b1111 gibt es im Dezimalsystem nicht. Für diese Ziffern wird im Hexadezimalsystem A bis F verwendet. Dezimal 12345 übersetzt sich zu hexadezimal 0x3039, denn

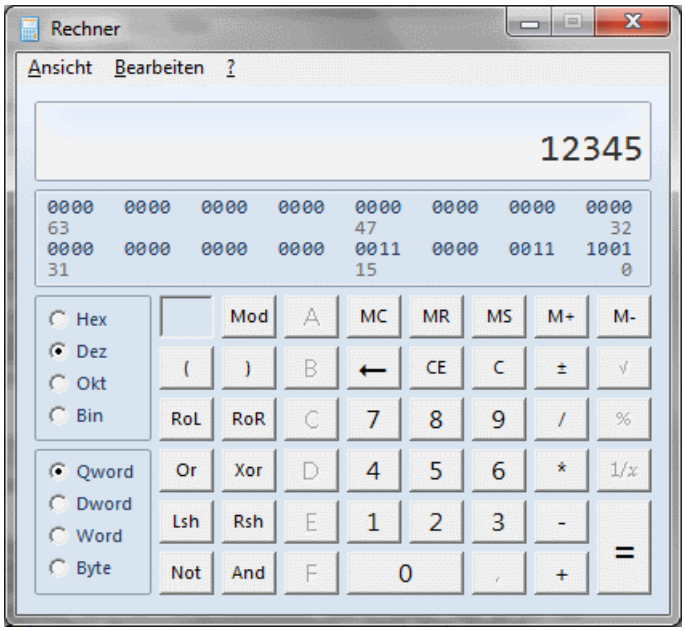
$$3 * 16^3 + 0 * 16^2 + 3 * 16^1 + 9 * 16^0$$

ist, mit den 16-er-Potenzen 1, 16, 256 und 4096 dasselbe wie 3*4096+3*16+9*1 = 12345. Aus den 14 Binärziffern macht das Hexadezimalsystem gerade mal vier Ziffern, eine gewaltige Vereinfachung.

Werkzeuge

Für die Umrechnerei von und in das Binär- und Hexadezimalsystem braucht man nicht unbedingt Kopfrechnen. Das machen so nette kleine Helferlein wie der beim Windows im Zubehör enthaltene Rechner calc.exe. Da gibt es im Menue unter "Ansicht" den Programmierermodus. Und der kennt Binär- und Hexadezimalzahlen.

Durch Klicken auf "Hex" oder "Bin" wird die Dezimalzahl flugs umgewandelt. Das Tool kann hexadezimal locker 64 Bits (was z. B. Taschenrechner, auch spezielle, nicht schaffen).



Anhang 5: Bauteileliste

Bauteile

Preisstand: 28.12.2017

Gesamtsumme ohne Allgemein =		36,39 €		Gesamtsumme =	101,09 €
#	Bauteil	Lektion	Anzahl	Reichelt-Name	Preis
Allgemein					
1	Multimeter	alle	1	PEAKTECH 1075	16,95
2	Breadboard	alle	1	STECKBOARD 2K4V	11,70
3	Kabel für Breadboard	alle	1	STECKBOARD DBS	4,60
4	Programmiergerät Olimex ISP	alle	1	DIAMEX USB ISP	21,50
5	Akkupack 4*AAA	alle	1	NH TC 4XAAA-1Z	7,75
6	15 cm Litze schwarz/weiß	alle	1	ZL214SWW-5	2,15
7	Stiftleiste einreihig 2-polig	alle	1	MPE 087-1-002	0,05
Lektion 1: ISP-Interface					
8	Stiftleiste einreihig 3-polig	alle	2	MPE 087-1-003	0,14
9	Stiftleiste zweireihig 3-polig	alle	1	MPE 087-2-006	0,14
10	Lochrasterplatine 10*10 mm	alle	1	H25PR050	0,97
11	IC-Fassung 8 polig DIL	alle	1	GS 8P	0,19
12	ATtiny13A	1 .. 9, 12	1	ATTINY 13A-PU	0,99
13	Widerstand 10k	alle	1	1/4W 10K	0,10
14	Keramikkondensator 100nF	alle	1	Z5U-5 100N	0,05
Lektionen 2: LED einschalten bis 5: LED-Pwm					
15	Widerstand 220 Ohm	2 .. 9, 12	1	1/4W 220	0,10
16	LED rot 5mm	2 .. 9, 12	1	LED 5MM RT	0,06
Lektion 6: LED Interrupt					
17	Widerstand 220 Ohm	6	1	1/4W 220	0,10
18	LED rot 5mm	6	1	LED 5MM RT	0,06
Lektion 7: Taster Interrupt					
19	Duoled rot/grün 5mm	7	1	LED 5 RG	0,18
20	Eingabetaster rot	7	1	DT 6 RT	0,79
Lektion 8: LED-Helligkeitsregler					
21	Poti 100k lin	8	1	PO4M-LIN 100K	1,99
Lektion 9: Tongenerator					
22	Elko 47µF/16V	9	1	RAD 47/16	0,03
23	Lautsprecher 0,2W/45 Ohm	9	1	BL 50A	1,20
Lektion 10: LCD-Anzeige					
24	ATtiny24-20PU	10 .. 14	1	ATTINY 24A-PU	0,85
25	IC-Fassung 14 polig DIL	10 .. 14	1	GS 14P	0,21
26	LCD 4x20 blau	10 .. 14	1	LCD 204B BL	22,80
27	Stiftleite 16-polig	10 .. 14	1	MPE 087-1-016	0,25
Lektion 11: EEPROM					
28	Eingabetaster weiß	11	1	DT 6 WS	0,79
Lektion 12: IR-Sender-Empfänger					
29	Infrarotdiode 5mm	12	1	SFH 4546	0,38
30	IR-Empfänger-Modul 40 kHz	12	1	TSOP 4840	0,66
31	IR-Empfänger-Modul 40 kHz	12	1	TSOP 31240	0,74
32	Widerstand 10k	12	1	1/4W 10K	0,10
33	Keramikkondensator 100nF	12	1	Z5U-5 100N	0,05
32	Folienkondensator 22 nF	12	1	MMK 22N 63	0,14
Lektion 13: Frequenz- und Induktivitätsmessgerät					
33	Widerstand 100k	13	4	1/4W 100K	0,41
34	Widerstand 220 Ohm	13	1	1/4W 220	0,10
35	Folienkondensatoren 100 nF	13	3	MMK 100N 63	0,36
36	Elko 1µF	13	1	SM 1,0/63RAD	0,03
37	CMOS-IC 4011	13	1	MOS 4011	0,25
38	IC-Fassung 14 polig DIL	13	1	GS 14P	0,21
Lektion 14: Volt- und Amperemeter					
39	Metallschichtwiderstand 1 M	14	1	METALL 1,00M	0,08
40	Metallschichtwiderstand 56,2K	14	1	METALL 56,2K	0,08
41	5W-Drahtwiderstand 0,1 Ohm	14	1	5W AXIAL 0,1	0,31
42	Sicherung 2A flink	14	1	FLINK 2,0A	0,30
43	Sicherungshalter	14	1	PL 112000	0,18

Anhang 6: Links zu den asm-Quellcode-Dateien

Lekt.	Quellcode-Link	Kurzbeschreibung	für Typ
2	2_Led_An.asm	Eine LED an einem Portausgang einschalten	ATtiny13
3	3_Led_Blink_Fast.asm	Eine LED ganz schnell blinken lassen	
	3_Led_Blink.asm	Eine LED mit Verzögerungsschleife im Sekundenrhythmus blinken lassen	
4	4_Timer_Blink.asm	Eine LED mit dem Timer blinken lassen	
	4_Led_Blink_timer.asm	Eine LED mit dem Timer im Sekundenrhythmus blinken lassen	
	4_Blink_128kHz.asm	Einen ATtiny13 per Fuse auf den 128kHz-Oszillator umstellen und LED im Sekundenrhythmus mit dem Timer blinken lassen	
5	5_fast_pwm.asm	Eine LED mit einem Pulsweiten-Signal PWM ansteuern	
6	6_tc0_o_int.asm	Einen Timer im Interrupt-Modus kontrollieren	
	6_tc0_int_compA.asm	Einen Timer mit Compare im Interruptmodus	
7	7_Taster_Int.asm	Einen Tastendruck mit Interrupt erkennen	
8	8_Helligkeitsregler_1.asm	LED mit Helligkeitsregelung über einen AD-Wandler	
	8_Helligkeitsregler_2.asm	Helligkeitsregelung mit Farbwechsel bei einer Duo-LED	
	8_Helligkeitsregler_3.asm	Farbmischung bei einer Duo-LED	
	8_Helligkeitsregler_4.asm	Grün/Rot-Anteil einer Duo-LED variieren	
9	9_tonerzeuger_1.asm	Mit einem Lautsprecher Töne erzeugen mit Tonhöhenregler	
	9_tonerzeuger_2.asm	Die Tonleiter mit einem Lautsprecher spielen	
	9_tonerzeuger_3.asm	Auf Tastendruck ein Musikstück abspielen	
10	10_Lcd-Display_1.asm	Eine LCD an den Prozessor anschließen, mittels Warteschleifen ansteuern und Texte ausgeben	ATtiny24
	10_Lcd-Display_2.asm	Die LCD durch Abfragen des Busy-Flags ansteuern	
	10_Lcd-Display_3.asm	Der LCD neue Zeichen beibringen und darstellen	
	Lcd_Zeichengenerator.ods	Zeichengenerator im OpenOffice-Format	Open Office
	Lcd_Zeicheng._Pfeile.ods	Zeichengenerator mit Pfeilen im OpenOffice-Format	
	Lcd_Zeichengenerator.xls	Zeichengenerator in Excel	Excel
	Lcd_Zeicheng._Pfeile.xls	Zeichengenerator mit Pfeilen in Excel	
11	11_Eeprom/Lcd4Busy.inc	Assembler-Include-Datei zur Ansteuerung der LCD im Busy-Modus	ATtiny24
	11_Eeprom_1.asm	Die Anzahl Einschaltvorgänge bei einem Chip zählen, im EEPROM speichern und dezimal ausgeben	
	11_Eeprom_2.asm	Die Anzahl Einschaltvorgänge mit erweitertem Zählbereich	
12	12_IR-Rx_1.asm	Die Dauer von Signalen einer Infrarot-Fernsteuerung messen und auf der LCD anzeigen	
	12_IR-Rx_1a.asm	Anzahl Kopf- und Datensignale einer IR-Fernsteuerung bestimmen und anzeigen	
	12_IR-Rx_1b.asm	Die Signaldauer von Datensignalen einer IR-Fernsteuerung messen und anzeigen	
	12_IR-Rx_1c.asm	Die Tastencodes einer IR-Fernsteuerung ermitteln und darstellen	
	12_IR-Tx.asm	Eigene Fernsteuersignale erzeugen und die Stellung eines angeschlossenen Potis senden	ATtiny13
	12_IR-Tx_Analog.asm	Die Potistellung messen und als IR-Signal senden	
	12_IR-Rx_Analog.asm	Die erzeugten Signale empfangen und die Potiaussteuerung in Prozent anzeigen	ATtiny24
	12_IR-Rx_Schalter.asm	Die IR-Fernsteuersignale im Lernmodus ausmessen und zur Ein-/Aus-Steuerung von LEDs und Relais verwenden	ATtiny13 ATtiny24
13	13_F-Meter_1.asm	Die Frequenz eines Digitalsignals messen und anzeigen	ATtiny24
	13_F-Meter_2.asm	Ein Analogsignal mit dem Analogvergleicher messen, Frequenz anzeigen	
	13_F-L-Meter.asm	Frequenzen und die Induktivität von Spulen messen und anzeigen	
14	14_U-Meter.asm	Spannungen an einem Eingang messen, umrechnen und anzeigen	
	14_U-I-Meter.asm	Ströme an einem Eingang messen, umrechnen und anzeigen	
	14_U-I-T-Meter.asm	Die Temperatur messen, umrechnen und anzeigen	

Anhang 7: Themenverzeichnis

Prozessorhardware

Externe Hardware

Programmierung

Quellcode