Mikrocontrollertechnik

# MODUL

# B

# Copyright ©

# Inhaltsverzeichnis MODUL B

## Neue Befehle in den einzelnen Kapiteln:

**B0:** `ror, rol, brcc, brcs`

**B2:** `brsh, brlo, adc, swap`

**B3:** `sei, cli`

# B0 Wiederholung

Die Grundlagen der Assembler-Programmierung aus dem Modul A sollen anhand einiger Programmieraufgaben wiederholt werden.

## *Kurze Zusammenfassung Modul A*

- Es existieren **32 Arbeitsregister** `r0-r31`, die sich auf den unteren Adressen im SRAM befinden. Mit ihnen lassen sich alle Berechnungen durchführen (Akkumulatorregister). Die obersten acht (`r24-r31`) Register können als 16 Bit Doppelregister `W`[1], `X` , `Y` und `Z` angesprochen werden. Die obersten sechs (`r24-r31`) Register können auch als Adresszeiger `X`, `Y` und `Z` zur indirekten Adressierung benutzt werden
  Die unmittelbare Adressierung (`ldi`) funktioniert nur bei den oberen 16 Register (`r16-r31`). Deshalb werden meist diese Register verwendet.
- **64 Sonderfunktions-Register** (SF-Register) ermöglichen die Ein- und Ausgabe von Daten (z.B. Ausgaberegister `PORTD`), die Steuerung der Peripherie (z.B. Datenrichtungsregister `DDRD`) und die Abfrage von Status-Informationen (z.B. Flagregister `SREG`). Sie können nicht unmittelbar adressiert werden. Zum Beschreiben und Lesen des ganzen Registers dienen die Befehle "`in`" und "`out`". Ein bitweises Schreiben der unteren 32 SF-Register ist mit den Befehlen "`sbi`", "`cbi`" möglich. Ein bitweises Überprüfen der unteren 32 SF-Register kann mit den Befehlen "`sbis`", "`sbic`" durchgeführt werden.
- Um ein neues Programm zu schreiben wird die **Assemblervorlage** (`A2_template.asm`) verwendet, da diese einige Initialisierungsarbeit abnimmt und ein einheitliches Erscheinungsbild der Lösungen ermöglicht.
- Um die Ein-/Ausgabe-Ports A-D benutzen zu können müssen diese zuerst initialisiert werden. Das **Datenrichtungsregister** `DDRx` (***D****ata* ***D****irection* ***R****egister*) legt fest ob es sich um einen Eingang oder Ausgang handelt. Es ist lese- und schreibbar. Das erwünschte Bit (Pin) wird mit `DDxn` bezeichnet.
- Über das **Datenausgaberegister** `PORTx` werden die Daten ausgegeben. Es ist lese- und schreibbar.
- Über das **Dateneingaberegister** `PINx` (***P****ort* ***IN****put Pins*) werden die Daten eingelesen. Es ist nur lesbar.
- **Nicht benutzte Pins werden als Eingang initialisiert!** Um Strom zu sparen sollte der interne Pull-Up-Widerstand zugeschaltet werden.
- Um den **internen Pull-Up-Widerstand** zu aktivieren wird im Ausgaberegister `PORTx` der entsprechende Pin auf Eins gesetzt.
- Die **Maskierung** mit logischen Funktionen erlaubt es gezielt einzelne Bits zu löschen, setzen oder umzuschalten (toggeln). Bei der **AND-Verknüpfung löscht** eine Null in der Maske das **Bit**. Bei der **OR-Verknüpfung setzt** eine Eins in der Maske das `Bit`.

1 Das Doppelregister `W` wird in der Vorlage (A2_template.asm) definiert (`.DEF WL = r24`; `.DEF WH = r25`) und kann nicht als Adresszeiger verwendet werden. `X`, `Y` und `Z` sind in der AVR-Definitionsdatei (z.B.: m32def.inc) definiert. `r25` und `r24` werden auch zur Parameterübergabe bei Unterprogrammen genutzt.

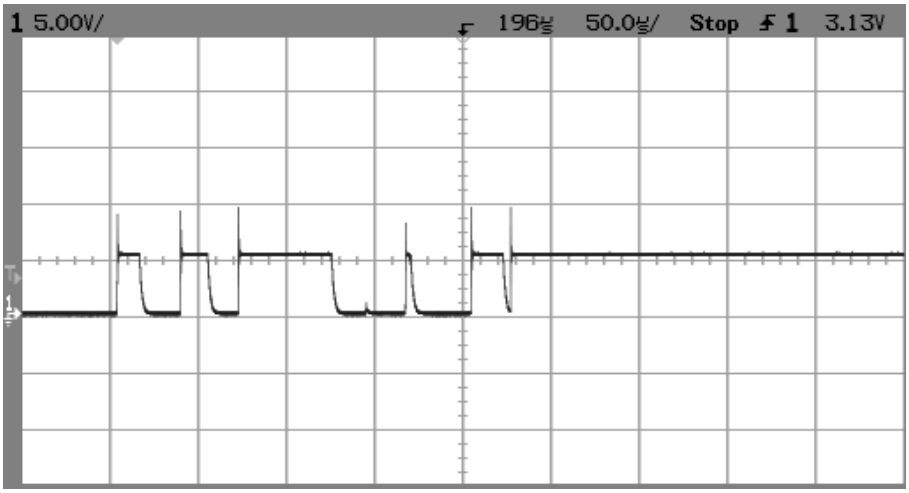
Bei der **XOR-Verknüpfung invertiert** eine Eins in der Maske das **Bit**.
Die Maskierung wird besonders zur Manipulation der oberen 32 SF-Register benötigt.

- Ein Befehl besteht aus dem Opcode und den Operanden. Es gibt Befehle ohne Operand, welche wo nur ein Operand benötigt wird und Befehle mit zwei Operanden. Bei zwei Operanden steht immer **zuerst das Ziel und dann die Quelle!**
- **Zeitschleifen** sind Programme, deren Aufgabe es ist, Zeit zu verbrauchen. Ändert man die Taktfrequenz des Controllers, so müssen in einem bestehenden Programm auch die Zeitschleifen verändert werden. Am einfachsten arbeitet man deshalb mit einer externen Unterprogramm-Sammlung (Zeitschleifen-Bibliothek, z.B.: **`SR_TIME_16M.asm`**) welche mit der Direktive "**`.INCLUDE`**" eingebunden wird.
- **Unterprogramme** sind Programmteile, die von einem Hauptprogramm aus (oder aus einem anderen Unterprogramm) aufgerufen werden und nach der Ausführung wieder an dieselbe Stelle des Hauptprogramms zurückspringen.
- **Unterprogramme können nur verwendet werden, wenn der Stapel vorher initialisiert wurde!** Register die ebenfalls im Hauptprogramm benötigt werden, müssen im Unterprogramm mit dem Befehl "**`push`**" gerettet werden und vor dem Verlassen des Unterprogramms mit dem Befehl "**`pop`**" wiederhergestellt werden. Ein Unterprogramm endet mit dem "**`ret`**"-Befehl.

# *Entprellen eines Tasters (Schalters)*

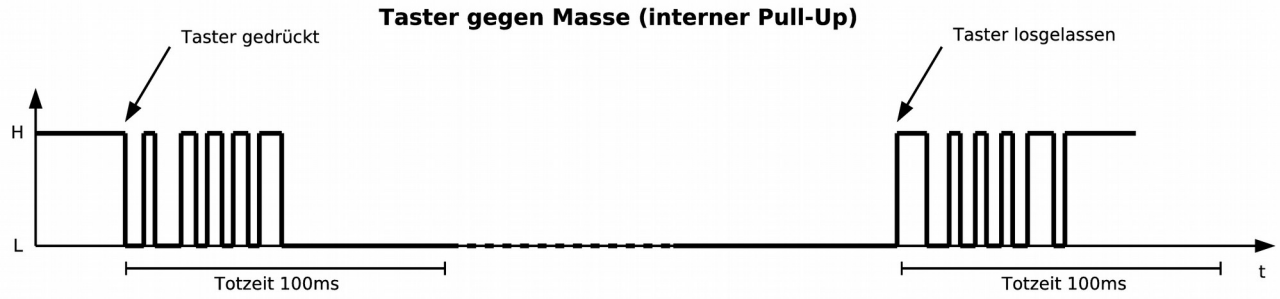
Wird ein Taster oder Schalter betätigt, so wird mechanisch ein Kontakt geschlossen. Hierbei federt der Kontakt mehrmals hin und her, das heißt, der Kontakt wird mehrmals geschlossen und geöffnet bevor ein stabiler Zustand eintritt. Das Prellen ist vom Schaltertyp abhängig und dauert meist einige hundert Mikrosekunden, kann bei großen Schaltern aber bis zu 50 ms dauern! In dieser Prellzeit schließen und öffnen sich die Kontakte oft einige zehn- bis hundertmal!

### Oszillogramm eines über etwa 250µs prellenden Tasters:




(Quelle: http://de.wikipedia.org/wiki/Prellen)

Das Prellen tritt auch beim Loslassen des Tasters auf:




Da Mikrocontroller einen Eingang mehrmals innerhalb von zehn Mikrosekunde abfragen können, besteht die Möglichkeit, dass fälschlicherweise mehrere Änderungen der Eingangssignale erkannt werden, obschon der Schalter manuell nur einmal betätigt wurde.

Durch Entprellung wird dieses Fehlverhalten vermieden. Die Entprellung kann hardwaremäßig oder softwaremäßig durchgeführt werden.

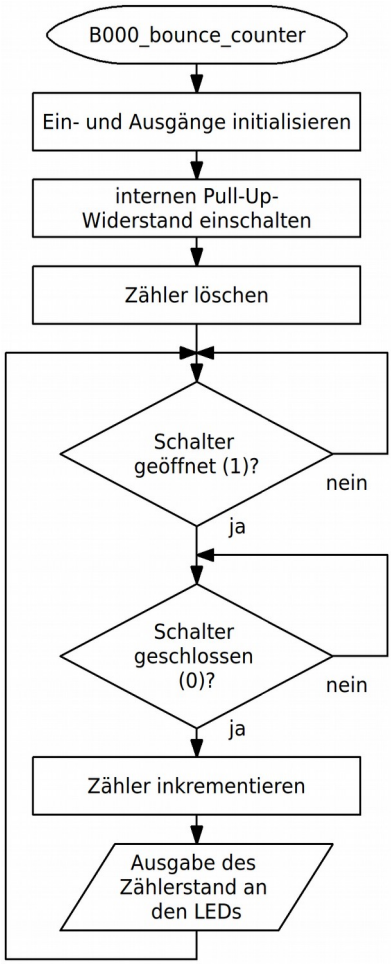
⌂ **B000**

a) Schreibe ein Programm, welches die mit einem gegen Masse geschalteten **Schalter** (Schließer, interner Pull-Up) an **PB0** erzeugten „Ein“-Schaltungen (1/0-Wechsel) beim Prellen zählt und laufend das Resultat binär an den LEDs von **PORTD** ausgibt. Das Flussdiagramm ist vorgegeben. Nenne das Programm "**B000_bounce_counter.asm**".

b) Wie muss das Programm verändert werden, damit die „Aus“-Schaltungen

(0/1-Wechsel) gezählt werden?

c) Teste das Programm ebenfalls mit einem hardwaremäßig entprellten Schalter.

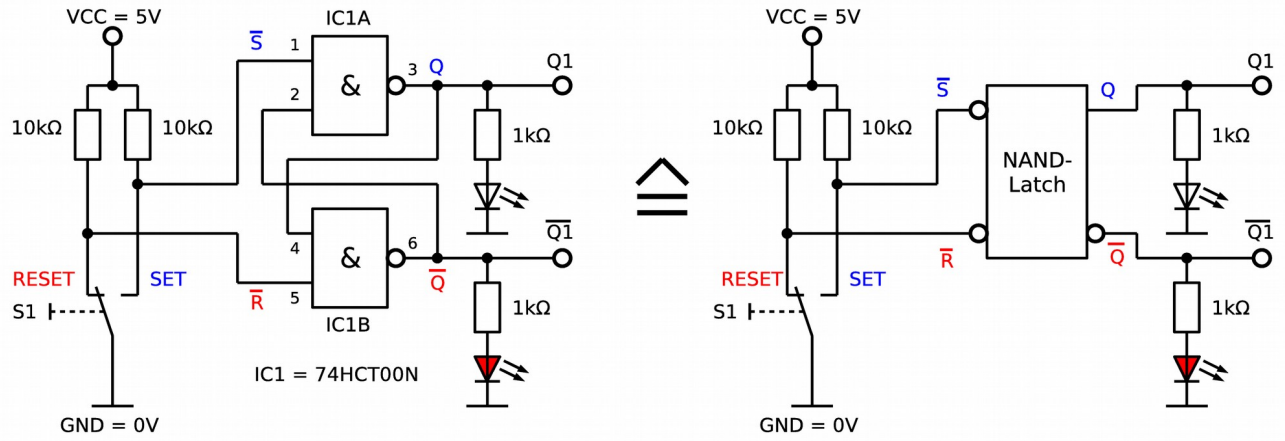



**Bemerkungen:** Der Zähler kann jeweils durch Drücken des RESET-Knopfes auf Null zurückgesetzt werden.
Die Taster des MICES2-Boards (im Gegensatz zum MICES-Board) prellen nur sehr wenig. Hier sollte mit unterschiedlichen externen Tastern und Schaltern experimentiert werden.

## *Hardwaremäßige Entprellung*

Die beste Lösung um das Prellen eines Schalters zu verhindern ist eine zusätzliche Schaltung am Eingang zum Beispiel mit einem Wechseltaster oder -schalter und einem **Flip-Flop**[2], so dass der Schalterzustand gleich eindeutig ist (hardwaremäßiges Entprellen). Hier eine Schaltung mögliche Schaltung mit zwei NAND-Gliedern (NAND-FF, *NAND-Latch*). Setzen und Rücksetzen wird hier mit einer logischen 0 erreicht. Die zwei Eingänge werden über Pull-Up-Widerstände auf definiertem Pegel gehalten wenn diese nicht mit Masse verbunden sind.

2 Beim Flip-Flop (FF, bistabile Kippstufe) handelt es sich um eine Speicherzelle. Mit einer Eins am SET-Eingang wird ein FF gesetzt (Information gespeichert). Eine Eins am RESET-Eingang wird das FF rückgesetzt (Information gelöscht). Ein 8 Bit-Register besteht zum Beispiel aus 8 Flip-Flops.




Eine Entprell-Schaltung mit einem einfachen Schließer mittels Tiefpass und Schmitt-Trigger ist auch möglich, hat allerdings den Nachteil, dass durch den Kondensator Verzögerungen beim Ein- und Ausschalten auftreten.

## ***Softwaremäßige Entprellung***

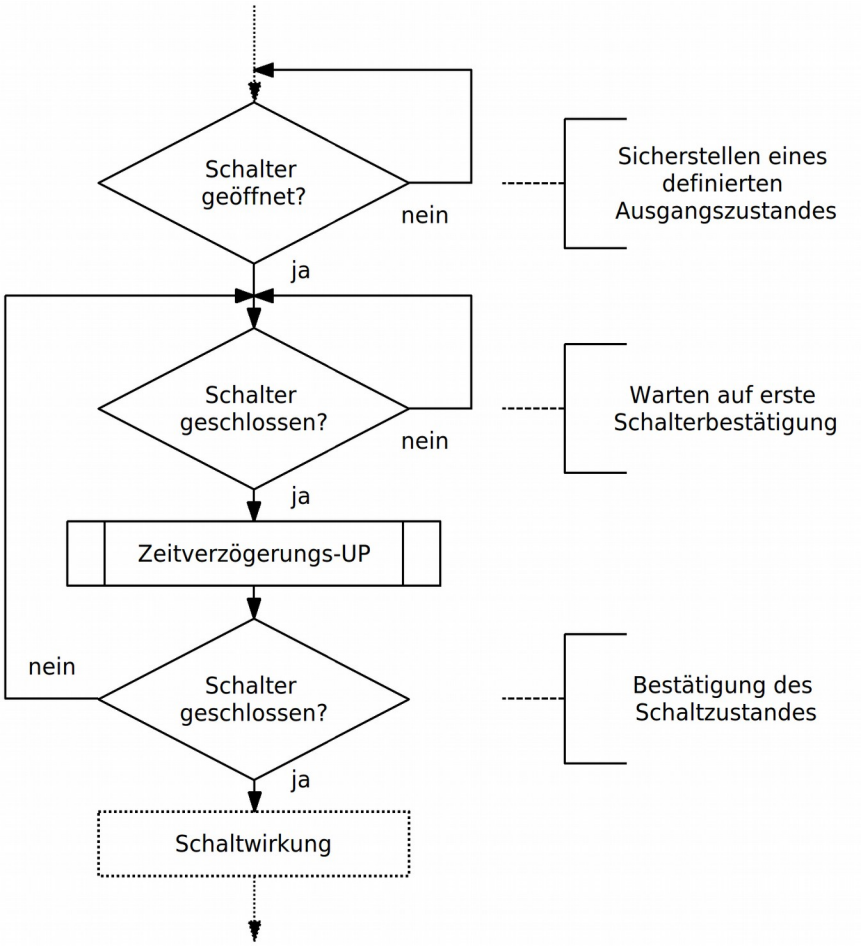
Zur softwaremäßigen Entprellung erzeugt man mit Hilfe einer Zeitschleifen eine Totzeit um das Prellen zu überbrücken. Nach der Sicherstellung eines definierten Anfangszustandes wird auf eine erste Schalterbetätigung gewartet. Ist diese erfolgt, wird nach Ablauf der Totzeit der Schalterzustand nochmals abgefragt, und bei erkanntem gültigen Schaltvorgang, fährt das Programm mit der gewünschten Schaltwirkung weiter.

Je nach Schaltertyp ergeben sich zwei verschiedene Möglichkeiten:

1. Der Schalter ist ein **Schließer:**
   Bei **Betätigung** wird der Stromkreis **geschlossen**.

2. Der Schalter ist ein **Öffner:**
   Bei **Betätigung** wird der Stromkreis **geöffnet**.

Das folgende Flussdiagramm zeigt das Prinzip, der softwaremäßigen Entprellung eines Schließers S. Im Falle eines Öffners braucht nur die Fragestellung entsprechend geändert zu werden (geschlossen? → geöffnet?)




⌂ **B001**

a) Ändere die vorige Aufgabe (**B000**) mit Hilfe des obigen Flussdiagramms so um, dass das Prellen des Schalters softwaremäßig unterdrückt wird. Als Totzeit sollen 10 ms gewählt werden. Hierzu kann das Unterprogramm **`W10MS`** aus der Zeitschleifen-Bibliothek "**`SR_TIME_16M.asm`**" verwendet werden. Nenne das Programm "**`B001_debounced_counter_1.asm`**".

b) Ändere das unter a) erstellte Programm jetzt so um, dass das Entprellen in einem Unterprogramm erfolgt. Das Unterprogramm soll den Namen **`DEBSWC`** (*debounce switch closer*) erhalten und in der Datei "**`SR_DEBOUNCE.asm`**" abgespeichert werden. Das verwendete PIN-Register und die Pinnummer werden über die definierten Namen **`SWPinC`** und **`SwitchC`** übergeben (Mittels **`.EQU`** – Anweisung in der Bibliothek oder im Hauptprogramm definiert.
Bsp.: **`.EQU SWPinC = PINB`** oder **`.EQU SwitchC = 0`**). Dies ermöglicht es ein beliebiges Pin auszuwählen ohne das Unterprogramm verändern zu müssen.
Nenne das Programm "**`B001_debounced_counter_2.asm`**".

⌂ **B002**

Die Datei "**`SR_DEBOUNCE.asm`**" soll mit weiteren kommentierten Entprell-Unterprogrammen (dokumentiert mit entsprechenden Flussdiagrammen!) ergänzt werden um so als Bibliothek für spätere Programme genutzt werden zu können.
Soll die Bibliothek verwendet werden, so sind die verwendeten Schalter oder Taster mit Masse zu verbinden und über Pull-Up-Widerstände an VCC anzuschließen.

a) Schreibe ein Unterprogramme zum Entprellen eines Öffners. Nenne das

Unterprogramme **DEBSW0** (debounce switch closer). Übergabe des Pin mit **SWPin0** und **Switch0.**

b) Schreibe zwei weitere Unterprogramme zum Entprellen eines schließenden Tasters und eines öffnenden Tasters. Hier soll das Loslassen des Tasters ebenfalls berücksichtigt werden! Das obige Flussdiagramm muss also praktisch doppelt ausgeführt werden, einmal für das Betätigen und einmal für das Loslassen des Tasters (siehe Zeitdiagramm am Anfang des Kapitels)!
Nenne die Unterprogramme **DEBPBC** (*debounce pushbutton closer*) und **DEBPB0** (Übergabe mit **PBPinC** und **PButtC** bzw. **PBPin0** und **PButt0**).

c) Teste die Unterprogramme mit einem Hauptprogramm, das die Bibliothek einbindet und die Schalter-Betätigungen mit einem Zähler an den LEDs ausgibt. Nenne das Programm "**B002_debounced_counter_3.asm**".

**Bemerkung:** Sollen mehrere Schalter entprellt werden so wiederholt man die entsprechenden Unterprogramme (**DEBSWC1**, **DEBSWC2** ...).

# *Leuchtmustergenerator*

Als weitere Wiederholungsaufgabe soll ein Leuchtmustergenerator programmiert werden:

⌂ **B003** Zeichne ein Flussdiagramm und schreibe ein Programm, welches 4 verschiedene Leuchtmuster auf acht LEDs ausgeben kann (**PORTD**).
Nenne das Programm "**B003_christmas_1.asm**".
Die Leuchtmuster werden in vier unabhängigen Unterprogrammen verwaltet.

Zwei Schalter S3 und S2 (**PB3** und **PB2**) sind für den Zeitabstand zwischen den Ausgaben zuständig (damit Änderungen optisch sichtbar sind). Die Abfrage der Schalter geschieht im Hauptprogramm. Alle Schalter sind gegen Masse geschaltet und benötigen interne Pull-Up-Widerstände. Eine Entprellung ist in diesem Programm nicht nötig. Zur Zeitverzögerung wird die Zeitschleife **WW10MS** im Hauptprogramm verwendet. Die Anfangswerte für **W** (**r25**:**r24**) werden in vier Zeit-Unterprogrammen festgelegt werden.

```
S3 S2  Zeit    Name des Unterprogramms
-------------------------------------
 0 0   50ms         WMS50
 0 1   100ms        WMS100
 1 0   200ms        WMS200
 1 1   500ms        WMS500
```

Mit Hilfe zweier weiterer Schalter S0 und S1 (**PB0** und **PB1**) wird das erwünschte Leuchtmuster im Hauptprogramm ausgewählt (0 bedeutet hier Schalter nicht betätigt!). Blinkmustern ändern dadurch, dass das Blinkmuster im Unterprogramm invertiert wird.

```
S1 S0  Aufgabe           Name des Unterprogramms    LEDs (* Ein, _ Aus)
-------------------------------------------------------------------------
 0  0  Lauflicht Rechts          RUNR                      ** ->
 0  1  Lauflicht Links           RUNL                      <- **
 1  0  Blinkmuster 1            BLINK1                _*_*_*_* / *_*_*_*_
 1  1  Blinkmuster 2            BLINK2                ****____ / ____****
```

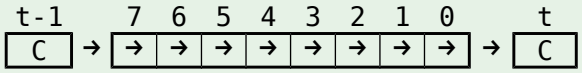
**Bemerkung:** Für das Lauflicht eignen sich die Befehle "**ror**" und "**rol**". Es muss aber eine Korrektur vorgenommen werden, sobald eine Eins zum Carry herausgeschoben wird.
Dies kann dadurch geschehen, dass das Flagregister **SREG** nach dem Rotieren gerettet wird und vor dem Rotieren wiederhergestellt wird, oder durch Setzen des höherwertigen Bits, sobald Eins im Carry auftritt (siehe Befehle "**brcs**" oder "**brcc**").

### ror Rd

**Rechtsverschiebung durch Carry-Flag um eine Stelle (*rotate right through carry*).**

| 1 | 0 | 0 | 1 | 0 | 1 | 0 | d | d | d | d | d | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Es erfolgt eine Rechtsverschiebung aller Bits im Register Rd um eine Stelle. Das bestehende Carry-Flag wird in Bit 7 geschoben. Bit 0 wird in das Carry-Flag hinein geschoben.



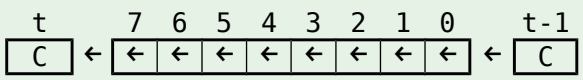

**Beeinflusste Flags: S, V, N, Z, C** **Taktzyklen: 1**

### rol Rd

**Linksverschiebung durch Carry-Flag um eine Stelle (*rotate left through carry*).**

| 0 | 0 | 0 | 1 | 1 | 1 | d | d | d | d | d | d | d | d | d | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Es erfolgt eine Linksverschiebung aller Bits im Register Rd um eine Stelle. Das bestehende Carry-Flag wird in Bit 0 geschoben. Bit 7 wird in das Carry-Flag hinein geschoben.




**Beeinflusste Flags: H, S, V, N, Z, C** **Taktzyklen: 1**

## brcc k

**Bedingter relativer Sprung falls kein Übertrag (*branch if carry cleared*).(k = Adresskonstante)**

| 1 | 1 | 1 | 1 | 0 | 1 | k | k | k | k | k | k | k | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Der Befehl wird meist gleich nach einer arithmetischen oder logischen Operation eingesetzt. **Der Sprung erfolgt falls kein Übertrag (Carry) aufgetreten ist** (C-Flag = 0).
**Beeinflusste Flags: keine**
**Taktzyklen: 1 (kein Sprung), 2 (Sprung)**

## brcs k

**Bedingter relativer Sprung falls Übertrag (*branch if carry set*).(k = Adresskonstante)**

| 1 | 1 | 1 | 1 | 0 | 0 | k | k | k | k | k | k | k | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Der Befehl wird meist gleich nach einer arithmetischen oder logischen Operation eingesetzt. **Der Sprung erfolgt falls ein Übertrag (Carry) aufgetreten ist** (C-Flag = 1).
**Beeinflusste Flags: keine**
**Taktzyklen: 1 (kein Sprung), 2 (Sprung)**

# *Aufgaben*

Zur weiteren Wiederholung können zusätzlich folgende Aufgaben gelöst werden:

⌂ **B004** Schreibe ein Programm (inkl. Flussdiagramm) für einen Vorwärtszähler, dessen Schrittweite an acht Schaltern während des Zählens eingestellt werden kann und, dessen Stand auf acht Leuchtdioden ausgegeben wird.

- Der Zähler soll mit einer Geschwindigkeit von einem Schritt pro halbe Sekunde zählen.
- Der Anfangswert des Zählers beträgt 0.
- Die Schalterabfrage und die Ausgabe an die LEDs soll als Unterprogramm realisiert werden.

Nenne das Programm "**`B004_up_counter.asm`**".

⌂ **B005** Schreibe ein Programm (inkl. Flussdiagramm) für einen Vorwärts-/Rückwärtszähler, dessen Schrittweite während des Zählens an sieben Schaltern (S0-S6) eingestellt werden kann und dessen Stand auf die acht Leuchtdioden ausgegeben wird.

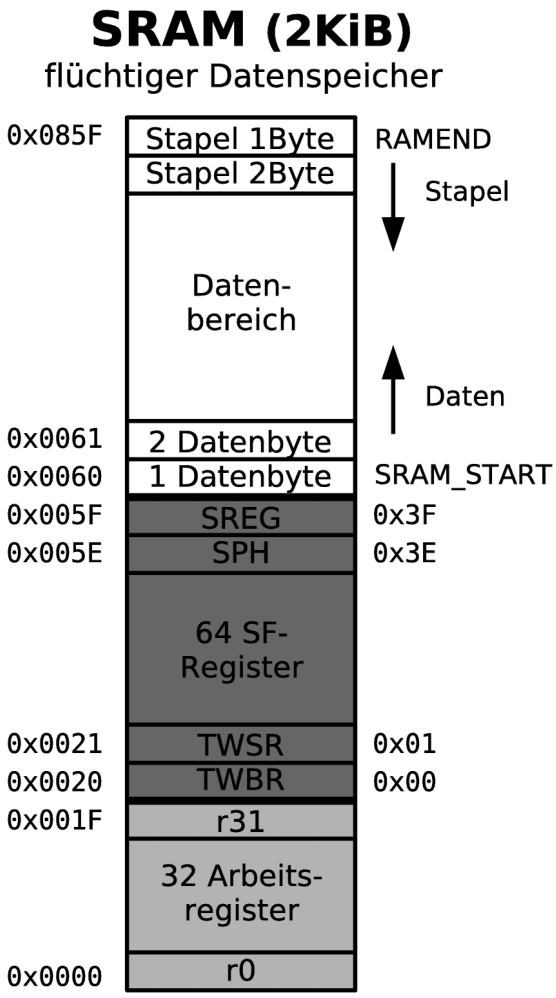
- Ein achter Schalter S7 entscheidet über die Zählrichtung und hat keinen Einfluss auf die Schrittweite.
- Wenn S7 = 0, soll der Zähler vorwärts zählen und für S7 = 1 rückwärts.
- Der Zählvorgang beginnt beim Zählerstand 0x10 und die Zählge schwindigkeit soll 1 Schritt pro 1,2 Sekunden betragen.
- Die Schalterabfrage und die Ausgabe an die LEDs soll als Unterprogramm realisiert werden.

Nenne das Programm "**`B005_up_down_counter.asm`**".

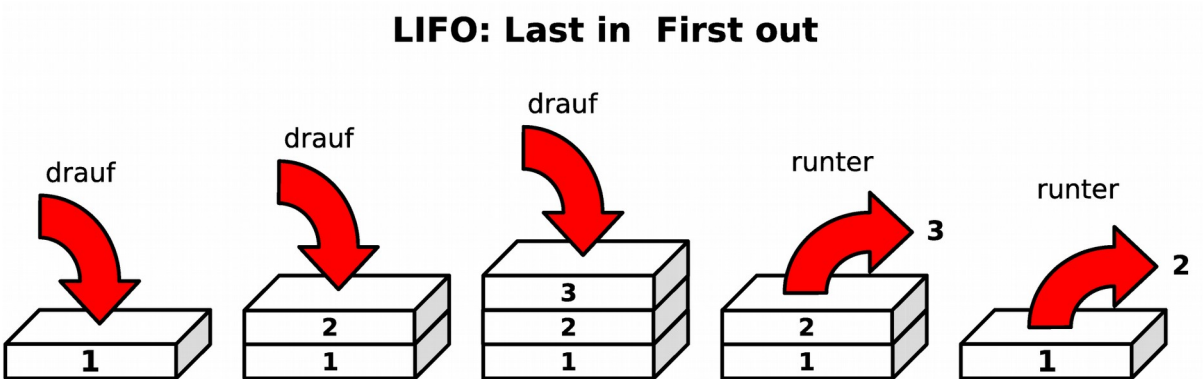
# B1 Stapelspeicher (stack)

## *Arbeitsweise des LIFO-Stapelspeichers*

Im Kapitel "Unterprogramme" wurde schon erwähnt, dass Unterprogramme einen so genannten Stapelspeicher (Kellerspeicher, **Stapel,** *stack*) benötigen um ihre Rücksprungadresse abzuspeichern. Dies gilt ebenfalls für Interrupt-Routinen. Auf den Stapel können auch Variablen zwischengespeichert werden, um sie vor Veränderung zu schützen oder zeitweise aufzubewahren.




Der Stapel ist ein frei definierbarer Speicherbereich im SRAM. Seine maximale Größe ist von der Größe des SRAM abhängig. Das Abspeichern der Variablen passiert in Form eines Stapels. Zuletzt abgespeichertes wird als Erstes wieder ausgelesen, genau so wie man beim Papierstapel das letzte Blatt Papier als Erstes wieder herunter nimmt. Aus dieser Eigenschaft entstand auch die Bezeichnung **LIFO (Last In First Out)**. Die zuletzt abgespeicherte Variable wird dem Stapelspeicher also als Erstes wieder entnommen.




# Der Stapelzeiger SP

Der Stapelspeicher wird über den **Stapelzeiger (*stack pointer*, SP)** angesteuert. Es handelt sich hierbei um ein SF-Register-Doppelregister (Sonderfunktions-Register), welches als Adresszeiger fungiert (vergleiche indirekte Adressierung). Das Doppelregister besteht aus den beiden SF-Registern **SPL** und **SPH.**

## Die SF-Register SPL und SPH

**SPL**-Register: SF-Register-Adresse **0x3D** (SRAM-Adresse **0x005D**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

### SPL = Stack Pointer Low

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **SPL 0x3D** | **SPL7** | **SPL6** | **SPL5** | **SPL4** | **SPL3** | **SPL2** | **SPL1** | **SPL0** |
| Startwert | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

**SPH**-Register: SF-Register-Adresse **0x3E** (SRAM-Adresse **0x005E**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

### SPH = Stack Pointer High

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **SPH 0x3E** | **SPH7** | **SPH6** | **SPH5** | **SPH4** | **SPH3** | **SPH2** | **SPH1** | **SPH0** |
| Startwert | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

# Die Initialisierung des Stapelspeichers

Zur Initialisierung des Stapelspeichers muss der Stapelzeiger **SP** am Anfang des Hauptprogramms mit einer Grundadresse geladen werden. Fehlt das Initialisieren des Stapelzeigers mit einer Grundadresse, so wird bei der Benutzung des Stapelspeichers der Inhalt an einer beliebigen Adresse im SRAM abgelegt, und kann dadurch Daten, die SF-Register oder die Arbeitsregister überschreiben (eventuell Absturz des Programms).

**Der Stapel muss zwingend initialisiert werden bei der Benutzung von**

1. **Unterprogrammen,**
2. **Interrupt-Routinen und**
3. **bei der Benutzung des Stapels als Zwischenspeicher mittels "push"- und "pop"-Befehlen.**

Bei vielen Mikroprozessoren und Mikrocontrollern wächst der Stapelzeiger nach "Unten"[3], so auch beim ATmega32. Der Stapelzeiger adressiert dabei mit zunehmender Füllung des Stapelspeichers immer niedrigere Adressen. Damit Konflikte mit den anderen Daten im SRAM welche von "Unten" nach "Oben" abgespeichert werden zu vermieden werden, wird der Stapel meist auf der höchst liegenden SRAM-Adresse initialisiert. In der Definitionsdatei des Controllers wurde dieser Wert im Namen **RAMEND** (**0x085F** für den ATmega32) festgelegt. In der Definitionsdatei wurde auch die Adresse des Stapelzeigers den Namen **SPL** (**0x3D**) und **SPH** (**0x3E**) zugeordnet.

Wie schon in der Assemblervorlage gesehen, kann die Initialisierung des Stapelbereichs also folgendermaßen aussehen:

```
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,HIGH(RAMEND)       ;RAMEND (SRAM) ist in der Definitions-
        out     SPH,Tmp1                ;datei festgelegt
        ldi     Tmp1,LOW(RAMEND)
        out     SPL,Tmp1
```

Im Ruhezustand (leerer Stapelspeicher) ist der Stapelzeiger mit der Anfangsadresse des Stapelspeichers geladen. Bei jedem Schreiben in den Stapelspeicher wird der Stapelzeiger um 1 ("**push**"-Befehl, Speichern eines Byte) oder 2 (Rücksprungadresse, Speichern von 2 Byte) erniedrigt, nachdem die Daten abgelegt wurden (*post-decrement*). Beim Lesen wird der Stapelzeiger um 1 ("**pop**"-Befehl) oder 2 (Rücksprungadresse) erhöht bevor die Daten abgeholt werden (*pre-increment*).

Durch den Stapelzeiger wird also so ein beliebig großer Stapelspeicher realisiert. Der Prozessor benötigt dazu nur ein 16-Bit-Register.

**Es ist darauf zu achten, dass der Stapelspeicher während des laufenden Programms keine anderen Speicherzellen in denen sich Programmcode oder Daten befinden überschreibt.**

3 Wenn man davon ausgeht, dass die niedrigste Adresse unten liegt.

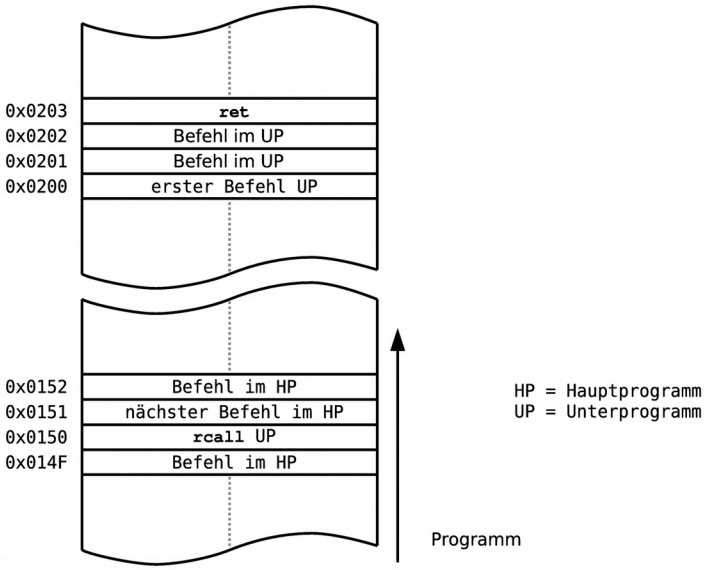
# *Das Abspeichern der Rücksprungadresse*

Beim Aufruf eines Unterprogramms oder einer Interrupt-Routine wird das Hauptprogramm unterbrochen um das Unterprogramm oder die Interrupt-Routine auszuführen. Dabei muss eine Rücksprungadresse gesichert werden damit bei einem "**`ret`**" bzw. "**`reti`**"-Befehl an der Stelle des Hauptprogramms fortgefahren werden kann, wo die Unterbrechung erfolgte.

Beim Aufruf des Unterprogramms oder der Interrupt-Routine wird der Inhalt des Befehlszählers **PC**, der bereits auf den folgenden Befehl des Hauptprogramms zeigt, als Rücksprungadresse an der momentanen Adresse des Stapelzeigers abgelegt. Hierbei wird zuerst niederwertige Byte (LByte) abgelegt, und dann erst das höherwertige Byte (HByte). Der Stapelzeiger wird dabei um zwei dekrementiert.

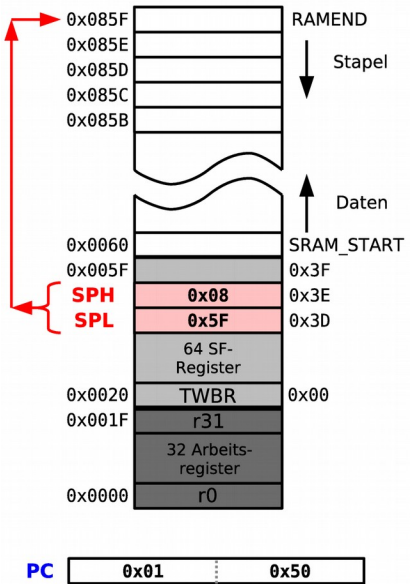
Nach dem Retten der Rücksprungadresse zeigt der Stapelzeiger auf die nächste freie Adresse im Stapel. Die auf dem Stapel abgespeicherte Rücksprungadresse darf während der Ausführung des Unterprogramms nicht verändert werden. Falls doch, muss der alte Inhalt vor Erreichen des Rückkehr-Befehls am Ende des Unterprogramms wiederhergestellt werden. Nachdem das Unterprogramm abgearbeitet wurde, zeigt der Stapelzeiger wieder auf den Anfang des Stapels. Der Stapel ist wieder leer, auch wenn die alten Werte sich noch in den Speicherzellen befinden. Beim nächsten Aufruf des Stapels werden sie überschrieben.

## Programmspeicher (Flash)



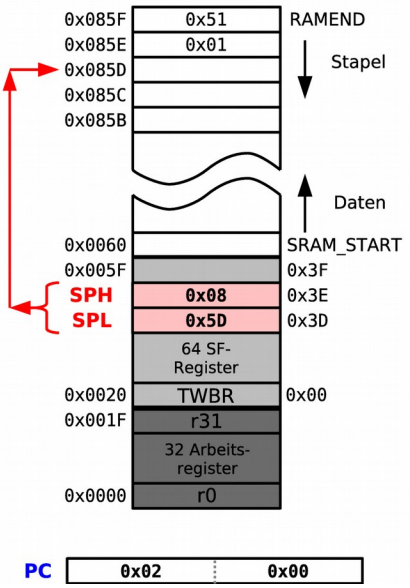

## Datenspeicher (SRAM)

**nach der Initialisierung**
(vor `rcall`)



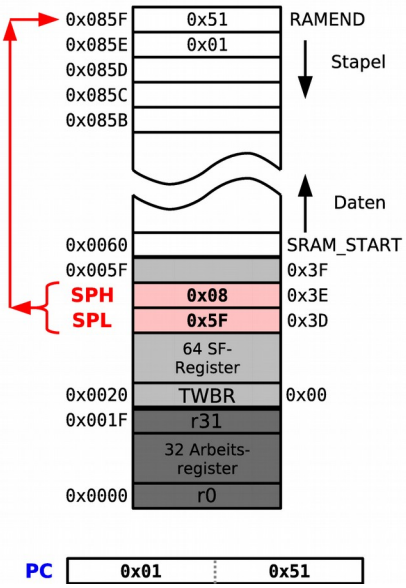

PC | 0x01 | 0x50

**nach dem Aufruf des Unterprogramms** (`rcall`)




PC | 0x02 | 0x00

**nach dem Rücksprung ins Hauptprogramm** (`ret`)
Stapel ist wieder leer!




PC | 0x01 | 0x51

⌂ **B100**

a) Teste das Programm "**B100_test_stack_return_adress_1.asm**" im Debug-Modus des Atmel Studio. Beobachte beim schrittweisen debuggen (Step Into, F11) den Stapelzeiger (I/O View unter CPU oder im Speicherfenster (View Memory), I/O ) sowie den Stapel (Speicherfenster, Data).

```
;-------------------------------------------------------------------------;
;       Hauptprogramm
;-------------------------------------------------------------------------
MAIN:   rcall   SR1             ;Unterprogramm aufrufen
        nop                     ;no operation
        rcall   SR1             ;Unterprogramm nochmals aufrufen
        rjmp    MAIN            ;Weiter

;-------------------------------------------------------------------------
;       Unterprogramme
;-------------------------------------------------------------------------
SR1:    ret                      ;Unterprogramm besteht nur aus Ruecksprungbefehl
```

b) Ändere das Unterprogramm so um, dass ein verschachtelter Aufruf zu einem zweiten Unterprogramm erfolgt, und teste das geänderte Programm nochmals wie in Punkt a).
Speichere das Programm als "**B100_test_stack_return_adress_2.asm**".

```
;-------------------------------------------------------------------------
;       Unterprogramme
;-------------------------------------------------------------------------
SR1:    rcall   SR2         ;verschachteltes Unterprogramm
        ret
SR2:    ret                 ;Unterprogramm besteht nur aus Ruecksprungbefehl
```

# *Funktionsweise bei "push" und "pop"*

Programme und Unterprogramme arbeiten mit den gleichen Arbeitsregistersatz. Möchte man in Unterprogrammen mit lokalen Variablen arbeiten, so muss man die benötigten Register unbedingt vor ihrer ersten Verwendung zwischenspeichern (retten) und sie kurz vor Verlassen des Programms wiederherstellen. Dazu bietet sich der Stapel an. Mit den Befehlen "**push**" und "**pop**" lässt sich dies sehr einfach bewerkstelligen.

Der Stapelspeicher kann aber auch im Hauptprogramm beliebig als Zwischenlager eingesetzt werden (Bsp:. Sicherstellung momentaner Flag-Zustände).

***Push*** bedeutet in den Stapel stoßen oder auf den Stapel laden. Bei jedem "**push**"-Befehl wird der Stapelzeiger **SP** automatisch um 1 dekrementiert. ***Pop*** bedeutet aus dem Stack ziehen oder vom Stack wegnehmen. Bei jedem "**pop**"-Befehl wird der Stapelzeiger **SP** automatisch um 1 inkrementiert.

Mit den "**push**"-Befehlen werden Registerinhalte auf den Stapel geladen, während die "**pop**"-Befehle die Registerinhalte wieder restaurieren.

Als Operand können alle Arbeitsregister verwendet werden.

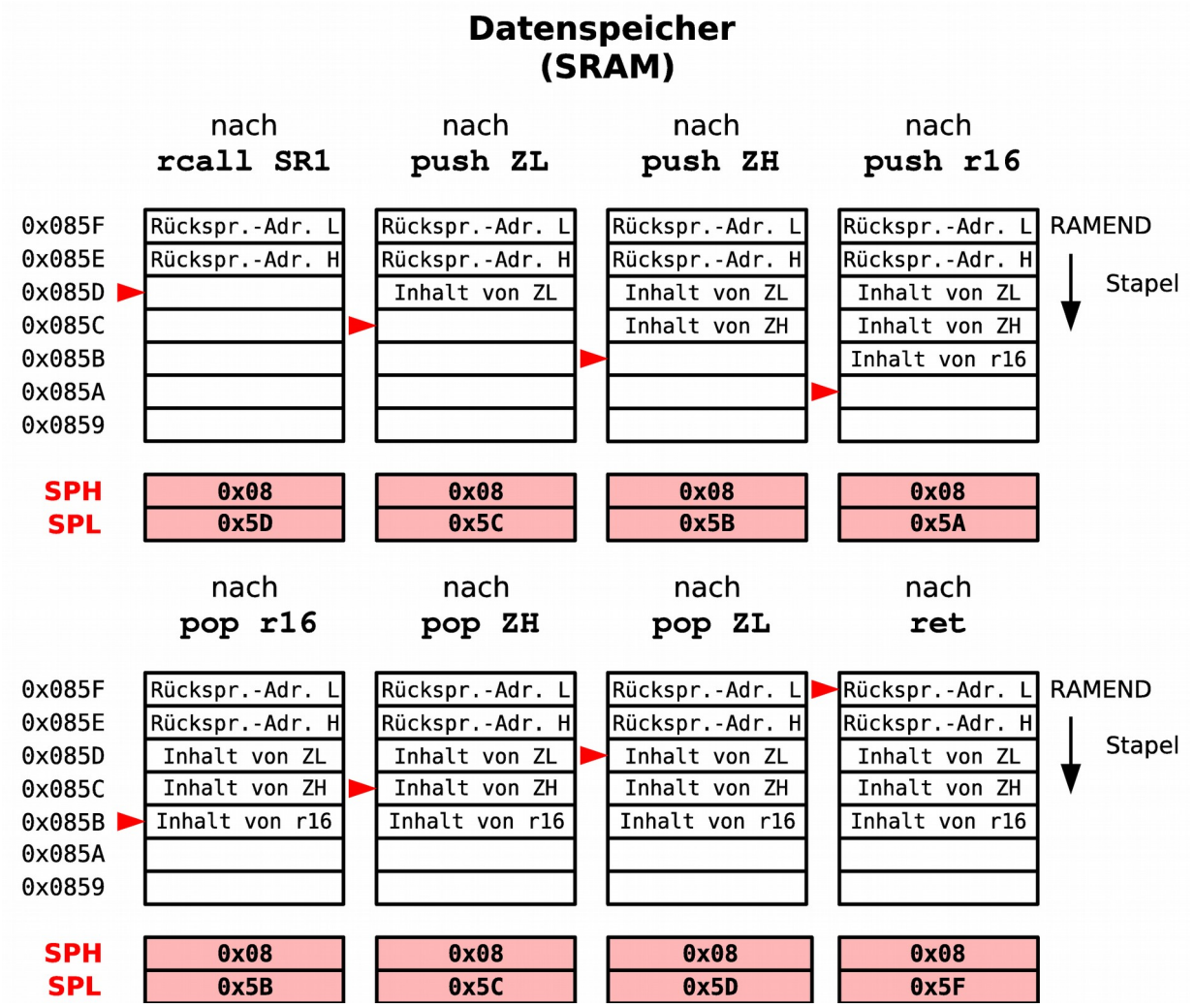
**Wegen des LIFO-Prinzip (Last In First Out) müssen die Register in umgekehrter Reihenfolge vom Stapel genommen werden ("pop") wie sie auf den Stapel gelegt wurden ("push")!**

Beispiel für ein Unterprogramm:

```
SR1:    push    ZL              ;Verwendeten Register retten
        push    ZH
        push    r16

        nop                     ;eigentlicher Code des Unterprogramms

        pop     r16             ;Verwendete Register wiederherstellen
        pop     ZH
        pop     ZL
        ret                     ;Zurueck ins Hauptprogramm
```

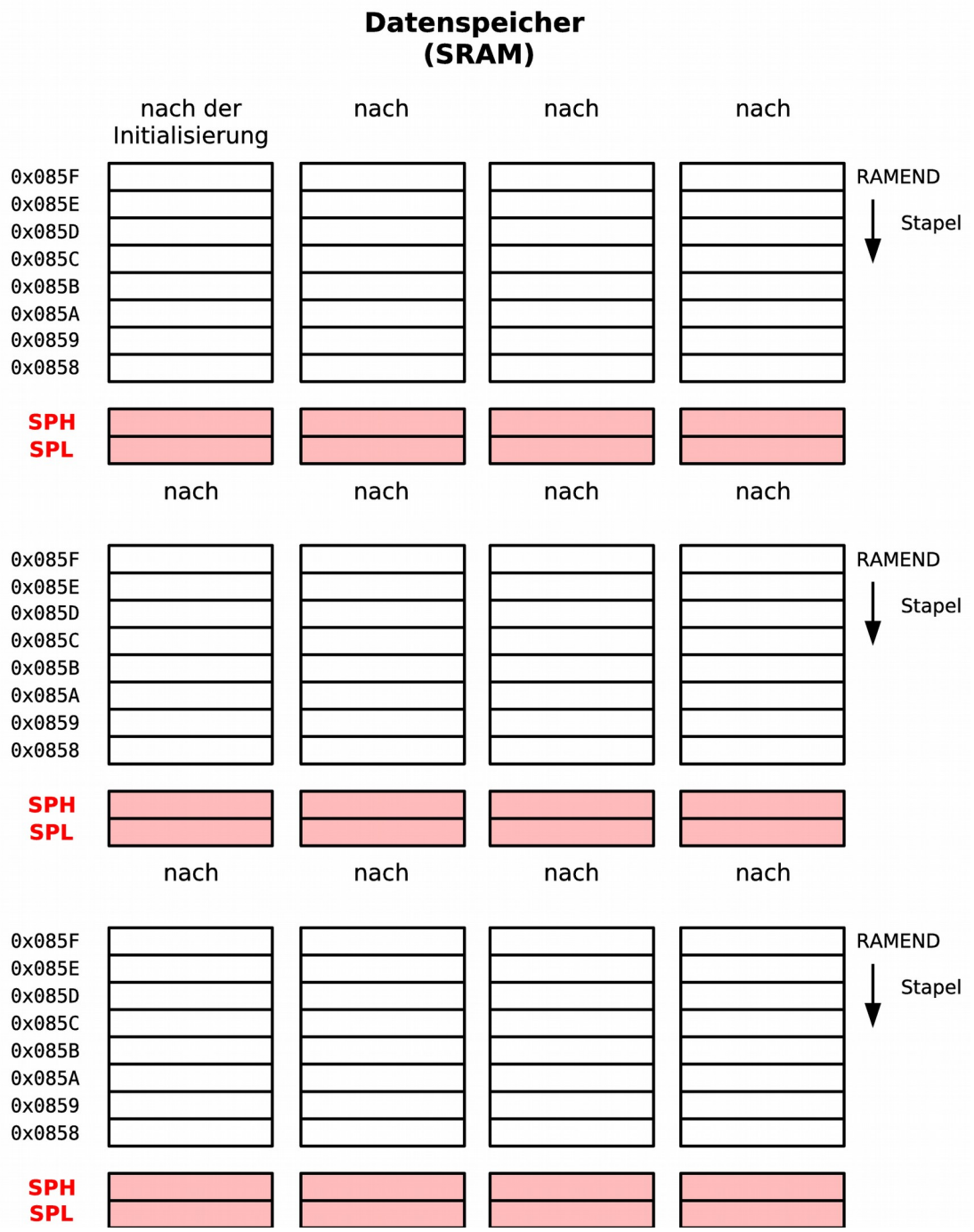



⌂ **B101** Teste das obige Beispiel für ein Unterprogramm im Debug-Modus des Atmel Studio. Beobachte beim schrittweisen Debuggen (Step Into, F11) den Stapelzeiger ("I/O

View" unter CPU oder im Speicherfenster (View Memory), I/O) sowie den Stapel (Speicherfenster, Data).
Speichere das Programm als "**B101_test_stack_push_pop_1.asm**".

⌂ **B102** Überlege beim Programm "**B102_test_stack_push_pop_2.asm**" (hier der Auszug aus der List-Datei) wie der Stapel sich verändert und trage die Inhalte der Speicherzellen und des Stapelzeigers in das folgende Diagramm ein (Der Stapel wurde natürlich initialisiert). Markiere ebenfalls die Position des Stapelzeigers:

```
49: ;-----------------------------------------------------------------------
50: ;       Hauptprogramm
51: ;-----------------------------------------------------------------------
52: 00002E   E111  ldi      r17,0x11        ;Register 17 initialisieren
53: 00002F   E222  ldi      r18,0x22        ;Register 18 initialisieren
54: 000030   D002  rcall    SR1             ;Unterprogramm aufrufen
55: 000031   0000  nop                      ;no operation
56: 000032   CFFB  rjmp     MAIN            ;Weiter
57:
58: ;-----------------------------------------------------------------------
59: ;       Unterprogramme
60: ;-----------------------------------------------------------------------
61: 000033   931F  push     r17             ;SR1: Verwendete Register retten
62: 000034   932F  push     r18
63: 000035   E313  ldi      r17,0x33        ;r17 fuer lokale Verwend. SR1 init.
64: 000036   D003  rcall    SR2
65: 000037   912F  pop      r18             ;Verwendete Reg. wiederherstellen
66: 000038   911F  pop      r17
67: 000039   9508  ret                      ;Zurueck ins Hauptprogramm
68: 00003A   931F  push     r17             ;SR2: Verwendeten Register retten
69: 00003B   E414  ldi      r17,0x44        ;r17 fuer lokale Verwend. SR2 init.
70: 00003C   911F  pop      r17             ;Verwendete Reg. wiederherstellen
71: 00003D   9508  ret                      ;Zurueck ins Unterprogramm SR1
```




⌂ **B103**

a) Entwickle das Flussdiagramm zu einem Programm, das in einer Schleife fünf im Zahlenraum nacheinander folgende 16-Bit-Zahlen im Stapelspeicher ablegt und diese in einem zweiten Schritt über eine Schleife wieder ausliest und im SRAM ab Adresse **`0x0060`** ablegt (*Lowest Byte First*).
b) Schreibe das Programm und nenne es "**`B103_push_pop_stack.asm`**"
c) Betrachte die Funktionsweise des Programms mit Hilfe des Atmel Studio.

# B2 Arbeiten mit Tabellen

In diesem Kapitel soll die direkte und die indirekte Adressierung des Datenspeichers bzw. des Programmspeichers wiederholt und vertieft werden.

Viele Programme müssen größerer Datenmengen verarbeiten. Da dann nicht ausreichend Arbeitsregister zur Verfügung um die Daten abzuspeichern, wird der Datenspeicher (SRAM) genutzt. Müssen nur einige wenige Bytes im Datenspeicher abgelegt werden, so nutzt man die **direkte Adressierung** des SRAM.

Soll mit mehr als 5-10 Bytes gearbeitet werden, so werden diese meist in Tabellen abgelegt. Um mit größeren Tabellen zu Arbeiten benötigt man die **indirekte Adressierung**.

Bei der indirekten Adressierung ist die Adresse des Operanden nicht direkt im Befehl enthalten, sondern in einem der drei Registerpaare **X**, **Y** oder **Z**, welche als Adresszeiger (Indexregister, *Pointer*) dienen. Vor jeder indirekten Adressierung muss der Adresszeiger initialisiert werden!

## *Variablen im Datenspeicher*

"Speichern" (*store direct to sram*, **sts**) und "Laden" (*load direct from sram*, **lds**) sind die beiden Befehle die der Verarbeitung von Variablen mittels direkter Adressierung im Datenspeicher dienen.

Am flexibelsten ist es den Speicher mit Hilfe von symbolischen Adressen (Labeln) zu organisieren. Dadurch ist man nicht an absolute Adressen gebunden und der Speicher kann optimal genutzt werden.

Mit der Assembleranweisung **.DSEG** teilt man dazu dem Assembler mit, dass die folgenden Zeilen sich auf den Datenspeicher (SRAM) beziehen. Dann reserviert man mit der Assembleranweisung **.BYTE** eine bestimmte Anzahl von Speicherzellen. Der vorangestellte Label dient als symbolische Adresse. Mit **.CSEG** wird dem Assembler mitgeteilt, dass der Rest im Programmspeicher landen soll.

Beispiel:

```
;---------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;---------------------------------------------------------------------------
.DSEG                           ;was ab hier folgt kommt in den SRAM-Speicher
VAR1:   .BYTE  1                ;1 Byte grosse Variable im Datensegment
VAR2:   .BYTE  4                ;4 Byte grosse Variable im Datensegment

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                            ;was ab hier folgt kommt in den FLASH-Speicher
```

Der Zugriff auf die Variablen erfolgt zum Beispiel mit:

```
        ldi     Tmp1,120
        sts     VAR1,Tmp1
        lds     r19,VAR2
        lds     r20,VAR2+1
        lds     r21,VAR2+2
        lds     r22,VAR2+3
```

# Tabellen im Datenspeicher

Zur Verarbeitung von Tabellen im Datenspeicher dienen ebenfalls ein "Speicher"-Befehl (*store indirect*, **st**) und ein "Laden"-Befehl (*load indirect*, **ld**). Besonders praktisch sind Befehle zur indirekte Adressierung mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers.

Auch hier ist es am flexibelsten den Speicher mit Hilfe von symbolischen Adressen (Labeln) zu organisieren. Dadurch ist man nicht an absolute Adressen gebunden und der Speicher kann optimal genutzt werden.

Im Kapitel zur Adressierung wurde ein Assemblerprogramm erstellt, dessen Aufgabe es war den SRAM-Speicher mit den Dezimalzahlen 0 bis 255 aufzufüllen. Dieses Programm soll jetzt ohne absolute Adresse programmiert werden. Eine mögliche Lösung könnte folgendermaßen aussehen:

```
;-----------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;-----------------------------------------------------------------------------
.DSEG                           ;was ab hier folgt kommt in den SRAM-Speicher
TAB1:   .BYTE  256              ;256 Byte grosse Tabelle im Datensegment

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET:  rjmp    INIT            ;springe nach INIT (ueberspringe ISR Vektoren)

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Cnt1 = r18              ;Register 18 dient als erster Zaehler
        clr     Cnt1            ;Zaehler (Zahl) mit 0 initialisieren
        ldi     XL,LOW(TAB1)    ;Adresszeiger mit Tabellenangfang initialisieren
        ldi     XH,HIGH(TAB1)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   st      X,Cnt1          ;Speichere den Inhalt des Zaehlers in
                                ;den Datenspeicher. Die Adresse befindet sich
                                ;im Doppelregister X
        adiw    XL,1            ;Inkrementiere den 16-Bit-Adresszeiger X
        inc     Cnt1            ;Inkrementiere den Zaehler (Zahl)
        cpi     Cnt1,0          ;Vergleiche mit Null (diese Zeile kann man auch weglassen!)
        brne    MAIN            ;Bleib solange in der Schleife bis 256
                                ;(Ueberlauf, Register wieder auf null)
        ;ab hier wird das Programm fuer die Aufgabe B200 erweitert
        ;Ende des Hauptprogramms (falls keine Endlosschleife im Hauptprogramm)
END:    rjmp    END             ;Endlosschleife

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

**Bemerkungen:** Die beiden Programmzeilen **`st X,Cnt1`** und **`adiw XL,1`** können durch **`st X+,Cnt1`** ersetzt werden.

Damit auch wirklich 256 Werte(0-255) geschrieben werden, wird erst abgebrochen, wenn das Register überläuft, also wieder Null wird! Die Zeile mit dem Vergleichsbefehl dient der Übersichtlichkeit. Man kann sie

weglassen, da das Inkrementieren eine Rechenoperation ist und somit auch das **Zero**-Flag im **SREG**-Register setzt.

⌂ **B200** Das obige Programm soll erweitert werden.

a) Zusätzlich zum Erstellen der Zahlentabelle sollen die ersten 128 Byte der erstellten Tabelle in eine zweite Tabelle "**Tab2**" kopiert werden. Die Zahlen in der zweiten Tabelle sollen als **ASCII**-Zeichen interpretiert werden (siehe **ASCII**-Tabelle im Anhang). Die Kopie soll dann nur noch die Zeichen "**A-Z**" enthalten. Die restlichen Zeichen sollen mit dem Nullbyte (**0x00**) aufgefüllt werden.
Gib dem Programm den Namen "**B200_sram_indirect_2.asm**" und dokumentiere das Programm mit Hilfe eines Flussdiagramms.

b) Teste das Programm im Atmel Studio mit dem Debugger. Lass das Programm durchlaufen (Autostep ("Alt+F5")) und beobachte das Resultat im Speicherfenster ("View/Memory" oder "Alt+4").

c) Für Fleißige: Ändere das Programm so um, dass zusätzlich "**0-9**", "**a-z**" gültig sind. Nenne das Programm "**B200_sram_indirect_3.asm**" und teste es ebenfalls.

## Tipps zur Programmieraufgabe:

Beim Kopieren von Tabellen wird mit zwei verschiedenen Adresszeigern (z.B. **X** und **Y**) gearbeitet. Zum Filtern von **ASCII**-Zeichen kann man die Befehle "**brsh**" (Verzweige falls gleich oder größer) und "**brlo**" (Verzweige falls kleiner) verwenden.

**brsh k**

**Bedingter relativer Sprung falls (vorzeichenlos) größer oder gleich (*branch if same or higher*).**

| 1 | 1 | 1 | 1 | 0 | 1 | k | k | k | k | k | k | k | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Der Befehl wird meist gleich nach einem Vergleichsbefehl (**cp**, **cpi**) eingesetzt. **Der Sprung erfolgt falls kein Übertrag (Carry) aufgetreten ist** (**C**-Flag = 0). Beim Vergleich vorzeichenloser Zahlen erfolgt der Sprung wenn der Zieloperand (Rd) ⩾ dem Quelloperand (Rr, K). Der Befehl entspricht dem Befehl brcc.
**Beeinflusste Flags: keine**
**Taktzyklen: 1 (kein Sprung), 2 (Sprung)**

**brlo k**

**Bedingter relativer Sprung falls (vorzeichenlos) kleiner (*branch if lower*).(k = Adresskonstante)**

| 1 | 1 | 1 | 1 | 0 | 0 | k | k | k | k | k | k | k | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Der Befehl wird meist gleich nach einem Vergleichsbefehl (**cp**, **cpi**) eingesetzt. **Der Sprung erfolgt falls ein Übertrag (Carry) aufgetreten ist** (**C**-Flag = 1). Beim Vergleich vorzeichenloser Zahlen erfolgt der Sprung wenn der Zieloperand (Rd) < dem Quelloperand (Rr, K). Der Befehl entspricht dem Befehl brcs.

**Beeinflusste Flags: keine**
**Taktzyklen: 1 (kein Sprung), 2 (Sprung)**

# Tabellen im Programmspeicher

Öfter benötigt man kleine Tabellen mit festen Werten. Da die Programmierung des EEPROM recht aufwändig ist, bietet es sich an diese Tabellen im Programmspeicher mit abzulegen.

"Lade Programmspeicher" (*load program memory*, **lpm**) ermöglicht es ein beliebiges Datenbyte aus dem Programmspeicher (Flash) in ein Arbeitsregister zu laden. Die indirekte Adresse muss sich dazu im **Z**-Pointer (Adresszeiger) befinden (siehe Kapitel "Befehle und Adressierung).

Mit der Direktive "**.ORG**" besteht die Möglichkeit den Beginn der Tabelle festzulegen. Ohne diese Anweisung wird die Tabelle gleich hinter dem Programm abgespeichert[4]. Am sichersten ist es die "**.ORG**" Direktive nicht einzusetzen, sondern nur mit Labeln (symbolischen Adressen) zu arbeiten. Der Programmieraufwand ist dadurch eventuell größer, jedoch kann dann der Speicher optimal genutzt werden und man läuft nicht Gefahr den Speicher bei großen Projekten mit eingebundenen Bibliotheken mehrfach zu belegen.

Die Direktive "**.DB**" (define Byte) ermöglicht es die Tabelle (Liste mit Bytekonstanten) zu definieren. Da der Programmspeicher in Worten (2 Byte) organisiert ist, ist es nötig die Wortadresse mit dem Faktor zwei zu multiplizieren, da der Adresszeiger nur byteweise adressieren kann. Dies kann einfach im Assembler mit Multiplikationszeichen geschehen.

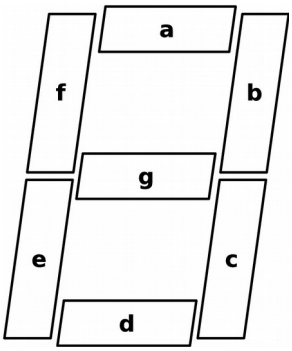
## Ansteuerung eines Sieben-Segment-Displays

### Ansteuerung einer Displaystelle

⌂ **B201** Das Beispielprogramm "**A408_flash_indirect_textstring.asm**" kann als Vorlage dienen (Initialisierung des Stapels nicht vergessen, keine **.ORG** Anweisung vor dem Tabellenlabel).
a) 4 Schalter sind an die unteren 4 Pins von Port B anzuschließen. **PINB** ist dann in das Arbeitsregister **r24** einzulesen. Der an den Schaltern eingestellte Wert (unteres Nibble (4 Bit) des Registers) soll in hexadezimaler Schreibweise auf einer Siebensegment-Anzeige ausgegeben werden. Die sieben Segmente (**a-f**) werden

4 Eine Tabelle sollte sich immer zum Schluss der Assemblerdatei befinden, da sonst Teile des Programms oder eines Unterprogramms hinter der Tabelle landen und nicht mehr richtig angesprochen werden!

durch die unteren 7 Bit des Port D angesteuert. Das hochwertigste Bit aktiviert die Anzeige. Erstelle die Dekodiertabelle!




| Bit: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| Seg: | | g | f | e | d | c | b | a | Byte: |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0xBF |
| 1 | 1 | | | | | | | | |
| 2 | 1 | | | | | | | | |
| 3 | 1 | | | | | | | | |
| 4 | 1 | | | | | | | | |
| 5 | 1 | | | | | | | | |
| 6 | 1 | | | | | | | | |
| 7 | 1 | | | | | | | | |
| 8 | 1 | | | | | | | | |
| 9 | 1 | | | | | | | | |
| A | 1 | | | | | | | | |
| b | 1 | | | | | | | | |
| C | 1 | | | | | | | | |
| d | 1 | | | | | | | | |
| E | 1 | | | | | | | | |
| F | 1 | | | | | | | | |

b) Die Dekodiertabelle soll sich hinter dem Programmcode befinden. Die Addition des ausmaskierten unteren Nibbles des Registers `r24` zur Anfangsadresse der Tabelle dient als Adresszeiger!
Achtung! Es muss eine 16-Bit-Addition durchgeführt werden!
Zeichne das Flussdiagramm. Schreibe den Assemblercode und speichere ihn als "**`B201_flash_indirect_numdisplay_1.asm`**".
Teste dein Programm mit unterschiedlichen Schalterstellungen.

c) Erweitere die Hardware um 4 Schalter. Ändere das Programm (und Flussdiagramm) so, dass im Sekundenrhythmus abwechselnd das untere (Schalter $2^0$ bis $2^3$) und das obere Nibble (Schalter $2^4$ bis $2^7$) des Registers auf der

Siebensegmentanzeige dargestellt werden.
Speichere das erweiterte Programm als
"**B201_flash_indirect_numdisplay_2.asm**".

### Tipps zur Programmieraufgabe:

Für eine 16-Bit Addition benötigt man die Befehle "**add**" und "**adc**". Im ersten Schritt werden mit "**add**" die beiden **niederwertigen Register** addiert. Der eventuell auftretende Übertrag (*carry*) wird mit dem "**adc**"-Befehl berücksichtigt. Mit diesem werden die **hochwertigen Register** dann addiert.
Wird ein 8-Bit-Register zu einem 16-Bit-Register addiert (wie in der obigen Aufgabe), so wird bei der Addition der hochwertigen Register Null addiert. Da der "**adc**" Befehl nicht als unmittelbarer Befehl zur Verfügung steht, wird ein gelöschtes Hilfsregister (Zero (**clr r15**) wurde im Template definiert) benötigt.

```
        add     ZL,r24          ;Addition der niederwertigen Register
        adc     ZH,Zero         ;eventuelles Carry berücksichtigen
```

**adc Rd,Rr**

**Addition des Arbeitsregisters Rd mit dem Arbeitsregister Rr mit Berücksichtigung des Übertrags (*add with carry*).**

| 0 | 0 | 0 | 1 | 1 | 1 | r | d | d | d | d | d | r | r | r | r |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Resultat in Rd. Carry-Flag wird zur Summe hinzuaddiert.
**Beeinflusste Flags: H, S, V, N, Z, C  Taktzyklen: 1**

Mit dem "**swap**"-Befehl können die beiden Nibble (4 Bit-Gruppe, eine hexadezimale Stelle) eines Byte einfach vertauscht werden (Unterpunkt c))!

**swap Rd**

**Niederwertiges und hochwertiges Nibble (4 Bit) eines Registers vertauschen (*swap nibbles*).**

| 1 | 0 | 0 | 1 | 0 | 1 | 0 | d | d | d | d | d | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Beeinflusste Flags: keine  Taktzyklen: 1**

## Ansteuerung von vier Displaystellen

Es besteht eine Unterprogramm-Bibliothek mit dem Namen "**SR_NUMDISPLAY.asm**".
(http://www.weigu.lu/tutorial/avr_assembler).

Sie erlaubt die Darstellung des Registerinhalts des Doppelregisters **W (r25:r24)** während einer bestimmten Zeit auf einem 4-stelligen 7-Segment-Display. Dazu müssen nur im Hauptprogramm die beiden Unterprogramme **D7SINI** und **D7SHEX** aufgerufen werden. Das erste Unterprogramm initialisiert die Port Pins (können im Zuweisungsteil der Bibliothek geändert werden), das zweite stellt den Inhalt des Doppelregisters **W** dar.

```
;*****************************************************************************
;*
;*      Titel:  Unterprogramme zur Darstellung von Daten auf einer 4-stelligen
;*              7-Segment-Anzeige (SR_NUMDISPLAY.asm)
;*
;*      Datum:  22/11/11        Version:        0.4 (02/01/13)
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32A        Quarzfrequenz:  16MHz
;*      Eingaenge:
;*      Ausgaenge:      Pin 0-7 (a-g, DP) am SegPort
;*                      4 Pin am DigPort
;*
;*      Informationen zur Funktionsweise:
;*
;*      Eine Zeitschleifenbibliothek (UP W1ms) muss im Hauptprogramm eingebunden
;*      sein. Bei den Zuweisungen wird festgelegt, welche Ports und Pins
;*      verwendet werden sollen (1 ganzes Port für die Segmente, 4 Bit eines
;*      Ports fuer die Stellen).
;*      Das Unterprogramm D7SINI muss im Hauptprogramm in der Initialisierungs-
;*      phase aufgerufen werden (Initialisiert Portpins als Ausgang).
;*
;*      Beim Aufruf von D7SHEX wird der Inhalt des Doppelregisters W (r25:r24)
;*      waehrend einer definierten Zeit in Millisekunden in Hexadezimal am
;*      Display ausgegeben.
;*
;*      Unterprogramme:
;*
;*      D7SINI          Muss im Hauptprogramm in der Initialisierungsphase
;*                      aufgerufen werden (Initialisiert Portpins als Ausgang)
;*      D7SHEX          Der Inhalt des Doppelregisters W (r25:r24) wird waehrend
;*                      einer definierten Zeit in Millisekunden in Hex am
;*                      Display ausgegeben. Die Dekodiertabelle befindet sich im
;*                      Programmspeicher. Die Hexstelle addiert zur Anfangs-
;*                      adresse dient als Adresszeiger.
;*
;*
;*      Copyright (c) 2011   Guy WEILER         weigu[at]weigu[dot]lu
;*
;*      Hiermit wird unentgeltlich, jeder Person, die eine Kopie der Software
;*      und der zugehoerigen Dokumentationen (die "Software") erhaelt, die
;*      Erlaubnis erteilt, uneingeschraenkt zu benutzen, inklusive und ohne
;*      Ausnahme, dem Recht, sie zu verwenden, kopieren, aendern, fusionieren,
;*      verlegen, verbreiten, unterlizenzieren und/oder zu verkaufen, und
;*      Personen, die diese Software erhalten, diese Rechte zu geben, unter den
;*      folgenden Bedingungen:
;*
;*      ;Der obige Urheberrechtsvermerk und dieser Erlaubnisvermerk sind in alle
;*      ;Kopien oder Teilkopien der Software beizulegen.
;*      ;
;*      ;DIE SOFTWARE WIRD OHNE JEDE AUSDRUECKLICHE ODER IMPLIZIERTE GARANTIE
;*      ;BEREITGESTELLT, EINSCHLIESSLICH DER GARANTIE ZUR BENUTZUNG FUER DEN
;*      ;VORGESEHENEN ODER EINEM BESTIMMTEN ZWECK SOWIE JEGLICHER RECHTS-
;*      ;VERLETZUNG, JEDOCH NICHT DARAUF BESCHRAENKT. IN KEINEM FALL SIND DIE
;*      ;AUTOREN ODER COPYRIGHTINHABER FUER JEGLICHEN SCHADEN ODER SONSTIGE
;*      ;ANSPRUECHE HAFTBAR ZU MACHEN, OB INFOLGE DER ERFUELLUNG EINES
;*      ;VERTRAGES, EINES DELIKTES ODER ANDERS IM ZUSAMMENHANG MIT DER SOFTWARE
;*      ;ODER SONSTIGER VERWENDUNG DER SOFTWARE ENTSTANDEN.
;*
;*****************************************************************************

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
;       Zuweisungen
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EQU    SegDDR  = DDRC          ;Segment-Port definieren (8 Bit)
.EQU    SegPort = PORTC
.EQU    DigDDR  = DDRB          ;Digit-Port (Stellen) definieren
.EQU    DigPort = PORTB
.EQU    D1PinNr = 0
.EQU    D2PinNr = 1
.EQU    D3PinNr = 2
.EQU    D4PinNr = 3

.EQU    DispTime = 1000         ;Zeit der Darstellung in Millisekunden
.                               ;(min. 4ms; max. 262140 ms)

.EQU    D1_Dot = 0              ;hier kann der Dezimalpunkt (Bit 7) fuer jede
.EQU    D2_Dot = 0              ;Stelle freigeschaltet werden (1 = on, 0 = off)
.EQU    D3_Dot = 1
.EQU    D4_Dot = 0

;-------------------------------------------------------------------------------
;       7 Segmentanzeige initialisieren
;-------------------------------------------------------------------------------
D7SINI: push    r16
        ser     r16
        out     SegDDR,r16
        sbi     DigDDR,D1PinNr
        sbi     DigDDR,D2PinNr
        sbi     DigDDR,D3PinNr
        sbi     DigDDR,D4PinNr
        pop     r16
        ret

;-------------------------------------------------------------------------------
;       Unterprogramm zur Darstellung des Doppelreg. W auf dem 7-Seg.-Display
;-------------------------------------------------------------------------------
D7SHEX: push    r16             ;alle verwendeten Register (inkl. SREG) retten
        in      r16,SREG
        push    r16
        push    r17
        push    r18
        push    r19
        push    r24
        push    r25
        push    ZL
        push    ZH

        ;16 Bit Zeitzaehler r19:r18 initialisieren
        ldi     r18,LOW(DispTime/4)     ;Minimale Zeit = 4*1ms=4ms)
        ldi     r19,HIGH(DispTime/4)

D7SHE1: ;erste Stelle anzeigen
        ldi     ZL,LOW(NDISP2*2)  ;Adresszeiger mit der Adresse der Tab.*2
        ldi     ZH,HIGH(NDISP2*2) ;initialisieren (Worte statt Bytes)
        mov     r16,r24         ;Adresszeiger errechnen
        andi    r16,0x0F        ;Maskieren des untersten Nibble von r24
        clr     r17             ;Zwischenspeicher = 0
        add     ZL,r16          ;Addiere Zaehlerstand zur Basisadresse
        adc     ZH,r17          ;ZH erhoehen falls Carry!
        lpm     r16,Z           ;Speichere das Zeichen der Speicherzeile deren
                                ;Adresse im Z-Pointer steht in das Arbeits-
                                ;register r16.
        .IF     D1_Dot == 1     ;Setze Dezimalpunkt Stelle 1 falls D1_Dot == 1
        ori     r16,0x80
        .ENDIF
        sbi     DigPort,D1PinNr ;Erste Stelle einschalten
        out     SegPort,r16     ;Zeichen am SegPort ausgeben
        rcall   W1ms            ;Warte 1ms
        cbi     DigPort,D1PinNr ;Erste Stelle ausschalten

        ;zweite Stelle anzeigen
        ldi     ZL,LOW(NDISP2*2)  ;Adresszeiger mit der Adresse der Tab.*2
```

```
        ldi     ZH,HIGH(NDISP2*2) ;initialisieren (Worte statt Bytes)
        mov     r16,r24           ;Adresszeiger errechnen
        andi    r16,0xF0          ;Maskieren des obersten Nibble von r24
        swap    r16               ;Vertausche beide Nibble
        clr     r17               ;Zwischenspeicher = 0
        add     ZL,r16            ;Addiere Zaehlerstand zur Basisadresse
        adc     ZH,r17            ;ZH erhoehen falls Carry!
        lpm     r16,Z             ;Speichere das Zeichen der Speicherzeile deren
                                  ;Adresse im Z-Pointer steht in das Arbeits-
                                  ;register r16.
        .IF     D2_Dot == 1       ;Setze Dezimalpunkt Stelle 2 falls D2_Dot == 1
        ori     r16,0x80
        .ENDIF
        sbi     DigPort,D2PinNr ;Zweite Stelle einschalten
        out     SegPort,r16       ;Zeichen am SegPort ausgeben
        rcall   W1ms              ;Warte 1ms
        cbi     DigPort,D2PinNr ;Zweite Stelle ausschalten


        ;dritte Stelle anzeigen
        ldi     ZL,LOW(NDISP2*2)  ;Adresszeiger mit der Adresse der Tab.*2
        ldi     ZH,HIGH(NDISP2*2) ;initialisieren (Worte statt Bytes)
        mov     r16,r25           ;Adresszeiger errechnen
        andi    r16,0x0F          ;Maskieren des untersten Nibble von r25
        clr     r17               ;Zwischenspeicher = 0
        add     ZL,r16            ;Addiere Zaehlerstand zur Basisadresse
        adc     ZH,r17            ;ZH erhoehen falls Carry!
        lpm     r16,Z             ;Speichere das Zeichen der Speicherzeile deren
                                  ;Adresse im Z-Pointer steht in das Arbeits-
                                  ;register r16.
        .IF     D3_Dot == 1       ;Setze Dezimalpunkt Stelle 3 falls D3_Dot == 1
        ori     r16,0x80
        .ENDIF
        sbi     DigPort,D3PinNr ;Dritte Stelle einschalten
        out     SegPort,r16       ;Zeichen am SegPort ausgeben
        rcall   W1ms              ;Warte 1ms
        cbi     DigPort,D3PinNr ;Dritte Stelle ausschalten

        ;vierte Stelle anzeigen
        ldi     ZL,LOW(NDISP2*2)  ;Adresszeiger mit der Adresse der Tab.*2
        ldi     ZH,HIGH(NDISP2*2) ;initialisieren (Worte statt Bytes)
        mov     r16,r25           ;Adresszeiger errechnen
        andi    r16,0xF0          ;Maskieren des obersten Nibble von r25
        swap    r16               ;Vertausche beide Nibble
        clr     r17               ;Zwischenspeicher = 0
        add     ZL,r16            ;Addiere Zaehlerstand zur Basisadresse
        adc     ZH,r17            ;ZH erhoehen falls Carry!
        lpm     r16,Z             ;Speichere das Zeichen der Speicherzeile deren
                                  ;Adresse im Z-Pointer steht in das Arbeits-
                                  ;register r16.
        .IF     D4_Dot == 1       ;Setze Dezimalpunkt Stelle 4 falls D4_Dot == 1
        ori     r16,0x80
        .ENDIF
        sbi     DigPort,D4PinNr ;Vierte Stelle einschalten
        out     SegPort,r16       ;Zeichen am SegPort ausgeben
        rcall   W1ms              ;Warte 1ms
        cbi     DigPort,D4PinNr ;Vierte Stelle ausschalten

        ;Zeitzaehler ueberpruefen
        subi    r18,1             ;Zaehler dekrementieren und Zeit ueberpruefen
        sbci    r19,0
        brne    D7SHE1

        pop     ZH
        pop     ZL
        pop     r25
        pop     r24
        pop     r19
        pop     r18
        pop     r17
        pop     r16
```

```
        out     SREG,r16
        pop     r16
        ret                     ;Zurueck ins Hauptprogramm

;-----------------------------------------------------------------------------
;       Tabelle         (7-Segment-Dekodiertabelle (hoechstwertigstes Bit = 0))
;-----------------------------------------------------------------------------
NDISP2:
.DB     0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07     ;Dekodiertabelle fuer
.DB     0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71     ;alle Hex-Zahlen
```

⌂ **B202**

a) Analysiere die Bibliothek, und erstelle ein Flussdiagramm für das Unterprogramm **D7SHEX**.

b) Schreibe ein Assemblerprogramm, das im Sekundentakt alle möglichen Zustände des **W**-Registers durchläuft und teste es auf mit einem vierstelligen LED-Display (z.B. MICES2-Board, siehe Anhang).
Das Programm soll als "**B202_numdisplay_4.asm**" abgespeichert werden.

c) Wie lange benötigt das Programm, um alle Zustände zu durchlaufen?

## *Lauflicht aus einer Tabelle:*

⌂ **B203**

Ein einfaches Lauflicht über 16 LEDs (2 Ports) soll mit Hilfe einer Flash-Tabelle realisiert werden.
Schreibe das Assemblerprogramm und nenne es "**B203_christmas_2.asm**".

# *Blinkmuster aus einer Tabelle:*

Zur Festigung des Gelernten kann zusätzlich folgende Aufgabe gelöst werden:

⌂ **B204** Ein Programm soll 32 verschiedene Blinkmuster ausgeben können. Die Blinkmuster befinden sich in einer Tabelle im Programmspeicher. Am Anfang des Programms soll die ganze Tabelle ins SRAM kopiert werden.
Über 5 Schalter soll das Bitmuster aus der SRAM-Tabelle geladen werden. Die Schalterstellung + Basisadresse soll hierbei gleich die Adresse des Bitmusters ergeben.
Ein Unterprogramm holt das Bitmuster aus der Tabelle ab, invertiert es, speichert es in die Tabelle zurück und gibt das Bitmuster an den LEDs aus.
Die Blinkzeit kann über weitere drei Schalter als Vielfaches von 100ms eingestellt werden.

a) Wieso ist dieses Umkopieren notwendig?
b) Erstelle das Flussdiagramm und die Bitmustertabelle!
c) Schreibe das Assemblerprogramm und speichere es als "**`B204_christmas_3.asm`**".

### Beispiel für eine Bitmustertabelle mit 16 Blinkmustern:

| Adresse | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | Wert |
|---|---|---|---|---|---|---|---|---|---|
| Label+0x00 | ★ | | ★ | | ★ | | ★ | | 0xAA |
| Label+0x01 | ★ | ★ | | | ★ | ★ | | | 0xCC |
| Label+0x02 | ★ | ★ | ★ | ★ | | | | | 0xF0 |
| Label+0x03 | ★ | | | ★ | ★ | | | ★ | 0x99 |
| Label+0x04 | ★ | | | | | | | ★ | 0x81 |
| Label+0x05 | ★ | ★ | | | | | ★ | ★ | 0xC3 |
| Label+0x06 | ★ | ★ | ★ | | | ★ | ★ | ★ | 0xE7 |
| Label+0x07 | ★ | ★ | ★ | ★ | ★ | ★ | ★ | ★ | 0xFF |
| Label+0x08 | | | | ★ | | | | ★ | 0x11 |
| Label+0x09 | | | ★ | | | | ★ | | 0x22 |
| Label+0x0A | | ★ | | | | ★ | | | 0x44 |
| Label+0x0B | ★ | | | | ★ | | | | 0x88 |
| Label+0x0C | | ★ | ★ | ★ | | ★ | ★ | ★ | 0x77 |
| Label+0x0D | ★ | ★ | ★ | | ★ | ★ | ★ | | 0xEE |
| Label+0x0E | | ★ | ★ | | | ★ | ★ | | 0x66 |
| Label+0x0F | ★ | | ★ | ★ | ★ | ★ | | ★ | 0xBD |

# Kleine Wiederholung zur Adressierung:

## Programmspeicher (FLASH, ROM)

Tabellen zum Abspeichern von **Konstanten** (nur lesbar!). **Nur indirekte Adressierung und nur über den Adresszeiger Z!** Die Initialisierung der Tabelle erfolgt **am Ende des Programms** im Codesegment (Assembleranweisung **.CSEG**). Die Tabelle besteht aus dem Label (Word-Adresse), der Assembleranweisung **.DB** (*define Byte*)[5] und den Konstanten in beliebiger Schreibweise. Bei der Initialisierung des Adresszeigers **Z** muss mit zwei multipliziert werden, da der Programmspeicher 16 Bit (Word) breit ist!

- Initialisierungen (Beispiele):

```
        ldi     ZL,LOW(FTAB*2)
        ldi     ZH,HIGH(FTAB*2)

am Ende des Programms:
FTAB:   .DB     3,144, 0x32, 0b11101110, "Hall", 'o', 0
```

- Befehl zum Lesen eines Byte: **lpm** (*load programm memory*). Bsp.:

```
        lpm     Tmp1,Z+
```

## Arbeitsspeicher (SRAM, RAM)

### Arbeitsregister r0 bis r31

Alle Berechnungen und Datenmanipulationen werden über die Arbeitsregister ausgeführt. Auf sie entfallen die meisten Befehle. Die meisten Befehle die den Buchstaben **i** (*immediate*) enthalten verwenden die unmittelbare Adressierung (direktes Laden von Konstanten) und können nur auf die Register **r16**-**r31** angewendet werden. **r24**-**r31** können als Doppelregister (16 Bit, **W**, **X**, **Y** und **Z**) verwendet werden.

- Einige Befehle die auf Arbeitsregister angewendet werden:

```
r0-r31:  mov,mul,add, cp, swap, inc, sbrs, …
r16-r31: ldi, cpi, adiw, ori, andi, ….
```

5 Es besteht auch die Möglichkeit **.DW** (*define Word*) zu benutzen.

## 64 SF-Register

Um die 64 SF-Register anzusprechen existieren die beiden Befehle **in** und **out**. Die untersten 32 SF-Register können zusätzlich über **sbi**, **cbi**, **sbis** und **sbic** angesprochen werden!

- **Alle** Befehle die auf SF-Register angewendet werden:

```
alle 64 SF-Register:        in, out
unterste 32 SF-Register:    sbi, cbi, sbis, sbic
```

## Datenbereich

### Direkte Adressierung

Die direkte Adressierung wird verwendet um Variablen oder sehr kurze Tabellen im SRAM zu schreiben oder zu lesen. Es wird eine **feste Adresse, meist in Form eines Labels** übergeben. Die Reservierung der Variablen bzw. der Tabelle erfolgt ganz am Anfang des Programms im Datensegment (Assembleranweisung **.DSEG)**. Dies geschieht mit einem Label (Word-Adresse), der Assembleranweisung **.BYTE** gefolgt von der Anzahl der zu reservierenden Bytes.

- Reservierung des Speicherplatzes im Datensegment (Beispiele):

```
.DSEG                       ;Anfang des Programms vor dem Codesegment
VAR1:   .BYTE 1
VAR32B: .BYTE 4
```

- **lds** (*load direct from sram)* ist der Befehl zum Lesen eines Byte
  **sts** (*store direct to sram)* ist der Befehl zum Schreiben eines Byte. Bsp.:

```
        lds     Tmp1,VAR1
        sts     VAR32B+2,Tmp1   ;beschreibt drittes Byte der Variablen
```

### Indirekte Adressierung

Zum Bearbeiten größerer Tabellen wird die indirekte Adressierung verwendet. Es können die Adresszeiger **X**, **Y** oder **Z** verwendet werden. Die Reservierung erfolgt wie bei der direkten Adressierung im Datensegment.

- Reservierung und Initialisierungen (Beispiele):

```
.DSEG                       ;Anfang des Programms vor dem Codesegment
STAB:   .BYTE 128

        ldi     XL,LOW(STAB)
        ldi     XH,HIGH(STAB)
```

- **ld** (*load indirect*) ist der Befehl zum Lesen eines Byte
  **st** (*store indirect*) ist der Befehl zum Schreiben eines Byte. Bsp.:

```
        ld      Tmp1,X
        st      Y+,Tmp1    ;+ für Autoinkrement des Adresszeigers
```

## Stapelbereich

Der Stapelbereich funktioniert ebenfalls mit indirekter Adressierung. Der Adresszeiger heißt Stapelzeiger **SP** und besteht aus den beiden SF-Registern **SPL** und **SPH**. Der Stapelzeiger wird von der Hardware automatisch verwaltet. Er muss nur am Anfang des Programms initialisiert werden. Die Befehle für Unterprogramme und Interrupt-Routinen (**rcall**, **call**, **ret**, **reti**, ...) verwenden automatisch den Stapel.

- Initialisierung:

```
        ldi     Tmp1,LOW(RAMEND)
        out     SPL,Tmp1
        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1
```

- **push** ist der Befehl zum Schreiben eines Byte.
  **pop** ist der Befehl zum Lesen eines Byte. Bsp.:

```
        push    r16
        pop     r24
```

# B3 Interrupts (Unterbrechungen)

## *Einführung*

Eine ankommende SMS unterbricht kurz die eben begonnene Arbeit. Ein kurzer Blick genügt, um die SMS in ihrer Priorität einzuordnen und zum Beispiel die gelieferte Information zu behalten.

In den Mikrocontrollern sind neben dem Prozessor viele periphere Bausteine integriert die gleichzeitig (parallel) zum Prozessor arbeiten können. In einem solchen Ein-Chip-Computersystem[6] müssen diese peripheren Bausteine auch die Möglichkeit besitzen ein laufendes Programm zu unterbrechen. Diese Unterbrechung (*Interrupt*) muss kurzfristig erfolgen können und dauert meist nur kurz.

Das Ereignis (*service*), das die Unterbrechung auslöst, wird Unterbrechungsanforderung (*Interrupt Request, IRQ*) genannt und es bewirkt, dass eine Unterbrechungsroutine (*Interrupt Service Routine, ISR*) aufgerufen wird. Bei der Unterbrechungsroutine handelt es sich um ein spezielles Unterprogramm, das es ermöglicht angemessen auf die Anforderung zu reagieren. Nach der Ausführung des ISR-Codes wird das Hauptprogramm an der Unterbrechungsstelle fortgesetzt.

Interrupts sind ein mächtiges Werkzeug, da sie erlauben praktisch in Echtzeit auf Ereignisse zu reagieren. Ohne Interrupts ist ein gleichzeitiges paralleles Arbeiten mehrerer Bausteine nicht möglich. Auch Multitasking-Betriebssysteme sind ohne Interrupts nicht möglich.

Interrupts sind allerdings auch gefährlich. Der Programmierer muss seine Denkweise beim Programmieren ändern. Hardware-Interrupts können Programme (auch die Unterprogramme!) zu jedem beliebigen Zeitpunkt unterbrechen und damit deren Variablen und Zeiger (Arbeitsregister, SF-Register und SRAM-Daten) verändern. Hardware-Interrupts erfolgen absolut asynchron gegenüber dem unterbrochenen Programm. Die Interrupt-Programmierung erfolgt eine große Sorgfalt.

Die Alternative zu den Interrupts ist das zyklische Abfragen (*Polling*), um den Status von zum Beispiel Schaltern oder Ein-/Ausgabebausteinen zu erfahren. Diese Methode ist zwar einfacher, aber wesentlich ineffizienter, da der Prozessor beim Polling für keine anderen Aufgaben mehr zur Verfügung steht.

## *Die Interrupts der AVR-Controller*

Die AVR-Controller benutzen **maskierbare Hardware-Interrupts**. Software-Interrupts[7] und so genannte *Exceptions* zur strukturierte Ausnahmebehandlung sind nicht vorhanden.

Die AVR-Controllern besitzen mit dem Hardware-**RESET** einen einzigen nicht-maskierbaren Interrupt (***N**on **M**askable **I**nterrupt **NMI***). Ein Reset kann im Programm nicht abgeschaltet werden. Er erfolgt immer zwingend. Beim Reset wird der Programmzeiger automatisch auf die Flash-

6 Manchmal wird hier der englische Begriff „*System on a Chip*“ oder SoC verwendet.

7 Externe Interrupts können bei Bedarf softwaremäßig durch Setzen des entsprechenden Portpins (das dann als Ausgang konfiguriert sein muss) ausgelöst werden. Dies stellt allerdings einen Spezialfall dar, der hier nicht behandelt wird.

Adresse **0x0000** gesetzt. An dieser Adresse befindet sich dann Sprungbefehl (**rjmp** oder **jmp**) zum eigentlichen Programm (siehe Assemblervorlage).

Die Hardware-Interrupts der AVR-Controller sind maskierbar, d.h. sie können abgeschaltet werden. Das Abschalten (bzw. Einschalten) erfolgt mit einzelnen Bits, sogenannten Flags, in unterschiedlichen SF-Registern.

Hardware-Interrupts können global geschaltet werden mit dem **I**-Flag in Statusregister **SREG**. Dies geschieht mit den Befehlen **cli** (***cl****ear* ***i****-flag*) und **sei** (***se****t* ***i****-flag*). Diese Flag dient als Hauptschalter, um also alle maskierbaren Interrupts zu verbieten oder zu erlauben.

Daneben kann aber auch jedes einzelne Interrupt mit dem ihm eigenen Bit maskiert werden. Zum Beispiel ist dies beim externen Interrupt **INT0** Bit 6 im SF-Register **GICR**. Mit einer ODER-Maskierung (**ori Tmp1,0b01000000**) kann das Interrupt freigegeben werden. Eine UND-Maskierung (**andi Tmp1,0b10111111**) sperrt das Interrupt[8].

Bsp. für das Freigeben des Interrupt:

```
        in      Tmp1,GICR               ;General Interrupt Control Register GICR einlesen
        ori     Tmp1,0b01000000         ;INT0 = 1
        out     GICR,Tmp1               ;Wert ins GICR Register zurueckschreiben
```

Welche maskierbaren Interrupts von welchem Baustein ausgelöst werden können ist aus der Interrupt-Vektortabelle ersichtlich. Jeder Controller hat seine eigene Vektortabelle! Damit der Programmcode portierbar bleibt sollen nicht die absoluten Adressen, sondern die von ATMEL® in den Definitionsdateien (Bsp. **m32def.inc**) vorgeschlagenen Namen für die Adressen verwendet werden.

# *Die Interrupt-Vektortabelle*

Tritt ein Hardware-Interrupt auf, so wird eine zu diesem Interrupt fest zugehörige Adresse in den Befehlszähler (PC) geladen und der Controller springt auf diese Flash-Adresse (Einsprungadresse). Diese nicht veränderbare Adresse wird als Interrupt-Vektor bezeichnet (Vektor steht hier für Adresszeiger) und die hardwaremäßig festgelegte Tabelle als Interrupt-Vektortabelle.

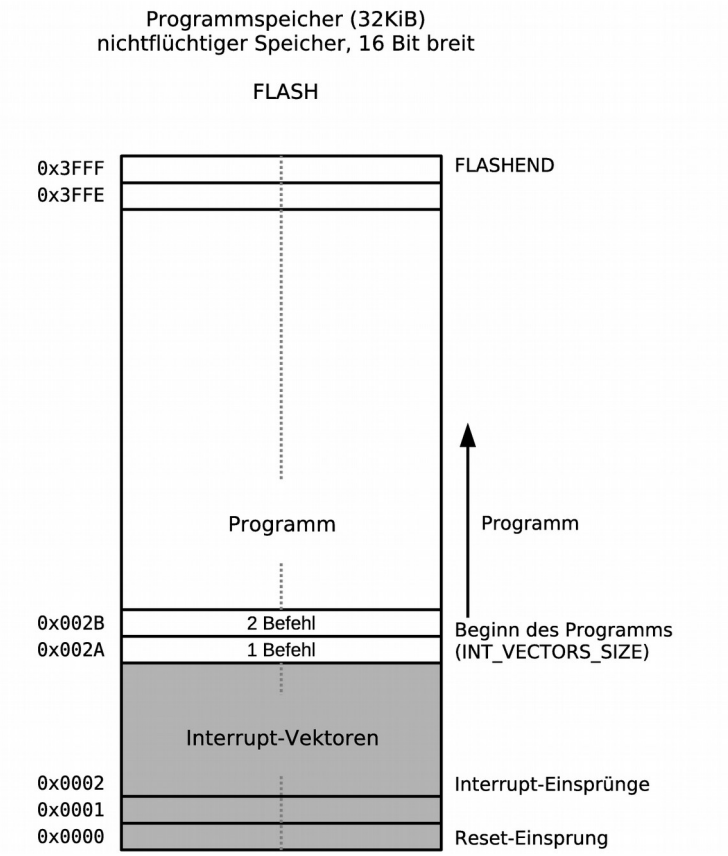
Der ATmega32 besitzt 21 solcher Adressen. Da ein nicht relativer Sprung-Befehl (**jmp**) zwei Flash-Speicherplätze benötigt, sind 42 Flash-Speicherplätze durch die Vektortabelle belegt. Das eigentliche Programm kann also erst an der Adresse **0x002A**[9] (= 42d) beginnen.

Da der Platz in der Interrupt-Vektortabelle natürlich nicht für den gesamten Code der Unterbrechungsroutine (ISR) reicht, wird hier mit einem Sprungbefehl auf die ISR verzweigt. Dieser Sprungbefehl ist veränderbar und muss am Anfang des Programms initialisiert werden, damit ein entsprechender Interrupt erfolgen kann[10].

8 Befindet sich ein Interrupt-Flag in einem der unteren 32 SF-Register (wie zum Beispiel das Flag für den AD-Wandler) so können natürlich statt einer Maskierung die Befehle **sbi** und **cbi** verwendet werden

9 Der in der Definitionsdatei vorgesehene Name für diese Adresse heißt: **INT_VECTORS_SIZE**

10 Da der veränderbare Sprungbefehl auf die ISR zeigt, kann man hier von einer zweiten softwaremäßigen Interrupt-Vektortabelle sprechen.




Da die Interrupt-Vektortabelle hardwaremäßig festgelegt wurde, ist die Reihenfolge der Interrupt-Vektoren nicht veränderbar. Die Reihenfolge bestimmt die Prioritäten der Interrupts. Umso niedriger die Adresse, desto höher die Priorität. Der externe Interrupt **INT0** hat also nach dem Reset die höchste Priorität. Treten zum Beispiel gleichzeitig ein externer Interrupt und ein Interrupt an der seriellen Schnittstelle auf, so wird zuerst der externe Interrupt ausgeführt.

**Bemerkungen:**

Bei anderen Controllern oder Prozessoren werden Interrupts oft anders behandelt als bei der AVR-Familie. Hier sind eventuell die Vektortabellen frei programmierbar und ein Interrupt-Controller übernimmt einen Teil der Interrupt-Steuerung.

Das eigentliche Programm kann erst an der Adresse **0x002A** beginnen. Der in der Definitionsdatei vorgesehene Name für diese Adresse heißt: **INT_VECTORS_SIZE**. Der Sprungbefehl an der Adresse **0x0000** verzweigt also normalerweise auf diese Adresse.

Die Vektortabelle beginnt mit der Adresse **0x0000**, (Einsprungadresse des **RESET**s). Wenn aber das **BOOTRST**-Fuse-Bit programmiert wurde, springt der Controller nach einem **RESET** automatisch in den Bootbereich (Bootloader-Programm) im oberen Adressbereich des Flash-Speichers. Mit Hilfe des **IVSEL**-Bit im SF-Register **GICR** kann dann die Interrupt-Vektortabelle in den Anfang des Bootbereichs verlegt werden.

Die Interrupt-Vektortabellen des ATmega8 und des ATmega16 unterscheiden sich von der Interrupt-Vektortabelle des ATmega32!

### Interrupt-Vektor-Tabelle des ATmega32:

| Vektor-nummer | Adresse im Flash | Name in "m32def.inc" | Quelle des Interrupts | verantwortlich für den Interrupt |
|---|---|---|---|---|
| 21 | 0x0028 | SPMRaddr | SPM_RDY | Interface für Programmspeicher |
| 20 | 0x0026 | TWIaddr | TWI | $I^2C$-Interface |
| 19 | 0x0024 | ACIaddr | ANA_COMP | Analog-Komparator |
| 18 | 0x0022 | ERDYaddr | EE_RDY | EEPROM-Interface |
| 17 | 0x0020 | ADCCaddr | ADC | A/D-Wandlung vollständig |
| 16 | 0x001E | UTXCaddr | USART, TXC | USART, Senderegister leer |
| 15 | 0x001C | UDREaddr | USART, UDRE | USART, UDR-Register leer |
| 14 | 0x001A | URXCaddr | USART, RXC | USART, Empfangsregister voll |
| 13 | 0x0018 | SPIaddr | SPI, STC | Serielle Übertragung beendet |
| 12 | 0x0016 | OVF0addr | TIMER0 OVF | Overflow von Timer 0 |
| 11 | 0x0014 | OC0addr | TIMER0 COMP | Compare Match von Timer 0 |
| 10 | 0x0012 | OVF1addr | TIMER1 OVF | Overflow von Timer 1 |
| 9 | 0x0010 | OC1Baddr | TIMER1 COMPB | Compare Match B von Timer 1 |
| 8 | 0x000E | OC1Aaddr | TIMER1 COMPA | Compare Match A von Timer 1 |
| 7 | 0x000C | ICP1addr | TIMER1 CAPT | Capture Event von Timer 2 |
| 6 | 0x000A | OVF2addr | TIMER2 OVF | Overflow Match von Timer 2 |
| 5 | 0x0008 | OC2addr | TIMER2 COMP | Compare Match von Timer 2 |
| 4 | 0x0006 | INT2addr | INT2 | externer Interrupt-Eingang 2 |
| 3 | 0x0004 | INT1addr | INT1 | externer Interrupt-Eingang 1 |
| 2 | 0x0002 | INT0addr | INT0 | externer Interrupt-Eingang 0 |
| 1 | 0x0000 | | RESET | externer Pin, Power-On-Reset, Brown-Out-Reset, Watchdog Reset, ... |

# *Interrupt-Behandlung*

Tritt ein Interrupt auf, so wird das zu diesem Interrupt zugehörige Flag (Bit in einem SF-Register) gesetzt. Sind Interrupts noch nicht zugelassen, so behält (speichert) dieses Flag das Auftreten des Interrupts, so dass dieses zu einem späteren Zeitpunkt ausgelöst werden kann.

Drei Bedingungen müssen erfüllt sein, damit der Controller Interrupts ausführt:

1. **Interrupts müssen global freischaltet sein.**
   (Hauptschalter, Befehle **`sei`** und **`cli`**).

2. **Interrupts müssen zusätzlich einzeln freigeschaltet werden.**
   (Setzen des spezifischen Interrupt-Bits (Flags) im entsprechenden SF-Register)

3. **Es muss ein Ereignis auftreten, das einen Interrupt auslöst** (und ein zum Interrupt gehörendes spezifisches Flag setzt).

Sind die drei Bedingungen erfüllt, so wird das Programm unterbrochen. Die Hardware (Interruptsteuerung) des AVR-Controllers übernimmt folgende Aufgaben:

- Der gerade abgearbeitete **Befehl wird beendet**.
- Die **Rücksprungadresse** (Adresse des folgenden Befehls) wird automatisch **auf den Stapel** gerettet[11].
- Weitere **Interrupts werden unterbunden** indem das globale Interrupt-Flag gelöscht wird (entspricht dem Befehl `cli`)
- Das **spezifische Interrupt-Flag** der Baugruppe **wird gelöscht**, damit der Interrupt nicht erneut auftritt.
- Der Befehlszähler (PC[12]) wird mit der dem Interrupt entsprechenden Adresse der Vektortabelle geladen und der sich an dieser Adresse befindende **Sprungbefehl zur Unterbrechungsroutine (ISR) wird ausgeführt**.

Nachdem die Unterbrechungsroutine abgearbeitet wurde, übernimmt die hardwaremäßige Interruptsteuerung wieder:

- Die **Rücksprungadresse** wird vom Stapel **in den Befehlszähler** geladen.
- **Interrupts werden wieder zugelassen** indem das globale Interrupt-Flag gesetzt wird (entspricht dem Befehl `sei`).
- Mindestens **ein Befehl des Hauptprogramms wird ausgeführt**[13].

11 Dies setzt voraus, dass der Stapel initialisiert wurde!!

12 *Program Counter* oder auch noch *Instruction Pointer*

13 Ein erneuter Interrupt kann erst auftreten, wenn mindestens ein Befehl des laufenden Programms ausgeführt wurde.

**sei**

**Setze globales Interrupt-Flag im SREG-Register I = 1 (*set global interrupt flag*).**

| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Der Befehl nach dem sei-Befehl wird noch ausgeführt bevor Interrupts zugelassen sind.
**Beeinflusste Flags: I** **Taktzyklen: 1**

**cli**

**Lösche globales Interrupt-Flag im SREG-Register I = 0 (*clear global interrupt flag*).**

| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Interrupts werden sofort unterbunden. Tritt ein Interrupt gleichzeitig mit den cli-Befehl auf, so wird dieses Interrupt nicht mehr ausgeführt.
**Beeinflusste Flags: I** **Taktzyklen: 1**

## *Die Behandlungsroutine (ISR)*

Die Interruptsteuerung kümmert sich nicht um das Retten wichtiger Registerinhalte, bzw. der Statusflags. Alle von der Behandlungsroutine veränderten Register (Arbeitsregister, SF-Register (z.B.: Statusflags im **SREG**-Register) und SRAM-Speicherzellen) müssen unbedingt gleich nach dem Eintritt in die Behandlungsroutine auf den Stapel gerettet werden und vor dem Verlassen der ISR wiederhergestellt werden.

**Alle Registerinhalte, die in der Behandlungsroutine (ISR) verändert werden, müssen am Anfang der ISR auf den Stapel gerettet werden. Dies gilt auch für das Statusregister SREG.**
**Am Ende der ISR werden die Registerinhalte wiederhergestellt.**

**Der Befehl zum Rücksprung aus einer ISR unterscheidet sich vom Rücksprungbefehl der Unterprogramme durch ein angehängtes „i“ (`reti`).**

Der Code für eine typische ISR, welche die Register **r16** und **r17** als interne Variablen benutzt, könnte folgendermaßen aussehen:

```
;-------------------------------------------------------------------------
;       ISR mit zwei internen Variablen
;-------------------------------------------------------------------------

ISR_I1: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten
        push    r17

        ;Code der ISR

        pop     r17             ;Achtung !! Umgekehrte Reihenfolge !!
        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung ins Hauptprogramm aus einer
                                ;Interrupt-Behandlungsroutine
```

**reti**

**Rücksprung aus einer ISR (*return from interrupt*).**

| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Die Rücksprungadresse wird vom Stapel genommen. Interrupts werden global wieder zugelassen (I = 1).
**Beeinflusste Flags: I** **Taktzyklen: 4**

## *Weitere Informationen zur Interruptbehandlung*

- Tritt ein Interrupt auf, so wird das spezifische Interruptflag gesetzt. Dieses Flag kann nicht manuell gesetzt werden. Das Löschen des Interrupt-Flags ist durch Schreiben einer Eins in das entsprechende Bit des Registers möglich!!
- Treten Interrupts auf, während sie global oder spezifisch gesperrt sind, so wird mittels des jeweiligen Flags behalten, dass das Interrupt auftrat. Sobald die Sperren aufgehoben sind, werden die Interrupts nach ihrer Priorität ausgeführt. Es besteht aber die Möglichkeit das Flag manuell zu löschen bevor die Sperren aufgehoben sind. So kann man zum Beispiel verhindern, dass ein Interrupt zweimal ausgeführt wird.
- Treten mehrere Interrupts gleichzeitig auf, so wird das Interrupt mit der höchsten Priorität als erstes ausgeführt. Die Interruptflags der anderen Interrupts werden gesetzt, und somit ihr Auftreten gespeichert. Nach der Abarbeitung des ersten Interrupts wird ein Befehl des Programms ausgeführt, bevor dann das Interrupt mit der zweit höchsten Priorität ausgeführt wird usw.
- Befindet sich der Controller in einer ISR, so sind Interrupts global gesperrt und auch das zu diesem Interrupt gehörende Flag wurde gelöscht. Tritt jetzt das gleiche Interrupt zu diesem Zeitpunkt noch einmal auf, so wird die gleiche ISR nach dem Verlassen und dem Abarbeiten eines Befehls noch einmal ausgeführt. Dies kann durch das Löschen des Flags (durch Schreiben einer Eins) kurz vor Verlassen der ISR unterbunden werden.

- Es besteht die Möglichkeit Interrupts mittels des **sei**-Befehls innerhalb einer ISR wieder zuzulassen. So können verschachtelte Interrupts erfolgen. Hierbei ist allerdings allergrößte Vorsicht geboten. Insbesondere dürfen die Stapeloperationen nicht unterbrochen werden.
- Tritt der **cli**-Befehl gemeinsam mit einem Interrupt auf, so überwiegt der **cli**-Befehl. Der auftretende Interrupt wird nicht mehr ausgeführt.
- Wird ein Interrupt zugelassen, und tritt auf, so muss der Sprungbefehl in der Vektortabelle für diesen Interrupt vorhanden sein und es muss eine ISR zu diesem Interrupt mit mindestens einem **reti**-Befehl existieren. Ist dies nicht der Fall, so durchläuft der Controller die Vektor-Tabelle weiter bis er auf den nächsten Sprungbefehl trifft und führt diesen aus. Die Wirkung wäre, dass die ISR des folgenden Interrupt fälschlicherweise ausgeführt wird.

# *Externe Interrupts*

Der ATmega32 verfügt über drei externe Interrupts, d. h. an drei speziellen Pins des Controllers können Signale ein Interrupt auslösen[14].

Es handelt sich hierbei um die Pins **PD2** (**INT0**), **PD3** (**INT1**)und **PB2** (**INT2**).

Diese Pins sollten, falls sie für externe Interrupts benutzt werden, auch als Eingang initialisiert werden[15]. Die Pull-Up-Widerstände können auch bei Bedarf zugeschaltet werden!

**INT2** hat die geringste Priorität und kann nur auf fallende oder steigende Flanken des Eingangssignals reagieren. Die beiden anderen Interrupts **INT0** und **INT1** können zusätzlich auf einen Low-Pegel (Zustand), und auf beliebige Pegeländerung (beide Flanken) reagieren.

## *Die Initialisierung*

Zur **Initialisierung und Statusabfrage** der externen Interrupts dienen vier SF-Register:

- **MCUCR = MCU Control Register**
- **MCUCSR = MCU Control and Status Register**
- **GICR = General Interrupt Control Register**
- **GIFR = General Interrupt Flag Register**

Damit ein externes Interrupt genutzt werden kann sind folgende Initialisierungsschritte notwendig:

1. Der richtige Sprungbefehl zur ISR muss in die Vektortabelle eingetragen werden.
2. Interrupts müssen global erlaubt werden (**sei**).
3. Im SF-Register **MCUCR** (für **INT0** und **INT1**) bzw. **MCUCSR** (für **INT2**) muss festgelegt werden auf welche Flanke bzw. auf welchen Pegel der Interrupt reagieren soll.

14 Reichen drei externe Interrupts nicht aus, so bietet sich als Alternative zum ATmega32 der pinkompatible ATmega324PA an mit 32 externen Interrupts.

15 Sie dürfen, falls eine Interrupt-Quelle angeschlossen ist, keinesfalls als Ausgang initialisiert werden, da sonst ein Kurzschluss entstehen kann. Sind sie ohne Interrupt-Quelle als Ausgang initialisiert, so kann durch Schalten des Ausgangs ein Interrupt ausgelöst werden, und somit ein Software-Interrupt simuliert werden.

4. Der spezifische Interrupt muss im SF-Register **GICR** freigeschaltet werden.

Zusätzlich dazu muss eine ISR geschrieben werden, in der alle benutzten Register und Flags gerettet und wiederhergestellt werden. Im SF-Register **GIFR** befinden sich die Interrupt-Flags, die bei Bedarf durch Schreiben einer 1 gelöscht werden können.

# *Die SF-Register für externe Interrupts*

## Das Interrupt-Kontrollregister GICR

**GICR**-Register: SF-Register-Adresse **0x3B** (SRAM-Adresse **0x005B**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)
Eine Maskierung beim Zugriff ist sinnvoll um **IVSEL** nicht zu verstellen. Es werden hier nur die drei oberen Bits benötigt um den jeweiligen Interrupt zu aktivieren.

### GICR = General Interrupt Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **GICR 0x3B** | **INT1** | **INT0** | **INT2** | **-** | **-** | **-** | **IVSEL** | **IVCE** |
| Startwert | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Read/Write | R/W | R/W | R/W | R | R | R | R/W | R/W |

**INTn** ***External INTerrupt Request n Enable***

**0** Das betreffende Interrupt ist nicht aktiviert.
**1** Das betreffende Interrupt ist aktiviert, wenn ebenfalls das **I**-Bit in **SREG** gesetzt ist (Interrupts global erlaubt, **sei**). Ein Interrupt wird auch erkannt, wenn das betreffende Pin als Ausgang initialisiert wurde.

## Das Interrupt-Flagregister GIFR

**GIFR**-Register: SF-Register-Adresse **0x3A** (SRAM-Adresse **0x005A**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

Die Flags zeigen an ob ein Interrupt aufgetreten ist. Sind Interrupts zum Zeitpunkt des Auftretens gesperrt, so speichert das Flag das Ereignis.

### GIFR = General Interrupt Flag Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **GIFR 0x3A** | **INTF1** | **INTF0** | **INTF2** | **-** | **-** | **-** | **-** | **-** |
| Startwert | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Read/Write | R/W | R/W | R/W | R | R | R | R | R |

**INTFn** ***External INTerrupt Flag n***

**0** Es trat kein Interrupt auf. Das Flag wird automatisch gelöscht wenn die Interruptroutine aufgerufen wird. Es ist beim **INT0** bzw. **INT1** dauernd gelöscht, wenn die Interrupts auf Pegel reagieren sollen.

**1** Es wurde ein Interrupt erkannt und dies wird im Flag gespeichert. Passiert dies innerhalb einer Interruptroutine (wo standardmäßig Interrupts global gesperrt sind), so wird das Interrupt nach dem Verlassen der Routine ausgeführt.
Mit dem **Schreiben einer Eins!** kann das Interrupt-Flag manuell gelöscht werden.

# Das MCU-Kontrollregister MCUCR

**MCUCR**-Register: SF-Register-Adresse **0x35** (SRAM-Adresse **0x0055**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)
Eine Maskierung beim Zugriff ist sinnvoll. Es werden hier nur die vier niederwertigsten Bit zur Flankensteuerung vom INT0 (Bit 0 und 1) und INT1 (Bit 2 und 3) benötigt.

## MCUCR = MCU Control Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **MCUCR 0x35** | **SE** | **SM2** | **SM1** | **SM0** | **ISC11** | **ISC10** | **ISC01** | **ISC00** |
| Startwert | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

**ISCn** ***Interrupt Sense Control n ISCn1, ISCn0***

Mit je zwei Bit kann bei **INT0** und bei **INT1** ausgewählt werden, ob der Interrupt auf einen Zustand (Pegel) oder auf eine Flanke reagieren soll (zustandsgesteuert bzw. flankengesteuert).

**00** **Der Low-Zustand löst einen Interrupt aus.**
**01** **Jede Zustandsänderung löst einen Interrupt aus.**
**10** **Eine fallende Flanke löst einen Interrupt aus.**
**11** **Eine steigende Flanke löst einen Interrupt aus.**

# Das MCU-Kontrollregister und Statusregister MCUCSR

**MCUCSR**-Register: SF-Register-Adresse **0x34** (SRAM-Adresse **0x0054**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

Eine Maskierung beim Zugriff ist sinnvoll, da hier nur Bit 6 (für INT2) benötigt wird.

## MCUCSR = MCU Control and Status Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **MCUCSR 0x34** | **JTD** | **ISC2** | **-** | **JTRF** | **WDRF** | **BORF** | **EXTRF** | **PORF** |
| Startwert | 0 | 0 | 0 | - | - | - | - | - |
| Read/Write | R/W | R/W | R | R/W | R/W | R/W | R/W | R/W |

**ISC2** ***Interrupt Sense Control 2***

**INT2** kann nur flankengesteuert betrieben werden.

**0 Eine fallende Flanke löst einen Interrupt aus.**
**1 Eine steigende Flanke löst einen Interrupt aus.**

## *Programmierbeispiel:*

Das folgende Programmierbeispiel soll aufzeigen, wie externe Interrupts eingesetzt werden können.

Zwei Interrupt-Signale an **INT0** (**PD2**) und **INT1** (**PD3**) steuern eine LED am Ausgang **PC0** des Controllers. Zur Erzeugung der Interrupt-Signale werden zwei entprellte Taster verwendet. Eine steigende Flanke an **INT0** soll die LED einschalten. Eine fallende Flanke schaltet die LED wieder aus.

Im Hauptprogramm soll eine zweite LED (**PC1**) im Sekundentakt blinken.

```
;***************************************************************************
;*
;*      Titel:  Beispiel zur Interrupt-Steuerung (B300_int_example.asm)
;*      Datum:  30/12/09                Version:        0.4 (03/01/13)
;*      Autor:  www.weigu.lu
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32        Quarzfrequenz:  16MHz
;*      Eingaenge:      entprellter Taster an PORTD2 (INT0)
;*                      entprellter Taster an PORTD3 (INT1)
;*      Ausgaenge:      LED an PORTC0
;*
;*      Informationen zur Funktionsweise:
;*
;*      Eine steigende Flanke an INT0 soll LED einschalten.
;*      Eine fallende Flanke an INT1 soll die LED ausschalten.
;*      Im Hauptprogramm blinkt eine LED
;*
;***************************************************************************

;---------------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;---------------------------------------------------------------------------
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET:  rjmp    INIT            ;springe nach INIT (ueberspringe ISR Vektoren)

;---------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;---------------------------------------------------------------------------
; Vektortabelle (im Flash-Speicher)
.ORG    INT0addr                ;interner Vektor für INT0 (alt.: .ORG 0x0002)
        rjmp    ISR_I0          ;Springe zur ISR von INT0
.ORG    INT1addr                ;interner Vektor für INT1 (alt.: .ORG 0x0004)
        rjmp    ISR_I1          ;Springe zur ISR von INT1

;---------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;---------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 1 wird zum Rechnen benoetigt
```

```
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister W und zur Parameteruebergabe
.DEF    W = r24

.EQU    LEDI = 0                ;LED ISR wird an Pin 0 angeschlossen
.EQU    LEDM = 1                ;LED MAIN an Pin 1 angeschlossen

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,LOW(RAMEND) ;RAMEND (SRAM) ist in der Definitions-
        out     SPL,Tmp1         ;datei festgelegt
        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1

        ;externe Interrupts initialisieren
        in      Tmp1,MCUCR              ;MCU Control Register einlesen
        andi    Tmp1,0b11111011         ;ISC11=1, ISC10=0 => fallende Flanke
        ori     Tmp1,0b00001011         ;ISC01=1, ISC00=1 => steigende Flanke
        out     MCUCR,Tmp1              ;Wert ins MCUCR zurückschreiben

        in      Tmp1,GICR               ;General Int. Control Reg. einlesen
        ori     Tmp1,0b11000000         ;INT1=1, INT0=1 => INT0+INT1 aktivieren
        out     GICR,Tmp1               ;GICR setzen

        ;Port C und Port D konfigurieren
        cbi     DDRD,2                  ;INT0 = Eingang
        cbi     DDRD,3                  ;INT1 = Eingang
        sbi     DDRC,LEDI               ;Richtungsbit fuer LEDI auf Ausgang
        sbi     DDRC,LEDM               ;Richtungsbit fuer LEDM auf Ausgang

        ;globales Interrupt-Flag setzen (Int. erlauben)
        sei

;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------
MAIN:   sbi     PORTC,LEDM              ;LED im Sekundentakt toggeln
        rcall   W500MS
        cbi     PORTC,LEDM
        rcall   W500MS
        rjmp    MAIN                    ;Endlosschleife

;-------------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-------------------------------------------------------------------------------
; Interrupt-Behandlungsroutine von INT0
ISR_I0: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten

        sbi     PORTC,LEDI      ;LED ein

        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung ins Hauptprogramm aus einer
                                ;Interrupt-Behandlungsroutine

; Interrupt-Behandlungsroutine von INT1
ISR_I1: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten

        cbi     PORTC,LEDI      ;LED aus

        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
```

```
        reti                    ;Ruecksprung ins Hauptprogramm aus einer
                                ;Interrupt-Behandlungsroutine

.INCLUDE "lib/SR_TIME_16M.asm"  ;Zeitschleifenbibliothek einbinden
;***************************************************************************
.EXIT                           ;Ende des Quelltextes
```

**Bemerkung:** Bei den Steuerregistern **MCUCR**, **GICR** und dem Portregister **PORTC** müssen nur einige Bits gesetzt oder gelöscht werden. Im Programm wird das durch eine Maskierung erreicht (siehe Modul A Kapitel 3). Die Befehle **sbi** und **cbi** können hier leider nicht eingesetzt werden, da nur die unteren 32 SF-Register mit diesen Befehlen erreicht werden können. Jeweils das ganze Register mit **ldi** zu setzen ist keine gute Idee, da die nicht benutzte, allerdings nicht unbedingt bedeutungslose Bits verstellt werden, und zu schwer auffindbaren Fehlern führen können.

# *Aufgaben*

⌂ **B300** Teste das obige Programm.

Nenne das Programm **"B300_int_example"**.

⌂ **B301** Ein Programm soll die positiven Flanken eines prellenden Schalters an **PD2** (**INT0**) zählen (Pull-Up-Widerstand zuschalten!) und an 8 LEDs ausgeben. Die Zählervariable (**COUNT1**) soll sich im **SRAM** befinden!

a) Zeichne das entsprechende Flussdiagramm.
b) Schreibe das kommentierte Assemblerprogramm.

Nenne das Programm **"B301_int_counter_1"**.

⌂ **B302** Entwickle einen binären 8-Bit-Vorwärtszähler, der um eins inkrementiert wird, wenn eine negative Flanke am Interrupt-Eingang **INT0** erscheint. Der Zählerwert soll über **PORTC** ausgegeben werden. Die Zählervariable (**COUNT1**) soll sich im **SRAM** befinden.

a) Entwerfe ein Flussdiagramm für das Programm.
b) Schreibe das zum Flussdiagramm passende Assemblerprogramm für den ATmega32 und kommentiere das Programm sinnvoll.

Nenne das Programm **"B302_int_counter_1"**.

⌂ **B303** Das vorige Programm soll erweitert werden. Eine positive Flanke an **PD3** (**INT1**) soll bewirken, dass der die Zählrichtung verändert wird (aus einem Vorwärtszähler wird ein Rückwärtszähler). Eine positive Flanke an **INT2** setzt den Zähler auf Null zurück. Treten **INT1** oder **INT2** auf, so soll deren Auftreten jeweils durch eine Flagvariable[16] im SRAM markiert werden (z.B.: **COUNTD** und **COUNTZ**). Die ISR von **INT0** muss dann diese Flags auswerten.

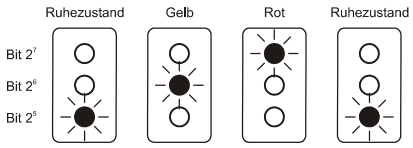
Nenne das Programm **"B303_int_counter_2"**.

16 Unter Flagvariable ist hier ein Byte zu verstehen, dass nur zwei verschiedene Zustände kennt. Dazu kann dann zum Beispiel Bit 0 verwendet werden. Die Variable wird also als Wert entweder 0x00 oder 0x01 enthalten.

⌂ **B304** In dieser Aufgabe soll eine einfache Interrupt-gesteuerte Ampelanlage für einen Fußgängerüberweg programmiert werden.

a) In einer ersten Phase soll zuerst nur die Ampel für die Autofahrer berücksichtigt werden. Für die Ampelanlage gelten folgende Punkte:

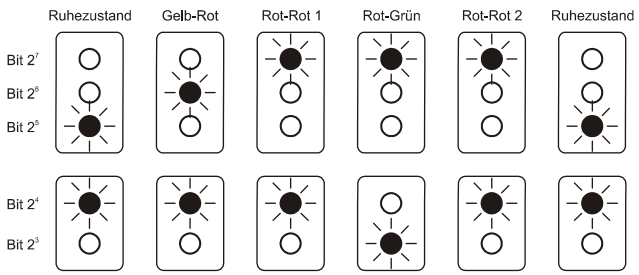
- Im Ruhezustand ist die Ampel Grün.
- Das Signal des Fußgängertasters löst mit seiner positiven Flanke am
- Eingang `INT0` einen Interrupt aus und ein Schaltzyklus beginnt (siehe Zeichnung).
- Der Einfachheit halber soll jeder Zyklus (bis auf den Ruhezustand) zwei Sekunden lang sein.
- Des Weiteren soll der Ruhezustand mindestens 6 Sekunden lang sein, bevor wieder geschaltet wird.
- Die Leuchten der Ampelanlage sollen mit PORTD verbunden werden!
- Benutze eine Tabelle für die einzelnen Phasen und greife mit Hilfe indirekter Adressierung auf diese zu!
- Ist ein Zyklus gestartet, wird erst wieder ein Interrupt angenommen, wenn Zustand *Rot* erreicht ist.




Erstelle ein Flussdiagramm und schreibe ein kommentiertes Assemblerprogramm für den ATmega32. Nenne das Programm **`"B304_int_traffic_light_1"`**.

**Tipp:** Damit die letzte Anforderung funktioniert muss das entsprechende Bit im Interrupt-Flag-Register **`GIFR`** nach der roten Phase gelöscht werden (durch Setzen einer Eins!).

b) Erweitere nun das Programm, damit auch die Fußgängerampel korrekt angesteuert wird.
Der Interrupt wird nun ab Zustand *Rot-Rot-2* wieder zugelassen.
Es braucht nur das Assemblerprogramm geschrieben zu werden.
Nenne das Programm **`"B304_int_traffic_light_2"`**.




⌂ **B305** Interessanterweise reicht es das **`MCUCR`**-Register (bzw. **`MCUCSR`**) zu initialisieren, damit die Interruptsflags bei zum Beispiel einer fallenden Flanke am entsprechenden Eingang gesetzt werden. So kann, **ohne die Benutzung von Interrupts**, ein

Ereignis über einen längeren Zeitraum gespeichert werden. Teste diese Eigenschaft, indem du ein Programm schreibst, das alle 10 Sekunden das Flag von **INT0** abfragt. **MCUCR** wurde für eine fallende Flanke initialisiert. Das Flag muss danach mit einer Eins gelöscht werden. An **PD2** wird ein Taster (Pull-Up aktivieren) angeschlossen.

Nenne das Programm **"B305_noint_flagtest"**.

**Bemerkungen:**

Wird der Pull-Up-Widerstand erst nach der Initialisierung des **MCUCR**-Register zugeschaltet, so kann nach einem **RESET** durch Störstrahlung bereits eine Interruptanforderung gespeichert werden, und somit ein Interrupt ausgelöst werden, der nicht auftreten soll.

Auch ist es wichtig nach dem Zuschalten des Pull-Up ungefähr 1µs zu warten (NOPs oder Zeitschleife) bevor das **MCUCR**-Register zugeschaltet wird, da es bei ungünstigem Layout bis zu 10 Taktzyklen (bei 16 MHz) dauern kann bis der Pull-Up wirklich aktiv ist.

# B4 Ansteuerung von Schrittmotoren

## *Einführung*

Wird ein hoch präziser Antrieb benötigt, so werden gerne Schrittmotoren eingesetzt.

Gegenüber anderen Motoren bieten Schrittmotoren folgende Vorteile:

- Sehr exakte Positionierung und konstante Drehzahl (keine Schrittfehler, falls sie nicht überlastet werden)
- Haltemoment in der Ruhelage
- Preiswert bei hoher Genauigkeit.
- Von einigen Milliwatt bis zu einem Kilowatt wirtschaftlich einsetzbar.
- Treiber können im Aufbau einfach gehalten werden. Dadurch ist die Ansteuerelektronik preiswert.
- Es ist eine direkte digitale Steuerung mittels Controller möglich.

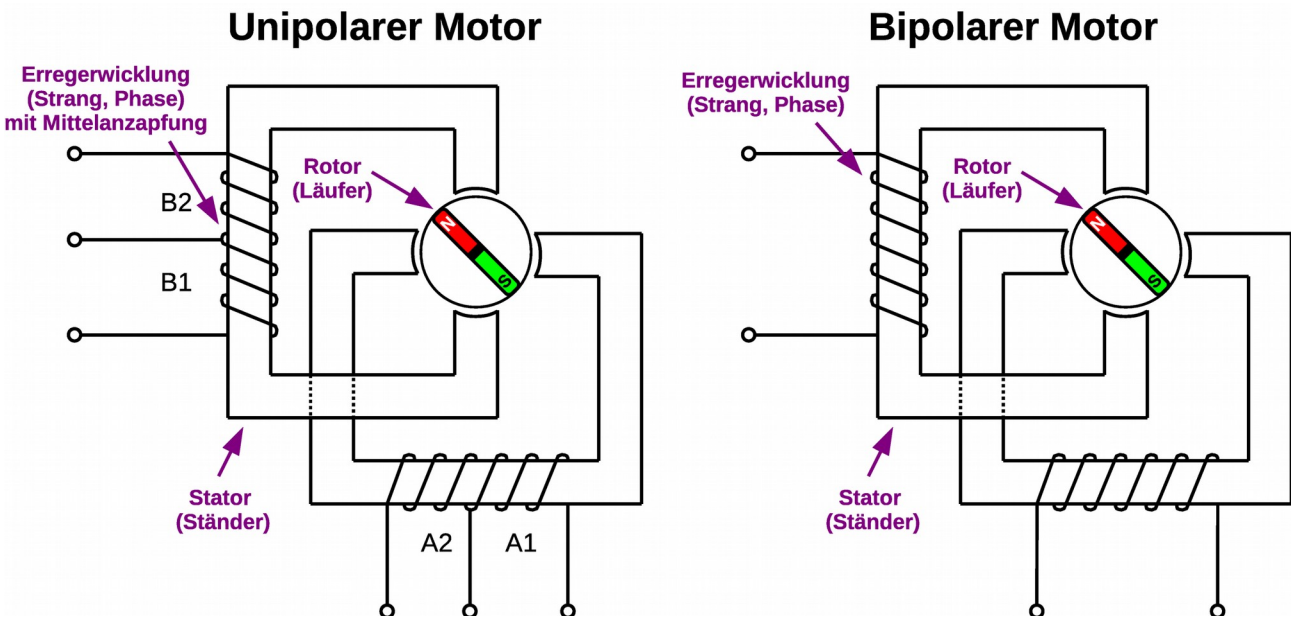
Man findet heute Schrittmotoren in fast allen Geräten, wo eine präzise Positionierung nötig ist. Dies sind zum Beispiel Diskettenlaufwerke, alten Festplatten, Faxgeräte, Scanner, Drucker, CD-Playern, computergesteuerten Werkzeugmaschinen (CNC)[17], Roboter und Kraftfahrzeuge. In heutigen Kraftfahrzeugen der mittleren und gehobenen Kategorie sind bis über 50 Schrittmotoren im Einsatz!

## *Aufbau des Motors*

Ein Schrittmotor besteht aus einem feststehenden Stator (Ständer) und einem darin drehenden Rotor (Läufer). Beim Schrittmotor befinden sich nur im Stator Spulen. Der Rotor besteht aus einem oder mehreren Permanentmagneten.

Da sich der Rotor immer so dreht, dass sich der größtmögliche magnetische Fluss ausbildet, entsteht ein Drehmoment, wenn die Magnetfelder im Stator und Rotor unterschiedlich ausgerichtet sind. Durch gezieltes Ein- und Ausschalten einzelner Wicklungen lässt sich so eine Drehbewegung erzeugen, wobei sich sehr einfach Drehsinn und Drehzahl des Motors steuern lassen.

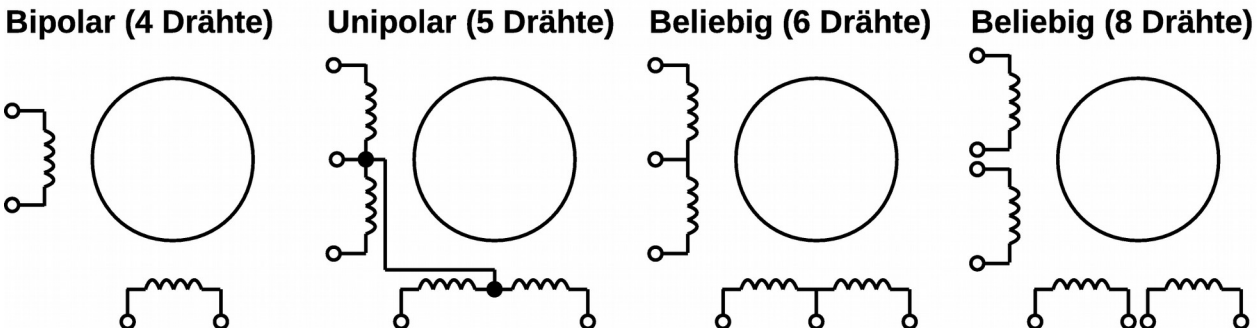
17 aus Wikipedia: **CNC** (*Computerized Numerical Control*) ist eine elektronische Methode zur Steuerung und Regelung von Werkzeugmaschinen (CNC-Maschinen), bzw. die dafür eingesetzten Geräte (Controller, Computer).



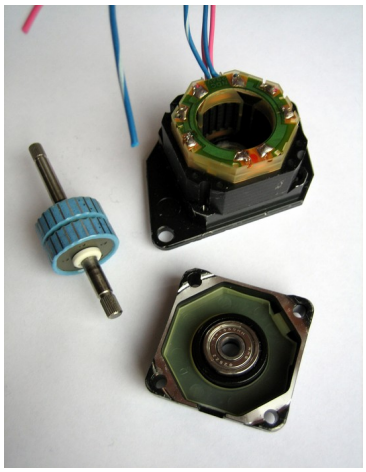

Es gibt zwei verschiedene Schrittmotormodelle.

Der **unipolare Schrittmotor** mit zwei Spulen mit Mittelanzapfung, der mit einer fest angelegten Spannung auskommt. Unipolare Motoren haben fünf oder sechs Anschlüsse. Mit einem Ohmmeter kann man schnell feststellen, wo sich die Mittenabgriffe befinden. Der Unipolarmotor mit 6 Anschlüssen kann auch bipolar angesteuert werden.

Der **bipolare Schrittmotor** mit zwei oder vier Spulen, bei dem die Spannung ständig umgepolt werden muss. Sind vier Spulen (2 Spulenpaare, acht Anschlüsse) vorhanden, so können je zwei Spulen parallel oder in Reihe geschaltet werden. Die Parallelschaltung erhöht das Drehmoment. Dadurch verdoppelt sich allerdings auch der zu liefernde Strom. Bipolare Motoren mit 8 Anschlüssen können natürlich auch unipolar angesteuert werden.



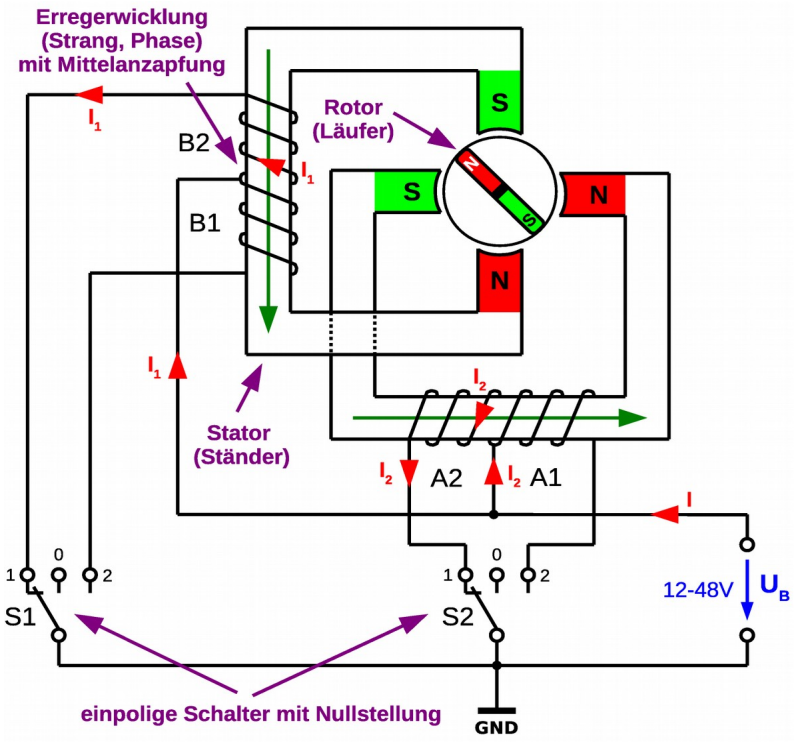

Heute werden vorwiegend Hybrid-Schrittmotor verwendet. Der Hybrid-Schrittmotor vereint die Vorzüge des Reluktanz-Schrittmotor mit dem des Permanentmagnet-Schrittmotors. Sein Rotor besteht aus einem axialen Permanentmagneten, an dessen Enden gezahnte Kappen befestigt sind. Diese sind um eine halbe Zahnbreite gegeneinander versetzt, so das sich Nord- und Südpole abwechseln. Der Vorteil des Hybrid-Schrittmotors ist das hohe Drehmoment und der genaue Schrittwinkel.



# Ansteuerungsarten und Betriebsarten

## Unipolarbetrieb (unipolare Ansteuerung)

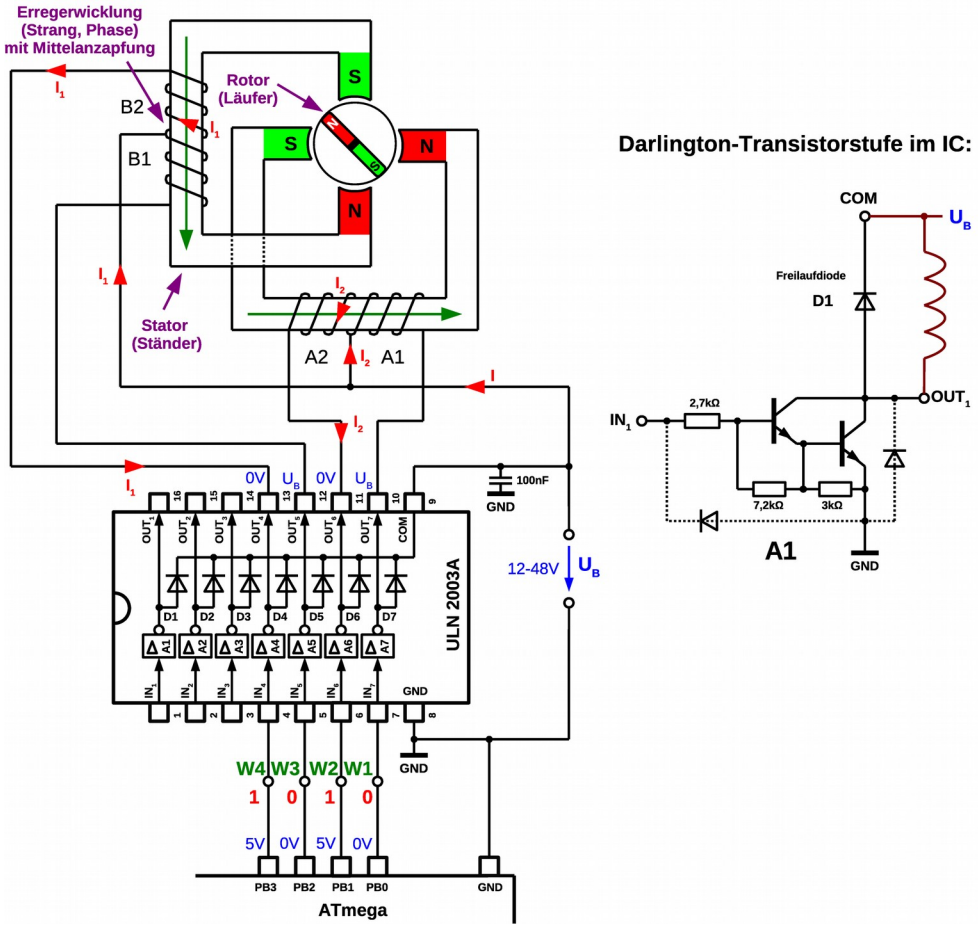
**Die Bezeichnung Unipolarbetrieb kommt, von den bei dieser Ansteuerung eingesetzten, unipolaren (einpoligen) Wechselschaltern.** Der Unipolarbetrieb ist nur möglich, wenn jede Erregerwicklung (Strang, Phase) eine Mittelanzapfung besitzt. Es wird immer nur eine der beiden Hälften der Erregerwicklung vom Strom durchflossen. Diese bestimmt dann die Polarität.



Schrittmotoren mit Mittelanzapfung werden als unipolare Schrittmotoren bezeichnet. Sie besitzen sechs bzw. mindestens fünf[18] Anschlussdrähte. Die Mittelanzapfung der Wicklungen wird fest mit der Betriebsspannung (meist zwischen 12 V und 48 V) verbunden. Die Wicklungsenden werden mittels Schalter mit Masse verbunden. Der Verlauf der Feldlinien (grün) bzw. die Pole lassen sich mit der Spulenregel[19] bestimmen.

Durch die Einfachheit der Ansteuerung wurde die unipolare Steuerung früher bevorzugt. Da aber jeweils nur eine Hälfte der Wicklung genutzt wird, ist das Drehmoment geringer als bei der bipolaren Ansteuerung.

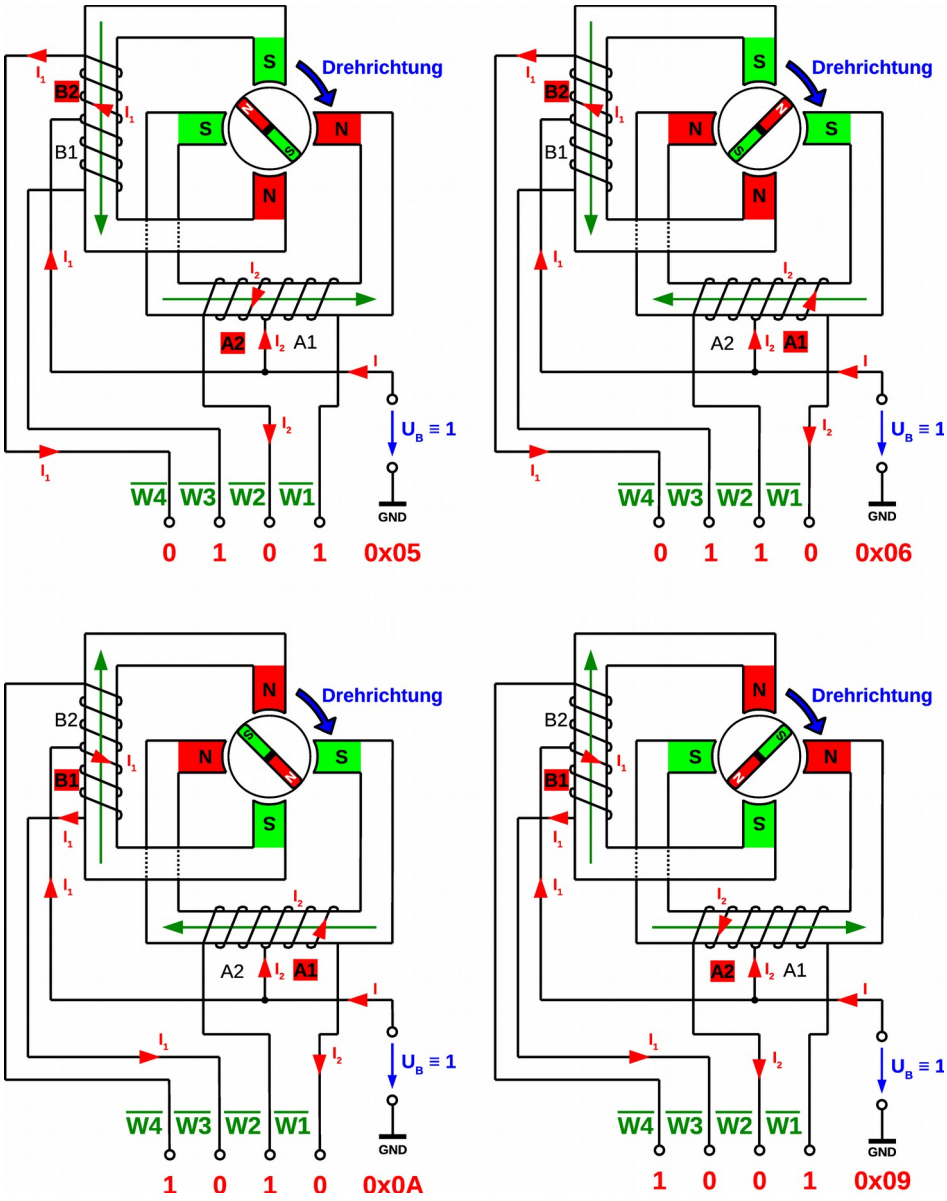
Bei der Ansteuerung mit dem Mikrocontroller werden elektronische Schalter (Transistoren) verwendet. Für Motoren bis 0,5 A kann die Ansteuerung sehr leicht mit dem IC ULN 2003A erfolgen. In ihm sind sieben invertierende Darlington-Transistorstufen (Emitterschaltung) inklusive der nötigen Freilaufdioden[20] integriert. Die Widerstände sind so gewählt, dass die Stufe mit 5 V angesteuert werden kann.




18 Wenn beide Mittelanzapfungen bereits im Motor miteinander verbunden sind!

19 Legt man die rechte Hand so um eine Spule, dass die Finger in Stromrichtung zeigen, so zeigt der abgespreizte Daumen zum Nordpol der Spule.

20 Die Freilaufdiode (*catch diode*) dient zum Schutz der Transistoren vor zu hohen induzierten Spannungen beim Abschalten einer induktiven Last.




Damit der Motor nach rechts dreht, muss der Controller die im folgenden Bild erarbeitete Sequenz senden. Um die Übersichtlichkeit zu steigern wurde die invertierende Stufe weggelassen. Da ja alle Anschlüsse invertiert werden, ändert sich nichts an der Drehrichtung, wenn wir die invertierten Werte senden würden. Die Sequenz ist nur verschoben. Um die Drehrichtung zu ändern, muss die Sequenz rückwärts durchlaufen werden.

| | | | | | $\overline{\text{Steuerbyte}}$ | Buchsenbezeichnung | | | | Steuerbyte |
|---|---|---|---|---|---|---|---|---|---|---|
| Schritt | $\overline{W4}$ | $\overline{W3}$ | $\overline{W2}$ | $\overline{W1}$ | | W4 | W3 | W2 | W1 | |
| 1 | 0 | 1 | 0 | 1 | 0x05 | 1 | 0 | 1 | 0 | 0x0A |
| 2 | 0 | 1 | 1 | 0 | 0x06 | 1 | 0 | 0 | 1 | 0x09 |
| 3 | 1 | 0 | 1 | 0 | 0x0A | 0 | 1 | 0 | 1 | 0x05 |
| 4 | 1 | 0 | 0 | 1 | 0x09 | 0 | 1 | 1 | 0 | 0x06 |

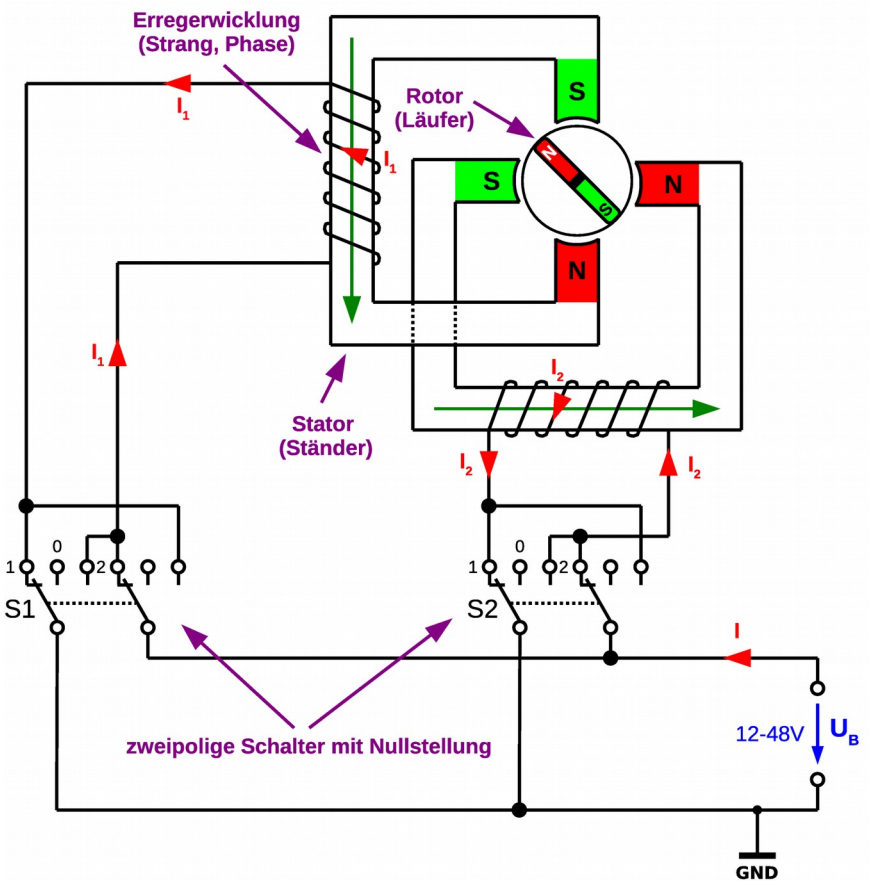
Der Schrittmotor dreht bei jeder Umpolung der Spulen um einen bestimmten Winkel weiter. Dieser Winkel wird auch als "Schritt" bezeichnet, was dem Motor seinen Namen verlieh.

Erfolgt die Umschaltung schnell genug, dann geht der Rotor in eine Drehbewegung über. Die Ansteuerung darf allerdings dabei nicht zu schnell erfolgen, so dass die maximale Drehgeschwindigkeit des Motors nicht überschritten wird. Tritt dieser Fall auf, so stottert der Motor.

## *Bipolarbetrieb (bipolare Ansteuerung)*

Werden beide Spulen vollständig umgepolt um die Stromrichtung in der Phase zu verändern, so spricht man vom Bipolarbetrieb. Die Spulen für diesen Betrieb brauchen keine Mittelanzapfung.

Für den Bipolarbetrieb sind zweipolige Umschalter erforderlich! Der Schaltungsaufwand ist größer. **Die Bezeichnung Bipolarbetrieb kommt von den bei dieser Ansteuerung eingesetzten bipolaren (zweipoligen) Wechselschaltern.**



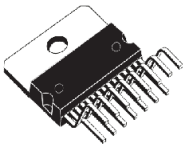

Mit dem Aufkommen preiswerter, integrierter Schaltungen begann sich die bipolare Ansteuertechnik mehr und mehr durchzusetzen. Für die Ansteuerung des bipolaren Motors sind je zwei Vollbrücken mit 4 Transistoren erforderlich.

Die einfachste und preiswerteste Ansteuerung eines Schrittmotors ist die Ansteuerung mit konstanter Spannung (L/R-Drive, siehe Unipolarbetrieb). Die Spannung wird so gewählt, das bei Motorstillstand gerade der erlaubte Nennstrom fließt (Gleichstrombetrieb, der Strom errechnet sich aus dem Spulenwiderstand). Damit ist allerdings das Drehmoment und die maximale Drehzahl begrenzt. Außerdem treten beim Stillstand hohe Verluste auf. Für Motoren bis 0,6 A kann diese Ansteuerung leicht mit dem IC L293D erfolgen. Die Leerlaufdioden sind schon mit integriert.

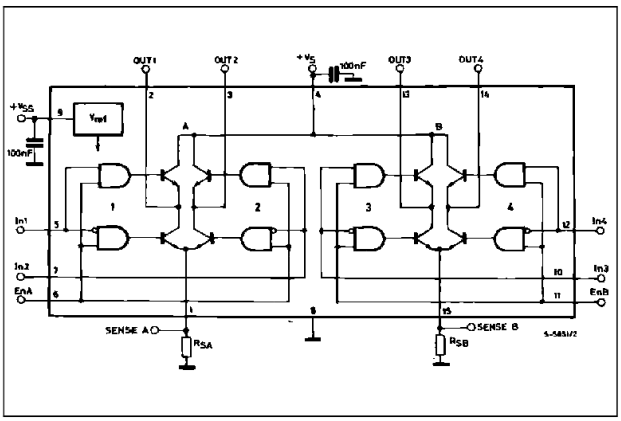
Durch die Induktivität der Ständerspulen wird eine Gegeninduktivität erzeugt, die umso höher ist, desto größer die Drehzahl (Umschaltgeschwindigkeit, Frequenz). Die Gegeninduktivität bewirkt, dass der Strom nicht mehr auf seinen Maximalwert ansteigen kann. Das maximale Drehmoment und die maximale Drehzahl werden nicht erreicht. Eine stromgeregelte Steuerung (sogenannter Chopperbetrieb) kann diesen Effekt vermeiden. Dazu wird eine viel höhere Spannung angelegt und mittels Shuntwiderständen (*sense*) der Strom gemessen. Je nach Stromstärke wird die Spannung dann zu oder abgeschaltet.

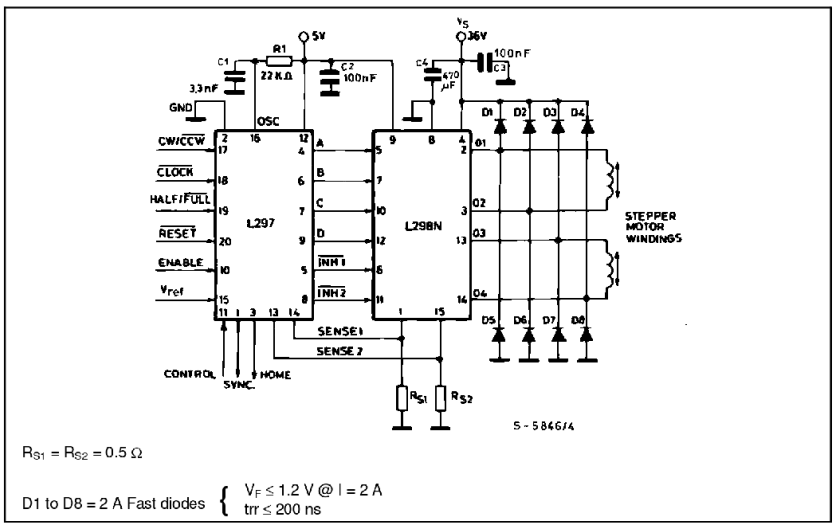
Heute werden alle Schrittmotoren, die etwas Leistung bringen müssen so angesteuert. Zwei integrierte Schaltkreise L298N und L297 erleichtern diese Ansteuerung ungemein.

**L298N**








Beim Bipolarbetrieb sind die Steuersequenzen die gleichen wie beim Unipolarbetrieb!

## Vorteile und Nachteile des Bipolarbetriebes:

Weil im Bipolarbetrieb die gesamte Wicklung eines Stranges ausgenutzt wird, ist das Drehmoment höher als im Unipolarbetrieb. Der Bipolarmotor ist preiswerter als der Unipolarmotor, jedoch erfordert der Antrieb eine aufwendigere Ansteuerelektronik, weil zwei Umschalter (Vollbrücke mit 4 Transistoren) je Strang benötigt werden.

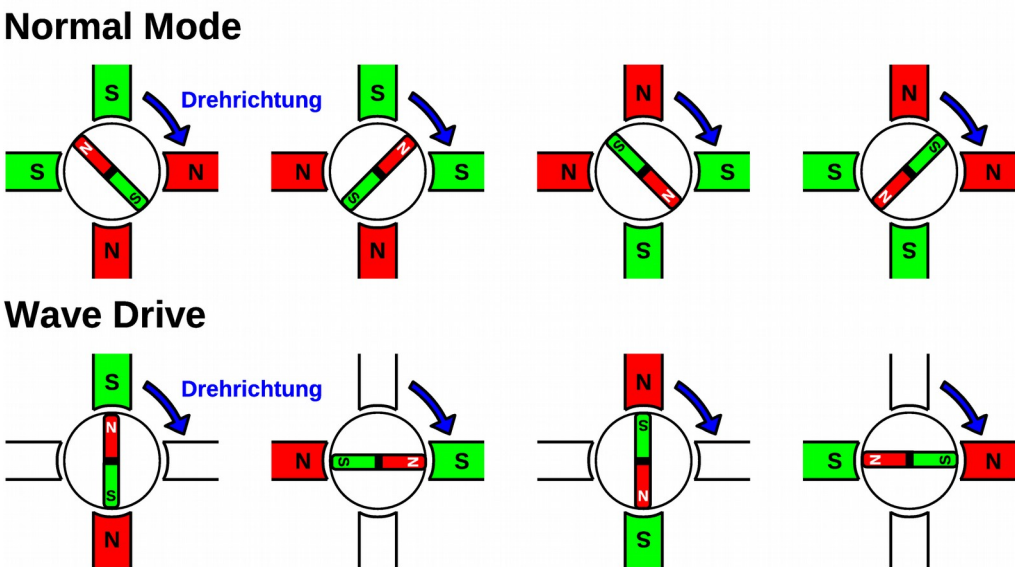
# Betriebsarten (Schrittarten)

Es gibt 4 sinnvolle Möglichkeiten das Drehfeld in einem Schrittmotor zu erzeugen:

- ***Wave Drive***: Ansteuern von jeweils einer Spule (*one phase on*)
- ***Normal Mode***: Ansteuern von zwei Spulen gleichzeitig (*full step drive*, *two phase on*)
- ***Half Stepping***: Halbschrittbetrieb, eine Kombination der beiden oberen Möglichkeiten.
- ***Microstepping***: Mikro-Schritte können erzeugt werden, wenn man den Strom in den zwei Wicklungen entsprechend einer Sinus- und Kosinus-Funktionen fließen lässt. Dieser Modus wird nur verwendet, wenn eine sehr kleine Schrittweite, sehr wenig Vibrationen oder sehr geringer Lärm erfordert sind.

## Vollschrittbetrieb.

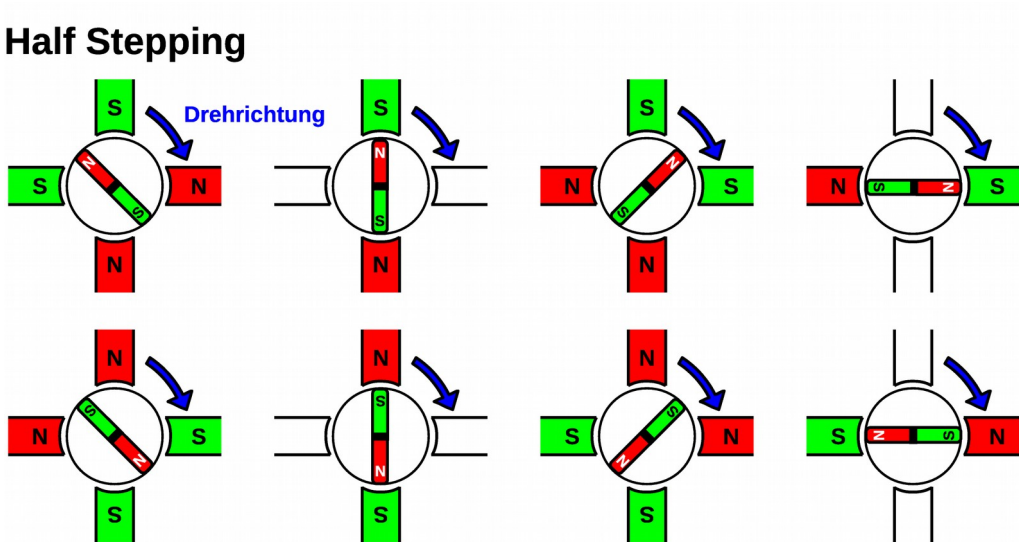
Bei einem Schrittmotor mit 90° Schrittwinkel sind 4 Vollschritte für eine Umdrehung erforderlich. Hierzu kann entweder nur eine Phase (*Wave Drive*) angesteuert werden, oder es können 2 Phasen gleichzeitig aktiv sein (*Normal Mode*).




Die Bezeichnung „Normal Mode“ deutet darauf hin, dass diese Methode die gebräuchlichste Art der Ansteuerung ist. Sie erlaubt ein höheres Drehmoment bei gleich großen Phasenströmen oder dasselbe Drehmoment bei kleineren Phasenströmen!

## Halbschrittbetrieb

Die Kombination von beiden zuvor genannten Möglichkeiten führt zum Halbschrittbetrieb. Die Motorphasen werden so geschaltet, dass der Rotor abwechselnd eine Vollschrittstellung und danach eine Zwischenstellung oder Halbschrittstellung annimmt. Damit hat man auf einfache Weise die Auflösung eines gegebenen Motors verdoppelt. Für eine volle Umdrehung sind also acht Schritte erforderlich.




Hält man die Phasenströme unabhängig von der momentanen Position konstant, wird in einer Stellung, in der beide Phasen bestromt sind, das Drehmoment höher sein. Ohne weitere Maßnahmen wird der Motor einen harten und danach einen weichen Schritt ausführen, also fühlbar unrund laufen.

**Die Steuersequenzen zur Ansteuerung von Bipolar- und Unipolarmotor unterscheiden sich nicht!**

# Wichtige Kenngrößen beim Schrittmotor

**Phasenzahl:** Schrittmotoren mit ein bis fünf Phasen (Spulen) sind gebräuchlich. Üblich ist der 2-Phasenmotor.

**Schrittwinkel:** Je Steuerimpuls dreht sich der Rotor um den Schrittwinkel. Sind die **Ständerphasenzahl m** und die **Polpaarzahl p** des Läufers bekannt, so lässt sich der Schrittwinkel errechnen mit:

$$\alpha = \frac{360^0}{2pm}$$

m = Strangzahl (Phasenz.) des Ständers
p = Polpaarzahl des Läufers

Mit zwei Spulen (m = 2, 2-Phasenmotoren) im Ständer und einer heute üblichen Polpaarzahl von p = 50 erreicht man Auflösungen im Vollschrittbetrieb von 1,8° und 0,9° im Halbschrittbetrieb (200 bzw. 400

Vollschritte pro Umdrehung). Mit geringem Schrittwinkel kann eine Position genauer anfahren werden. Der Motor läuft er ruhiger und leiser. Mit großem Schrittwinkel können höhere Drehzahlen erreicht werden. Es ist darauf zu Achten, ob die Herstellerangaben sich auf Vollschritt oder Halbschrittbetrieb beziehen!

**Nennstrom:** (Phasenstrom) Der maximal zulässige Strom im Dauerbetrieb pro Wicklung (Phase). Bei Motorstillstand soll eine Stromabsenkung vorgenommen werden, um den Motor nicht unnötig thermisch zu belasten.

**Nennspannung:** Spannung, welche im Stillstand angelegt werden muss, damit der Nennstrom fließt. Wird mit konstantem Strom angesteuert, so ist die angelegte Spannung wesentlich höher als die Nennspannung.

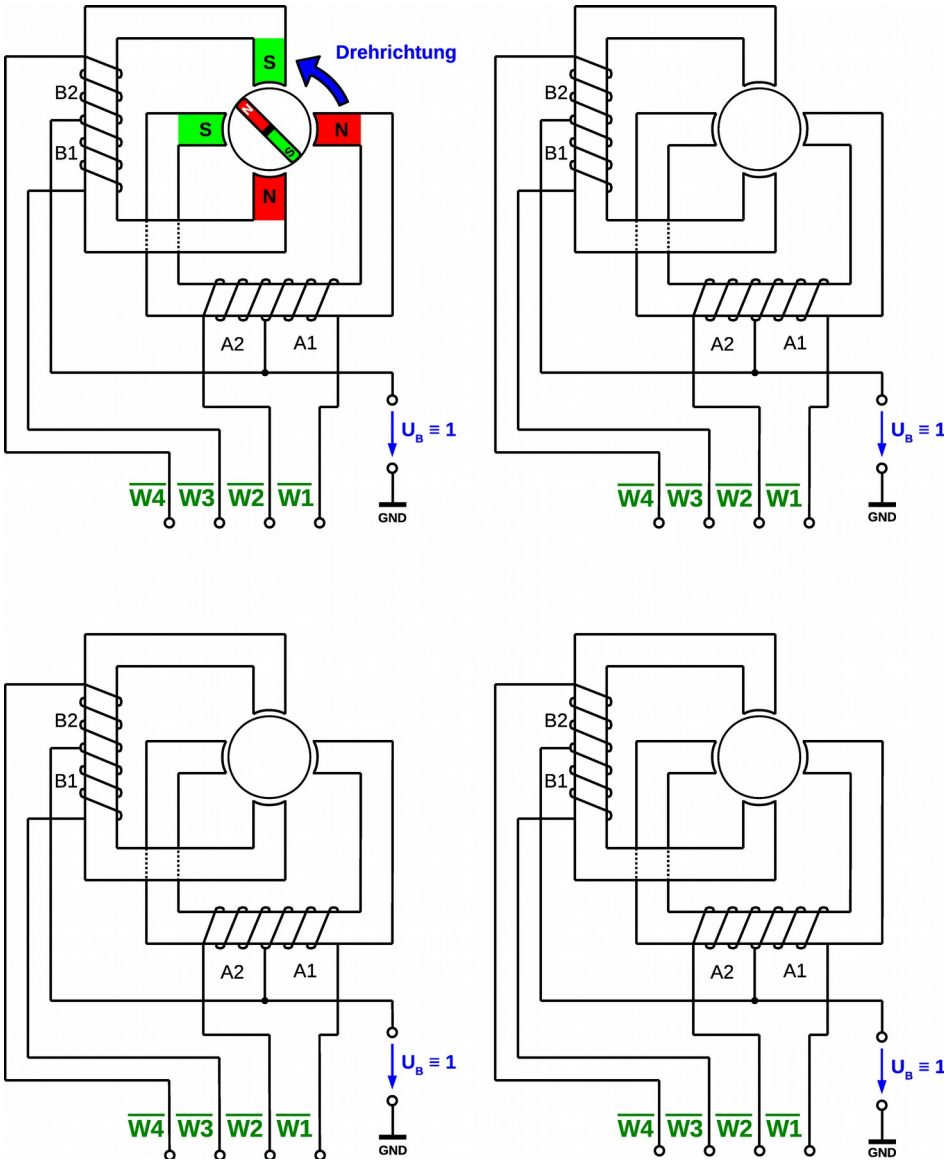
**Haltemoment:** Bis zu diesem Drehmoment kann ein Schrittmotor im Stillstand (volle Betriebsspannung) belastet werden, ohne dass er sich durch die Belastung dreht.

**Drehmoment:** Maximales Moment bei einer bestimmten Drehzahl. Meist aus einer Kennlinie ablesbar.

# *Aufgaben*

⌂ **B400** Der dargestellte Schrittmotor soll im Normal Mode (Linkslauf) angesteuert werden. Trage für jeden Schritt den Strom I sowie die Polarität des Magnetfeldes im Stator und im Rotor ein. Ergänze die beigefügte Tabelle.

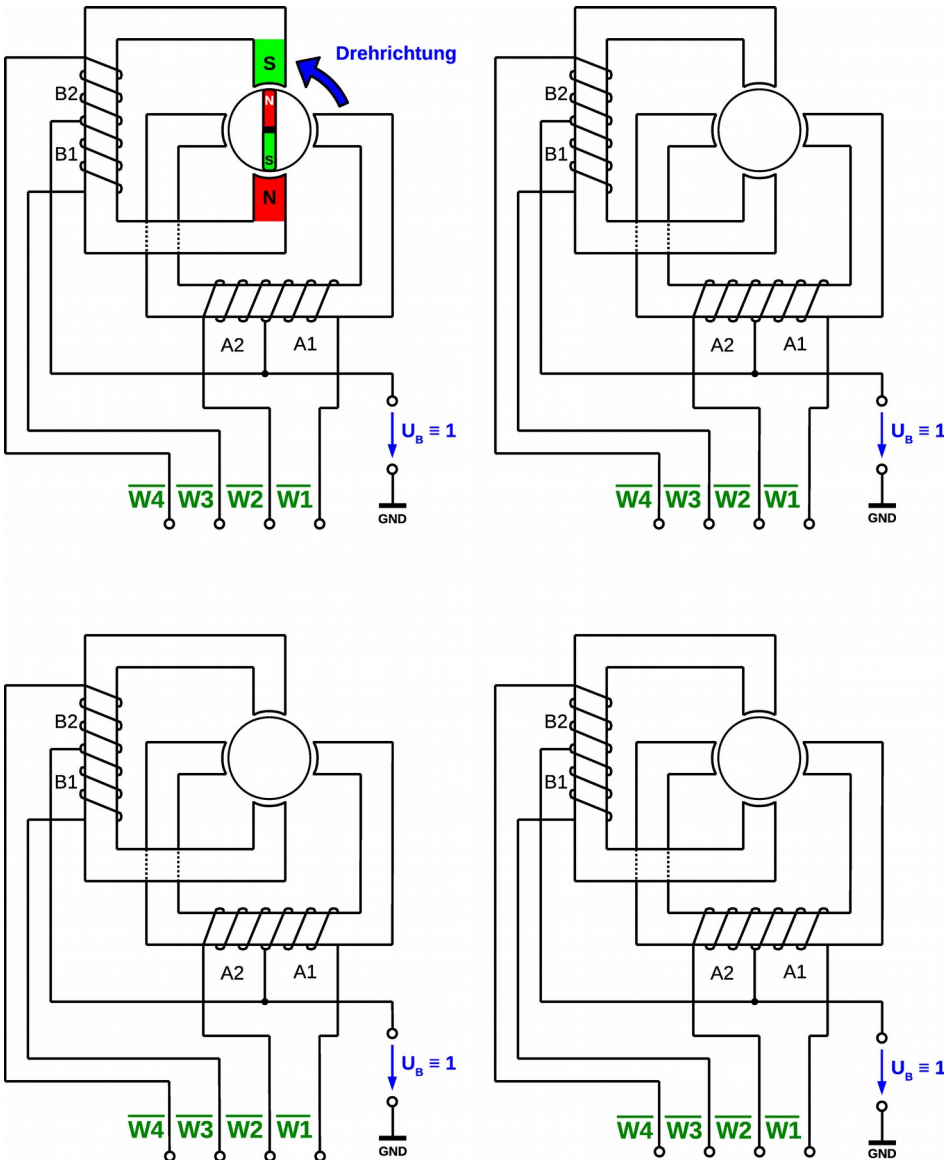
**Vollschrittbetrieb (Normal Mode)**




| | | | | | $\overline{\text{Steuerbyte}}$ | Buchsenbezeichnung | | | | Steuerbyte |
|---|---|---|---|---|---|---|---|---|---|---|
| Schritt | $\overline{W4}$ | $\overline{W3}$ | $\overline{W2}$ | $\overline{W1}$ | | W4 | W3 | W2 | W1 | |
| 1 | | | | | | | | | | |
| 2 | | | | | | | | | | |
| 3 | | | | | | | | | | |
| 4 | | | | | | | | | | |

⌂ **B401** Der dargestellte Schrittmotor soll im Wave Drive (Linkslauf) angesteuert werden. Trage für jeden Schritt den Strom I sowie die Polarität des Magnetfeldes im Stator und im Rotor ein. Ergänze die beigefügte Tabelle.

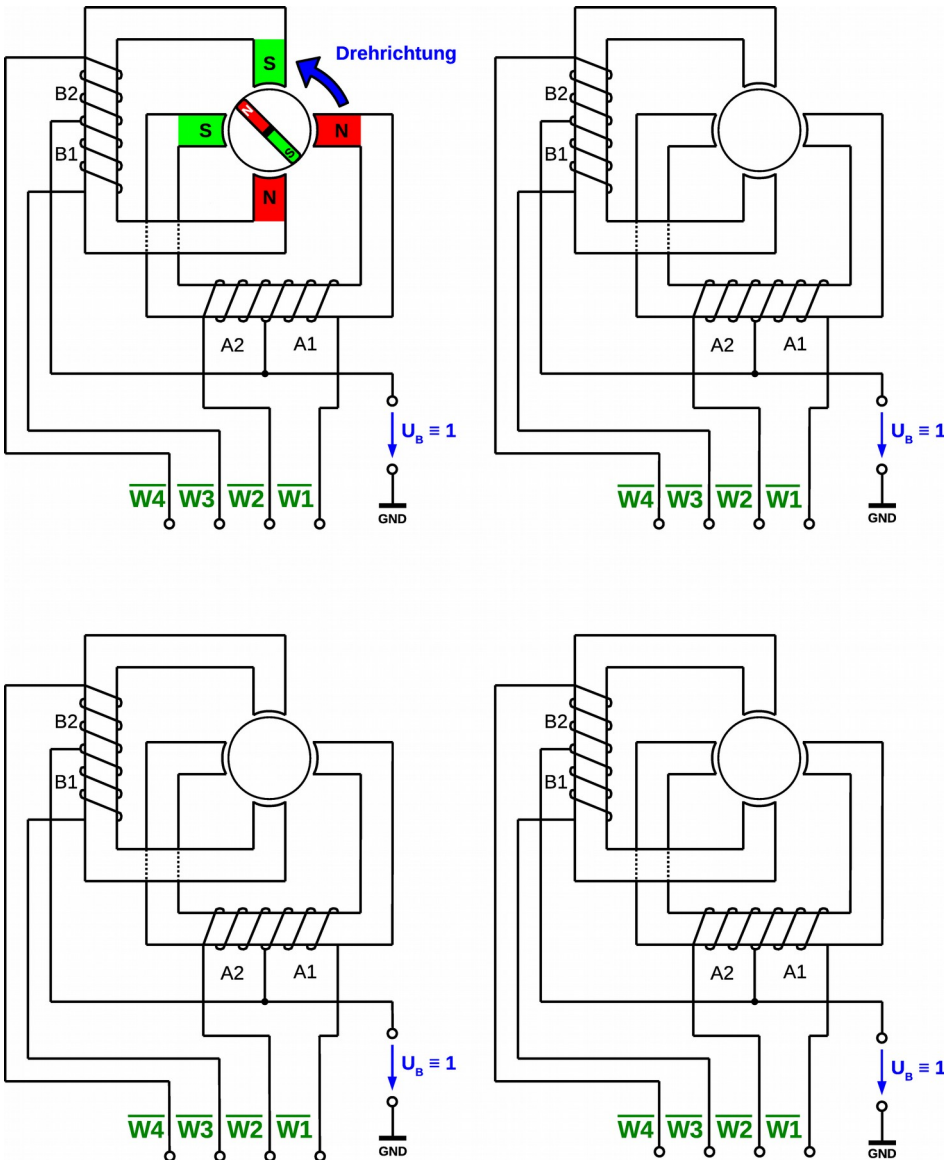
**Vollschrittbetrieb (Wave Drive)**



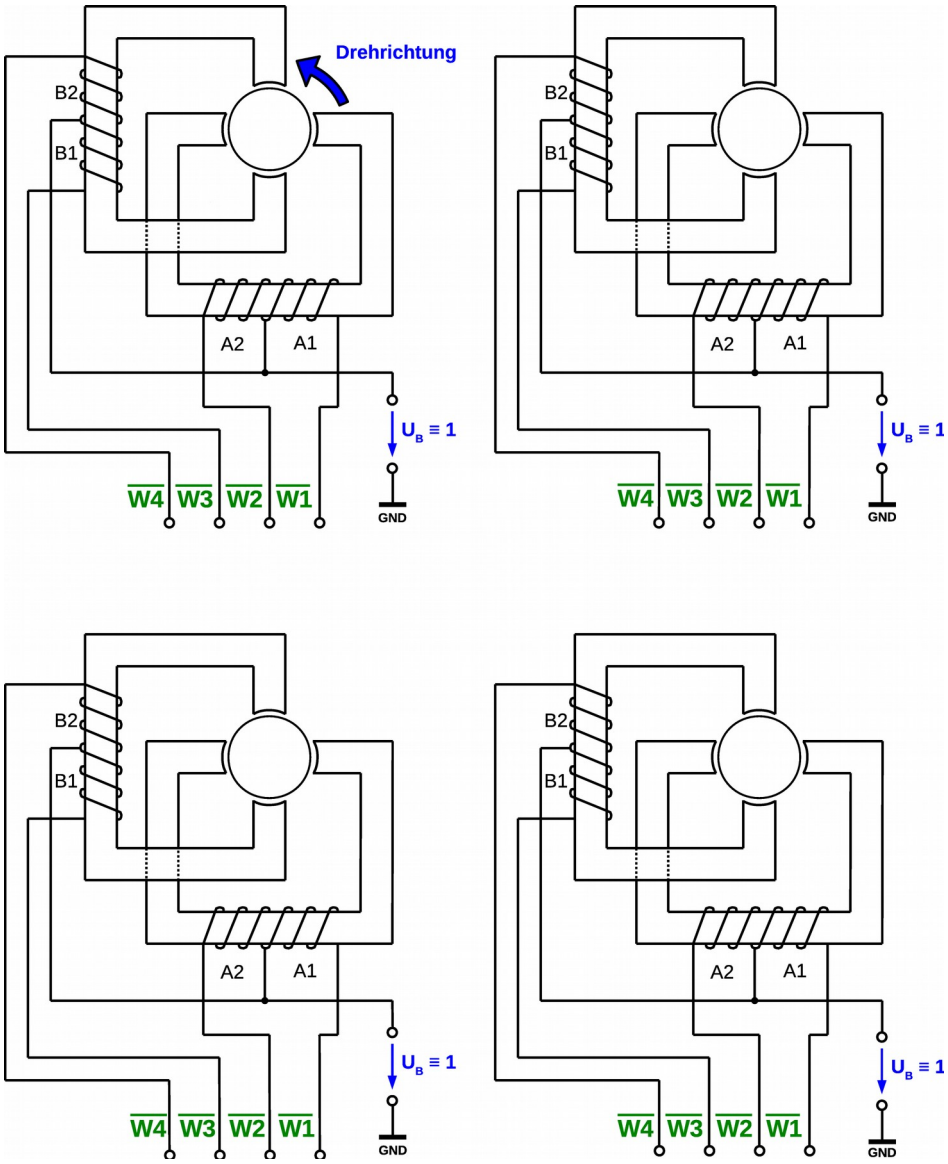

| | | | | | Steuerbyte | Buchsenbezeichnung | | | | Steuerbyte |
|---|---|---|---|---|---|---|---|---|---|---|
| Schritt | $\overline{W4}$ | $\overline{W3}$ | $\overline{W2}$ | $\overline{W1}$ | | W4 | W3 | W2 | W1 | |
| 1 | | | | | | | | | | |
| 2 | | | | | | | | | | |
| 3 | | | | | | | | | | |
| 4 | | | | | | | | | | |

⌂ **B402** Der dargestellte Schrittmotor soll im Halbschrittbetrieb (Linkslauf) angesteuert werden. Trage für jeden Schritt den Strom I sowie die Polarität des Magnetfeldes im Stator und im Rotor ein. Ergänze die beigefügte Tabelle.

**Halbschrittbetrieb (Half Stepping)**




| | | | | | Steuerbyte | Buchsenbezeichnung | | | | Steuerbyte |
|---|---|---|---|---|---|---|---|---|---|---|
| Schritt | $\overline{W4}$ | $\overline{W3}$ | $\overline{W2}$ | $\overline{W1}$ | | W4 | W3 | W2 | W1 | |
| 1 | | | | | | | | | | |
| 2 | | | | | | | | | | |
| 3 | | | | | | | | | | |
| 4 | | | | | | | | | | |




| | | | | | Steuerbyte | Buchsenbezeichnung | | | | Steuerbyte |
|---|---|---|---|---|---|---|---|---|---|---|
| Schritt | W4 | W3 | W2 | W1 | | W4 | W3 | W2 | W1 | |
| 5 | | | | | | | | | | |
| 6 | | | | | | | | | | |
| 7 | | | | | | | | | | |
| 8 | | | | | | | | | | |

⌂ **B403** Die erstellten Tabellen sollen jetzt getestet werden. Dazu werden drei Schalter verwendet. Schalter S0 dient als Ein/Ausschalter. **Im Aus-Zustand muss der Motor stromlos sein.** Mit den Schalter S1 und S2 kann zwischen den vier folgenden Tabellen ausgewählt werden:

**FTab1** Normal Mode Rechtslauf
**FTab2** Normal Mode Linkslauf
**FTab3** Wave Drive Linkslauf
**FTab4** Half Stepping Linkslauf

Die jeweilige Tabelle wird einmal ganz durchlaufen bevor die Schalter wieder überprüft werden. Zwischen jedem Schritt ist eine Wartezeit 20 ms vorzusehen.

a) Zeichne ein Flussdiagramm.
b) Schreibe das zum Flussdiagramm passende kommentierte Assembler-Programm und nenne es "**B403_stepper_motor_1.asm**".
c) Verkleinere schrittweise die Zeitschleife und ermittle so die maximale Drehzahl des Motors.

**Hinweis:**
Steht kein Schrittmotor zur Verfügung, oder soll die Lösung optisch kontrolliert werden, so erhöht man einfach die Zeitschleife (1 s) und gibt die Signale an den LEDs aus.

⌂ **B404** Die vorige Aufgabe soll nun mit nur einziger Tabelle (Half Stepping) realisiert werden. Dazu wird beim für Vollschrittbetrieb einfach immer nur der zweite Wert gewählt. Zum Ändern der Drehrichtung wird die Tabelle rückwärts abgefragt.
a) Zeichne ein Flussdiagramm.
b) Schreibe das zum Flussdiagramm passende kommentierte Assembler-Programm und nenne es "**B404_stepper_motor_2.asm**".

⌂ **B405** Verändere die vorige Aufgabe, so dass das Stoppen des Motors nach jedem beliebigen Schritt möglich ist.
a) Zeichne ein Flussdiagramm.
b) Schreibe das zum Flussdiagramm passende kommentierte Assembler-Programm und nenne es "**B405_stepper_motor_3.asm**".

⌂ **B406** Mit Hilfe von zwei Schaltern S0 (Bitstelle $2^0$) und S1 (Bitstelle $2^1$) muss das folgende Betriebsverhalten eines Schrittmotors (Vollschrittbetrieb, Rechtslauf) eingestellt werden können:

Der Schalter S0 ist der Einschalter . S0 = 0 schaltet den Motor vollständig aus. Der Motor muss dann stromlos sein .

Mit S1 kann man zwischen zwei verschiedenen Geschwindigkeiten umschalten.
S1 = 0 Normale Geschwindigkeit (Wartezeit zwischen den Schritten 200ms)
S1 = 1 Schnelle Geschwindigkeit (Wartezeit zwischen den Schritten 100ms)

**Das Stoppen des Motors darf nur bei der niedrigen Drehgeschwindigkeit möglich sein, d.h. zuerst muss S1 auf 0 gesetzt werden; ein erneutes Starten kann dann auch nur mit der niedrigen Drehgeschwindigkeit erfolgen.**

Änderungen der Betriebsbedingungen brauchen erst an einem Zyklusende erfasst und ausgewertet zu werden.

a) Erstelle ein detailliertes Flussdiagramm.
b) Schreibe den zum Flussdiagramm passenden Quellcode in AVR-Assembler und kommentiere ihn sinnvoll.
   Nenne das Programm "**B406_stepper_motor_4.asm**".