

- Sprachanfang ist CPU - abhängig

↳ Wenn Befehl nicht existiert, macht die CPU auch einfach nichts

↳ Fehlermeldung „illegal instructions“

- „Mnemonics“ ($\hat{=}$ engl. „Gedächtnissstütze“)

ansstelle von hex- kodierenden Befehlen

↳ Assemblieren macht daraus Hex- Code

- Fun- Fact: Band- Raten sind häufig ganzzahlige

Vielfache von 75, da Fernschreiber früher so

gearbeitet haben und mit einem Getriebe auf

300 umgesetzt werden konnten.

Parallel Port Programmer - IC: 74LS245 /
74HC245
DIY

Assembler - Output:

.EEP \Rightarrow EEPROM

.HEX \Rightarrow Programm .OBJ \Rightarrow Simulation .LST \Rightarrow Referenzinfos 2

- Kommentare: Beginnen mit einem Semikolon ;
- Nicht nur kommentieren, warum man etwas macht, sondern auch, was man macht
(anders als bei Hochsprache)
- Programm - Bspf:
 - VOLIST ; Auflisten der Datei abschalten
 - INCLUDE "... dcf.inc"; Import Header
 - LIST ; Auflisten anschalten
- DEVICE ATTINY85 ; Gerät festlegen
- Register - Definition: .DEF tmp = R16
- Konstanten - Definition Präprozess or:
 - EQU fq= 6000 000 ; Frequenz
 - CSEG und .DSEG ?
 - ORG 0x---- (4-stellige Hex-Zahl)

Arbeitsweise des AVR

↳ Einschalten \Rightarrow Reset

↳ Ausführen des Codes an Adresse 0x0000

↳ Fetch during Execution:

Der nächste Befehl wird geladen

während der aktuelle ausgeführt
wird

↳ Programm-Counter wird während der
Ausführung um 1 erhöht

↳ Jeder Taktzyklus entspricht einer
ausgeführten Operation

↳ Am Anfang des Programms:

Interrupt-Vektor-Tabelle

- Programme laufen linear ab (von oben nach unten)

Register : 8-Bit Speicher

- an ALU bzw. Akkumulator angeschlossen
- Single clock-cycle instructions
- können Quelle, Daten & Ziel von Operationen sein

R0 bis R31

. DEF tmp = R16 ; Text-Ersatzung im Präprozessor

Zahl in Register laden:

LDT tmp, 150 ; lädt Zahl 150 in R16
„load immediate“

MOV tmp 2, tmp ; kopiere Inhalt von tmp in tmp 2
„Move“ : Ziel, Quelle

Vergleich „Ziel = Quelle;“ aus Hochsprache

R0 bis R15: Low - Register

R16 bis R31: High - Register

- konstante laden . LD1: Nur High

- Reg. auf 0 setzen: High und Low

C L R Rx

eigentlich nichts anderes als XOR Rx, Rx

Befehle nur für High - Register

- ANDI Rx, h ; Binäres Und mit Register Rx

und Konstante h, Ergebnis wird in Rx gespeichert

- CB R Rx, M ; Lösche alle Bits in Register Rx,

die in der Maske M (konstante) gesetzt sind.

- CPI Rx, h ; Vergleiche Register Rx mit

der Konstante h

- SBC I Rx, h ; Subtrahiere die Konstante

h und das Bit im Carry-Flag vom Register Rx

- $SBR Rx, M$; Setze alle Bits im Register Rx , die in der Maske M (konstante) gesetzt sind.
- $SET Rx$; Setze alle Bits im Register Rx , entspricht: $LDI Rx, 0xFF$
- $CLR Rx$; Lösche alle Bits im Register Rx , entspricht: $LDI Rx, 0x00$
- $SBT Rx, u$; Subtrahiere die konstante u vom Inhalt des Registers Rx

Poiner-Register: Doppel-Register

- X: R 26 / R 27
- Y: R 28 / R 29
- Z: R 30 / R 31

=> 16-Bit Register

=> Verwendet für Adressierung im Ram oder als Zeiger in den Programmspeicher.

- Niedriger-wertiges Byte im niedrigeren Register,
Höher-wertiges Byte im höheren Register.
- => eigene Namen: ZH, ZL (R31, R30)

Bsp.:

- EQU Wert = Ox----- (16-Bit Zahl)
- LDI XH, HIGH(Wert)
LDI XL, LOW(Wert)
- zur Demonstration
von HIGH() und LOW()

Instruktionen für Pointer-Zugriffe

- X, Y und Z: Pointer für Addressierung
im RAM
- Z: Pointer für Addressierung im
Programmspeicher
- LD: „Load“, Lade Adresse aus dem
Ram LD R1, X
- ST: „Store“, Speichere in Ram an
der Pointer-Adresse ST X, R1

- Poiner - Manipulatoren:

- LD R1, X; Lese / schreibe von / zu
Adresse in X und inkrementiere
den Poiner danach
- LD R1, -X; Dekrementiere den
Poiner und lese / schreibe danach
von / zu Adresse in X

- LOD und STD: Addiere vor
Zugriffs - Operation noch das
„Displacement“ q an die Poiner -
Adresse

- LOD R1, Z + q
Werte von 0 bis 63

\Rightarrow Analog auch mit Y und Z als Poiner
möglich

Programmspeicher - Pointer : 2

- LMP ; lädt das Byte an der Adresse 2 in das Register R0

Befehle bestehen aus 2 Wörtern

=> Adresse mit 2 multiplizieren

=> unterstes Bit gibt an, ob das untere oder das obere Word adressiert wird.

L DI 2H, HIGH (2 * Adresse)

L DI 2L, LO W (2 * Adresse)

LMP ; Lese unteres Byte / Word

ADI w 2L, 1 ; Inkrementiere 2 - Pointer

LMP ; Lese oberes Byte / Word

- ADI w: „ ADD Immediate Word“,

kann bis zu max. 63 zu dem Word addieren.

Als Parameter wird immer das untere Zielregister angegeben.

- SB IW : „Subtract Immediate Word“, ähnlich wie AD IW
- anwendbar auf: X, Y, Z und R24/R25

R24/R25:

- 16-Bit Doppelregister
- keine Verwendung als Poiner möglich
- AD IW und SB IW Operationen dafür möglich

Daten im Programmspeicher ablegen:

- .DB - und .DW - Direktive: speichernd die dahinter folgenden Konstanten als Code
 - .DB 1234, 45, 67, 89; Liste mit 6 Bytes
 - .DB "Text"; Liste mit einem Text, 6 Bytes lang
- ⇒ wichtig: ist die Anzahl an Bytes nicht gerade, kommt hinter ein O-Byte dran.

- DW 12345, 6789 ; zwei Worte,
Listen werden wortweise aufgebaut
(2 Bytes pro Word)
- Listen können auch mit Sprungadressen /
Labels aufgebaut werden:

Label 1:

...

Label 2:

...

Sprungtabelle:

- . DW Label 1, Label 2

=> Beim Lesen mit LPM erscheint das
niedrigste Byte zuerst

Regeln zum Umgang mit Registern:

- immer mit .DEF verwenden
- Pointer für RAM: X, Y und Z bzw.

R26 bis R31

- 16-Bit-Wert: R24/R25, da ADIW, SBW
- Lesen aus Programmspeicher: R0 als Ziel und Z (R30/R31) als Pointer
- Einzelne Bits: R16 bis R23
- Interrupts: R15 zum Sichern des Flage-Registers
- Alles andere: R1 bis R14

Ports, aka „Register“

- 64 direkt adressierbare Ports, nicht alle sind bei allen AVR-Typen vorhanden
- bei größeren ATmega noch mehr indirekt adressierbare Ports
- Kommunikation mit externen und internen Geräten, z.B. SREG
- nehmen ganze Zahlen oder Steuer-Bits auf
- in der Include-Datei sind wichtige Ports und Steuer-Bits mit Namen belegt:
 - .include "tiny18def.inc"

Port-Zugriffe:

Bsp.: Setze den Port MCUCR

LDI R16, 0b 10000000

OUT MCUCR, R16 ; Setze Port

- Oad : Schiebt Register - Inhalt in Port raus
- Im Beispiel wurde das sleep - Bit im Port MCUCR gesetzt. Dies geht auch eleganter:

LDI R16, 1 < SE

↑

Schiebt eine Binäre 1 (0b 00000001)

SE - mal nach links,

wird vom Präprozessor ausgewertet.

Binäres Oder:

LDI R16, (1 < SE) | (1 < SMO)

oder - Operator

Wichtiger Vorteil: Portierung des Quellcodes einfacher

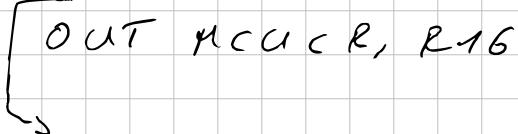
- Ports auslesen:

IN R16, MCUCR

- Bsp.: Nur bestimmte Bits verändern:

IN R16, MCUCR;

SBR R16, (1<SE) | (1<SM0);


OUT MCUCR, R16

SBR: Setze alle Bits im Register Rx, die
in der Maske (konstante) gesetzt sind

- Setzen einzelner Bits:

.EQU Bit Zeiger = 0 ; konstante

SBI PORTB, Bit Zeiger; auf „1“ setzen

CBI PORTB, Bit Zeiger; auf „0“ setzen

=> lässt sich nur auf Ports bis 0x1F anwenden

- IN und OUT lässt sich nur auf Ports

bis 0x3F anwenden

- Zugriff auf Ports über Zeiger

• DEF tmp = R16

LDI ZH, HIGH(PORTB + 32)

LDI ZL, LOW(PORTB + 32)

LD tmp, Z

die ersten 32 Adressen
im SRAM sind die Register

Wichtige Ports:

Akkumulator

SREG

Status Register

SREG

Stack

SPL / SPLI

Stackpointer

SPL / SPH

Ext. SRAM /

MCUCR

"MCU General
control Register"

MCUCR

Ext. Interrupt

Exd. Ind.

INT

Interrupt Mask Reg.
Flag Register

GMISK
GIFR

Timer Ind.

Timer Int.

Int Mask Reg.
Int Flag Reg.

TIMSK
TIFR

... und noch viele weitere

Status-Register: S REG

- Z: Zero-Flag \Rightarrow Ergebnis der letzten Rechenoperation ist 0
- C: Carry-Flag \Rightarrow Überlauf der letzten Rechenoperation
- N: Negative-Flag \Rightarrow Ergebnis der letzten Rechenoperation ist negativ
- V \Rightarrow Überlauf Zweierkomplement
- S \Rightarrow Vorzeichen negativ & zweierkomplement ($N \& V$, binäres und)
- H: Half-Carry Flag \Rightarrow Überlauf des unteren Nibbles
- T: Transfer-Bit \Rightarrow beliebig nutzbares Flag
- I: Interrupt-Flag: Schaltet Interrupts ein / aus

SRAM

(Static RAM)

- Speicherzellen, die im Vergleich zu Registern nicht mit der ALU verschaltet sind
- mehr Speicherplatz als Register, da mehr Zellen zur Verfügung stehen.
- Zugriffe langsamer, da zuerst in ein CPU-Register geladen werden muss.
- externer SRAM evtl. Möglich, Anssteuerung identisch zu internem RAM
- Zugriff über feste Adressen oder über Zeiger möglich
- Pointer-Zugriff über offset-Addition auf die Adresse möglich
- Stack befindet sich im SRAM (von unten nach oben)

- Speichern : STS $\underbrace{0x0060}_{SRAM\text{-Adresse}}, R1$
- Lesen : LDS $R1, \underbrace{0x0060}_{})$

- Beginn des SRAMs an Adresse:

SRAM - START

- Organisation des SRAMs in Datensegmente :

- . DS EGS ; Beginn des Datensegments

- . ORG SRAM - START ; Datensegment an

- den Anfang des SRAMs setzen

- Label 1 ; Label definieren

- . BYTE 1 ; 1 Byte reservieren

- Label 2 ; Label definieren

- . BYTE 2 ; 2 Byte reservieren

. EQ u Puffergroesse = 32

Buffer- Stand:

. BYTE Puffergroesse ; 32 Speicherzellen reserv.

. CS EG ; Ende Datensegment, Beginn

Code - Segment

▷ Reine Organisation des SRAMs,

keine Inhalte die eingelegt werden.

- Poiner - Zugriff über X, Y und Z - Register

möglich :

LD I XH, High (Adresse)

LD I XL, Low (Adresse)

LD R1, X ; (x+ und -X möglich)

- Indizierte Poinerzugriffe auf Y- und
Z - Poiner mit LDD und STD möglich

LDI YH, HIGH (Adresse)

LDI YL, LOW (Adresse)

LDI R1, $Y + \underbrace{q}_w$

$q \in \{0, \dots, 63\}$, 6 Bit

- Pointer - Adresse wird nur temporär für den Zugriff verändert, Register wird bleibt erhalten

Stack:

- Stack-Pointer Register: SPH: SPL

p

Nur verfügbar, wenn man mehr als 256 B Ram hat

- Initialisierung:

.DEF Temp = R16

LDI Temp, HIGH(RAMEND)

OUT SPH, TEMP

LDI Temp, LOW(RAMEND)

OUT SPL, TEMP

- Verwaltung des Stack-Pointers geschieht automatisch
- Ablegen: PUSH Rx
- Rücklesen: POP P Rx
- Abholen aller zurückgelegten Werte wichtig, sonst läuft der Stack voll
- Rücksprung von Unterprogrammen mit Adresse oben im Stack:

CALL Label ; Springe zu "Label"

...

Label:

...

RET ; Springe zurück

=> legt einen 16-Bit Wert auf dem Stack ab

- Interrupt-Service-Routine legt Adresse auf dem Stapel ab

Reset

- Beim Start des µCs
- Externer Reset - Pin
- Sprung nach Adresse 0x 0000
 - Initialisierung aller Register mit Default-Werten findet nicht statt
- WDT - Reset

- Muss gestartet werden
- Regelm. Zurücksetzen mit WDR
- kein Zurücksetzen => µC Reset
- .ORG 0x0000

- Organisation der Code-Segmente
- "ORG" = "Origin"
- lässt Lücken zwischen leeren Code-Abschnitten

- .ESSEG : EEPROM-Segment

- Beginn des Programms: Interrupt - Vektoren
 - Sprung - Adressen für die Interrupts
 - Beim Eindreten wird Zeile ... ausgeführt
 - Vorherige Position wird auf dem Stack hinterlegt
- Standard - Aufbau:

.CSEG
.ORG 0x 0000

RJMP start ; Reset - Interrupt
... interrupt - Vektoren

start: ; jetzt geht's richtig los

- RJMP ; Sprung zu Adresse / Label ...

- Labels :

- Wenn kein Doppelpunkt, dann ignoriert
- fehlendes Label: Fehlermeldung „Undefined label“

Programmablauf:

- linear: von oben nach unten, eine Instruction nach der anderen
- Programmzähler (Program counter)
zählst immer um 1 hoch
- Ausnahmen: gewollte Verzweigungen, Sprünge oder Interrupts

Bedingte Verzweigungen:

Bsp.: 32-Bit Zähler: 6 8-Bit Register

- Inkrementieren mit Befehl INC
→ Beim Überlauf wird 2-Bit im Status-Register gesetzt
→ C-Bit wird bei INC nicht verändert
- Inkrementieren der höheren Register nur, wenn das darunterliegende einen Überlauf hat

INC R1

B R N E Weiter

INC R2

B R N E Weiter

INC R3

B R N E Weiter

INC R4

Weiter:

Liste der Conditional Branches:

- BRCC / BRCS: Carry - Flag 0 / gesetzt
- BRS H : größer
- BRL O : kleiner
- BRMI : Minus } Negative - Flag: Ergebnis der
- BRLT : Plus } letzten Berechnung + / -
- BRGE: größer oder gleich } mit Vorzeichen, Signed -
- BRLT: kleiner } Flag gesetzt oder nicht.

- BR HC / BR HS : Halbübertrag 0 oder 1
- BR TC / BR TS : T-Bit 0 oder 1
- BR UC / BR US : U-Bit (overflow) 0 oder 1
- BR IE / BR IO : Interrupt ein- oder ausgeschaltet
- BR NE / BR EQ : Branch not Equal / Branch Equal, Z-Flag

Zeitzusammenhänge:

- Ausführung eines Befehls entspricht genau einem Prozessor-Takt

$$\text{Bsp.: } 4 \text{ MHz} \rightarrow \frac{1}{4} \mu\text{s} = 250 \text{ ns}$$

$$10 \text{ MHz} \rightarrow 100 \text{ ns}$$

- Instructions, die mehr Takte brauchen: sprung-
Instructions, Lese- & Schreibinstructions SRAM
- Rich-t - fun Operation, um einen Takt
"wegzuwerfen": NOP, "No Operation"

$$\text{Bsp.: } 4 \text{ MHz} \rightarrow 6 \text{ NOPs für } 1 \mu\text{s Verzögerung}$$

- Schleifen - konstruktion mit „NOP“s, um Zeit zu schinden
- 8-Bit-Register, das heruntergezählt wird:
 - " CLR R1; setze R1 auf 0x00

Zaehl:

DEC R1; Dekrementiere R1

" BRNE Zaehl; falls nicht 0: Springen

- 16-Bit-Zähler:

" LD I ZH, HIGH (0xFFFF)

LDI ZL, LO W (0xFFFF)

Zaehl:

SB I ZL, 1 ; subtrahiere 1 von Z

BRNE Zaehl

Macros:

. MACRO Name

...

, ENDMACRO

| Beim Aufruf des Macros werden
| die Code-Zeilen dazwischen
| dort rein kopiert

Unterprogramme

- Im Gegensatz zu Macros sparsamer mit dem Programm speicherplatz
- Sprung zur Unterprogramm-Sequenz und anschließend Rücksprung zu Sprungadresse auf dem Stack
- Bsp.:

Label With Name Of Subroutine:

...

RET ; Springe zurück

- Aufruf mit RCALL :

RCALL Label With Name Of Subroutine

- RET braucht 6 Takte, RCALL 3 Takte, da die Sprungadressen auf dem Stack hinterlegt werden.
→ Verwendung des Stacks setzt Initialisierung des Stacks voraus
- Ohne Stack: Verwenden von absoluten sprüngen

RJMP Sub1

Label: ;

Subroutine muss aber am Ende mit RJMP selber zu Label Label springen und daher kann die Subroutine nicht mehr von überall aus im Programm aufgerufen werden

- Branchings sind RJMP - Aufrufe
- Aufruf eines Unterprogramms kann an einen Branch gehängt werden:
 $SBR C R17,7$; überspringe, wenn Bit 7 in R1 0 ist
 $RCALL Sub1$; Ruft Subroutine auf
- SBRC / SBR S: ship if Bit in Register is clear / set, ein bis zwei
- SBIC / SBIS: ship if Bit in I/O is clear / set (nur unterer 32 Ports)
- CPS E: compare, ship if Equal

Sprungtabelle:

Ausgangssituation: 4 I/O - Pins, 16 Zustände

Alle 16 Zustände sollen 16 unterschiedliche Funktionen bereitstellen / 16 unterschiedliche Reaktionen hervorrufen.

Lösung 1: 16 bedingte Sprünge \Rightarrow langsam und aufwendig zu implementieren

Lösung 2: Sprungtabelle

Mein Tab:

R JMP Sub 1

R JMP Sub 2

...

R JMP Sub 16

; alle Sprünge definiert

; Beginn der Tabelle in Z-Register laden:

LD I ZH, HIG.H (Mein Tab)

LD I ZL, LOW (Mein Tab)

; Tabellenbeginn in Z-Register hinterlegt

IN R 16, PIN B ; 110-Bit B als Quelle einlesen

ANDI R 16, 0x0F ; Obere 4 Bits löschen

ADD ZL, R 16 ; Addiere Adresse auf Z-Reg.

BZCC kein Überlauf ; Überlauf-Prüfung

INC ZH

kein Überlauf:

ICALL ; Ruft Subroutine auf

• ICALL: „Indirect Call to Subroutine“ } Springe zu

• IJMP : „Indirect Jump“ } Adresse in Z

=> Prozessor lädt Inhalt des Z-Registerpaars in den Program-Counter und macht dort weiter

Interrupts:

- Methode 1: Polling, regelmäßiges Abfragen
- Methode 2: ISR, interrupt service routine

1.) Microcontroller mitteilen, dass er beim Eintreten eines Interrupts unterbrechen darf

SEI; „set global interrupts enabled“

CPI; „clear global interrupts enabled“

⇒ manipuliert I-Bit im SREG

2.) Interrupt - Flags setzen : Welche Interrupts?

3.) Interrupt tritt ein

⇒ Adresse wird auf den Stack abgelegt, dieser muss vorher eingerichtet worden sein

⇒ Prozessor springt an die vorgefahne Stelle in der Interrupt - Vektor - Tabelle

⇒ Später: Sprung zurück beendet das interrupt

Ausnahme: Reset legt keine Adresse auf dem Stack ab

- Nur 1-Word - Befehle in der Int. Vekt. Tab. zulässig

- Fähigkeit, Interrupts auszuführen, ist prozessor-spezifisch und von der Ausstattung abhängig
- 2 Interrupts gleichzeitig: jeweils oberes in der Liste gewinnt. Das darunter kommt erst dran, wenn das darüber abgearbeitet ist.
- Beenden eines Interrupts:

`RETI`; Schaltet Interrupt ein und springt zurück

Oder:

{ `SEI`; schaltet Interrupt ein
 { `RET`; springt zurück

→ Vorsicht: Status-Bit lässt vor dem Rückprung schon wieder Interrupt zu, dadurch können verschachtelte Interrupts entstehen

- Beim Betreten eines Interrupts sperrt der IC alle Interrupts und sie müssen danach wieder reaktiviert werden
- Bei großen AVRs reicht `RJMP` nicht aus, um zu einer bestimmten Speicheradresse zu springen. Da muss „JMP“ (2-Word-Instruktion) oder ein „RETI“ gefolgt von einem „MOP“ verwendet werden.
- Erste Instruktion im Interrupt: SREG sichern, um Isolierbarkeit gewährleisten zu können.

Schleifen & Timing

Einfache 8-Bit Schleife:

. equ C1 = 200 ; Anzahl Durchläufe
ldi R16, C1

Loop:

dec R16

brne Loop ; Wenn nicht 0, dann Schleife beginnen

Zeitverzögerung bei Schleifen abhängig von:

- Anzahl an Taktten, die eine Schleife benötigt
- Zeitdauer eines Prozessorzyklus

=> Anzahl an Instruktionen steht im Datenblatt, im „Instruction Set Summary“ in der Spalte „# clocks“

=>. equ C1 = 200 ; 0 Takte, Präprozessor
ldi R16, C1 ; 1 Takt

Loop: ; Label, Präprozessor

dec R16 ; 1 Takt

brne Loop ; 2 Takte ≠ 0, 1 Takt bei 0

1. Konstante laden: 1 Takt, 1 mal

2. Schleife mit Sprung: 3 Takte, ($C1 - 1$) mal

3. Schleife ohne Sprung: 2 Takte, 1 mal

Summe: $3 \times C1$ Takte

Taktfrequenz-Einstellungen.

- AVR ohne externen Oszillator brauchen einen externen Quarz, Keramikresonator oder Oszillator
- AVR mit einem oder mehreren internen Oszillatoren siehe Datasheet "System Clock and Clock option" bzw. Kapitel "Default Clock Source"
- AVR mit internem RC-Oszillator bieten die Möglichkeit, diesen zu verstehen. Dazu wird ein Wert in das OSCCAL Register geschrieben. Damit kann der AVR auf eine Spannungs- und eine Betriebstemperatur-abhängige Umgebung kalibriert werden.
- Das Wechseln zwischen interner und externer Taktquelle wird über das Verstellen von Fuses durchgeführt, sieh Datasheet Kap. "System Clocks and Clock options". Weiterhin kann ein Verteiler (clock prescaler) verwendet werden und eine Wartezeit nach dem Einschalten des Prozessors eingesetzt werden, bis der Oszillator stabil schwingt.
Die Fuses werden mit dem Programmiergerät eingesetzt, weshalb ein Verstellen der Taktquelle schwerwiegende Folgen haben kann.
- Ein Clock Prescaler kann über das Register CLKPR auf einen Wert von 2^n von 1 bis 256 eingesetzt werden.

Zeitverzögerung

Bsp.: Einfache Schleife von vorhin ($nC = C1 + 3$,
 $C1 = 200$, $nC = 600$) und 1,2 MHz Takt:
500μs Verzögerung. $C1 = 0xFF$: 640 μs

Schleife „länger“ machen:

```
.equ c1 = 200 ; 0 Takte, Assemblerdirektive  
ldi R16, C1 ; 1 Takt  
loop:           ; Schleifenbeginn  
    nop          ; Nichts tun, 1 Takt  
    nop          ; -- --  
    nop          ; -- --  
    nop          ; -- --  
    nop          ; -- --  
    dec R16      ; 1 Takt  
    brne loop    ; 2 Takte wenn ≠ 0, 1 Takt bei = 0
```

⇒ Laufzeit: $8 * c1$, Bei $256 * 8$ Taktten = 2048 Takte
und 1,2 MHz: 1,7 ms Verzögerung

Summ - Programm:

- Lautsprecher an Port B, Bit 0 über einen 100μF / 6V Elko. Ziel: 586 Hz Brummen
- Microcontroller: Attiny 13

.include "fh13def.inc"

.equ C1 = 0 ; bestimmt die Tonhöhe
sbi DDRB, 0 ; Portbit als Ausgang

Loop:

sbi PORTB, 0 ; Portbit auf High
Ldi R16, C1

Loop 1:

nop
nop
nop
nop
nop
nop

dec R16

brne Loop1

cbi PORTB, 0 ; Portbit auf Low
Ldi R16, C1

Loop2:

nop
nop
nop
nop
nop

dec R16

brne Loop2

rjmp Loop

16-Bit Schleife

- Verwendete Doppel-Register R24 / R25

• equ C1 = 50000

ldi R25, HIGH(C1)

ldi R24, LOW(C1)

Loop:

sbiw R24, 1 ; Doppel-Register R24 / R25
brne Loop; } dekrementieren

- Register X, Y und Z können ebenfalls ADIW und SBIW

- Prozessordauerte:

- 2x 1 Takt, konstante laden

- 1x 2 Takte, Subtraktion

- 2x Sprung: 2 Takte ≠ 0, 1 Takt = 0

$$\Rightarrow nt = 2 + 4 * (C1 - 1) + 3 = 4 * C1 + 1$$

Zeitverzögerung

• equ fC = 1200000; Prozessordauert 1,2 MHz

• equ fch = fc / 1000; Takt in kHz

Delay 1ms:

• equ c1ms = (1000 * fch) / 1000 - 1

ldi R25, HIGH(C1ms)

ldi R24, LOW(C1ms)

rjmp delay

Delay 100 ms:

$$\cdot \text{eqn } c100ms = (100 * fch) / 4 - 1$$

|ldi R25, HIGH(c 100 ms)

|ldi R24, LOW(c 100 ms)

rjmp Delay

; Schleife erwartet Konstante in R25:R24

Delay:

sbiw R24, 1 ; dekrementieren

brne Delay ; zähle bis 0

nop ; 1 Takt Zusatz-Vergögerung

=> C1ms und C100ms werden auf untersch. Weisen berechnet, um Rundungsfehler oder Überläufe zu vermeiden.

=> Verlängern der Schleife mit weiteren NOPs möglich.

Blinkprogramm:

.eqn c1 = 60000

sbi DDRB, 0

Loop:

sbi PORTB, 0

ldi R25, HIGH(c1)

ldi R24, LOW(c1)

Loop1:

nop (5x)

sbiw R24, 1

brne Loop1

| cbi PORTB, 0

| ldi R25, HIGH(c1)

| ldi R24, LOW(c1)

| Loop:

| nop (5x)

| sbiw R24, 1

| brne loop2

| rjmp loop

Interrupts - Kap 2

Interrupts sind Stromsparender als Polling

Regeln für den Umgang mit Interrupts:

- Stack muss initialisiert sein (SPH: SPL) und zu Beginn auf RAMEND gesetzt werden.
- Jedes Interrupt muss durch das zugehörige Interrupt-Enable-Bit gesetzt werden.
- Das I-Flag im SREG wird zu Beginn gesetzt und bleibt möglichst während des gesamten Betriebes gesetzt.
- In der Interrupt-Vektor-Tabelle wird jedem Interrupt eine Interrupt-Service-Routine (ISR) zugeordnet. Dafür ist in der LUT ein Ein-Wort Sprung befehl vorgesehen. Bei großen AT-Megas sind zwei-Wort-Sprünge vorgesehen (RJMP: 1 Word, JMPL: 2 Word)
- ISR-Adressen sind prozessorspezifisch angeordnet
⇒ aufpassen beim Portieren
- Jede ISR muss mit einem "RET," beendet werden (außer Reset-Interrupt)
- Tritt ein Interrupt ein, so wird in einem Steuerregister ein flag gesetzt. Dieses wird fast immer beim Sprung in die ISR gelöscht.
- Bei gleichzeitig eintretenden Interrupts wird die ISR mit der niedrigsten Adresse in der Tabelle bevorzugt.
- Jede ISR beginnt mit einer Sicherung des SREG und endet mit der Wiederherstellung des SREG.

- Beim Eintreten eines Interrupts wird die Rücksprungadresse auf dem Stack abgelegt.
- Das Anspringen einer ISR löscht das I-Flag im SREG und das Beenden einer ISR mit RETI setzt es wieder.
- Da während der Ausführung einer ISR für gewöhnlich keine weiteren Interrupts ausgeführt werden können, sollte diese so kurz wie möglich gehalten werden.

Tipp: alle ISRs können die gleichen Temp-Register verwenden

- Kommunikation zwischen ISR und Hauptprogramm läuft über Flagues ab, die in der ISR gesetzt und im Hauptprogramm zurückgesetzt werden. Zum Zurücksetzen kommen ausschließlich 1-Vord-Instruktionen zum Einsatz oder Interrupts werden währenddessen blockiert
- Die Übergabe von Werten zwischen Hauptprogramm und ISR erfolgt über dedizierte Register/DRAM-Adressen. Bei mehr als einem Byte muss ein Übergabemechanismus verhindern, dass keine Fehler durch weitere Interrupts auftreten.
- Das Hauptprogramm kann den Prozessor in den Schlafmodus versetzen. Jedes Interrupt weckt ihn wieder auf.

Interrupt-Vektor-Tabelle:

≈ Auflistung an Sprung-Instruktionen zu den ISRs

kein „Displacement“ (aka Vektor) bei AURs

- Nicht benutzte Sprungadressen sind Refi befüllten

- o Anzahl an RSTP / RETI - Instruktionen in der Tabelle hat exakt der Anzahl der Einträge der IVT zu entsprechen
- o nicht benutzte Stellen werden mit Opcode 0xFFFF befüllt (NOP)
- Tabelle sieht bei jedem AVR anders aus
 - kleiner AVR: 1-Wort Sprung oder RETI
 - großer AVR: 2-Wort Sprung oder RETI + NOP

Ablauf

1.) Reset, Init

- Einschalten: PC bei 0x0000, hier muss ein Sprung-Befehl zu einer Stelle im Hauptprogramm stehen
- Hauptprogramm initialisiert Stack bzw. Stackpointer, da dieser von den Interrupts benötigt wird (Red, Redi)
- Hardware-Initialisierung: ADC, Timer, evtl. Schlaf-modus so setzen, dass Prozessor beim Interrupt wieder aufwacht
- Interrupts setzen, Prozessor arbeiten lassen (schlafen legen)

2.) Interrupt tritt ein, der Microcontroller

- wacht auf, deaktiviert die globalen Interrupts
- legt den Wert des PC (oder den Wert dahinter) auf dem Stack ab und setzt den PC auf den Wert des Eintrags in der IVT

- führt den (Sprung-) Befehl in den IVT im nächsten Zyklus aus. Damit wird die ISR ausgeführt, bis sie mit einem RETI beendet wird. Während der ISR kann ein Flagen-Bit in einem Register gesetzt werden, welches später vom Hauptprogramm ausgewertet wird.
- die ISR endet mit einem RETI, welcher dazu führt, dass die Ausführung des Programms an der im Stack abgelegten Stelle fortgesetzt wird. Dabei wird das global-Interrupt-Flag wieder abhängig.
- ohne oder mit Nachbearbeitung wird der Microcontroller wieder schlafen gelegt.

Bei mehreren Interrupts:

- Reihenfolge der Interrupt-Hierarchie beachten
- Jedes Interrupt hat ein eigenes Flague-Bit
- ISRs sollten so kurz wie möglich gehalten werden, um sich nicht zu blockieren

Ressourcen-Management

SR EGz :

- im Register zunächst legen:
in R15, SREG 2 Takte Zeitbedarf
C...I
out SREG, R15
- SREG im Stack zunächst legen
Push R0 6 Takte Zeitbedarf
C...I
in R0, SREG
out SREG, R0
P0 P R0

- SREG im SRAM zurück legen

GT alte Zeitverbrauch

```

STS 0x00xx, R0
    in R0, SREG
    out SREG, R0
LDS R0, 0x00xx

```

=> T-Bit im SREG kann als Flagge verwendet werden

=> Schreib- & Lese-Zugriffe dürfen manchmal von ISR nicht gestört werden

- Flaggen sollten so schnell wie möglich wieder gelöscht werden, da sonst ein Ereignis verpasst wird.
- Bei mehreren Interrupts: Häufigkeit => Abwägung, was zuerst ausgewertet wird.

Loop:

sleep; schlafen legen

nop; dummy nach aufwachen

sbrcc rFlag. Bit A; frage Bit A ab

rcall int A; bearbeite A

sbrcc rFlag. Bit B; frage Bit B ab

rcall int B; bearbeite B

} hier könnte ISR A eintreten
ISR A eintreten

sbrcc rFlag. Bit A; frage Bit A erneut ab

rcall int A; bearbeite A

rjmp loop; gehe schlafen

int A; Bearbeite A

cbr rFlag, 1 <= Bit A; lösche Flag
[..] ref

int B; Bearbeite B

cbr rFlag, 1 <= Bit B; lösche Flag
[..] ref

- immer einzelne Bits & niemals Register zurücksetzen
- Doppel-Register wird außerhalb der ISR gesetzt und in der ISR ausgesehen
=> Aussetzen der Interrupts mit

CLI

C...J

SEI

Regeln für das Verwenden von Registern bei Interrupts:

- möglichst klare alleinige Verwendung innerhalb / außerhalb eines Registers zuordnen
- vermeindliche Konflikte vor Nutzung überdenken
- Bei Doppelregistern: Interrupts blockieren und danach wieder ausführen

! Interrupts müssen immer per Software aktiviert werden, bevor sie benötigt werden

Zahlenarten in Assembler:

- positive Granzahlen
- Granzahlen mit Vorzeichen
- Binary Code Digit (BCD)
- Grapach des BCD
- ASCII - Format

Positive Granzahlen:

- vielfache von Bytes
 - 1 Byte: 0 bis 255
 - 2 Byte: 0 bis 65 535
 - 3 Byte: 0 bis 16 777 215
 - 4 Byte: 0 bis " 294 967 295
- Register können verschieden verwendet werden,
da zur Manipulation jedes Register einzeln
angegeben werden muss
- Organisation eines Doppelwörter:
 - DEF dw0 = r16
 - DEF dw1 = r17
 - DEF dw2 = r18
 - DEF dw3 = r19

=> Beschreiben:

EQU dwi = 4.0.0.0 0 0
LDI dw0, LOW(dwi); R16
LDI dw1, Byte2(dwi); R17
LDI dw2, Byte3(dwi); R18
LDI dw3, Byte4(dwi); R19

Vorzeichenbehaftete Zahlen

- z.B. negative Zahlen
 - höchstwertiges Bit für das Vorzeichen reserviert
 - 0 \Rightarrow positiv
 - 1 \Rightarrow negativ,
- alle Zahlen werden im binär invertierten Format dargestellt
(2er-Komplement)

Binary coded Digit

- jedes Bit enthält eine Ziffer einer Dezimal-Zahl
- => Gruppierung BCD
- Pro Byte 2 BCD-Zahlen
 - weniger Speicherverbrauch als bei BCD
 - Ox -- Darstellung zum Schreiben sinnvoll

ASCII

- 0 $\hat{=}$ 48 / 0x30
- BCD \rightarrow ASCII:
 - 48 dazu addieren
 - Bits 4 und 5 auf '1' setzen
- Anführungszeichen für das Laden sinnvoll
 - LDI R16, '2'

BCD \Rightarrow ASCII: Bits 4 und 5 setzen

- ORI R16, 0x30 (ORI nur R16...R31)
- LD R2, R16, R2 (OR alle Register)
- SBR R16, 0b 00110000 (SBR nur R16...R31)

ASCII \Rightarrow BCD: nur die unteren 4 Bits

- ANDI R16, 0x0F (ANDI nur R16 ... R31)
- LDRI R2, 0x0F (AND alle Register)
- AND R16, R2
- CBR R16, 0b 00 11 0000 (CBR nur R16...R31)

Bits eines Registers invertieren: Exklusiv-Oder

- LDRI R2, 0b 10101010 (Mos6c)
- EOR R16, R2 (EOR alle Register)
- COM Rx (COM alle Register)
 \hookrightarrow invertiere alle Bits (1er komplement)
($x \mapsto (255 - x)$)

Vergieren eines Registers:

- NEG Rx (Neg alle Register)
 \hookrightarrow Vorzeichen-Bit (Bit 7) umdrehen
 \hookrightarrow Inhalt von 0 subtrahieren

T-Bit im SREG:

- CLT ; T-Bit auf 0 setzen
- SET ; $\begin{matrix} \text{---} \\ \text{---} \end{matrix}$ $\begin{matrix} 1 \\ \text{---} \end{matrix}$ $\begin{matrix} \text{---} \\ \text{---} \end{matrix}$
- BLO R1, 0 ; T-Bit ins Bit 0 im Register R1 kopieren
- BST R2, 2 ; Bit 2 aus Register R2 ins T-Bit laden

Mit 2 Multiplizieren: nach links schieben

- LSL Rx (R0 ... R31)
 \Rightarrow freies Bit 0 wird mit 0 gefüllt
 \Rightarrow Überlauf: Bit 7 ins Carry-Bit im SREG

Durch 2 dividieren:

- LSR Rx ($R_0 \dots R_{31}$)
=> freies Bit 7 wird mit 0 gefüllt
=> über auf: Bit 0 ins Carry-Flag im SREG

=> Aufrunden nach dem Dividieren:

BRCC Dir F

INC Rx

DIVE:

Arithmetisches Schieben: Vorzeichen-Bit (Bit 2) bleibt erhalten

- ASR Rx
=> Bit 0 landet im Carry
=> Bit 6 wird mit 0 aufgefüllt

16-Bit Zahl mit 2 multiplizieren

- links aus dem unteren Byte heraus geschobene Bit von rechts in das obere Byte rein schieben
- Rollen auffüllen nicht mit 0 sondern mit Carry-Bit, über auf ins Carry-Bit
- LSL R1 ; unteres Byte schreiben
ROL R2 ; oberes Byte rollen
=> weitere Bits angeben möglich

16-Bit Zahl mit 2 dividieren:

- LSR R2;
ROR R1;

Oberes Nibble einer gepackten BCD Zahl isolieren:

ROR Rx
ROR Rx
ROR Rx
ROR Rx

oder:

SWAP Rx \Rightarrow vertauschen der oberen
ANDI 0x0F und unteren Nibble
 \rightarrow Isolieren

Zwei 16-Bit Integer addieren:

R1: R2 und R3: R4 (HIGH: LOW)

ADD R2: R4 ; zuerst Low-Bytes addieren

ADC R1: R3 ; dann High-Bytes mit Übertrag addieren

Subtrahieren 2 16-Bit Integer: (R1: R2 - R3: R4)

SUB R2, R4 ; Low-Bytes zuerst

SBC R1, R3 ; High-Bytes mit Übertrag

Vergleich zweier 16-Bit integer:

CP R2, R4 ; low-Bytes vergleichen

CPC R1, R3 ; HIGH-Bytes vergleichen

\Rightarrow Carry-Flag: R3: R4 > R1: R2

Vergleich Register mit Konstante:

CPI R16, 0x AA

=> Z-Bit: R16 = 0x AA gleich

Register kleiner oder gleich Null?

TST RX

=> Z-Flag: RX ist 0x 00

=> Nach Vergleich Sprung mit BREQ, BRNE,
BRMI, BRPL, BRLO, BRSR, BRGE, BRVC
oder BRVS möglich

Addieren von BCD-Zahlen

- Überläufe im Carry- & Half-Carry Flag prüfen

R2 } + (R11:R3) => LDI R16, 0x 66 ; Konstante nur in high-Reg.
LDI R17, 0x 66 ; Korrektur: Subtraktion Ergebnis
ADD R2, R3

BRCC No Cy 1
INC R1
ANDI R17, 0x 0F

No Cy 1:

BRHC No He 1
ANDI R17, 0x F0

No He 1:

ADD	R2, R16
BRCC	No Cy 2
INC	R1
ANDI	R17, 0xF0

No Cy 2:

BRHC	No He 2
ANDI	R17, 0xF0

No He 2:

Sub	R2, R12
-----	---------

1.) Zahlen addieren
=> Überlauf?

Ja \hookrightarrow Übertrags-Register erhöhen, Korrektur des oberen Nibbles nicht nötig (obere 6 im Korrekturspeicher R17 löschen)

2.) INC und ANDI beeinflussen H-Bit nicht,
ist also im Zustand von nach der Addition
=> H-Flag gesetzt?

Ja \hookrightarrow Korrektur des anderen Nibble entfällt

3.) 0x66 addieren

4.) Carry & Half-Carry wie davor behandeln

5.) Korrektur-Register vom Ergebnis abziehen

längere alternative zu: BCD-Zahlen addieren

LDI R16, 0x66

ADD R2, R16

ADD R2, R3

BRCC No Cy

INC R1

ANDI R16, 0x0F

No Cy:

BR HC No He

ANDI R16, 0x0F

No HC:

SUB R2, R16

Packed BCD \rightarrow BCD

1.) SWAP \rightarrow HIGH Nibble ist in der Position des Low Nibble

2.) ANDI , 0x0F \Rightarrow HIGH Nibble löschen

BCD \rightarrow Packed BCD:

1.) Höherwertige BCD mit SWAP ins HIGH-Nibble verfrachten

2.) Mit Niedrigwertiger BCD verODERN

BCD \rightarrow Binär

1.) Niedrigswertige BCD-Zahl in Ziel-Register kopieren

2.) Mit 10 multiplizieren:

- 2 mal nach links schieben ($\times 4$)
- altes Ergebnis drauf addieren ($\times 5$)
- 1 mal nach links schieben ($\times 10$)

3.) Wieder bei 1 mit höherwertiger BCD-Zahl

Carry \Rightarrow BCD-Zahl passt nicht in vorhandene Bytes

Binär \rightarrow BCD

(1)

16-Bit-Zahl:

- 10.000 subtrahieren bis Überlauf
 \Rightarrow erste BCD-Ziffer
- 1.000 subtrahieren bis Überlauf
 \Rightarrow zweite BCD-Ziffer
- ⋮
- Speichern der 10er Potenzen im Programm-Speicher:

Dez Tab:

• DW 10 000, 1000, 100, 10

\Rightarrow mit LPM darauf zugreifen

(2) Wertigkeit der Bits im Programm-Speicher
zurücklegen

- . DB 0, 3, 2, 7, 6, 8
- . DB 0, 1, 6, 3, 8, 9
- . DB 0, 0, 8, 1, 9, 2
- . DB 0, 0, 4, 0, 9, 6
- . DB 0, 0, 2, 0, 4, 8
- :
.
- . DB 0, 0, 0, 0, 0, 1

=> einzelne Bits nach links heraus schreiben.

Falls es eine 1 ist:

Tabellenwert per LPM auslesen und zu dem Ergebnisbyte addieren

Addieren & subtrahieren in Software

Zu kompliziert um es hier aufzuschreiben :(

Siehe Buch