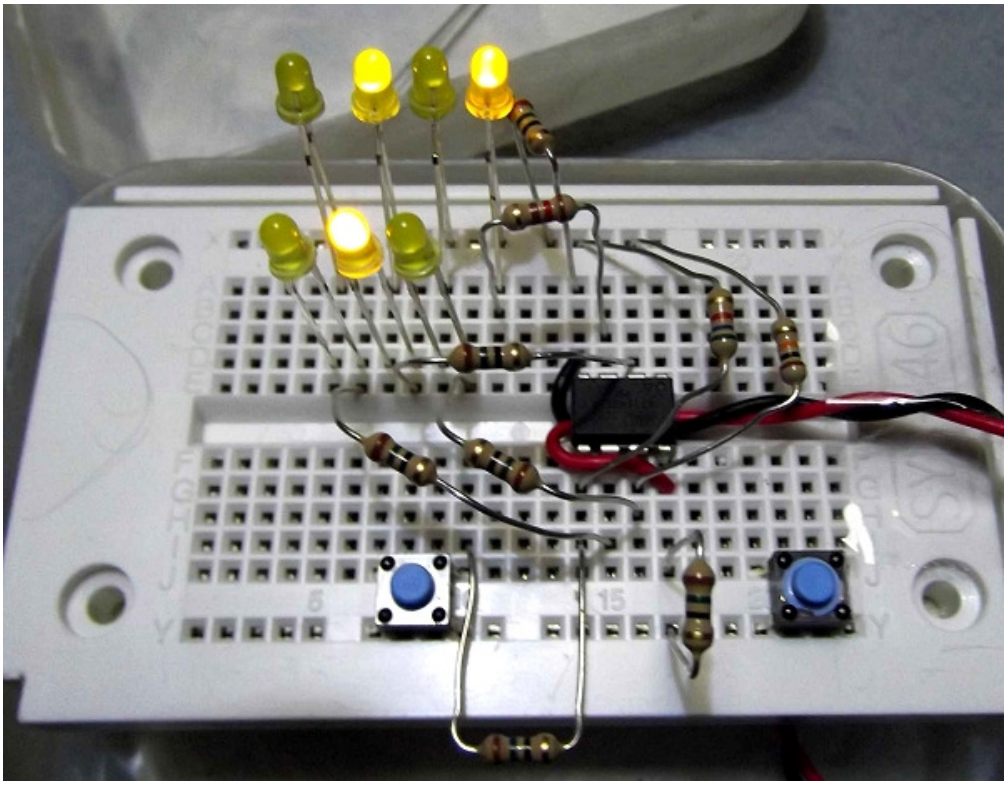
Urlaubskurs ATtiny13-Assembler

TPA (Tasten programmierbarer AVR)

von Heinz D.



Inhalt

Siehe auch: **Der Programmierwettbewerb**

Im Jahre 13 muss sich ein Beitrag mit dem Tiny13 beschäftigen. Das Programmieren des Tiny13 in Assembler ermöglicht geradezu winzige Programme. Dieser Beitrag spaltet die Anwender in zwei Lager. Die eine Hälfte wird nach drei Tagen aufgeben, weil der Zeit- und Lernaufwandt zu gross erscheint. Die andere Hälfte 'leckt Blut', nimmt die Herausforderung an und stürzt sich in die Arbeit. Das Denken in binären Zahlen ist am Anfang wirklich schwer. Bitte nehmen Sie sich täglich bis zu einer Stunde Zeit, um das Dargestellte zu begreifen. Holen Sie sich täglich ein Erfolgserlebnis (versuchen Sie nicht einen Tag auszulassen oder zu überfliegen). Auch wenn das Lernpensum enorm ist, sollen Sie den Spass nicht verlieren.

Hinweis: Für den Beitrag wurden nur wenige wichtige Befehle ausgewählt. Die Beschreibung der Hardware beschränkt sich nur auf das Minimum. Im Rahmen des Beitrages ist kein vollständiger Assemblerkurs möglich. Im Netz finden Sie von anderen Autoren zum Teil sehr gute Anfängerkurse zur Assemblerprogrammierug, die Sie als ergänzende Information lesen können, sie sind meist etwas anders dargestellt werden.

Falls Sie im Urlaub hiermit anfangen wollen, laden Sie sich die Datenblätter auf Ihr eBook und diesen Text natürlich, und legen die Bauteile in eine Seifendose.

24 Bauteile
Batterie (4,5V), Steckbrett, Tiny13, 2x Tasten, Draht, 5x 1k, 2x 10k, je 2x LED grün/gelb/rot, 1 Piezo, 1uF-10uF, 10k-Poti, D9-Buchse, Flachkabel

Sie können fast jeden AtTiny oder AtMega benutzen, obwohl hier nur auf den AtTiny13 eingegangen wird! Wenn Sie keinen Tiny13 benutzen, sollten Sie immer beide Datenblätter (13 + Ihren) im Blick behalten.
[www.atmel.com/images/doc0856] alle AVR-Befehle mit binärem Aufbau
[www.atmel.com/images/doc2535] AtTiny 13V (der ältere, oder)
[www.atmel.com/images/doc8126] AtTiny 13A (der jüngere, fast gleich (kompatibel) mit 13V)

Später benötigen Sie noch den Assembler und die IDE
[www.avr-asm-tutorial.net/gavrasm/index_de-html] [gavrasm_win_de_33.zip]
[www.avr-asm-tutorial.net/gavrasm/index_de-html#caller] [gavrasmW.zip]

Zum 'Brennen' des .HEX-File benötigen Sie noch (aus dem Lern Paket Mikrocontroller)
[www.b-kainka.de/LPmicrosUpdate.zip]

Hardware

Ein AVR besteht im wesentlichen aus

der Ablaufsteuerung mit der Takterzeugung, sie holt der Reihe nach die Befehle zur Ausführung,
dem Befehlsinterpreter, er steuert die Baugruppen zur Ausführung der Befehle,
dem Flash-Speicher, er enthält die Programm-Befehle (nicht flüchtig),
dem SRAM-Speicher, enthält (flüchtige) Ergebnisse während der Laufzeit,
dem EEPROM-Speicher, er erhält ggf. (nicht flüchtige) Daten,
der Arithmetisch/Logischen Einheit (ALU), hier erfolgen die mathematischen/logischen Berechnungen,
dem Couter/Timer, er erzeugen z.B. PWM-Signale,
dem AD-Wandler, um externe analoge Spannungen intern digital zu verarbeiten,
den Ports, von 'innen' wie Speicherzellen, nach 'außen' die Leitungen (für Tasten/LED usw.),

Die Register und die Adressierung hat Atmel etwas seltsam angelegt.

	Low Reg. n		High Reg. n		Low I/O-Reg.		I/O-Reg.		High I/O-Reg.		I/O-Reg.		SRAM		SRAM		SRAM		SRAM
0	R0	16	R16	32	=0x00	48	=0x10	64	=0x20	80	BODCR	96	=0x60	112	=0x70	128	=0x80	144	=0x90
1	R1	17	R17	33		49		65	WDTCR	81	OSCAL	97		113		129		145	
2	R2	18	R18	34		50		66		82	TCNT0	98		114		130		146	
3	R3	19	R19	35	ADCSRB	51		67	=0x23	83	TCCR0B	99		115		131		147	
4	R4	20	R20	36	ADCL	52	DIDR	68		84	MCUSR	100		116		132		148	
5	R5	21	R21	37	ADCH	53	PCMSK	69	PRR	85	MCUCR	101		117		133		149	
6	R6	22	R22	38	ADCSRA	54	PinB	70	CLKPR	86	OCR0A	102		118		134		150	
7	R7	23	R23	39	ADMUX	55	DDRB	71		87	SPMCSR	103		119		135		151	
8	R8	24	R24	40	ACSR	56	PortB	72	GTCCR	88	TIFR0	104		120		136		152	
9	R9	25	R25	41		57		73	OCR0B	89	TIMSK0	105		121		137		153	
10	R10	26	XL	42		58		74		90	GIFR	106		122		138		154	
11	R11	27	XH	43		59		75	=0x2B	91	GIMSK	107		123		139		155	
12	R12	28	YL	44		60	EECR	76		92	=0x3C	108		124		140		156	
13	R13	29	YH	45		61	EEDR	77		93	SPL	109		125		141		157	
14	R14	30	ZL	46		62	EEARL	78	DWDR	94	=0x3E	110		126		142		158	
15	R15	31	ZH	47	=0x0F	63	=0x1F	79	TCCR0A	95	SREG	111	=0x6F	127	=0x7F	143	=0x8F	159	RAMend

Von links nach rechtes:
- die (Low) Register haben spezielle Aufgaben,
- die ersten 8 (High) Register, R16-R23 sind universell und fast immer zu verwenden,
- die I/O-Register-Adressierung (Low) fängt wieder bei 0 (-32) an (alle Portbefehle),
- die High-I/O-Register nur von IN+OUT-Befehlen,
- die normalen SRAM-Zellen haben wieder ihre 'richtigen' Adressen.

1.Tag

Grundsätzliches: die AVR verstehen nur binäre Ziffern als Befehle und Daten. Die meisten Befehle (samt Daten) sind in je 16 Bit = 2 Byte untergebracht. Damit sich Menschen die 16 Bit merken können, werden die Befehle durch sog. Mnemonics beschrieben (z.B. 'rjmp k').

Die erste Regel, die ein Programmierer lernt: Ein Programm muss enden! Es darf nicht wahllos durch den Programmspeicher rasen und versuchen irgendwelche zufälligen (unsinnigen) Befehle auszuführen.

Einen direkten Befehl, um die Programmausführung anzuhalten gibt es zwar nicht, aber wir können einen Befehl benutzen, der immer wieder sich selbst (Endlosschleife) und keine weiteren Befehle ausführt.

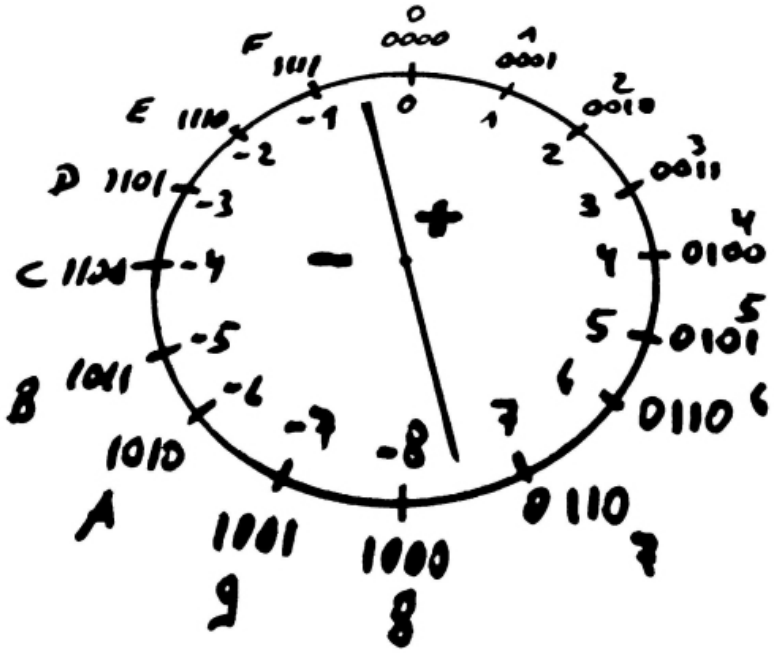
rjmp -1

In der Befehlstabelle (doc0856.pdf) finden Sie:
Relativer Sprung +-2047 (4096 signed); rjmp k; code 1100 kkkk kkkk kkkk

Nun müssen wir noch herausfinden, wie der AVR eine negative Zahl erkennt. Es gibt kein '-' -Zeichen! Eine negative Zahl muss mit binären Ziffern darstellbar sein.

Stellen Sie sich das Zifferblatt einer Uhr vor. Darum legen wir den Zahlenstrahl mit der '0' in der Mitte über der '12'. Im Uhrzeigerzinn haben wir dann 1, 2, 3, usw. Gegen den Uhrzeigersinn haben wir
'11Uhr'='-1', '10Uhr'='-2', usw.

Ersetzen wir nun die dezimalen Zahlen durch binäre, dann erhalten wir nach rechts 0001, 0010, 0011, 0100 usw. nach links 1111, 1110, 1101, 1100, usw. Wir erkennen sofort, das eine positive Zahl immer mit einer '0..' beginnt, eine negative mit '1..'. Die Länge spielt dabei keine Rolle: '-1 dezimal' entspricht '1111' = '1111 1111' = '1111 1111 1111' und so weiter, je nach Speicherplatz.



Nun können wir uns den Befehl rjmp -1 zusammenbauen: 1100 1111 1111 1111

Machen Sie sich bitte ein wenig mit dem Datenblatt vertraut.

2.Tag

Bevor ein Programm etwas an seinen Ausgängen ausgeben kann, müssen Vorbereitungen getroffen werden. Bei Atmel sind die Ports in den (64) I/O-Registern untergebracht.

Port-Befehle für die unteren a=0-31 (PortLow) I/O-Register, die ein Bit (b=Bit 0-7) ändern können:
Setze Port a.bit=1; sbi a,b
Lösche Port a.bit=0; cbi a,b

Für den Befehl sbi DDRB,1 (PortB.1=Ausgang) bauen wir mal den Binärcode:
In der Befehlstabelle (doc0856.pdf) finden wir: sbi a,b; 1001 1010 aaaa abbb
Im Datenblatt für den Tiny13 finden wir an I/O-Registeradresse 0x17 (=10111): DDRB
das ergibt dann: 1001 1010 1011 1bbb;
mit Bit=1 = 001: 1001 1010 1011 1001

Der cbi (clear Bit) Befehl funktioniert genauso, nur fängt er mit 1001 1000 .. an.

Versuchen Sie bitte ein Programm zu schreiben, welches die LED an PortB.1 einschaltet und dann anhält.

3.Tag

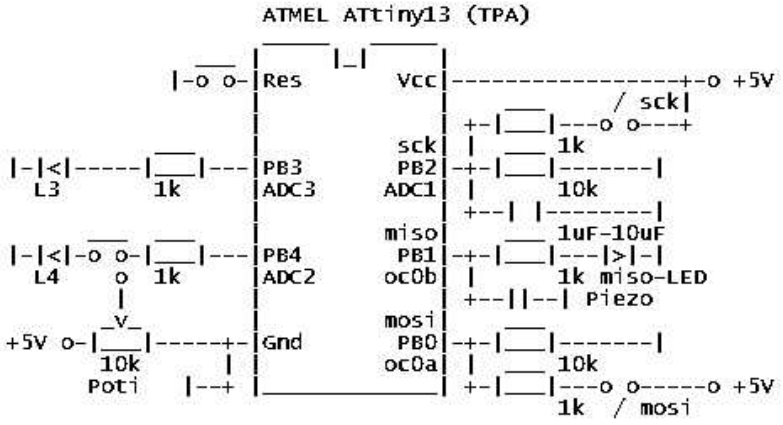
Lösung:
vorbereiten des PortB.1 als Ausgang: sbi ddrb,1 ; 1001 1010 1011 1001
einschalten der LED durch eine '1' : sbi portb,1; 1001 1010 1100 0001
anhalten durch Sprung auf sich selbst: rjmp -1 ; 1100 1111 1111 1111
Das war hoffentlich nicht zu schwer.

Heute kommen noch 2 Portbefehle dazu, die alle I/O-Register (p=0-63) lesen und schreiben können. Außerdem kann mehr als 1 Bit manipuliert werden. Die Werte müssen zwischen gespeichert werden. Dazu dienen die Register r0 bis r31 (bitte nicht mit I/O-Register verwechseln). Da einige Register spezielle Aufgaben haben, benutzen wir erst mal nur r16-r23.

Lade Register mit Port; in rd,p
Lade Port mit Register; out p,rr

Wir holen alle Bit vom PortB in das Register r16: in r16,PinB
Der Code dazu: 1011 0ppr rrrr pppp
Mit R16=10000: 1010 0pp1 0000 pppp
Mit PinB=0x16: 1011 0011 0000 0110

Das ist etwas gewöhnungsbedürftig und Sie sollten es üben. Der out Befehl geht genauso, nur fängt er mit 1011 1... an.

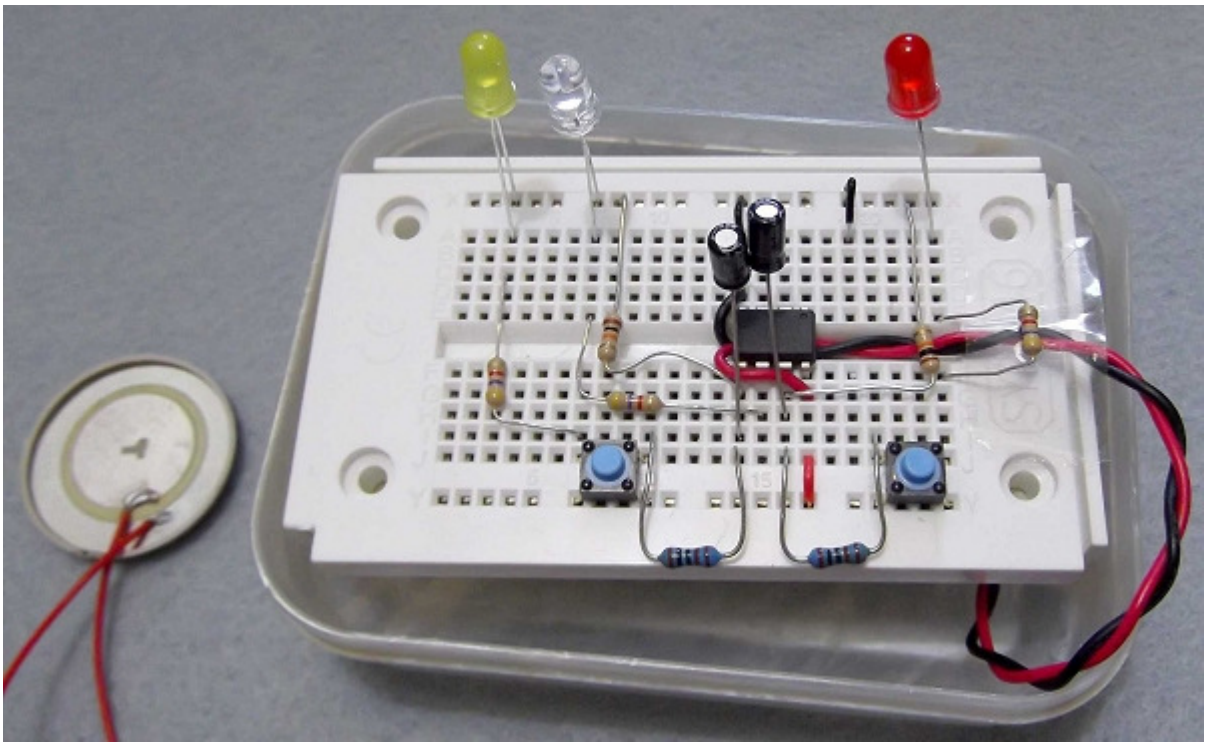


Für Morgen können Sie die folgende Schaltung stecken (ggf. nur die rechte Seite ohne Piezo):

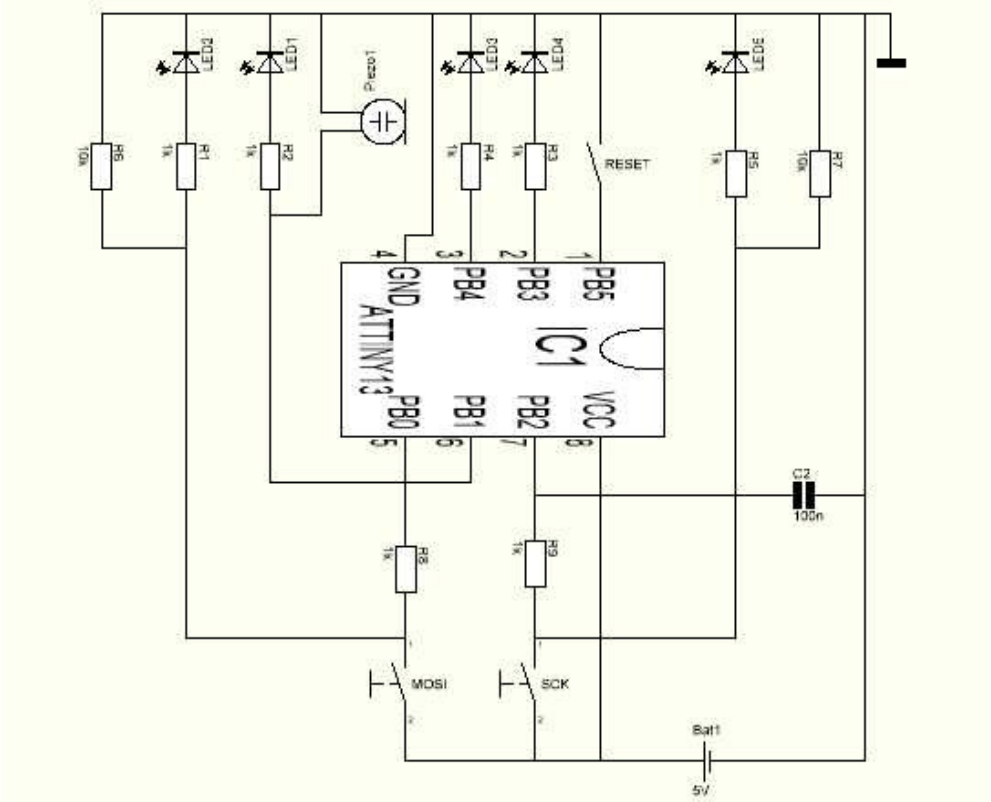
```
.DEVICE ATtiny13              ;für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.CSEG                         ;CodeSegment, muss nicht immer angegeben
```

```
werden
.ORG 0                          ;für Adresse '0', muss nicht immer angegeben
werden
;--------------------------------Programmrumpf
init:                           ;Vorbereitung
        sbi DDRB,1              ;Ausgang definieren
main:   ;------------------------Hauptprogramm
        rjmp main               ;Endlos-Schleife
;--------------------------------
Tag01:                          ;
        rjmp Tag01              ;Endlos-Schleife
Tag02:                          ;
        sbi ddrb,1              ;set bit I/O-Reg(Low),Bit
        cbi portb,1             ;clear bit I/O-Reg(Low),Bit
Tag03:                          ;
        out portb,r16           ;I/O-Reg mit dem Byte in R16 laden
        in r16,pinb             ;R16 mit dem Byte des I/O-Reg laden
```

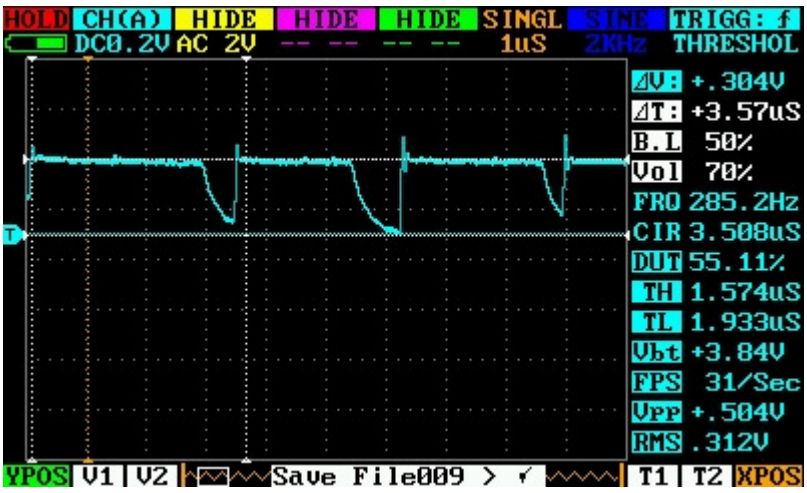
4.Tag



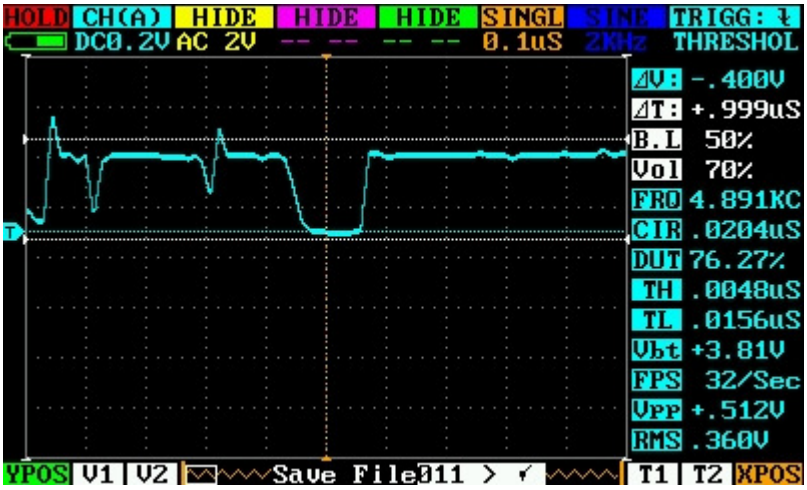
Die meisten AVR werden (seriell) über die SPI/ISP-Schnittstelle programmiert. Dazu gehören folgende Anschlüsse: RESET, MOSI, MISO, SCK und GND. Nach dem anlegen der Versorgungsspannung wird der RESET mit einer Brücke auf GND gelegt. Dadurch werden im Inneren des AVR die Anschlüsse PB.0-PB.2 auf die SPI/ISP-Hardware umgeschaltet. Soll eine '1' gesendet werden, wird MOSI-Taste festgehalten und ein Impuls über die SCK-Taste eingegeben. Bei einer '0' muss nur ein Impuls über die SCK-Taste eingegeben werden.



Tastenprellen: drückt man eine Taste, um eine LED einzuschalten, sieht man nicht, das die LED mehrere mal ein- und ausgeschaltet wurde bevor sie dauernd leuchtet.

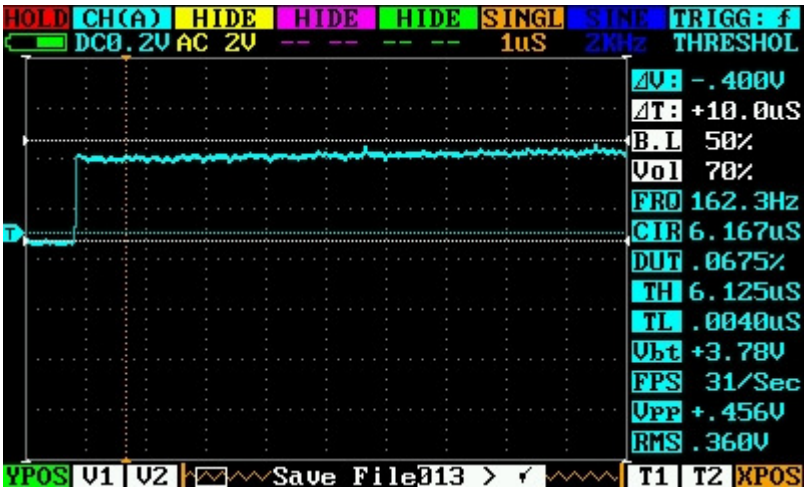


ohne Kondensator



ohne Kondensator

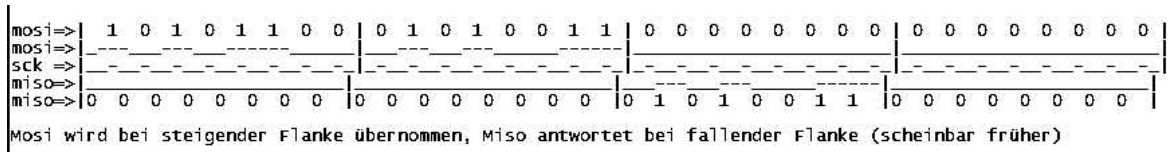
Um bei genau einem Tastendruck auch nur genau einen Impuls zu erhalten, muss ein Kondensator mit seiner Ladung die Zeit (1ms-100ms = 100nF-1uF) des Tastenprellen überbrücken.



mit Kondensator 100nF

Manche Taster sind von so miserabler Qualität, das auch ein Kondensator an MOSI durchaus Sinn macht.

Zu jeder Sequenz gehören 4 Byte (= genau 32 Sck Tastendrücke). Miso antwortet mit der fallenden Flanke. Zur Übung, ob alles richtig angeschlossen ist, geben Sie bitte NUR den 'Prog Enable'-Befehl ein:



Miso antwortet mit 0x53 im 3.Byte! Danach lösen Sie Reset erst einmal auf!
Für den Fall, das es noch nicht klappt:
überprüfen Sie bitte die Schaltung, insbesondere den korrekten Sitz des Entprell-

Kondensators.
Ggf. tauschen Sie den Taster an Sck aus.

Bitte wiederholen Sie 'Prog Enable', bis Sie sicher sind, das Miso eindeutig antwortet und die Tasten nicht prellen. (Reset zwischendurch aufheben)

Programmierung einleiten	prog	enable	Mosi	1010	1100	0101	0011	xxxx	xxxx	xxxx	xxxx	AC 53 00 00
			Miso					0101	0011			00 00 53 00
Lese Signaturbyte ad=0-2	read	signature	Mosi	0011	0000	000x	xxxx	xxxx	xx00	xxxx	xxxx	30 00 00 00
			Miso			0011	0000			0001	1110	00 30 00 1E
Lese Signaturbyte ad=0-2	read	signature	Mosi	0011	0000	000x	xxxx	xxxx	xx01	xxxx	xxxx	30 00 01 00
			Miso			0011	0000			1001	0000	00 30 00 90
Lese Signaturbyte ad=0-2	read	signature	Mosi	0011	0000	000x	xxxx	xxxx	xx10	xxxx	xxxx	30 00 02 00
			Miso			0011	0000			0000	0111	00 30 00 07

Nun können Sie die drei Signatur-Byte lesen. Im 2.Byte muss Miso mit '0x30' antworten! Andernfalls unterbrechen Sie die Eingabe und beginnen von vorn.

WARNUNG: Über eine ähnliche Sequenz werden auch die FUSE-Bits verändert !!!
Falls Sie sich verzählt haben, sollten Sie zuerst RESET freigeben und dann noch einmal anfangen!!!

Wer es nachlesen möchte, findet im Datenblatt Kapitel 17.6 Serial Programming weitere Info's.

Bitte üben, üben, üben, man verzählt sich leicht.

5.Tag

Heute kann der Tiny13 sein erstes Programm über ISP/SPI bekommen. Es ist das Programm vom 2.Tag und schaltet nur die LED an PB1 ein.

- Zum programmieren wird zuerst die Versorgungsspannung angelegt,
- dann wird RESET mit einer Drahtbrücke auf GND gelegt,
- die einzelnen Bit werden mit MOSI geschaltet und mit SCK übertragen,
- mit ProgEnable und Chip-Erase wird die Programmierung eingeleitet,
- mit LoadPage.. werden die Programmbyte in den Zwischspeicher übertragen,
- mit Write-Page werden die (6) Byte ins Flash-Memory übertragen,

Die gesamte Sequenz binär (wenn die Miso-LED nicht stimmt, brechen Sie bitte ab):

Programmierung einleiten	prog	enable		Mosi	1010	1100	0101	0011	xxxx	xxxx	xxxx	xxxx	AC 53 00 00	
									0101	0011			00 00 53 00	<=Miso
Chip löschen (Flash+EEPROM)	chip	erase		Mosi	1010	1100	100x	xxxx	xxxx	xxxx	xxxx	xxxx	AC 80 00 00	
							1010	1100					00 AC	<=Miso
Zwischenspeicher addr=0	load	prog mem	L	Mosi	0100	0000	000x	xxxx	xxxx	0000	1011	1001	40 00 00 B9	Sbi ddrb,1
							0100	0000					00 40	<=Miso
Zwischenspeicher addr=0	load	prog mem	H	Mosi	0100	1000	000x	xxxx	xxxx	0000	1001	1010	48 00 00 9A	
					1011	1001	0100	1000					B9 48	<=Miso
Zwischenspeicher addr=1	load	prog mem	L	Mosi	0100	0000	000x	xxxx	xxxx	0001	1100	0001	40 00 01 C1	Sbi portb,1
					1001	1010	0100	0000				0001	9A 40	<=Miso
Zwischenspeicher addr=1	load	prog mem	H	Mosi	0100	1000	000x	xxxx	xxxx	0001	1001	1010	48 00 01 9A	
					1100	0001	0100	1000				0001	C1 48	<=Miso
Zwischenspeicher addr=2	load	prog mem	L	Mosi	0100	0000	000x	xxxx	xxxx	0010	1111	1111	40 00 02 FF	Rjmp -1
					1001	1010	0100	0000				0010	9A 40	<=Miso
Zwischenspeicher addr=2	load	prog mem	H	Mosi	0100	1000	000x	xxxx	xxxx	0010	1100	1111	48 00 02 CF	
					1111	1111	0100	1000				0010	FF 48	<=Miso
Umladen in aaddr=0	write	prog mem		Mosi	0100	1100	0000	000a	addr	xxxx	xxxx	xxxx	4C 00 00 00	
					1100	1111	0100	1100					CF 4C	<=Miso

Nun kann RESET wieder freigegeben werden und die Miso-LED sollte leuchten.

Herzlichen Glückwunsch, Sie haben gerade einen AVR ohne 'Brenner' programmiert!

6.Tag

Sie haben schon kurz etwas von Registern gehört. Es sind Speicherplätze (je 1 Byte), die für Zwischenergebnisse zur Verfügung stehen. Einige der 32 Register haben besondere Aufgaben.

Die Register R0 bis R15 sind nicht von jedem Befehl erreichbar.

Die Register R24 bis R31 werden meist für Word-Ergebnisse (2 Byte) gebraucht (nicht mit jedem Befehl). Als Universal-Register bleiben R16 bis R23 (8 Stück) übrig.

Befehle für die (Speicher-) Register.

Lade Register (r16-r31) direkt mit einer Konstenten Zahl; ldi rd,k
Lade Register (r0-r31) mit einem beliebigen anderem Register; mov rd,rr

Lade r16 mit 255: 1110 kkkk dddd kkkk
Mit ldi r16,255: 1110 1111 0000 1111
da hier nur r16-r31 zulässig sind, entspricht r16 = 0000; r31 = 1111

Lade r16 mit r17: 0010 11rd dddd rrrr
Mit mov r16,r17: 0010 1111 0000 0001
hier sind alle Register zulässig; r16 = 10000; r17 = 10001

Das Ziel ist immer das erste Register rd <- rr (d= Destination= Ziel)

Es kommt vor, das 2 Byte (= 1 WORD) am Stück verarbeitet werden müssen. Dazu werden die geraden Register (r24; r26; r28; r30) benutzt und, ohne es zu erwähnen, das ungerade höhere Register noch dazu.

Lade Doppelregister r0(r1)-r30(r31) mit einem beliebigen anderem Word-Reg.; movw rd,rr
Im geraden Register steht die untere Hälfte (LSB) des Wertes, im Oberen die obere Hälfte (MSB).

Lade Doppelregister r24 mit r26: 0000 0001 dddd rrrr
Mit movw r24,r26: 0000 0001 1100 1101
Aus r24 wird (24/2) dez 12 = bin 1100; aus r26 wird dez 13 = bin 1101

(movw gibt es bei Tiny11/12 nicht.) ... Sie sollten es üben.

```
.DEVICE ATtiny13                ;für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.CSEG                           ;CodeSegment, muss nicht immer angegeben
werden
.ORG 0                          ;für Adresse '0', muss nicht immer angegeben
werden
;--------------------------------Programmrumpf
init:                           ;Vorbereitung
        sbi DDRB,1              ;Piezo als Ausgang definieren
main:   ;------------------------Hauptprogramm
        rjmp main               ;Endlos-Schleife
;--------------------------------
Tag01:                          ;
        rjmp Tag01              ;Endlos-Schleife
Tag02:                          ;
        sbi ddrb,1              ;set bit I/O-Reg(Low),Bit
        cbi portb,1             ;clear bit I/O-Reg(Low),Bit
Tag03:                          ;
        out portb,r16           ;I/O-Reg mit dem Byte in R16 laden
        in r16,pinb             ;R16 mit dem Byte des I/O-Reg laden
Tag06:                          ;
        ldi r16,7               ;lade R16 mit 7(dez)(Byte)
        mov r16,r17             ;lade R16 mit dem Byte in R17
        movw r24,r26            ;R24<=R26 und R25<=R27 (Word)
```

7.Tag

Sie können den Flash-Speicher auslesen, ohne den Inhalt zu verändern. Lesen Sie bitte Low- und High-Byte an der Adresse 0.

Programmierung einleiten	prog	enable		Mosi	1010	1100	0101	0011	xxxx	xxxx	xxxx	xxxx	AC 53 00 00	
				Miso					0101	0011			00 00 53 00	
Lesen von aaddress=0-511	read	prog mem	L	Mosi	0010	0000	0000	000a	aadd	ress	xxxx	xxxx	20 0= == 00	
				Miso			0010	0000			data	data		1011 1001
Lesen von aaddress=0-511	read	prog mem	H	Mosi	0010	H000	0000	000a	aadd	ress	xxxx	xxxx	28 0= == 00	
				Miso			0010	1000			data	data		1001 1010

0xB9 und 0x9A sind die erwarteten Ergebnisse.

Als nächstes benötigen wir Befehle, um die Registerinhalte zu verändern. Mathematische Befehle:

Addiere zwei beliebige Register; add rd,rr
Addiere Doppelregister (r24-r30) mit Konstanten; adiw rd,k

Addiere r16 und r17: 0000 11rd dddd rrrr
Mit add r16,r17: 0000 1111 0000 0001

Addiere zu r24 1: 1001 0110 kkdd kkkk
Mit adiw r24,1: 1001 0110 0000 0001

Beim adiw-Befehle wird beim hochzählen ein Übertrag ins höhere Register automatisch ausgeführt. Erst beim 65536. Aufruf von adiw r24,1 ist das Doppelregister einmal durchgezählt. Lädt man R25 auf den PortB, wird PB.1 (die miso-Leitung) das 2.Bit von unten anzeigen. (adiw gibt es bei Tiny11/12 nicht)

Aufgabe: Welche Wirkung hat add r16,r16 ?

8.Tag

Lösung: Richtig! Multiplikation mit 2; r16 <= r16 * 2

Lesen Sie bitte ein Byte an Adresse '7', es sollte '0xF', weil gelöscht lauten.

Falls Sie BAScom installiert haben, können Sie es zum assemblieren benutzen. Es erzeugt einen größeren Code als ein Assembler, weil die Vorbereitungen für Basic-Programme erforderlich sind. Programm 1 in BAScom:

```
' 13UK, Tag 5, Programm 1, LED an PortB.1 = 1
'--------------------------------------------------------------------------
-----
' alle Fuse-Byte ab Werk
' kompiliert mit Bascom-Demo 2.0.7.1 (sollte auch mit 2.0.7.5
funktionieren)
'--------------------------------------------------------------------------
-----
$regfile = "attiny13.dat"
$crystal = 1200000            ' 9,6MHz/8
$hwstack = 8
$asm                          ' Assembler-Befehle einfügen
'--------------------------------------------------------------------------
-----
   sbi DDRB,1                 ' PortB.1 = Ausgang
   sbi portb,1                ' Set Bit PortB.1
   rjmp -1                    ' Endlosschleife
'--------------------------------------------------------------------------
```

```
-----
$end Asm                        ' End Assembler
End
' End Program
```

Das .ASM-File aus dem Disassembler:

```
          ; Atmel AVR Disassembler v1.30
          ;
          .cseg
          .org  0

          rjmp   avr000A      ; 0000 C009     ;Vector Tabelle
          reti                ; 0001 9518
          reti                ; 0002 9518
          reti                ; 0003 9518
          reti                ; 0004 9518
          reti                ; 0005 9518
          reti                ; 0006 9518
          reti                ; 0007 9518
          reti                ; 0008 9518
          reti                ; 0009 9518
avr000A:  ldi    r24, 0x9F    ; 000A E98F
          out    SPL, r24     ; 000B BF8D     Stack-Pointer-Low=RamEnd
          ldi    YL, 0x98     ; 000C E9C8
          ldi    ZL, 0xA0     ; 000D EAE0
          mov    r4, ZL       ; 000E 2E4E     r4=0xA0
          clr    YH           ; 000F 27DD     Y=0x0098
          mov    r5, YH       ; 0010 2E5D     r5=0
          wdr                 ; 0011 95A8     WD-Timer restart
          in     r24, ?0x34?  ; 0012 B784     MCUSR = Reset-Flags
          mov    r0, r24      ; 0013 2E08
          andi   r24, 0xF7    ; 0014 7F87     WD-Reset abschalten
          out    ?0x34?, r24  ; 0015 BF84     zurückschreiben
          ldi    r24, 0x18    ; 0016 E188
          clr    r25          ; 0017 2799
          out    WDTCR, r24   ; 0018 BD81     WDTCR(0x21) =0x18 enable
          out    WDTCR, r25   ; 0019 BD91     WDTCR(0x21) =0
          ldi    ZL, 0x3E     ; 001A E3EE
          ldi    ZH, 0x00     ; 001B E0F0     Z=0x003E
          ldi    XL, 0x60     ; 001C E6A0
          ldi    XH, 0x00     ; 001D E0B0     X=0x0060
          clr    r24          ; 001E 2788
avr001F:  st     X+, r24      ; 001F 938D     SRam löschen von 0x60 bis
0x9E
          sbiw   ZL, 0x01     ; 0020 9731
           brne  avr001F      ; 0021 F7E9
          clr    r6           ; 0022 2466     r6=0
          sbi    DDRB, 1      ; 0023 9AB9 ### Init:
avr0024:  sbi    PORTB, 1     ; 0024 9AC1 ### Main:
          rjmp   avr0024      ; 0025 CFFE ### Endlosschleife
          cli                 ; 0026 94F8     clear Interrupt-Flag   wird
nicht aufgerufen
avr0027:  rjmp   avr0027      ; 0027 CFFF     HALT                   wird
nicht aufgerufen
avr0028:  sbiw   ZL, 0x01     ; 0028 9731     sub_wait ZL (256)      wird
nicht aufgerufen
           brne  avr0028      ; 0029 F7F1     springe wenn Z=0
          ret                 ; 002A 9508     end sub
          set                 ; 002B 9468     Müll...                wird
nicht aufgerufen
          bld    r6, 2        ; 002C F862
```

```
          ret                 ; 002D 9508
          clt                 ; 002E 94E8
          bld     r6, 2       ; 002F F862
          ret                 ; 0030 9508

         .exit
```

An der Adresse 0x0023 finden wir unser Programm wieder (Der Rest ist im Moment nicht wichtig).

Wir finden unsere 3 Befehle im .HEX-File:

:1000000009C01895189518951895189518956C
:10010000189518958FE98DBFC8E9E0E84E2EDD27C9
:100020005D2EA89584B7082E877F84BF88E1992725
:1000300081BD91BDEEE3F0E0A0E6B0E088278D93AE

----Adr-----------------0x46-0x48-0x4A
-----vv-----------------v-v--v-v--v-v-
:10004000 3197 E9F7 6624 B99A C19A FECF F894 FFCF A9
:100050003197F1F70895689462F80895E89462F88A
:02006000089501
:00000001FF

Hier werden die Adressen in Byte gezählt, im Listing in Word, aus 0x0023 wird deshalb 0x0046!

Bei den Befehlen kommt immer LSByte zuerst, dann das MSByte.

Nur diese 6 Byte mussten wir eingeben.

In der Befehlsbeschreibung ist Ihnen das Status Register (Sreg) aufgefallen. Sreg wird nach jeder mathematischen (und logischen) Operation neu gesetzt. Hier nur die drei wichtigsten Flags:

- N(egativ)-Flag ist eine Kopie des obersten Bit im Register (für signierte Zahlen).
- Z(ero)-Flag wird gesetzt, wenn das Ergebnis =0 war.
- C(arry)-Flag wird bei einem Überlauf/Unterlauf (>255/Byte >65535/Word) gesetzt.

r16=100; r17=200; add r16,r16; r16=200; N=1; Z=0; C=0
r16=100; r17=200; add r16,r17; r16= 44; N=0; Z=0; C=1; r16=300-256=44 weil das Byte 'übergelaufen' ist.

Zum Rechnen gehört auch das Abziehen:

Subtrahiere ein Register vom einem anderen Register; sub rd,rr; code 0001 10rd dddd rrrr
Subtrahiere eine Konstante vom Register (r16-r31) ; subi rd,k; code 0101 kkkk dddd kkkk
Subtrahiere eine Konstante vom Word-Reg. (r24-r30) ; sbiw rd,k; code 1001 0111 kkdd kkkk
(sbiw nicht bei Tiny11 und Tiny12)

Zum subi gibt es kein addi. Da k von 0-255 betragen darf, kann k auch signiert (+-127) sein.

Die Aufgabe 'r16=r16+15' => r16=r16-(-15) => r16=r16-11110001 => subi r16,0b11110001
Ein wenig seltsam, aber es geht.

Aufgabe: Bitte finden Sie den Code für sub r16,r17 und suchen Sie die NZC-Flags mit r16=100, r17=200.

Antwort: 0 0 0 1 1 0 _ _ _ _ _ _ _ _ _ _ ; N= __ ; Z= __ ; C= __

9.Tag

Lösung: 0001 1011 0000 0001; r16=155; N=1; Z=0; C=1; weil Carry auch den Unterlauf anzeigt.

Heute soll ein Ton mit dem Piezo an der Miso-LED erzeugt werden, indem ein Word-Register (R24/R25) hochgezählt wird und R25 in PortB geladen wird (PortB.1 = R25.1).

```
;
;bei r24(25) = n * 512 erfolgt das Toggeln von PB.1 (Puls+Pause=1024)
;Frequenz: 1200000Hz / 5 Cyclen / 1024 = ~234Hz
;
init:               ;Vorbereitungen
   9AB9  sbi ddrb,1    ;DDRB.1=Ausgang

main:               ;Hauptprogramm
   9601  adiw r24,1    ;r25/r24 hochzählen (2 Cyclen)
   BB98  out portb,r25    ;r25 ausgeben (1 Cyclus)
   CFFD  rjmp main        ;Jr-3 nochmal (2 Cyclen)
```

Programmierung einleiten	prog	enable		Mosi	1010	1100	0101	0011	xxxx	xxxx	xxxx	xxxx	AC 53 00 00	
									0101	0011			00 00 53 00	<=Miso
Chip löschen (Flash+EEPROM)	chip	erase		Mosi	1010	1100	100x	xxxx	xxxx	xxxx	xxxx	xxxx	AC 80 00 00	
							1010	1100					00 AC	<=Miso
Zwischenspeicher addr=0	load	prog mem	L	Mosi	0100	0000	000x	xxxx	xxxx	0000	1011	1001	40 00 00 B9	Sbi ddrb,1
							0100	0000					00 40	<=Miso
Zwischenspeicher addr=0	load	prog mem	H	Mosi	0100	1000	000x	xxxx	xxxx	0000	1001	1010	48 00 00 9A	
					1011	1001	0100	1000					B9 48	<=Miso
Zwischenspeicher addr=1	load	prog mem	L	Mosi	0100	0000	000x	xxxx	xxxx	0001	0000	0001	40 00 01 01	Adiw r24,1
					1001	1010	0100	0000				0001	9A 40	<=Miso
Zwischenspeicher addr=1	load	prog mem	H	Mosi	0100	1000	000x	xxxx	xxxx	0001	1001	0110	48 00 01 96	
					0000	0001	0100	1000				0001	C1 48	<=Miso
Zwischenspeicher addr=2	load	prog mem	L	Mosi	0100	0000	000x	xxxx	xxxx	0010	1001	1000	40 00 02 98	Out portb,r25
					1001	0110	0100	0000				0010	9A 40	<=Miso
Zwischenspeicher addr=2	load	prog mem	H	Mosi	0100	1000	000x	xxxx	xxxx	0010	1011	1011	48 00 02 BB	
					1001	1000	0100	1000				0010	FF 48	<=Miso
Zwischenspeicher addr=3	load	prog mem	H	Mosi	0100	1000	000x	xxxx	xxxx	0011	1111	1101	48 00 02 FD	Rjmp -3
					1011	1011	0100	1000				0011	FF 48	<=Miso
Zwischenspeicher addr=3	load	prog mem	H	Mosi	0100	1000	000x	xxxx	xxxx	0011	1100	1111	48 00 02 CF	
					1111	1101	0100	1000				0011	FF 48	<=Miso
Umladen in aaddr=0	write	prog mem		Mosi	0100	1100	0000	000a	addr	xxxx	xxxx	xxxx	4C 00 00 00	
					1100	1111	0100	1100					CF 4C	<=Miso

Nach freigeben des RESET, sollte ein Ton erklingen.

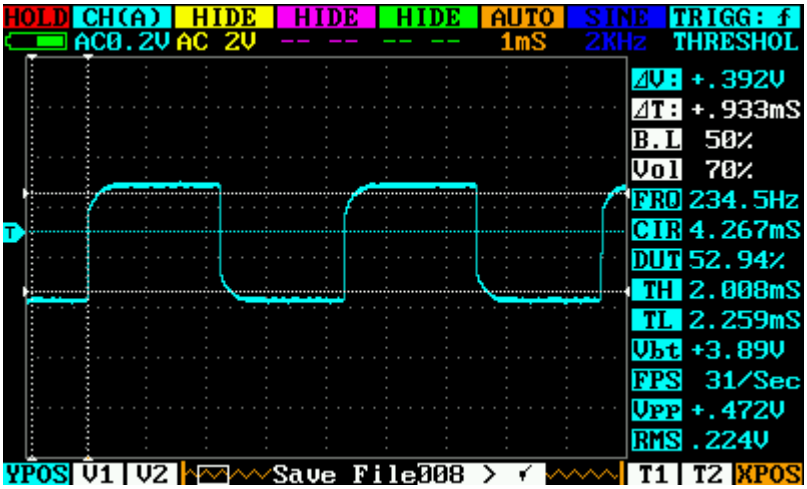
Hier das .BAS-File dazu:

```
' 13UK, Tag 9, Programm 3, LED an PortB.1 = 1
'
' 9600000Hz/8 /256(r24) /5(Cyclen) /2(r25) /2(toggle) = 234,4 Hz
' mit der Konstanten (1) hinter adiw können höhere Frequenzen eingestellt
werden
' mit z.B. 10 werden 2344 Hz erzeugt
'-------------------------------------------------------------------------
-----
' alle Fuse-Byte ab Werk
' kompiliert mit Bascom-Demo 2.0.7.1 (sollte auch mit 2.0.7.5
funktionieren)
'-------------------------------------------------------------------------
-----
$regfile = "attiny13.dat"
$crystal = 1200000             ' 9,6MHz/8
$hwstack = 8
$asm                           ' Assembler-Befehle einfügen
'-------------------------------------------------------------------------
-----
Init:                          ' Vorbereitungen
   sbi DDRB,1                  ' PortB.1 = Ausgang
Main:                          ' Hauptprogramm
   adiw r24,1                  ' r25/r24 hochzählen (2 Cyclen)
   Out Portb , R25             ' R25 Ausgeben       (1 Cyclus)
   rjmp main                   ' nochmal            (2 Cyclen)
'-------------------------------------------------------------------------
-----
$end Asm                       ' End Assembler
End
' End Program
```

Das .HEX-File:

```
:1000000009C01895189518951895189518951895189518956C
:10010000189518958FE98DBFC8E9E0E84E2EDD27C9
:100020005D2EA89584B7082E877F84BF88E1992725
:1000300081BD91BDEEE3F0E0A0E6B0E088278D93AE

----Adr------------------0x46-0x48-0x4A-0x4C
-----vv------------------v-v--v-v--v-v--v-v-
:10004000 3197 E9F7 6624 B99A 0196 98BB FDCF F894 E9
:10005000FFCF3197F1F70895689462F80895E89416
:0400600062F80895A5
:00000001FF
```

Und das Oszillogramm:



Das Oszi misst netterweise selbst die Frequenz, (rechts) 'FRQ 234,5Hz'

Bei diesem Programm hatte ich beim 1.Versuch beim rjmp nicht 'CFFD' eingegeben, deshalb war die Frequenz niedriger...bin eben nur ein Mensch. Mit größeren Werten bei adiw r24,WERT können Sie höhere Frequenzen einstellen. (20 => 234Hz * 20 = ~4680Hz)

Können Sie das .ASM-File selbst kommentieren? [13uk-prog3.asm.txt]

```
          ; Atmel AVR Disassembler v1.30
          ;
          .cseg
          .org   0

          rjmp    avr000A       ; 0000 C009
          reti                  ; 0001 9518
          reti                  ; 0002 9518
          reti                  ; 0003 9518
          reti                  ; 0004 9518
          reti                  ; 0005 9518
          reti                  ; 0006 9518
          reti                  ; 0007 9518
          reti                  ; 0008 9518
          reti                  ; 0009 9518
avr000A:  ldi     r24, 0x9F     ; 000A E98F
          out     SPL, r24      ; 000B BF8D
          ldi     YL, 0x98      ; 000C E9C8
          ldi     ZL, 0xA0      ; 000D EAE0
          mov     r4, ZL        ; 000E 2E4E
          clr     YH            ; 000F 27DD
          mov     r5, YH        ; 0010 2E5D
          wdr                   ; 0011 95A8
          in      r24, ?0x34?   ; 0012 B784
          mov     r0, r24       ; 0013 2E08
          andi    r24, 0xF7     ; 0014 7F87
          out     ?0x34?, r24   ; 0015 BF84
          ldi     r24, 0x18     ; 0016 E188
          clr     r25           ; 0017 2799
          out     WDTCR, r24    ; 0018 BD81
          out     WDTCR, r25    ; 0019 BD91
          ldi     ZL, 0x3E      ; 001A E3EE
          ldi     ZH, 0x00      ; 001B E0F0
          ldi     XL, 0x60      ; 001C E6A0
```

```
          ldi    XH, 0x00     ; 001D E0B0
          clr    r24          ; 001E 2788
avr001F:  st     X+, r24      ; 001F 938D
          sbiw   ZL, 0x01     ; 0020 9731
           brne  avr001F      ; 0021 F7E9
          clr    r6           ; 0022 2466
          sbi    DDRB, 1      ; 0023 9AB9     ###
avr0024:  adiw   r24, 0x01    ; 0024 9601     ###
          out    PORTB, r25   ; 0025 BB98     ###
          rjmp   avr0024      ; 0026 CFFD     ###
          cli                 ; 0027 94F8
avr0028:  rjmp   avr0028      ; 0028 CFFF
avr0029:  sbiw   ZL, 0x01     ; 0029 9731
           brne  avr0029      ; 002A F7F1
          ret                 ; 002B 9508
          set                 ; 002C 9468
          bld    r6, 2        ; 002D F862
          ret                 ; 002E 9508
          clt                 ; 002F 94E8
          bld    r6, 2        ; 0030 F862
          ret                 ; 0031 9508

         .exit
```

Zwei weitere Befehle ermöglichen das abziehen/addieren mit 1:
Register -1; dec rd; rd<=rd-1; code 1001 010d dddd 1010
Register +1; inc rd; rd<=rd+1; code 1001 010d dddd 0011

Beim Subtrahieren sind Ihnen die seltsamen Ergebnisse beim Register-Unterlauf aufgefallen. Das sind einfach nur Zahlen, die man als negativ (-) interpretieren muss. Will man mit negativen Zahlen arbeiten, muss man sich darüber klar sein, das nur +127 bis -127 in ein Byte passen. +0 bis +127 -> oberes Bit nicht gesetzt, -1 bis -127 -> oberes Bit gesetzt. Für -128 gibt es keine positive Zahl.

Um aus einer positiven Zahl eine negative zu machen oder aus einer negativen eine positive müssen alle Bit invertiert (aus 1 wird 0, aus 0 wird 1) und eine 1 addiert werden. Fertig! Dafür gibt es den Befehl: neg rd; code 1001 010d dddd 0001 Die Wirkung ist wie: Zahl = Zahl * -1, egal ob Zahl positiv oder negativ war.

Will man nur alle Bits invertieren (Complement-Befehl): com rd; code 1001 010d dddd 0000

Aufgabe: Wandeln Sie -128 in eine positive Zahl und sehen, warum das nicht geht.

10.Tag

Lösung: -128 = 1000 0000 -> com -> 0111 1111 -> Inc -> 1000 0000 = -128 ; +128 gibt es nicht!

Das Programm von gestern hat einen Ton erzeugt, indem ein Port ein- und ausgeschaltet wurde. So eine (primitive) Aufgabe kann man durch einen Timer erledigen lassen. Wir

benutzen den Timer im 'normal'-Mode, das heisst er zählt von 0-255 und dann wieder von vorn. Wir legen fest, das bei einem Überlauf PortB.1 umgeschaltet (toggle) wird:
TCCR0A=0x00010000=16.

Dann schalten wir den höchsten Vorteiler (prescaler) ein und starten den Timer im Normal-Mode:
TCCR0B=0x00000101=5.

Die Frequenz ist dann: 9,6MHz /8(devide) /1024(prescaler) /256(Timer) /2(toggle) =2,288Hz

Das war schon alles, der Timer schaltet den Port ohne weiteres Zutun. Das Progamm 4 übersetzen (assemblieren) Sie bitte mit 'gavrasm':

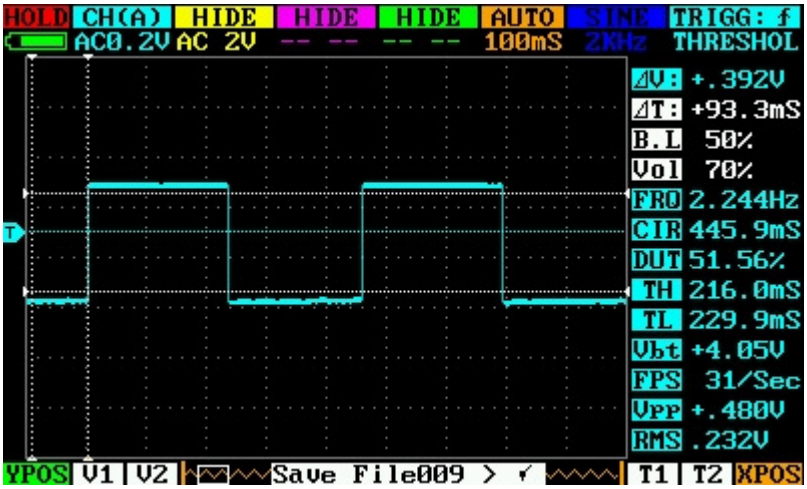
```
.DEVICE ATtiny13       ;für gavrasm, für Symbolische Registerbezeichnungen
(z.B.'DDRB')
.cseg
.org 0
;
; der Timer blinkt selbständig an PortB.1 (=OC0B)
;
; 9600000Hz /8 /1024(prescaler) /256(Timer) /2(toggle) =2,288Hz
;
Init:
   sbi DDRB,1           ; PortB.1 = Ausgang
Init_timer:
   ldi r16,16           ; 0b00010000
   Out Tccr0a , R16    ;     ^^-----toggle Ausgang B bei Überlauf

   ldi r16,5            ; 0b00000101
   Out Tccr0b , R16     ;        ^^^-Vorteiler /1024 und Timer starten
Main:
   rjmp main            ; Endlosschleife
```

Im .HEX-File finden Sie die Codes (12Byte):

```
von gavrasm erzeugt:
:020000020000FC

------Adr-L-M--L-M--L-M--L-M--L-M--L-M
-------vv-v-v--v-v--v-v--v-v--v-v--v-v  vv-Checksumme
:0C000000 B99A 00E1 0FBD 05E0 03BF FFCF 7F
:00000001FF

L=LSByte das niederwertige Byte zuerst
M=MSByte
```

Können Sie es eintippen? Sie sehen dann ein schnelles blinken.

Im Oszillogramm sehen wir, das die berechnete Frequenz fast stimmt:



Weitere mathematische Befehle:
Division (/2) gibt es auch und heisst arithmetic shift right: asr rd; code 1001 010d dddd 0101
Alle Bits werden nach rechts verschoben, wobei das Vorzeichenbit 7 erhalten bleibt und das untere
Bit ins C(arry)-Flag geschoben wird.

r16=9dez=0000 1001; asr r16; r16=4dez=0000 0100; S=0; Z=0; C=1
r16=-4dez=1111 1100; asr r16; r16=-2dez=1111 1110; S=1; Z=0; C=0

Ein ähnlicher Befehl von den logischen Befehlen: logical shift right: lsr rd: code 1001 010d dddd 0110
Hier bleibt das Vorzeichenbit nicht erhalten und ist deshalb nur für positive Zahlen anwendbar.
r16=255dez=1111 1111; lsr r16; r16=127dez=0111 1111; C=1

Wenn mehrere Byte nacheinander nach links (*2) oder rechts (/2) geschoben werden müssen, muss ein Übertrag oder Unterlauf für das nächste Byte gespeichert werden. Dafür ist das C(arry)-Flag ideal.

adc rd,rr (add with Carry) mit rd=rr (*2) ;0001 11rd dddd rrrr
ror rd (rotate right through Carry) (/2) ;1001 010d dddd 0111

11.Tag

Das Programm von gestern in BAScom:

```
' 13UK, Tag 10, Programm 4, blinkt mit Timer, LED an OC0B (PortB.1)
'
' 9600000Hz /8 /1024(prescaler) /256(Timer) /2(toggle) =2,288Hz
'
```

```
'-------------------------------------------------------------------------
-----
' alle Fuse-Byte ab Werk
' kompiliert mit Bascom-Demo 2.0.7.1 (sollte auch mit 2.0.7.5
funktionieren)
'-------------------------------------------------------------------------
-----
$regfile = "attiny13.dat"
$crystal = 1200000             ' 9,6MHz/8
$hwstack = 8
$asm                           ' Assembler-Befehle einfügen
'-------------------------------------------------------------------------
-----
Init:
   sbi DDRB,1                  ' PortB.1 = Ausgang
Init_timer:
   ldi r16,16                  ' &B00010000
   Out Tccr0a , R16            '     ^^-----toggle Ausgang B bei Überlauf

   ldi r16,1                   ' &B00000001<-kein Vorteiler = 2343Hz
'  ldi r16,5                   ' &B00000101
   Out Tccr0b , R16            '        ^^^-Vorteiler /1024 und Timer
starten
Main:
   rjmp main                   ' Endlosschleife
'-------------------------------------------------------------------------
-----
$end Asm                       ' End Assembler
End
' End Program
```

Weitere logische Befehle:
Und-Verknüpfung zwischen zwei Registern; and rd,rr; code 0010 00rd dddd rrrr
Exclusiv - Oder zwischen zwei Registern; eor rd,rr; code 0010 01rd dddd rrrr
Oder-Verknüpfung zwischen zwei Registern; or rd,rr; code 0010 10rd dddd rrrr
Oder-Verknüpfung zwischen Reg und Konst.; ori rd,k; code 0110 kkkk dddd kkkk
Und-Verknüpfung zwischen Reg und Konst.; andi rd,k; code 0111 kkkk dddd kkkk

Dazu noch ein paar Tricks:

r16 soll gelöscht werden: eor r16,r16; Z=1
S und Z-Flag soll gesetzt werden ohne den Inhalt zu verändern: or r16,r16
oberes Nibble auf 1 setzen, untere Bits erhalten: ori r16,240
unteres Nibble auf 0 setzen, obere Bits erhalten: andi r16,240

Hier noch eine Aufgabe für Profis (Anfänger sollten es nicht versuchen):
Was passiert mit den Inhalten von r16 und r17 nach den drei Befehlen?

eor r16,r17
eor r17,r16
eor r16,r17

12.Tag

Lösung mit r16 = 0101 = 5 und r17 = 0011 = 3:
eor r16,r17; r16 = 0110; r17 wie vorher = 0011
eor r17,r16; r17 = 0101; r16 wie vorher = 0110
eor r16,r17; r16 = 0011; r17 wie vorher = 0101
r16 = 3; r17 = 5; -> die Inhalte der Register haben ihre Plätze getauscht, ohne ein Hilfsregister!
Probieren Sie es mal mit anderen Zahlen (es klappt immer).

Damit nicht immer nur ein Befehl nach dem anderen abgearbeitet werden muss, gibt es Sprung-Befehle. Den unbedingten (springt immer) Sprung rjmp k kennen Sie schon. Dazu gibt es noch Sprünge, die nur springen, wenn eine BEDINGUNG erfüllt ist. Dazu dient das Sregister mit seinen Flags. Je nachdem, ob ein Flag gesetzt oder gelöscht ist, wird ein Sprung zu einer anderen Programmzeile ausgeführt oder nicht.

Bedingter Sprung wenn ein Bit im Sreg=0; brbc s,k; code 1111 01kk kkkk ksss
Bedingter Sprung wenn ein Bit im Sreg=1; brbs s,k; code 1111 00kk kkkk ksss

Springe wenn Z-Flag=0: brbc 1,k; da k nur +-63 betragen darf, kann man nur 'in die Nähe' springen.
Springe wenn C-Flag=1; brbs 0,k;
Diese Befehle machen nur Sinn, wenn vorher ein mathematischer/logischer Befehl die Flags setzt.

Manchmal ist es einfacher nur den nächsten Befehl zu überspringen. SKIPBEFEHLE (von Skippy das Känguru)

Überspringe den nächsten Befehl wenn Register rd=rr; cpse rd,rr; code 0001 00rd dddd rrrr
Überspringe den nächsten Befehl wenn I/O-Reg.bit=0; sbic a,b; code 1001 1001 aaaa abbb
Überspringe den nächsten Befehl wenn I/O-Reg.bit=1; sbis a,b; code 1001 1011 aaaa abbb
Überspringe den nächsten Befehl wenn Register.bit=0; sbrc rr,b; code 1111 110r rrrr 0bbb
Überspringe den nächsten Befehl wenn Register.bit=1; sbrs rr,b; code 1111 111r rrrr 0bbb

Die Verzweigungen (Sprung- und Skip-Befehle) verändern keine Register-Inhalte.

Nun müssen wir die neuen Befehle mal ausprobieren. An anderer Stelle (TPS-Zufall) wurde bereits über Pseudozufallszahlen geschrieben. Wir wollen heute weißes Rauschen erzeugen. Als reine digitale Zufallszahl würde eine Portleitung genügen. Benutzen wir alle 5 Ausgänge, können wir einen DA-Wandler anschließen und erhalten ein 32-stufiges Analogsignal. Eigentlich müssen wir beide Signale durch einen Tiefpass leiten, der die (dominante) Trägerfrequenz wegfiltert. Für den Piezo sparen wir uns das heute.

Die Software macht nichts anderes, als ein Hardwaregenerator (Schieberegister + Exor). Zunächst suchen wir mit 22 Stufen eine günstige Bit-Länge, die in 3 Byte passt. Dazu hängen wir die Register 16, 17 und 18 (per Software) aneinander. Die beiden letzten Bit werden Exor

verknüpft und an der Position R18.5 gespeichert. Im Hilfsregister R19 werden die Berechnungen durchgeführt.

Das Programm in Assembler:
[13uk-prog5.asm]

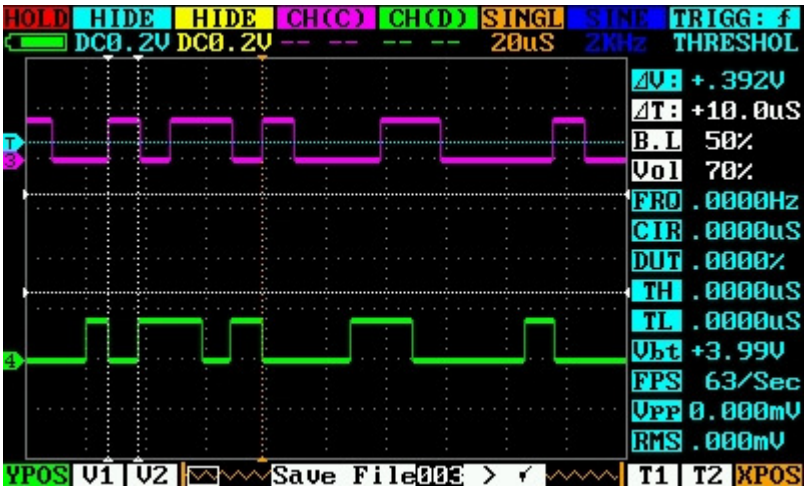
```
.DEVICE ATtiny13       ;für gavrasm, für Symbolische Registerbezeichnungen
(z.B.'DDRB')
.cseg
.org 0
;
; Pseudo-Zufallszahlen, weisses Rauschen durch Simulation eines
; 22-Bit-Schieberegister (mit Exor der beiden rechten Bit)
;
; fo = 1,2 MHz / 11,5 Cyclen / 2 = 52174 Hz
; fu = fo / 2^22-1 = 52174 Hz / 4194303 = 0,012 Hz
;
init:                  ;
       ldi r16,31      ; alle Ports auf Ausgang
       out DdrB,r16    ;
Main:                  ;
       out PortB,r16   ; die letzten 5 Bit ausgeben (1.Ausgabe = 31 =
11111)
       mov r19,r16     ; obwohl alle Bit gespeichert werden, geht es nur um
das letzte Bit
       ror r18         ; alles 1 Bit nach rechts, das rechte in Carry
       ror r17         ; von Carry in Bit 7, alles 1 Bit nach rechts, das
rechte in Carry
       ror r16         ; nocheinmal, nun ist das ursprüngliche Bit 2 in Bit
1
       eor r19,r16     ; EXOR, hier interessiert nur r19,1 xor r16.1
       ror r19         ; das rechte Bit in Carry schieben
       brbs 0,eins     ; wenn C=1 springe nach eins, wenn nicht dann ist in
C=0
       andi r18,31     ; also wird eine '0' in r18.5=0 geschrieben, r18 =
xx0x xxxx
       rjmp Main       ; nächster Schiebetakt
eins:                  ; hier landen wir nur, wenn in C=1 war
       ori r18,32      ; also wird eine '1' in r18.5=1 geschrieben, r18 =
xx1x xxxx
       rjmp Main       ; nächster Schiebetakt
```

Ein Beispiel des 1. Durchlaufs:

```
Beispiel 1. Durchlauf:

r18      r17      r16      r19      Carry
xxxxxxxx xxxxxxxx 00011111 00011111 x

nach ror
xxxxxxxx xxxxxxxx x0001111 00011111 1

nach eor-------------------vvvvvvvv 1
xxxxxxxx xxxxxxxx x0001111 x0010000

nach ror r19
xxxxxxxx xxxxxxxx x0001111 00000000 0

nach andi r18,31
00011111
000xxxxx xxxxxxxx xxxx1111 00000000 0
  ^

wenn Carry=1: nach oii r18,32
00100000
001xxxxx xxxxxxxx xxxx1111 00000000 0
  ^
```

Bitte assemblieren Sie mit gavrasm, weil in BAScom später einige Programme nicht machbar sind. Für einen Piezo hört sich das Signal an jedem Ausgang gleich an.

Am Oszilloskop ist ein Teil des Musters erkennbar, unten PB.0, oben PB.1 und man sieht die Verschiebung um eine Zeiteinheit.



Zum Schluss noch die Abschätzung der Bandbreite, für die, die es genauer wissen wollen. Die obere Frequenz richtet sich nach der AVR-Taktfrequenz / Zeit eines Durchlaufs / 2 wg. Puls+Pause. Aus der Befehlstabelle holen Sie sich die Cyclen für alle Befehle und addieren sie (=11,5 Cylen). Der brbs wird zu Hälfte mit 1Cy und 2Cy berücksichtigt. fo = 1,2MHz / 11,5 Cyclen / 2 = 52174 Hz Die untere Frequenz ist von der Registerlänge abhängig. Nach 2^22-1 wiederholt sich das Muster.
fu = fo / 2^22-1 = 52174Hz / 4194303 = 0,012Hz

Die nutzbare obere Frequenz muss mit einem Tiefpass noch einmal durch 2 geteilt werden = 26087Hz

Die Verwendung eines 5-Bit DA-Wandlers reduziert die Registerlänge um 5 auf 17 oder fo / 32. Die nutzbare fo5 = 26087Hz / 32 = 815 Hz, fu bleibt.

Die bisher gelernten Befehle:

```
.DEVICE ATtiny13              ;für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.CSEG                         ;CodeSegment, muss nicht immer angegeben
werden
.ORG 0                        ;für Adresse '0', muss nicht immer angegeben
werden
main:   ;-----------------------Hauptprogramm
        rjmp main             ;Endlos-Schleife
;--------------------------------
;--------------------------------
Tag01:                        ;
        rjmp Tag01            ;Endlos-Schleife
Tag02:                        ;
        sbi ddrb,1            ;set bit I/O-Reg(Low),Bit
        cbi portb,1           ;clear bit I/O-Reg(Low),Bit
Tag03:                        ;
        out portb,r16         ;I/O-Reg mit dem Byte in R16 laden
        in r16,pinb           ;R16 mit dem Byte des I/O-Reg laden
Tag06:                        ;
        ldi r16,7             ;lade R16 mit 7(dez)(Byte)
        mov r16,r17           ;lade R16 mit dem Byte in R17
        movw r24,r26          ;R24<=R26 und R25<=R27 (Word)
Tag07:                        ;
        add r16,r17           ;R16<=R16+R17          ->SReg
        adiw r24,15           ;R24<=R24+15           ->Sreg
Tag08:                        ;
        sub r16,r17           ;R16=R16-R17           ->SReg
        subi r16,31           ;R16<=R16-31           ->SReg
        sbiw r24,1            ;R24(R25)<=R24(R25)-1  ->SReg
Tag09:                        ;
        inc r16               ;R16<=R16 +1           ->SReg
        dec r16               ;R16<=R16 -1           ->SReg
        neg r16               ;R16<=R16 * -1         ->SReg
        com r16               ;invertiere alle Bits im R16 ->SReg
Tag10:                        ;
        asr r16               ;R16<=R16/2            ->Sreg
        lsr r16               ;R16<=R16/2            ->Sreg
        adc r16,r17           ;R16<=R16+R17+Carry    ->Sreg
        ror r16               ;R16<=R16/2+128*C      ->SReg
Tag11:                        ;
        and r16,r17           ;R16<=R16 AND R17      ->SReg
        eor r16,r17           ;R16<=R16 EXOR R17     ->SReg
        or r16,r17            ;R16<=R16 OR R17       ->SReg
        ori r16,5             ;R16<=R16 OR 0101      ->SReg
        andi r16,5            ;R16<=R16 AND 0101     ->SReg
TAG12:                        ;
        brbc 1,10             ;springe 10 weiter, wenn SREG.1=0 -> Z=0
        brbs 0,-5             ;springe 5 zurück, wenn SReg.0=1 -> C=1
        cpse r16,r17          ;skip if R16=R17
        sbic portb,1          ;skip if PortB.1=0; Einzelbit-Prüfung
        sbis portb,2          ;skip if PortB.2=1; Einzelbit-Prüfung
        sbrc r16,3            ;skip if R16.3=0; Einzelbit-Prüfung
        sbrs r16,4            ;skip if R16.4=1; Einzelbit-Prüfung
```

13.Tag

Bisher haben Sie Befehle für die 'lineare' Programmierung mit Sprüngen kennen gelernt. Um z.B. eine Warteschleife nicht immer wieder im Programm zu wiederholen, wäre es gut, wenn eine Warteschleife von überall aufgerufen werden kann. Das nennt man dann Unterprogramm (Subroutine). Um wieder ins Hauptprogramm zurück zu springen, muss die Adresse gespeichert werden. Dazu benutzen alle CPUs einen Bereich am oberen Ende des Speichers, Stapel (Stack) genannt. Um ihn zu benutzen, muss er am Programmanfang definiert werden. Dazu wird das I/O-Reg: SPL auf das RAM-Ende eingestellt.

ldi r16,ramend; der Assembler ist so nett und holt uns RAMEND=0x9F als Zahl in r16
out spl,r16; spl=0x3D; out 61,159 (dezimal)

(Bei meinen Tiny13 ist SPL schon ab Werk mit 0x9F geladen und ich konnte die beiden Befehle sparen. Das muss aber nicht für jede Produktionscharge gelten.)

Aus dem Hauptprogramm dürfen wir nun ein Unterprogram aufrufen.

rcall +-2047 (die nächste Adresse wird im Stack gespeichert und dann zur Subroutine gesprungen)

Jedes einzelne Unterprogramm muss mit dem RET-Befehl enden.

RET (holt die Adresse vom Stack und arbeitet an der Stelle im Hauptprogramm weiter)

Ist der Stack definiert, kann man noch andere schöne Sachen machen. Bei nur 8 Universalregistern (r16-r23) kann es schon mal eng werden. Benötigt man den Inhalt von r19 später nocheinmal, aber das Register könnte man gerade gut gebrauchen, dann legt man den Inhalt auf den Stack und holt ihn später zurück.

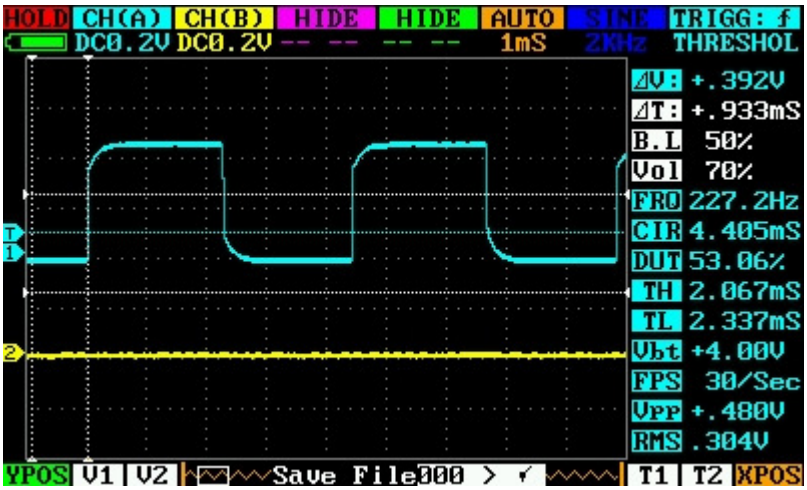
Lege ein Reg. auf den Stack; push rd; code 1001 001d dddd 1111
Lade ein Register vom Stack; pop rd ; code 1001 000d dddd 1111
Die Befehle MÜSSEN paarweise verwendet werden! Sonstkommt der Stack durcheinander.

Aufgabe: Bitte übersetzen Sie das folgende Programm und tippen es ein.

```
.DEVICE ATtiny13       ;für gavrasm, für Symbolische Registerbezeichnungen
(z.B.'DDRB')
.cseg
.org 0
;
; Übung zu rcall und ret, Stack
;
; (#) weil Atmel spl schon ab werk auf 9F gesetzt hat,
; können die 2 ersten Befehle weggelassen werden.
; Das ist bei meinen AtTiny13 tatsächlich so, kann aber abweichen !
;
init:                  ;Vorbereitungen
;       ldi r16,ramend ;der Assembler ist so nett und holt uns RAMEND=0x9F
als Zahl in r16 (#)
;       out spl,r16    ;spl=0x3D; out 61,159 (dezimal) (#)
                       ;
```

```
        sbi DDRB,1      ;PB1=OC0B als Ausgang setzen für den Piezo
                        ;
main:                   ;Hauptprogramm
        rcall sub_wait  ;+1
        rjmp main       ;nochmal
                        ;
sub_wait:               ;Unterprogramm
        adiw r24,1      ;r25/r24 hochzählen
        out portb,r25   ;r25 ausgeben (toggle bei 512 = 1,2MHz/512=2343Hz)
        brbc 1,sub_wait ;springe wenn Z=0
        ret             ;zurück ins Hauptprogramm (wenn R25/r24=0)
;
; Cyclen: (2+3+4)*1   (2+1+2)*512
; 1,2 MHz / 512 Toggle / 2 Puls+Pause / 5 Cyclen = 234Hz
;
```

Das Oszillogramm dazu (227Hz):



14.Tag

Wenn wir einen Stack definiert haben, dann dürfen wir auch die Interrupts benutzen. Am Beispiel der Tastenabfrage soll es erklärt werden. Will man feststellen, ob eine Taste gedrückt oder losgelassen ist, müsste man im Programm ständig den Tasten-Port abfragen, damit ein Ereignis am Port rechtzeitig erkannt wird. Die AVR gestatten es, von einem (mehreren) Port einen Interrupt (=Unterbrechung) auszulösen. Ein Interruptprogramm arbeitet wie ein Unterprogramm! Es wird jedoch NUR von der Hardware aufgerufen, unterbricht das laufende Programm, führt Befehle aus die nur zum Interrupt gehören und kehrt danach (mit RETI) ins laufende Programm zurück. Zur Nutzung sind einige Vorbereitungen notwendig:

An der Adresse '0' muss die (Vector-) Tabelle stehen, die den Aufruf des Interrupt-Programms steuert; vor dem Hauptprogramm (main) müssen alle Vorbereitungen (Init) abgeschlossen sein; und alle Interrupts ermöglicht (enable) werden (SEI).

Da BAScom seine eigene Vectortabelle erzeugt, müssen Programme mit eigener Vectortabelle mit 'gavrasm' assembliert werden.

```
;-------------------------------------------------------------------------
-----
; Beim betätigen oder beim loslassen der Taste an PB.2 wird der Interrupt
(int_pc) ausgelöst.
; Beim drücken wird die LED an PB.1 eingeschaltet und mit RETI ins
Hauptprogramm zurückgekehrt.
; Beim loslassen wird nocheinmal ein Interrupt ausgelöst !, LED=aus und
zurück.
;-------------------------------------------------------------------------
-----
; dieses Programm geht so nicht in BAScom !!!
;-------------------------------------------------------------------------
-----
.DEVICE ATtiny13              ; für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.CSEG                         ; fürs CodeSegment
.org 0                        ; für Adresse 0 übersetzen
;-------------------------------------------------------------------------
-----
        rjmp init             ; Reset vector
                              ; nach einem Reset wird zu init gesprungen
;-------------------------------------------------------------------------
-----
        reti                  ; Int0 interrupt vector
                              ; hierhin wird nach auslösen eines Int0
gesprungen
;-------------------------------------------------------------------------
-----
int_pc:                       ; wird NUR vom Interrupt PCINT aufgerufen !!!
        cbi PORTB,1           ; LED aus
        sbic PINB,2           ; skip if sck=0
        sbi PORTB,1           ; LED ein wenn sck=1
        reti                  ; ins Hauptprogramm zurück
;-------------------------------------------------------------------------
-----
init:                         ; Vorbereitungen
        sbi DDRB,1            ; LED als Ausgang
                              ;
        sbi pcmsk,2           ; Maske für PCINT auf PB.2 setzen
                              ;
        ldi r16,0b00100000    ; - 0 1 - - - - -
        out GIMSK,r16         ;   ^-^- PCINT Enable
                              ;
        sei                   ; Setze Interrupt Flag
;-------------------------------------------------------------------------
-----
main:                         ; Hauptprogramm
        rjmp main             ; Endlosschleife
;-------------------------------------------------------------------------
-----
```

Die Interrupt-Routine wird so schnell abgearbeitet, dass kein Tastendruck verloren geht.
Können Sie das Programm übersetzen und eintippen?

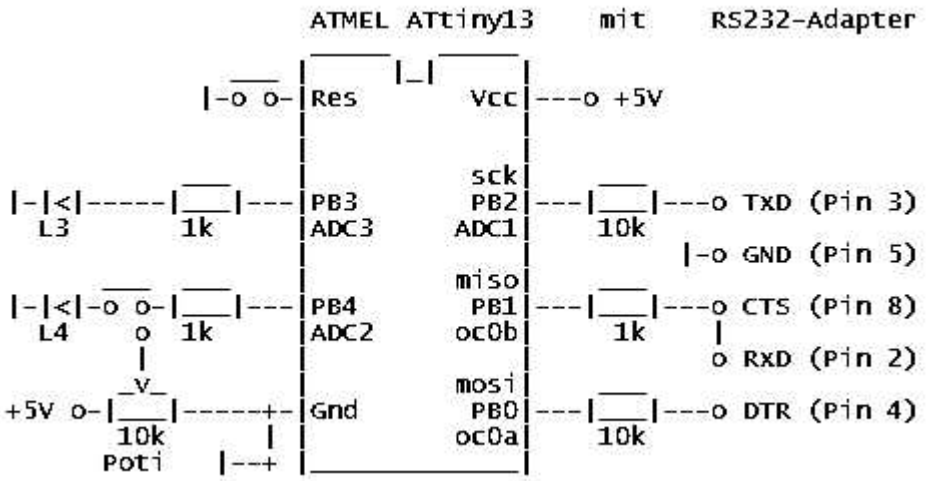
```
:020000020000FC
:10000000 05C0 1895 C19A B299 C198 1895 B99A AA9A 3B
:08001000 00E2 0BBF 7894 FFCF 62
```

```
:00000001FF
12 Befehle = 24 Byte
```

15.Tag

Es fehlt noch einen nützlicher Befehl um ganz wenig Zeit zu verzögern: mach nix; nop; code 0000
(Er kann auch als Platzhalter dienen, wenn später noch etwas eingefügt werden soll.)

Die folgenden Beispiele sind so umfangreich, das Sie die Hilfe eines SPI/ISP-Programmieradapter in Anspruch nehmen sollten. Um das Programm aus dem Lernpaket micro von Herrn Kainka zu benutzen wird ein vereinfachter Adapter benutzt.



Um das nötige .HEX-File zu erhalten, empfehle ich Ihnen den Assembler 'gavrasm', weil es die Fehlermeldungen in deutsch ausgibt. Passend dazu gibt es die IDE 'gavrasmW'.exe für Windows (bis Win7) Im Gegensatz zu anderen Assemblern benötigen Sie keine .INC-Dateien, es genügt '.DEVICE ATtiny13'.

Die Lösung von gestern als .LST-File:

```
gavrasm Gerd's AVR assembler version 3.3 (C)2012 by DG4FAC
-----------------------------------------------------------
Quelldatei:  13uk-prog7.asm
Hexdatei:    13uk-prog7.hex
Eepromdatei: 13uk-prog7.eep
Kompiliert:  17.03.2013, 09:22:19
Durchgang:   2
     1: ;-------------------------------------------------------------------
-------------
     2: ; Beim betätigen oder beim loslassen der Taste an PB.2 wird der
Interrupt (int_pc) ausgelöst.
     3: ; Beim drücken wird die LED an PB.1 eingeschaltet und mit RETI ins
Hauptprogramm zurückgekehrt.
     4: ; Beim loslassen wird nocheinmal ein Interrupt ausgelöst !, LED=aus
und zurück.
```

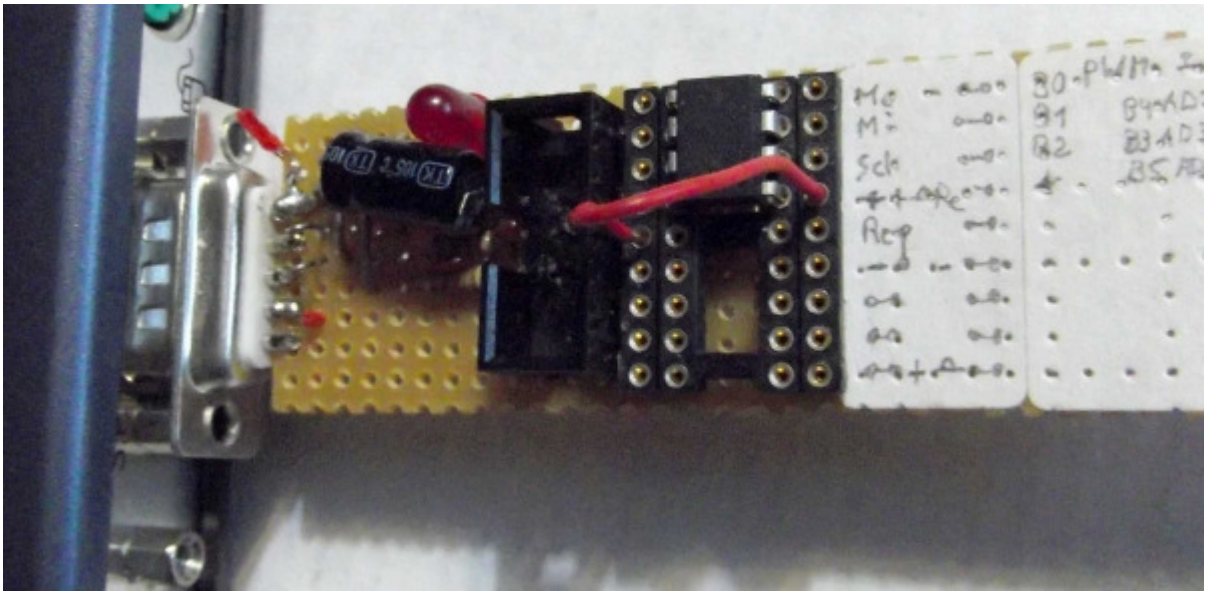
```
     5: ;-------------------------------------------------------------------
-------------
     6: ; dieses Programm geht so nicht in BAScom !!!
     7: ;-------------------------------------------------------------------
-------------
     8: .DEVICE ATtiny13              ; für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
     9: .CSEG                         ; fürs CodeSegment
    10: .org 0                        ; für Adresse 0 übersetzen
    11: ;-------------------------------------------------------------------
-------------
    12: 000000   C005  rjmp init              ; Reset vector
    13:                               ; nach einem Reset wird zu init
gesprungen
    14: ;-------------------------------------------------------------------
-------------
    15: 000001   9518  reti                   ; Int0 interrupt vector
    16:                               ; hierhin wird nach auslösen eines
Int0 gesprungen
    17: ;-------------------------------------------------------------------
-------------
    18: int_pc:                       ; wird NUR vom Interrupt PCINT
aufgerufen !!!
    19: 000002   9AC1  sbi PORTB,1            ; LED ein
    20: 000003   99B2  sbic PINB,2            ; skip if sck=0
    21: 000004   98C1  cbi PORTB,1            ; LED aus wenn sck=1
    22: 000005   9518  reti                   ; ins Hauptprogramm zurück
    23: ;-------------------------------------------------------------------
-------------
    24: init:                         ; Vorbereitungen
    25: 000006   9AB9  sbi DDRB,1             ; LED als Ausgang
    26:                               ;
    27: 000007   9AAA  sbi pcmsk,2            ; Maske für PCINT auf PB.2
setzen
    28:                               ;
    29: 000008   E200  ldi r16,0b00100000     ; - 0 1 - - - - -
    30: 000009   BF0B  out GIMSK,r16          ;   ^-^- PCINT Enable
    31:                               ;
    32: 00000A   9478  sei                    ; Setze Interrupt Flag
    33: ;-------------------------------------------------------------------
-------------
    34: main:                         ; Hauptprogramm
    35: 00000B   CFFF  rjmp main              ; Endlosschleife
    36: ;-------------------------------------------------------------------
-------------
 -> Warnung 001: 1 Symbol(e) definiert, aber nicht benutzt!

Programm        :       12 words.
Konstanten      :        0 words.
Programm Gesamt :       12 words.
Eepromnutzung   :        0 bytes.
Datensegment    :        0 bytes.
Kompilation fertig, keine Fehler.
Kompilation beendet 17.03.2013, 09:22:20
```

Zwischen Adresse und Befehl finden Sie die Hex-Codes. Der Assembler erzeugt auch ein .HEX-File, das über den Adapter und 'LPmikros.exe' geflasht werden kann.

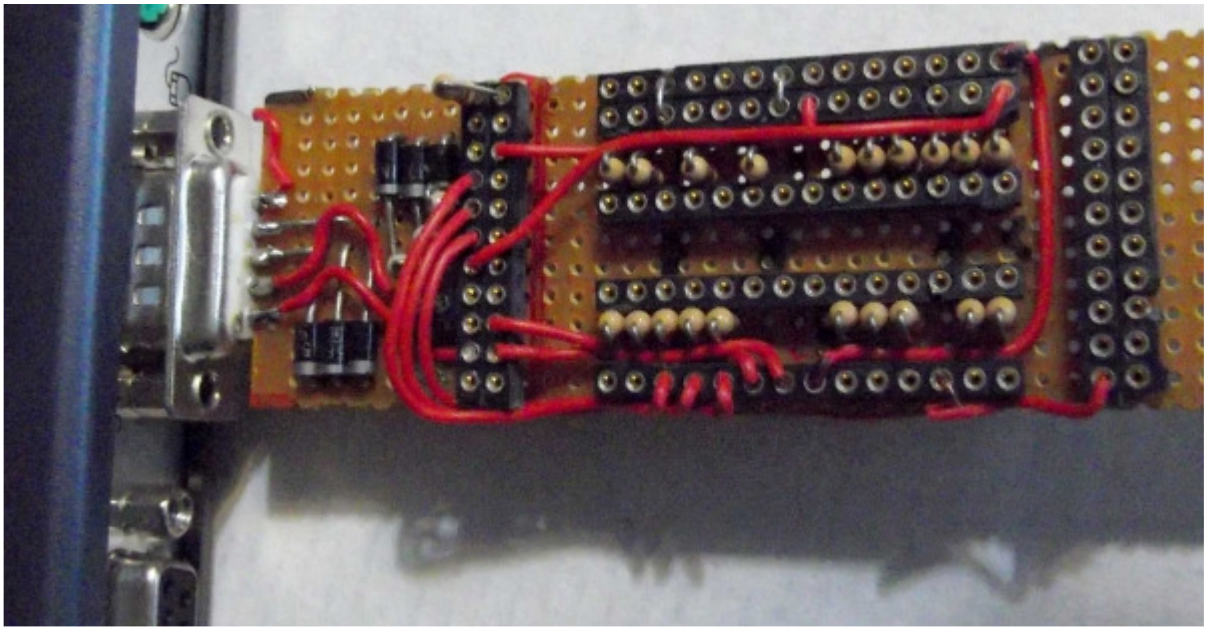
Wenn Sie den Programmier-Adapter nicht immer entfernen wollen, können Sie PB.3 und PB.4 für LED und Taste benutzen. Ändern Sie das Programm entsprechend ab. Der Kondensator und die LED's an 'sck' und 'mosi' müssen unbedingt entfernt werden, wenn der

Adapter benutzt wird. Die LED an miso nur mit eigenem Vorwidertand !!! kann ggf. bleiben.

Assemblieren (ändern) Sie alle bisherigen Programme und werden Sie sicher im Umgang mit 'gavrasm' und 'LPmikro'.



Im Bild sehen Sie einen Selbstbau-Adapter des LPmikro mit zusätzlichem ISP6-Sockel. Noch ein Wort zu den vorgeschlagenen 10k Schutzwiderständen: sie schützen die Eingänge des Tiny13 und sind für USB->RS232-Adapter mit etwa +-6V geeignet. Bei direktem Anschluss an einen PC mit 'echter' RS232-Schnittstelle und +-12V sollten Sie ggf. die Widerstände auf 22k-33k erhöhen. Auch andere Bauteilwerte sind nur Richtwerte. Meine Vorwiderstände für die highefficient LED konnte ich auf 4k7 !!! erhöhen. Im nächsten Bild sehen Sie einen Adapter für AVR bis 28pol (für LPmikro und ISP-Programmer-Mega8/ping-pong-Adapter).



Alle bisher gelernten Befehle:

```
.DEVICE ATtiny13               ;für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.CSEG                          ;CodeSegment, muss nicht immer angegeben
werden
.ORG 0                         ;für Adresse '0', muss nicht immer angegeben
werden
;-------------------------------Programmrumpf
        rjmp init              ;Reset vector
        reti                   ;Int0 interrupt vector
        rjmp int_pc            ;PCINT0 vector
        reti                   ;TC0 overflow vector
        reti                   ;Eeprom ready vector
        reti                   ;Analog comparator int vector
        reti                   ;TC0 CompA vector
        reti                   ;TC0 CompB vector
        reti                   ;WDT vector
        rjmp int_adc           ;ADC conversion complete vector
int_pc: ;-----------------------Pin-Change-Interrupt
        in r13,pinb            ;Lese PortB in R13
        reti                   ;zurück ins Hauptprogramm
int_adc:;----------------------Aufruf NUR, wenn neuer Wert verfügbar
        in r14,ADCh            ;Lese ADC-Wert, obere 8Bit
        reti                   ;zurück ins Hauptprogramm
init:   ;-----------------------nach Reset                  ;
        ldi r16,ramend         ;Stack vorbereiten
        out spl,r16            ;auf RAMEND setzen
init_pc:;----------------------Vorbereitung PCInt
        sbi pcmsk,2            ;Maske auf PB.2
        ldi r16,0b00100000     ;- 0 1 - - - - -
        out GIMSK,r16          ;pcie^=enable
init_adc:;---------------------Vorbereitung adc-Int
        ldi r16,0b00100010     ;  - 0 1 -  - - 1 0
        Out Admux , R16        ;ref-^ ^adlar   ^+^-adc2
        ldi r16,0b00000000     ;  - - - -  - 0 0 0
        Out Adcsrb , R16       ;             ^+^+^-Free Run
        ldi r16,0b11000111     ;    1 1 0 0  0 1 1 1
        Out Adcsra , R16       ;ena-^ ^-start  ^+^+^-prescale
init_timer:;-------------------Vorbereitung Timer
        sbi DDRB,1             ;Piezo als Ausgang definieren
        ldi r16,0b00010010     ;  0 0 0 1  - - 1 0
        Out Tccr0a , R16       ;toggleB-^      ^+^=ctc =Timer als Zähler
        ldi r16,0b00000010     ;  0 0 - -  0 0 1 0
        Out Tccr0b , R16       ;             ^+^+^-prescale 1/8
init_sleep:;-------------------Vorbereitung Sleep-Mode
        ldi r16,0b00100000     ;   0 0 1 0 0 0 0 0 SLEEP-Mode=Idle
vorbereiten
        out MCUCR,r16          ;enable-^ ^-^-Mode 00=idle (10=power down)
;-------------------------------
        sei                    ;Set Enable Interrupt (-Flag)
main:   ;-----------------------Hauptprogramm
        rjmp main              ;Endlos-Schleife
;-------------------------------Subroutines
sub_wait1:                     ;Unterprogramm Wait Byte
        (3+3*256-1+4)=645us
        subi r23,1             ;R23 255-0                              (1)
        brbc 1,sub_wait1       ;springe wenn Z=0                       (1/2)
        ret                    ;zurück ins Hauptprogramm (wenn R23=0)  (4)
;-------------------------------
```

```
sub_wait2:                      ;Unterprogramm Wait Word
        (3)(3+4*65536-1+4)=0,22s
        sbiw r24,1              ;r25/r24 65535-0                        (2)
        brbc 1,sub_wait2        ;springe wenn Z=0                       (1/2)
        ret                     ;zurück ins Hauptprogramm (wenn R25/r24=0)(4)
;--------------------------------End Program
;--------------------------------
Tag01:                          ;
        rjmp Tag01              ;Endlos-Schleife
Tag02:                          ;
        sbi ddrb,1              ;set bit I/O-Reg(Low),Bit
        cbi portb,1             ;clear bit I/O-Reg(Low),Bit
Tag03:                          ;
        out portb,r16           ;I/O-Reg mit dem Byte in R16 laden
        in r16,pinb             ;R16 mit dem Byte des I/O-Reg laden
Tag06:                          ;
        ldi r16,7               ;lade R16 mit 7(dez)(Byte)
        mov r16,r17             ;lade R16 mit dem Byte in R17
        movw r24,r26            ;R24<=R26 und R25<=R27 (Word)
Tag07:                          ;
        add r16,r17             ;R16<=R16+R17           ->SReg
        adiw r24,15             ;R24<=R24+15            ->Sreg
Tag08:                          ;
        sub r16,r17             ;R16=R16-R17            ->SReg
        subi r16,31             ;R16<=R16-31            ->SReg
        sbiw r24,1              ;R24(R25)<=R24(R25)-1   ->SReg
Tag09:                          ;
        inc r16                 ;R16<=R16 +1            ->SReg
        dec r16                 ;R16<=R16 -1            ->SReg
        neg r16                 ;R16<=R16 * -1          ->SReg
        com r16                 ;invertiere alle Bits im R16 ->SReg
Tag10:                          ;
        asr r16                 ;R16<=R16/2             ->Sreg
        lsr r16                 ;R16<=R16/2             ->Sreg
        adc r16,r17             ;R16<=R16+R17+Carry     ->Sreg
        ror r16                 ;R16<=R16/2+128*C       ->SReg
Tag11:                          ;
        and r16,r17             ;R16<=R16 AND R17       ->SReg
        eor r16,r17             ;R16<=R16 EXOR R17      ->SReg
        or r16,r17              ;R16<=R16 OR R17        ->SReg
        ori r16,5               ;R16<=R16 OR 0101       ->SReg
        andi r16,5              ;R16<=R16 AND 0101      ->SReg
TAG12:                          ;
        brbc 1,10               ;springe 10 weiter, wenn SREG.1=0 -> Z=0
        brbs 0,-5               ;springe 5 zurück, wenn SReg.0=1 -> C=1
        cpse r16,r17            ;skip if R16=R17
        sbic portb,1            ;skip if PortB.1=0; Einzelbit-Prüfung
        sbis portb,2            ;skip if PortB.2=1; Einzelbit-Prüfung
        sbrc r16,3              ;skip if R16.3=0; Einzelbit-Prüfung
        sbrs r16,4              ;skip if R16.4=1; Einzelbit-Prüfung
Tag13:                          ;
        rcall 9                 ;Unterprogrammaufruf
        ret                     ;Rücksprung ins Hauptprogramm
        push r16                ;(spl)<=R16
        pop r16                 ;R16<=(spl)
Tag14:                          ;
        sei                     ;set enable interrupt
        reti                    ;Rücksprung vom Interrupt ins Hauptprogramm
Tag15:
        nop                     ;no operation
```

16.Tag

Bei den Beispielen mit BAScom wurde stillschweigend vorausgesetzt, dass Sie über einen Adapter (STK500 oder USBASP) verfügen. Sie können jedoch die .HEX-Files auch überkreuz zum Flashen benutzen. Das bedeutet:
in BAScom programmiert, .HEX-File mit LPmikro und RS232-Adapter geflasht oder
in gavrasm programmiert, .HEX-File in BAScom laden und mit STK500/USBASP flashen.

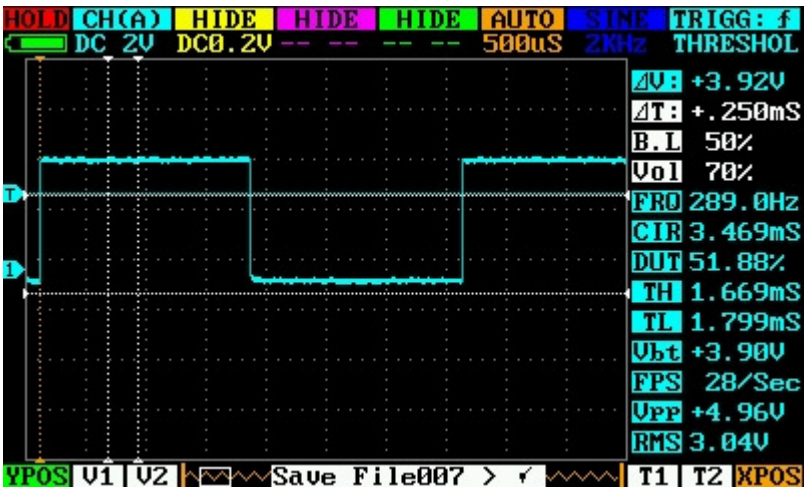
Für die Verwendung mit der Tasten-Eingabe wurden möglichst wenige Befehle ausgesucht. Mit dem SPI/ISP- Adapter kann etwas sauberer programmiert werden. Im nächsten Beispiel soll der ADC-Wandler abgefragt werden. Damit wir auch was merken, wird damit die Tonhöhe am Piezo verändert.

```
; Übung zum AD-Wandler,
;
; init_timer stellt PB.1 auf toggle, CTC-Mode (OCR0A) und den Prescaler
ein,
; init_adc stellt Uref auf Vcc, adlar=1 => 8Bit in ADCH, ADC(2), free run
und Autostart.
; Das Hauptprogramm tut dann nichts mehr, Wandlung und Ausgabe geschieht in
int_adc.
;
; Nach jeder Wandlung wird int_adc ausgelöst, der Wert mit 7 maskiert und
in OCR0A geladen.
; Da die Frequenz eine 1/x-Funktion ist, entspricht Uadc=0V der höchsten
Frequenz.
; Die maskierung mit 7 reduziert die Einstellung auf 32 Stufen und die
Werte auf 7..255.
;
;
;
.DEVICE ATtiny13              ; für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.cseg                         ; für den Flash-Speicher
.org 0                        ; für Adresse 0
        rjmp init             ; Reset vector
        reti                  ; Int0 interrupt vector
        reti                  ; PCINT0 vector
        reti                  ; TC0 overflow vector
        reti                  ; Eeprom ready vector
        reti                  ; Analog comparator int vector
        reti                  ; TC0 CompA vector
        reti                  ; TC0 CompB vector
        reti                  ; WDT vector
        rjmp int_adc          ; ADC conversion complete vector
Int_adc:                      ;
        in r16,adch           ; obere 8Bit holen
        ori r16,0b00000111    ; keine Zahl kleiner 7
        Out Ocr0a , r16       ; compare A laden
        reti                  ; Return from Interrupt
init:                         ;
        ldi r16,LOW(RAMEND)   ; Stapelzeiger auf Ende SRAM
```

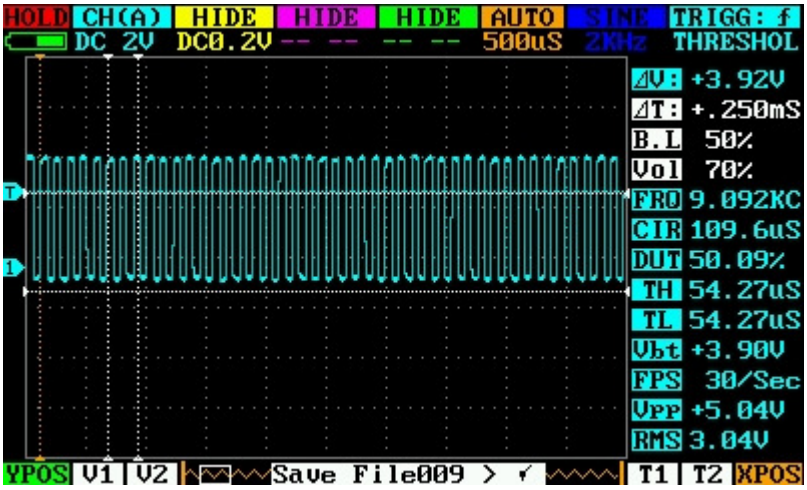
```
        Out Spl , R16           ;
        sbi DDRB,1              ; Piezo
init_timer:                     ;
        ldi r17,0b00010010      ;   0 0 0 1 0 0 1 0
        Out Tccr0a , R17        ;   ocA ocB     ^---CTC
        ldi r17,0b00000011      ;   0 0 0 0 0 0 1 1
        Out Tccr0b , R17        ;               ^-^-prescaler 1-5
init_adc:                       ;
        ldi r16,0b00100010      ;   0 0 1 0 0 0 1 0
        Out Admux , R16         ; ref-^ ^-adlar ^-^-adc(0-3)
        ldi r16,0b00000000      ;   0 0 0 0 0 0 0 0
        Out Adcsrb , R16        ;             ^-^-^-free run
        ldi r16,0b11101111      ;   1 1 1 0 1 1 1 1
        Out Adcsra , R16        ;             ^-^-^-prescaler, ADC starten,
Autostart
        sei                     ; Set Interrupt Flag
main:                           ;
        rjmp main               ; Endlosschleife
;
```

Mit ADLAR=1 wird der ADC-Wert auf die oberen 8Bit im ADCh-Register eingestellt. Vom Interrupt-Programm wird ein neuer ADC-Wert direkt (ohne Hauptprogramm) in den Vergleicher (Compare) des Timer geladen. Im CTC-Mode zählt der Timer hoch, bis der Wert im Vergleichsregister (OCR0A=ADCH) erreicht ist. Dann wird PortB (getoggelt) umgeschaltet (0->1 oder 1->0). Bei kleinen ADC-Werten wird früh getoggelt (hohe Frequenz). Damit die Frequenzen nicht zu hoch werden, wird jeder ADC-Wert mit 7 Oder verknüpft. Dadurch kommen in OCR0A nur Werte von 7 bis 255 an.

Die Frequenz: 9,6MHz / 8(devide) / 8(prescale) / 2(toggle) / OCR0A = 75000Hz / 255 = 294Hz



75000Hz / 8 = 9375Hz



Die hohe Frequenz wird nicht ganz erreicht, weil die Ladebefehle nicht berücksichtigt sind.

17.Tag

Der aufmerksame Leser fragt sich, wo das Programm 2 geblieben ist. Durch auslagern von Aufgaben ist das Hauptprogramm häufig arbeitslos. In diesem Fall ist der Sleep-Mode ideal.

Nachdem ich mir eine Übung ausgedacht hatte, schlief mein Tiny13 so tief und fest, das ich ihn mit BAScom nicht wecken konnte. Grund war der fehlende Interrupt. Mit dem aktive Adc kann das nicht mehr passieren.

```
init_sleep:          ; SLEEP-Mode=Idle vorbereiten
   ldi r16,0b00100000    ;   0 0 1 0 0 0 0 0
   out MCUCR,r16       ; enable-^ ^-^-Mode 00=idle (10=power down)
main:               ;
   sleep          ; würde eigentlich reichen, aber wenn er
   nop          ; wachgeworden !
   rjmp main       ; muss er wieder schlafen gelegt werden
```

Im Idle-Mode wird nur die Befehlsausführung gestoppt, Timer, ADC, Ports usw. arbeiten weiter. Im Power-Down-Mode bleiben nur die Hardware-Interrupts (Reset, Int0) wach, alles andere wird stromlos.

```
; Übung zum Sleep-Mode Idle, mit AD-Wandler,
;
; init_timer stellt PB.1 auf toggle, CTC-Mode (OCR0A) und den Prescaler
ein,
; init_adc stellt Uref auf Vcc, adlar=1 => 8Bit in ADCH, ADC(2), free run
und Autostart.
; Das Hauptprogramm tut dann nichts mehr, Wandlung und Ausgabe geschieht in
int_adc.
;
; Nach jeder Wandlung wird int_adc ausgelöst, der Wert mit 7 maskiert und
```

```
in OCR0A geladen.
; Da die Frequenz eine 1/x-Funktion ist, entspricht Uadc=0V der höchsten
Frequenz.
; Die maskierung mit 7 reduziert die Einstellung auf 32 Stufen und die
Werte auf 7..255.
;
; Die Frequenz: 9,6MHz / 8(devide) / 64(prescale) / 2(toggle) / OCR0A =
9375Hz / OCR0A
; PPB.1=Piezo, PB.4=Adc(2)=Poti
;
.DEVICE ATtiny13              ; für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.cseg                         ; für den Flash-Speicher
.org 0                        ; für Adresse 0
        rjmp init             ; Reset vector
        reti                  ; Int0 interrupt vector
        reti                  ; PCINT0 vector
        reti                  ; TC0 overflow vector
        reti                  ; Eeprom ready vector
        reti                  ; Analog comparator int vector
        reti                  ; TC0 CompA vector
        reti                  ; TC0 CompB vector
        reti                  ; WDT vector
        rjmp int_adc          ; ADC conversion complete vector
Int_adc:                      ;
        in r16,adch           ; obere 8Bit holen
        ori r16,0b00000111    ; keine Zahl kleiner 7
        Out Ocr0a , r16       ; compare A laden
        reti                  ; Return from Interrupt
init:                         ;
        ldi r16,LOW(RAMEND)   ; Stapelzeiger auf Ende SRAM
        Out Spl , R16         ;
        sbi DDRB,1            ; Piezo
init_timer:                   ;
        ldi r17,0b00010010    ;   0 0 0 1 0 0 1 0
        Out Tccr0a , R17      ;   ocA ocB     ^---CTC
        ldi r17,0b00000011    ;   0 0 0 0 0 0 1 1
        Out Tccr0b , R17      ;               ^-^-prescaler 1-5
init_adc:                     ;
        ldi r16,0b00100010    ;   0 0 1 0 0 0 1 0
        Out Admux , R16       ; ref-^ ^-adlar ^-^-adc(0-3)
        ldi r16,0b00000000    ;   0 0 0 0 0 0 0 0
        Out Adcsrb , R16      ;             ^-^-^-free run
        ldi r16,0b11101111    ;   1 1 1 0 1 1 1 1
        Out Adcsra , R16      ;             ^-^-^-prescaler, ADC starten,
Autostart
        sei                   ; Set Interrupt Flag
init_sleep:                   ; SLEEP-Mode=Idle vorbereiten
        ldi r16,0b00100000    ;    0 0 1 0 0 0 0 0
        out MCUCR,r16         ; enable-^ ^-^-Mode 00=idle (10=power down)
main:                         ;
        sleep                 ; würde eigentlich reichen, aber wenn er
        nop                   ; wachgeworden !
        rjmp main             ; muss er wieder schlafen gelegt werden
;
```

Der Stromverbrauch sinkt in diesem Beispiel nur wenig. Bei anderen Anwendungen und Batteriebetrieb kann Sleep sehr sinnvoll sein.

Bitte gewöhnen Sie sich an eine ausführliche Kommentierung. Schon nach wenigen Tagen/Wochen weiss man nicht mehr warum es so und nicht anders programmiert ist.

Trauen Sie sich, den Sleep-Mode in andere Programme einzubauen? Falls BAScom nicht reagiert, keine Panik, mit der Tasten-Sequenz: Prog Enable und Chip erase vom 5.Tag holen Sie ihn zurück.

18.Tag

Heute können Sie kreativ werden. Entwickeln Sie ein Programm, mit dem Sie die oberen 4 Bit des AD-Wandlers auf 4 LEDs an PortB.0 bis PortB.3 ausgeben.

Viel Spaß!

Inhalt:

19.Tag

Das Problem von gestern kann auf verschiedene Weisen gelöst werden. Hier nur zwei Beispiele:

```
; Lösung zu Tag 18 (es gibt natürliche mehrere Ansätze)
;
; init_adc stellt Uref auf Vcc, adlar=1 => 8Bit in ADCH, ADC(2), free run
und Autostart.
; Das Hauptprogramm tut dann nichts mehr, Wandlung und Ausgabe geschieht in
int_adc.
;
; Nach jeder Wandlung wird int_adc ausgelöst, der Wert nach links
verschoben und in PortB geladen.
;
.DEVICE ATtiny13              ; für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.cseg                         ; für den Flash-Speicher
.org 0                        ; für Adresse 0
        rjmp init             ; Reset vector
        reti                  ; Int0 interrupt vector
        reti                  ; PCINT0 vector
        reti                  ; TC0 overflow vector
```

```
        reti                    ; Eeprom ready vector
        reti                    ; Analog comparator int vector
        reti                    ; TC0 CompA vector
        reti                    ; TC0 CompB vector
        reti                    ; WDT vector
        rjmp int_adc            ; ADC conversion complete vector
Int_adc:                        ;
        in r16,adch             ; obere 8Bit holen
        lsr r16                 ; nach unten verschieben 0xxxx___
        lsr r16                 ; nach unten verschieben 00xxxx__
        lsr r16                 ; nach unten verschieben 000xxxx_
        lsr r16                 ; nach unten verschieben 0000xxxx
        Out portb,r16           ; compare A laden
        reti                    ; Return from Interrupt
init:                           ;
        ldi r16,LOW(RAMEND)     ; Stapelzeiger auf Ende SRAM
        Out Spl , R16           ;
        ldi r16,15              ; PB.0-3 auf Ausgang
        out DDRB,r16            ;
init_adc:                       ;
        ldi r16,0b00100010      ;   0 0 1 0 0 0 1 0
        Out Admux , R16         ; ref-^ ^-adlar ^-^-adc(0-3)
        ldi r16,0b00000000      ;   0 0 0 0 0 0 0 0
        Out Adcsrb , R16        ;             ^-^-^-free run
        ldi r16,0b11101111      ;   1 1 1 0 1 1 1 1
        Out Adcsra , R16        ;             ^-^-^-prescaler, ADC starten,
Autostart
        sei                     ; Set Interrupt Flag
main:                           ;
        rjmp main               ; Endlosschleife
;


' 13UK, Programm 10
' Die oberen 4 Bit des Adc werden binär an PB.0-3 angezeigt
'
'-------------------------------------------------------------------------
-----
' alle Fuse-Byte ab Werk
' kompiliert mit Bascom-Demo 2.0.7.1 (sollte auch mit 2.0.7.5
funktionieren)
'-------------------------------------------------------------------------
-----
$regfile = "attiny13.dat"
$crystal = 1200000            ' 9,6MHz/8
$hwstack = 8

'init
Config Adc = Single , Prescaler = Auto
Ddrb = 15                     'PortB.1 - PortB.3
Dim Temp As Word

Do                            'Hauptprogramm
   Temp = Getadc(2)           'Wert holen
   Temp = Temp / 64           'auf 4 Bit reduzieren
   Portb = Temp               'und ausgeben
Loop                          'nochmal

End
' End Program
```

Der Timer kann nicht nur als Zähler/Teiler arbeiten, sondern auch PWM-Signale erzeugen. PWM bedeutet das Puls und Pause nicht die gleiche Breite haben. Beim Piezo würden wir die Unterschiede kaum hören. Eine LED zeigt jedoch breitere Impulse (schmalere Pausen) durch grössere Helligkeit an.

Der Tiny13 erlaubt 2 PWM-Mode, die sehr ähnlich sind. Im Fast-PWM (wie der Name schon sagt) zählt der Timer immer von 0-255 und dann wieder von 0-255. Bei erreichen des Vergleichswertes wird der Ausgang umgeschaltet.

f(fast) = fosc / prescaler / 256

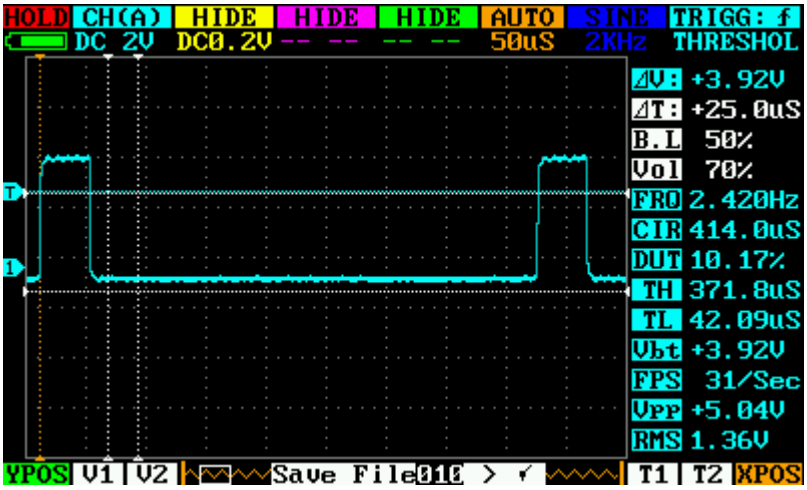
Beim Phase-Correct-Mode wird der Timer von 0-255 gezählt und dann wieder zurück 255-0. Bei erreichen des Vergleichswertes wird der Ausgang umgeschaltet.

f(phase) = fosc / prescaler / 510 (510 da 0 und 255 je nur einmal vorkommt)

Phase Correct PWM:

```
; Phase-Correct-PWM
;
; f(phase)=1,2MHz / prescaler / 510 = 2353Hz
;
.DEVICE ATtiny13              ;für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.cseg
.org 0
                              ;Vector-Tabelle
        rjmp init             ;Reset vector
        reti                  ;Int0 interrupt vector
        reti                  ;PCINT0 vector
        reti                  ;TC0 overflow vector
        reti                  ;Eeprom ready vector
        reti                  ;Analog comparator int vector
        reti                  ;TC0 CompA vector
        reti                  ;TC0 CompB vector
        reti                  ;WDT vector
        rjmp int_adc          ;ADC conversion complete vector
Int_adc:                      ;Aufruf NUR, wenn neuer Wert verfügbar
        in r14,ADCh           ;Lese ADC-Wert, obere 8Bit
        Out Ocr0b,R14         ;Pulslänge
        reti                  ;zurück ins Hauptprogramm
init:                         ;Vorbereitungen
        sbi ddrb,1            ;Piezo als Ausgang definieren
        ldi r16,ramend        ;Stack vorbereiten
        out spl,r16           ;
init_pwm:                     ;
        ldi r16,0b00100001    ;  0 0 1 0  - - 0 1
        Out Tccr0a , R16      ;A-^-^ ^-^-B    ^+^=PWM
        ldi r16,0b00000001    ;  0 0 - -  0 0 0 1
        Out Tccr0b , R16      ;       PWM-^ ^+^+^-prescale
init_adc:                     ;
        ldi r16,0b00100010    ;  - 0 1 -  - - 1 0
        Out Admux , R16       ;ref-^ ^adlar   ^+^-adc2
        ldi r16,0b00000000    ;  - - - -  - 0 0 0
```

```
        Out Adcsrb , R16        ;              ^+^+^-Free Run
        ldi r16,0b11101111      ;    1 1 1 0  1 1 1 1
        Out Adcsra , R16        ;ena-^ ^-start  ^+^+^-prescale
        sei                     ;Set Enable Interrupt (-Flag)
main:                           ;Hauptprogramm
        rjmp main               ;
```



Im praktischen Gebrauch sind beide Modes kaum zu unterscheiden.

20.Tag

Die Lösung in Basic macht fast das gleiche. Die Abweichung der Frequenz kommt daher, weil in BAScom Getadc(2) nicht per Interrupt sondern einzeln aufgerufen wird.

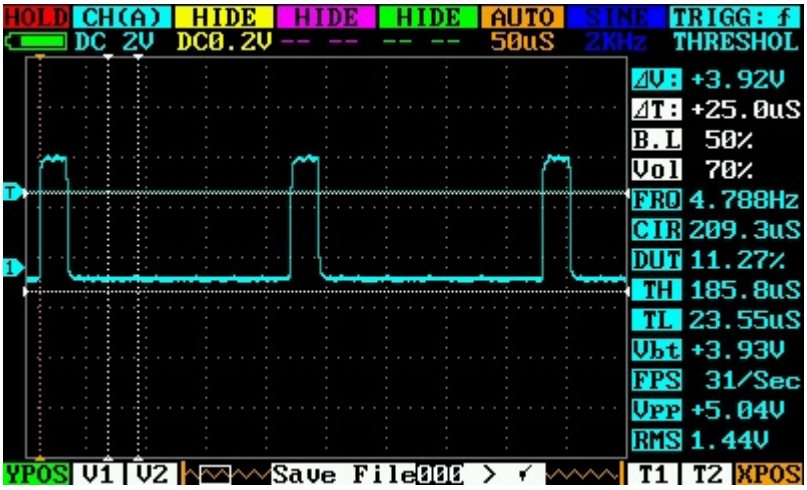
```
' 13UK, Programm 11 PWM
' Die Helligkeit der LED an PB.1 wird mit dem Poti von 0%-100% eingestellt
'
'-------------------------------------------------------------------------
-----
' alle Fuse-Byte ab Werk
' kompiliert mit Bascom-Demo 2.0.7.1 (sollte auch mit 2.0.7.5
funktionieren)
'-------------------------------------------------------------------------
-----
$regfile = "attiny13.dat"
$crystal = 1200000              ' 9,6MHz/8
$hwstack = 8

'init
Config Adc = Single , Prescaler = Auto

Config Timer0 = Pwm , Compare B Pwm = Clear Up , Prescale = 1

Ddrb = 2                        'PortB.1
Dim Temp As Word
```

```
Do                              'Hauptprogramm
   Temp = Getadc(2)             'Wert holen
   Temp = Temp / 4              'auf 8 Bit reduzieren
   Pwm0b = Temp                 'und ausgeben
Loop                            'nochmal
End
' End Program
```

Fast-PWM:

```
; Fast-PWM
;
; f(fast)=1,2MHz / prescaler / 256 = 4687,5Hz
;
.DEVICE ATtiny13                ;für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.cseg
.org 0
                                ;Vector-Tabelle
        rjmp init               ;Reset vector
        reti                    ;Int0 interrupt vector
        reti                    ;PCINT0 vector
        reti                    ;TC0 overflow vector
        reti                    ;Eeprom ready vector
        reti                    ;Analog comparator int vector
        reti                    ;TC0 CompA vector
        reti                    ;TC0 CompB vector
        reti                    ;WDT vector
        rjmp int_adc            ;ADC conversion complete vector
Int_adc:                        ;Aufruf NUR, wenn neuer Wert verfügbar
        in r14,ADCh             ;Lese ADC-Wert, obere 8Bit
        Out Ocr0b,R14           ;Pulslänge
        reti                    ;zurück ins Hauptprogramm
init:                           ;Vorbereitungen
        sbi ddrb,1              ;Piezo als Ausgang definieren
        ldi r16,ramend          ;Stack vorbereiten
        out spl,r16             ;
init_pwm:                       ;
        ldi r16,0b00100011      ;  0 0 1 0  - - 1 1
        Out Tccr0a , R16        ;A-^-^ ^-^-B    ^+^=Fast-PWM
        ldi r16,0b00000001      ;  0 0 - -  0 0 0 1
        Out Tccr0b , R16        ;       PWM-^ ^+^+^-prescale
init_adc:                       ;
        ldi r16,0b00100010      ;  - 0 1 -  - - 1 0
        Out Admux , R16         ;ref-^ ^adlar   ^+^-adc2
        ldi r16,0b00000000      ;  - - - -  - 0 0 0
        Out Adcsrb , R16        ;             ^+^+^-Free Run
        ldi r16,0b11101111      ;    1 1 1 0  1 1 1 1
        Out Adcsra , R16        ;ena-^ ^-start  ^+^+^-prescale
        sei                     ;Set Enable Interrupt (-Flag)
main:                           ;Hauptprogramm
        rjmp main               ;
```

Falls der Piezo noch angeschlossen ist, hören Sie den Unterschied.



Der AD-Wandler und sein Innenwiderstand: Am Eingang sorgt eine Sample and Hold Schaltung dafür, dass der Messwert während der Wandlungszeit gespeichert bleibt. Wenn der Wandler eine Messung nach der anderen macht, ergibt sich ein Innenwiderstand von ~1-10MOhm. Ruft man jedoch getadc() etwa im Sekundentakt auf, erhöht sich der Innerwiderstand um den Faktor 1000! Auf diese Weise kann man z.B. aus Fotodioden auch das letzte Elektron rausholen.

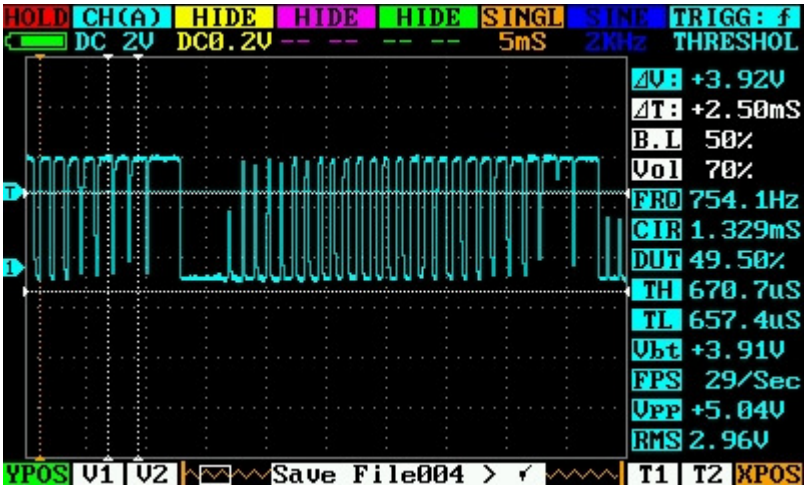
21.Tag

Die letzte Übung soll zeigen, wie ein Sägezahn mit PWM erzeugt wird. Hier kann einfach ein Register hochgezählt werden. Andere Kurvenformen werden mit vorberechneten Werten erzeugt.

```
; Sägezahn durch Pulsbreiten-Modulation
;
; Fast-PWM
;
; f(fast)=1,2MHz / prescaler / 256 = 4687,5Hz
;
.DEVICE ATtiny13               ;für gavrasm, für Symbolische
Registerbezeichnungen (z.B.'DDRB')
.cseg
.org 0
init:                          ; Vorbereitungen
        sbi ddrb,1             ; Ausgang definieren
;       ldi r16,ramend         ; Stack vorbereiten
;       out spl,r16            ;
init_pwm:                      ;
        ldi r16,0b00100011     ;   0 0 1 0  - - 1 1
        Out Tccr0a , R16       ; A-^-^ ^-^-B    ^+^=Fast-PWM
        ldi r16,0b00000001     ;   0 0 - -  0 0 0 1
        Out Tccr0b , R16       ;        PWM-^ ^+^+^-prescale
main:                          ; Hauptprogramm
```

```
        adiw r24,1              ; r25/r24 hochzählen
        out ocr0b,r25           ; r25 ausgeben, ergibt Sägezahn durch
aufwärtszählen
        rjmp main               ;
```

(Für das Oszilloskop wurde die devide-Fuse abgestellt =9,6MHz.)



Sie haben nun alles gelernt, was ich geplant hatte. Ich hoffe Sie hatten viel Spaß, auch wenn ich viel von Ihnen verlangt habe. Auf jeden Fall können Sie nun am CONTEST teilnehmen.

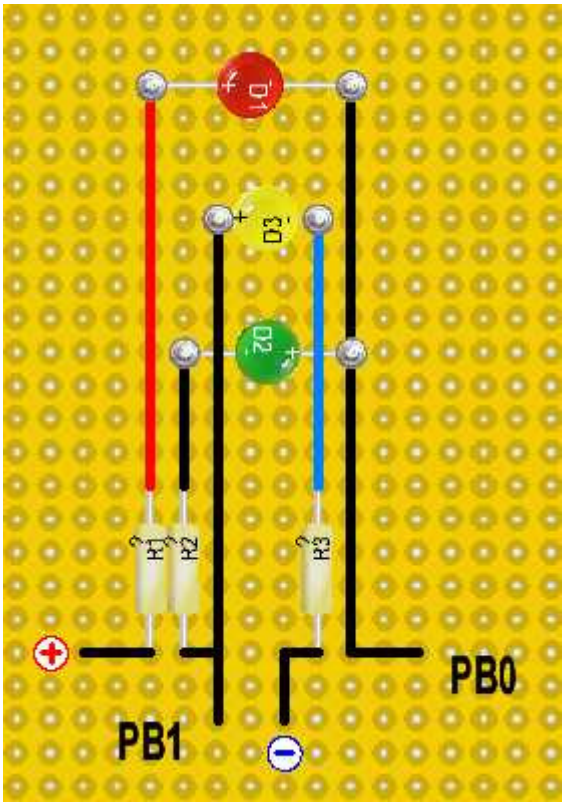
Scheuen Sie sich nicht, Ihre Problemlösungen vorzustellen, es gibt keine 'schlechten' Programme. Der Olympische Gedanke zählt.

Zum Schluss noch zwei Ideen und ich hoffe Sie bleiben am 'ASM-Ball'.

Würfel: Die Taste ist an PB.4 nach gnd geschaltet, die LED für 1, 2(+3), 2(4+6), 2(6) an PB.0-3.

```
;
;Würfel
;
;LED an PB0-PB3 gegen GND
;Taste an PB4 gegen GND
;
; ---          ---
;|PB1|        |PB2| in Reihe
; ---          ---
; ---    ---   ---
;|PB3| |PB0| |PB3| in Reihe
; ---    ---   ---
; ---          ---
;|PB2|        |PB1| in Reihe
; ---          ---
;
.DEVICE ATtiny13        ;für gavrasm, für Symbolische Registerbezeichnungen
(z.B.'DDRB')
.cseg
.org 0

init:
;       ldi r16,ramend ;der Assembler ist so nett und holt uns RAMEND=0x9F
```

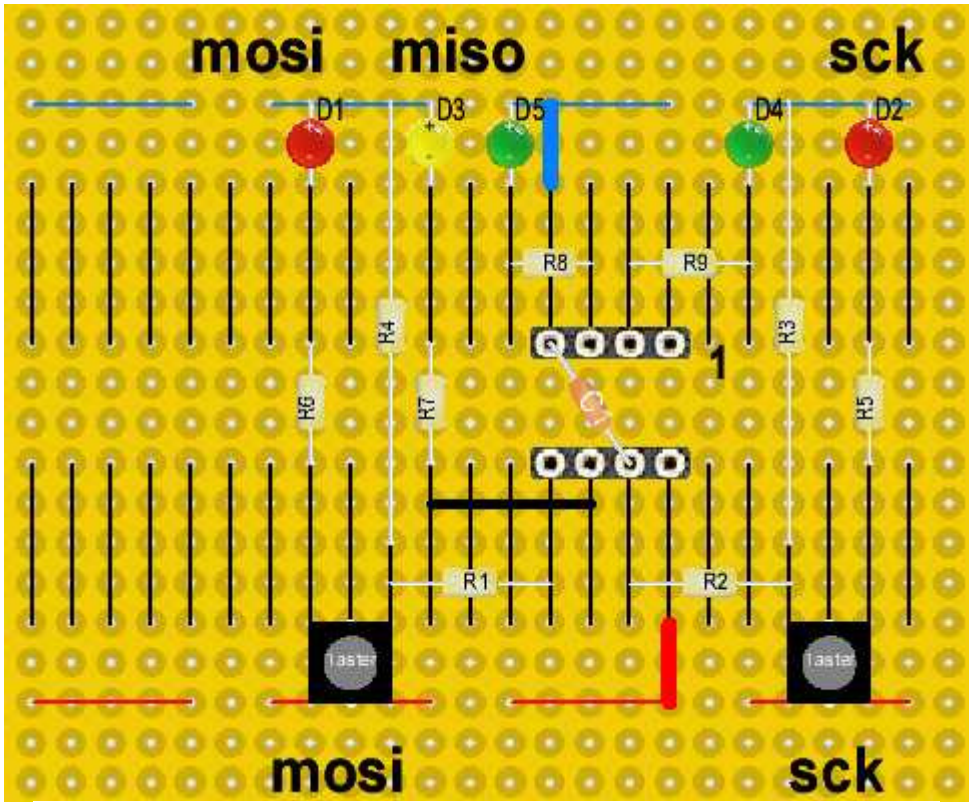
```
als Zahl in r16 (#)
;       out spl,r16     ;spl=0x3D; out 61,159 (dezimal) (#)
        ldi r16,15      ;PB.0 bis PB.3 als Ausgang
        out ddrb,r16    ;0b00001111
        sbi portb,4     ;Pullup für Taste
main:
eins:
        ldi r16,1       ;0b0001
        rcall sub_wait  ;
zwei:
        ldi r16,2       ;0b0010
        rcall sub_wait  ;
drei:
        ldi r16,5       ;0b0101
        rcall sub_wait  ;
vier:
        ldi r16,6       ;0b0110
        rcall sub_wait  ;
funf:
        ldi r16,7       ;0b0111
        rcall sub_wait  ;
sechs:
        ldi r16,14      ;0b1110
        rcall sub_wait  ;
        rjmp main       ;alles nochmal
                        ;
sub_wait:               ; Unterprogramm
        out portb,r16   ; ausgeben
        adiw r24,8      ; r25/r24 hochzählen
        brbc 1,sub_wait ; springe wenn Z=0
        sbis PINB,4     ; Ueberspringe Befehl wenn PB4 eins
        ret             ; zurück ins Hauptprogramm (wenn R25/r24=0)
sub_wait1:              ;
        sbic PINB,4     ; Ueberspringe Befehl wenn PB4 null
        rjmp sub_wait1  ;
        ret             ;
ende:
```



Ampel: Die Zeiten können mit den 4 Ladebefehlen (ldi R22,x) geändert werden.

```
;
;Ampel an PB0-PB3 (by Heinz D.; ausser Konkurenz)
;
;Um mit je 2 Leitungen pro Ampel auszukommen, müssen
;die LED in bestimmter Weise in Reihe geschaltet werden.
;
;       rot     grün    gelb
;+5V o--|>|--+--|>|--+--|>|--o GND
;       ___  |       |  ___
;PB0 o-|___|-+       +-|___|-o PB1
;
;       rot     grün    gelb
;+5V o--|>|--+--|>|--+--|>|--o GND
;       ___  |       |  ___
;PB2 o-|___|-+       +-|___|-o PB3
;
;                       Ampel 1  Ampel 2
;Phasen:                PB3 PB2  PB1 PB0
;2(8)rot     /grün       0   0    1   0
;3(8)rot     /gelb       0   0    1   1
;4(8)rot     /rot        0   0    0   0
;5(8)rot+gelb/rot        0   1    0   0
;6(8)grün    /rot        1   0    0   0
;7(8)gelb    /rot        1   1    0   0
;8(8)rot     /rot        0   0    0   0
;1(8)rot     /rot+gelb   0   1    0   0
;
;
;R21 in 2,5 us-Schritten, 0=640us
;R25<=R22 in 2,56ms *256 =0,655s-Schritte
;
;
.DEVICE ATtiny13       ;für gavrasm, für Symbolische Registerbezeichnungen
```

```
(z.B.'DDRB')
.cseg
.org 0

init:
        ldi r16,15      ;PB.0 bis PB.3 als Ausgang
        out ddrb,r16    ;0b00001111
main:
        ldi r16,2       ;Phase 2(8) rot/grün
        out portb,r16   ;
        ldi r22,16      ;2,56ms *256 *16 =10,5s
        rcall wait      ;warte lang

        ldi r16,3       ;Phase 3(8) rot/gelb
        out portb,r16   ;
        ldi r22,4       ;2,56ms *256 *4 =2,6s
        rcall wait      ;warte kurz

        ldi r16,0       ;Phase 4(8) rot/rot
        out portb,r16   ;
        rcall wait      ;warte kurz

        ldi r16,4       ;Phase 5(8) rot+gelb/rot
        out portb,r16   ;
        rcall wait      ;warte kurz

        ldi r16,8       ;Phase 6(8) grün/rot
        out portb,r16   ;
        ldi r22,16      ;2,56ms *256 *16 =10,5s
        rcall wait      ;warte lang

        ldi r16,12      ;Phase 7(8) gelb/rot
        out portb,r16   ;
        ldi r22,4       ;2,56ms *256 *4 =2,6s
        rcall wait      ;warte kurz

        ldi r16,0       ;Phase 8(8) rot/rot
        out portb,r16   ;
        rcall wait      ;warte kurz

        ldi r16,1       ;Phase 1(8) rot/rot+gelb
        out portb,r16   ;
        rcall wait      ;warte kurz

        rjmp main       ;alles nochmal
wait:
        mov r25,r22     ;Wartezeit laden
sub_wait2:
        subi r21,1              ;1200000MHz /256 /3 =1563Hz =640us
        brbc 1,sub_wait2        ;wenn R21=0 dann sub_wait1     (1+2)
sub_wait1:                      ;
        sbiw r24,1              ;640us *4 =2,56ms * R24(word) (max.168s)
        brbc 1,sub_wait2        ;wenn R24=0 dann return                (2+2)
        ret                     ;zurück ins Hauptprogramm      (4)
ende:
```



13uk-Ampel-Nachtrag:
um die Helligkeit der LED besser anpassen zu können,
bekommt jede LED einen eigenen Widerstand.

```
          rot       grün       gelb
+5V o--|>|---|___|-+--|>|---|___|-+--|>|---|___|-o GND
                    |                 |
                 PB0 o               o PB1
```

```
          rot       grün       gelb
+5V o--|>|---|___|-+--|>|---|___|-+--|>|---|___|-o GND
                    |                 |
                 PB2 o               o PB3
```

Übersicht:

Programm	ASM	BAS	
1 LED on	X	X	
3 Piezo	X	X	
4 Blink 2Hz	X	X	
5 Zufall	X	X	
6 Stack+Sub	X	X	
7 PCINT0	X	---	wg. Vectortabelle
8 Adc+Timer	X	X	
9 Sleep	X	---	
10 Adc-binär	X	X	
11 Phase-PWM	X	X	
12 Fast-PWM	X	---	
13 Sägezahn	X	X	
Ampel	X	---	