

# A4 Befehle und Adressierung

Verschiedene Befehle aus unterschiedlichen Befehlsgruppen wurden im vorigen Kapitel schon verwendet. In diesem Kapitel sollen alle Befehlsgruppen vorgestellt werden und auch die unterschiedlichen Adressierungsarten. Die Beeinflussung der Bedingungsbits (Flags, Zustandsbits) durch Befehle soll ebenfalls untersucht werden sowie die damit verbundenen bedingten Sprünge.

## Die Befehle der ATmega-Mikrocontroller

Beim ATmega-Controller stehen mehr Befehle zur Verfügung als bei einem klassischen 8-Bit-Prozessor. Erstens besitzt der Controller wesentlich mehr Arbeitsregister. Zweitens beschränken sich arithmetische Befehle nicht nur auf ein Akkumulator-Register. Es können alle 32 Arbeitsregister für Berechnungen herangezogen werden.

Insgesamt stehen beim ATmega32A 131 unterschiedliche Befehle (ATmega8A 130 Befehle) zur Verfügung.

Ein 150-seitiges Dokument zum Befehlssatz von ATMEL kann zum detaillierten Verständnis bei Bedarf an der folgenden Adresse bezogen werden:

[www.atmel.com/atmel/acrobat/doc0856.pdf](http://www.atmel.com/atmel/acrobat/doc0856.pdf)

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.

### Beispiele:

	Opcode	Operand
kein Operand:	<b>nop</b>	
1 Operand:	<b>inc</b>	<b>r16</b>
2 Operanden:	<b>ldi</b>	<b>r16, 0x08</b>

Bei zwei Operanden steht immer zuerst das Ziel und dann die Quelle!



Für die meisten Befehle wird nur ein Wort benötigt (16 Bit). Ausnahmen sind: "**lds Rd, k16**", "**sts k16, Rd**", "**call k**" und "**jmp k**" welche jeweils 2 Worte benötigen.

Im Befehlssatz werden für Operanden folgende Abkürzungen verwendet:

**Rd** Meist Ziel-Arbeitsregister (*destination*, bei einigen Befehlen auch Quelle)  
**r0-r31** (bei unmittelbaren (*immediate*) Befehlen nur **r16-r31**)

<b>Rd</b>	Niederwertiges Byte (LByte) eines 16 Bit Ziel-Arbeitsregister
<b>Rr</b>	Quell- oder Sende-Arbeitsregister ( <b>r0-r31</b> , <i>source</i> )
<b>K</b>	Daten-Konstante 8 Bit (0-255)
<b>k</b>	Adress-Konstante für Operationen mit dem " <i>program counter</i> PC". (Bsp.: Label für einen Sprung)
<b>b</b>	Bitkonstante 3 Bit (0-7), zum Auswählen eines Bits in einem Arbeits- oder SF-Registers
<b>P</b>	Adresse eines SF-Registers 6 Bit (0-63)
<b>s</b>	Bitkonstante 3 Bit (0-7), zum Auswählen eines Bits im Statusregister
<b>X, Y, Z</b>	Doppelregister (Pointer, Adresszeiger) zur direkten Adressierung
	<b>X</b> r27:r26; <b>Y</b> ⇒ r29:r28; <b>Z</b> r31:r30

Wie aus dem Befehlssatz ersichtlich unterscheidet man folgende Befehlsgruppen:

## Datentransferbefehle (Datentransportbefehle)

Transfer- oder Transportbefehle dienen dem Transport von Daten zwischen Arbeitsregistern, Arbeitsregistern und den SF-Registern bzw. dem SRAM. Auch werden sie benötigt um Konstanten in die Register, Registerpaare oder in den Speicher zu schreiben. **Datentransferbefehle beeinflussen die Zustandsbits (Flags) nicht.**

Beispiele für Transferbefehle: **mov Rd,Rr; in Rd,P; push Rr; sts k,Rr; LPM**

**Bemerkung:** Bei **mov**-Befehlen wird keine Verschiebung im eigentlichen Sinne durchgeführt, sondern der Inhalt wird kopiert! Der Inhalt des Quellregisters bleibt also erhalten!

## Arithmetische und logische Operationen (Befehle)

Arithmetik-Befehle sind Befehle zur Anwendung der Grundrechenarten. Logikbefehle umfassen die Booleschen Funktionen UND, ODER, NICHT sowie Exklusiv-ODER. Sie erlauben ein gezieltes Maskieren, Löschen oder Setzen von Bits in Datenworten.

Arithmetische und logische Operationen werden von der ALU ausgeführt und arbeiten nur mit den Arbeitsregistern!

**Arithmetische und logische Operationen beeinflussen die Zustandsbits.**

Beispiele für arithmetische Operationen: **add Rd,Rr; sbiw Rdl,K; inc Rd; mul Rd,Rr**

Beispiele für logische Operationen: **or Rd,Rr; adiw Rdl,K; com Rd; tst Rd**

## Bitorientierte Befehle

Bitorientierte Befehle dienen dazu Zustandsbits (Flags) zu beeinflussen, Programmunterbrechung (Interrupts) zu erlauben bzw. zu verbieten, einzelne Bits in den Arbeitsregistern bzw. SF-Registern zu setzen oder zu löschen oder um Register zu rotieren.



## Das Zustands- oder Statusregister SREG

Das Rechenwerk des Controllers addiert und subtrahiert nur Bitmuster. Es interpretiert die Resultate nicht. Um arithmetische Operationen richtig bewerten zu können benötigen wir ein Zustands- oder Statusregister (Flagregister). Seine Bits (0-5) kennzeichnen das Ergebnis einer Operation die in der **ALU** durchgeführt wurde. Diese Zustandsbits werden auch noch als **Flags** (Signal-Fahnen) bezeichnet.

Es sind also nur die arithmetischen Befehle (inkl. Vergleichsbefehle), logische Befehle, Rotationsbefehle und Befehle zur Bitbeeinflussung im Statusregister die das Statusregister beeinflussen. Zusätzlich sind im Zustandsregister noch ein Bit enthalten mit dem global Interrupts zugelassen oder gesperrt werden können (**I**-Flag) und ein Bit (**T**-Flag), das dazu dient dem Anwender das Zwischenspeichern eines Zustandes (Bits, Flags) zu vereinfachen.

Das **Statusregister** (Datenblatt S10) befindet sich bei den SF-Registern auf der SRAM-Adresse **0x005F** (SF-Register-Adresse **0x3F**) und wird mit der Abkürzung "**SREG**" angesprochen (Definitionsdatei).

### SREG = Status Register

Bit	7	6	5	4	3	2	1	0
<b>SREG</b> <b>0x3F</b>	<b>I</b> Interrupt	<b>T</b> Transfer	<b>H</b> Halfcarry	<b>S</b> Sign	<b>V</b> Overflow	<b>N</b> Negative	<b>Z</b> Zero	<b>C</b> Carry
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

- I** Das "**Global Interrupt Enable/Disable**"-Flag **I** wird vom Benutzer gesetzt (1) oder rückgesetzt (0) um global Unterbrechungen zu erlauben bzw. zu verbieten (Befehle: **sei**, **cli**).
- T** Das "**Transfer**"-Flag **T** erlaubt mit Hilfe der Befehle **bld** und **bst** ein einzelnes Bit aus einem Arbeitsregister abzuspeichern (retten) und wieder zu laden (wiederherstellen). Es kann mit den Befehlen **set** und **clt** auch einfach gelöscht oder gesetzt werden.
- H** Das "**Halfcarry**"-Flag **H** (Auxiliary-Carry) kennzeichnet einen Übertrag von Bit 3 auf Bit 4 und wird bei Berechnungen mit Nibbles (4 Bit) benötigt. Es kann mit den Befehlen **seh** und **clh** auch einfach gelöscht oder gesetzt werden.
- S** Das "**Sign**"-Flag **S** wird bei der Berechnung mit vorzeichenbehafteten Zahlen eingesetzt. Es entspricht der Exklusiv-Oder Verknüpfung des Negative- und des Overflow-Flags:  $S = N \text{ xor } V = (\bar{N} \wedge V) \vee (N \wedge \bar{V})$ . Es kann mit den Befehlen **ses** und **cls** auch einfach gelöscht oder gesetzt werden.
- V** Das "**Overflow**"-Flag **V** zeigt einen Überlauf bei der Zweierkomplement-Berechnung, also bei vorzeichenbehafteten Zahlen an. Es kann mit den Befehlen **sev** und **clv** auch einfach gelöscht oder gesetzt werden
- N** Das "**Negative**"-Flag **N** entspricht dem höchstwertigen Bit des Resultats.
  - 8 Bit:  $N = r7$
  - 16 Bit:  $N = r15$ .
 Es kann mit den Befehlen **sen** und **cln** auch einfach gelöscht oder gesetzt werden.
- Z** Das "**Zero**"-Flag **Z** wird auf Eins gesetzt wenn das Ergebnis einer Operation Null ist. Es kann mit

den Befehlen **sez** und **clz** auch einfach gelöscht oder gesetzt werden

**C** Das "Carry"-Flag **C** wird Eins, wenn ein Übertrag an der höchsten Stelle entsteht. Es kann mit den Befehlen **sec** und **clc** auch einfach gelöscht oder gesetzt werden.

Der Inhalt der Zustandsbit entscheidet über den weiteren Programmablauf bei Programmverzweigungen. Programmverzweigungen werden über bedingte Sprungbefehle (Verzweigung) verwirklicht.

**Beispiel:** Beim Befehl **breq** (*branch if equal*, springe wenn Ergebnis eines Vergleichs (Subtraktion) gleich Null) fragt die Ablaufsteuerung den Zustand des Zero-Flags ab. Bei **Z** = 1 wird das Programm zu der im Sprungbefehl enthaltenen Adresse (**k**) verzweigt, bei **Z** = 0 an der alten Adresse fortgesetzt. Solche bedingte Springbefehle werden meist gleich nach einer arithmetischen oder logischen Operation eingesetzt.

Entsprechend bedingte Sprungbefehle gibt es auch für die übrigen Flags. Wegen dieser Bedeutung der Zustandsbit muss der Programmierer genau wissen, welche Befehle Auswirkungen auf den Inhalt bestimmter Zustandsbit haben. Dies ist im Befehlssatz vermerkt.

Mit der Step-Into-Funktion des Studio 4 Programms im Debug-Modus lassen sich bequem einzelne Befehle eines Programms ausführen. Am Bildschirm kann anschließend die Wirkung der Befehle auf das Zustandsregister ausgewertet werden (siehe folgende Aufgaben).

**breq k**

**Bedingter relativer Sprung falls gleich (*branch if equal*).**(k = Adresskonstante)

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

Der Sprung erfolgt falls das Resultat einer vorigen Operation Null wurde (**Z**-Flag = 1). War diese Operation ein Vergleich (Subtraktion, siehe *compare*-Befehle) so waren beide Operanden gleich.

**Beeinflusste Flags: keine**

**Taktzyklen: 1 (kein Sprung), 2 (Sprung)**

## Adressierungsarten

**Wir unterscheiden:**

- 1.** Die **unmittelbare Adressierung** bei der die **Operanden Bestandteil des Befehls** sind.
- 2.** Die **direkte Adressierung** bei der die **unveränderbare Adresse im Befehl enthalten** ist.
- 3.** Die **indirekte Adressierung** wo die **Adresse in einem speziellen 16-Bit Adressregister** (Doppelregister **X**, **Y** oder **Z**) abgespeichert wird und somit dynamisch **veränderbar** ist. Das Doppelregister wird als Adresszeiger, Indexregister oder Pointer bezeichnet.

In der bei Controllern verwendeten Harvard-Struktur wird der Programm- und der Datenspeicher getrennt. Bei der Adressierung werden wir diese auch getrennt betrachten.

Alle Adressierungsarten lassen sich auf den SRAM-Datenspeicher anwenden. Der nichtflüchtige EEPROM-Datenspeicher kann nicht intern adressiert werden. Dazu stehen keine Befehle zur Verfügung. Seine Adressierung über die SF-Register wird später behandelt.

Da der Programmspeicher vorrangig nicht zur Verwaltung von Daten gedacht ist sind seine Adressierungsarten stark eingeschränkt.

## ***Adressierung des Datenbereichs:***

Die **direkte Adressierung** kann man unterteilen in die:

- **Direkte Registeradressierung**  
(Arbeiten mit wenigen Variablen (32 Arbeitsregister))
- **Direkte Adressierung der SF-Register**  
(Ein- und Ausgabe über die Peripherie)
- **Direkte Adressierung des SRAM-Datenspeichers**  
(Arbeiten mit vielen Variablen mit konstanten Adressen)

Bei der **indirekte Adressierung des SRAM-Speichers** unterscheidet man:

- **Indirekte Adressierung**  
(Viele Variablen mit variablen Adressen)
- **Indirekte Adressierung mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers**
- **Indirekte Adressierung mit festem (konstantem) Abstand**
- **Indirekte Adressierung mit "push" und "pop"**

## **Die unmittelbare Adressierung (*r15-r31*)**

Die unmittelbare Adressierung dient zum Arbeiten mit konstanten Werten. Die Konstante wird mit dem betroffenen Register unmittelbar hinter dem Opcode angegeben und befindet sich als Wert also unveränderbar im Flash.

Befehle für die unmittelbare Adressierung erkennt man am Buchstaben „**i**“ für das englische „**immediate**“.

Beispiele für eine unmittelbare Adressierung:

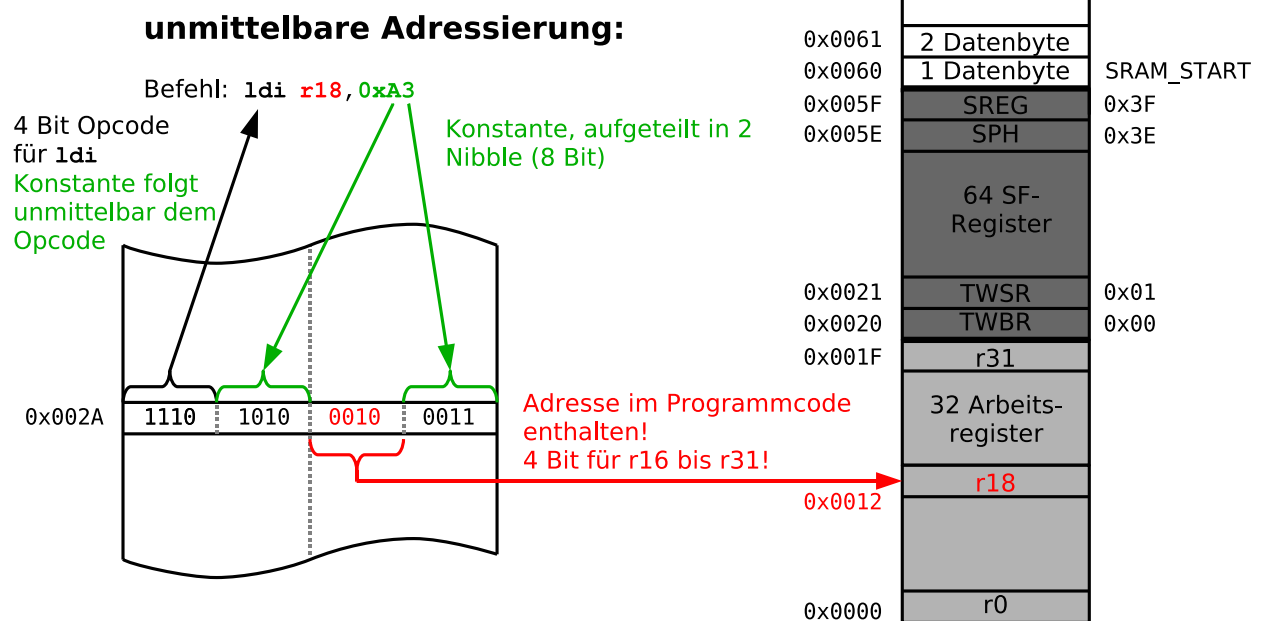
```
ldi r18,0xA3
andi Tmp1,0xFF
subi Tmp2,0x01
```

## Programmspeicher (Flash)

16 Bit breit

## Datenspeicher (SRAM)

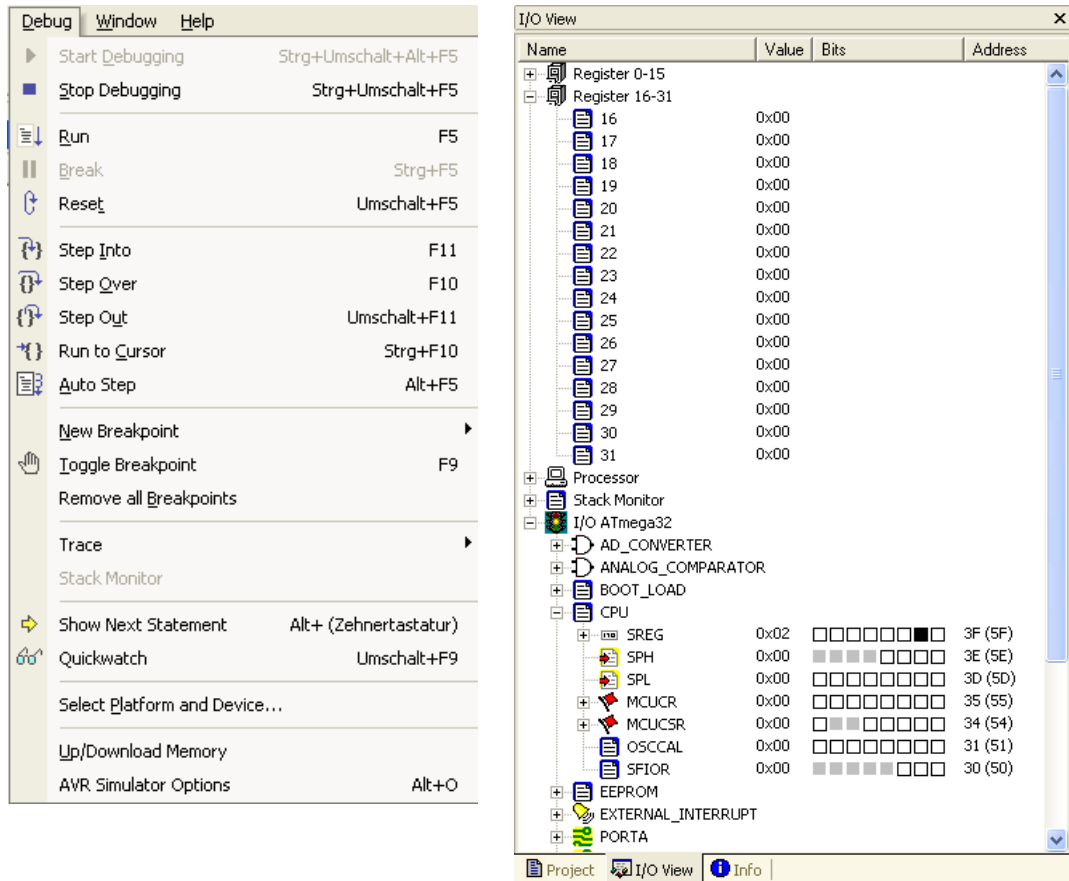
8 Bit breit



**A400** Zeichne ein Flussdiagramm und schreibe ein Programm das folgende logische Verknüpfungen durchführt:

Zuerst wird unmittelbar die Dezimalzahl **85** in die Variable (Register) **Tmp1** geladen. Dann wird eine logische UND-Verknüpfung mit der Zahl **170** durchführt (**andi**). Das Resultat wird dann noch mit der Zahl **128** ODER-Verknüpft (**ori**). Gib dem Programm den Namen "**A400\_immediate.asm**".

- Führe die Berechnung zuerst von Hand aus.
- Teste das Programm im Studio 4 ohne den Baustein zu programmieren (Nach dem "Assemble (Build F7)"-Befehl den Debugging-Modus einschalten mit "Start Debug" im Menu "Debug". Dann Einzelschritt mit "Step Into (F11)").
- In welchem Register steht jeweils das Resultat?
- Betrachte das Statusregister (Flagregister) **SREG** und die Resultate der Berechnungen im "I/O-View"-Fenster und interpretiere die Ergebnisse.



- A401** Schreibe ein Programm, das die Dezimalzahl **50** von **100** abzieht (Befehl: "**subi**"), dann zwei gleiche Zahlen (**100**) subtrahiert und schlussendlich die Zahl **200** von **100** subtrahiert. Danach soll das Programm nochmals die gleichen Berechnungen durchführen, allerdings mit dem Vergleichsbefehl "**cpi**" statt "**subi**".  
 Gib dem Programm den Namen "**A401\_subi\_cpi.asm**".
- Führe die Subtraktionen zuerst von Hand aus (Addition des Zweierkomplement!).
  - Teste das Programm im Studio 4 im Step-Modus
  - Betrachte das Statusregister (Flagregister) **SREG** und die Resultate der Berechnungen im "I/O-View"-Fenster und interpretiere die Ergebnisse.

**subi Rd, K**

**Subtrahiert Konstante K (8 Bit) von Register Rd  
(*subtract immediate*).**

0	1	0	1	K	K	K	K	d	d	d	d	K	K	K	K
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Unmittelbare Verknüpfung. Resultat in Rd. Nur Reg. r16-r31.  
**Beeinflusste Flags: H, S, V, N, Z, C**    Taktzyklen: 1

**cpi Rd, K**

Vergleicht Konstante K mit Inhalt von Register Rd (*compare with immediate*).

0	0	1	1	K	K	K	K	d	d	d	d	K	K	K	K
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Unmittelbare Verknüpfung. Nur Reg. r16-r31. Entspricht einer Subtraktion, allerdings wird der Inhalt von Rd nicht verändert! Wird meist vor bedingten Sprüngen eingesetzt.

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

**Bemerkung:** Wie schon oben erwähnt, kann **keine unmittelbare Adressierung** mit den Arbeitsregistern **r0 bis r15** durchgeführt werden.

## Die direkte Registeradressierung

Bei der Registeradressierung beziehen sich alle Operanden eines Befehls auf die Arbeitsregister. Es gibt Befehle die nur ein einzelnes Register benötigen, andere arbeiten mit zwei Registern. Manchmal wird die Registeradressierung auch als direkte Registeradressierung (siehe direkte Adressierung) bezeichnet, da die Adresse des Arbeitsregisters direkt im Operanden enthalten ist.

Befehle für die direkte Registeradressierung erkennt man daran, dass ausschließlich Arbeitsregister als Operanden fungieren (Ausnahme "**sbr**" und "**cbr**").

Beispiele für eine Registeradressierung:

```
inc    r23
com    Tmp1
mov    r5, r20
sub    r5, r20
cp     Tmp2, r22
```

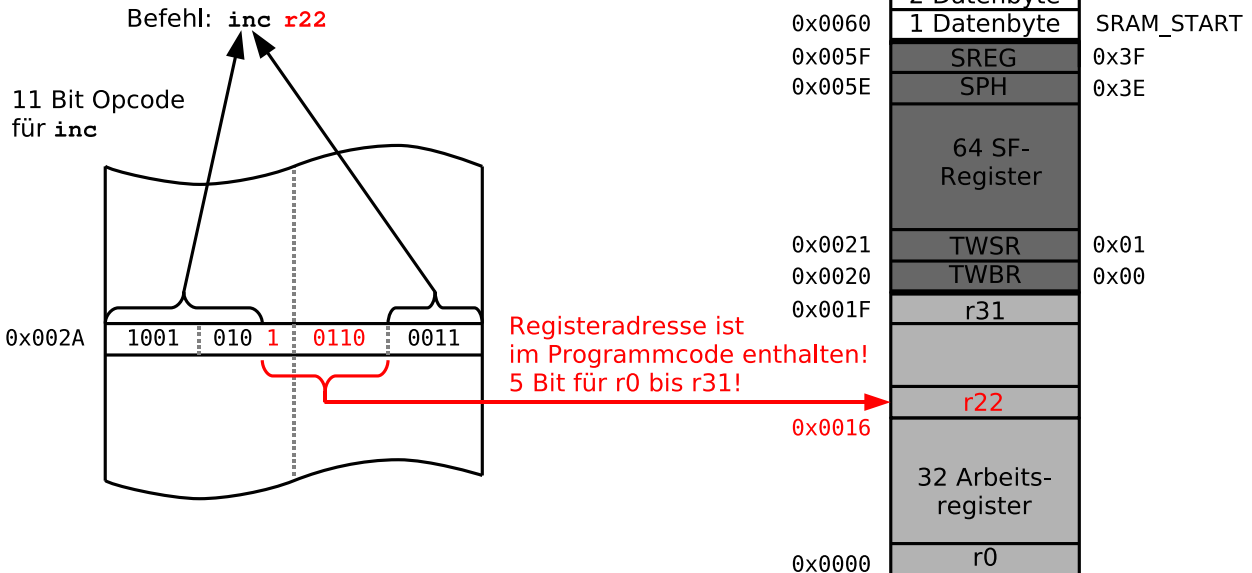
## Programmspeicher (Flash)

16 Bit breit

## Datenspeicher (SRAM)

8 Bit breit

### Registeradressierung 1 Register:



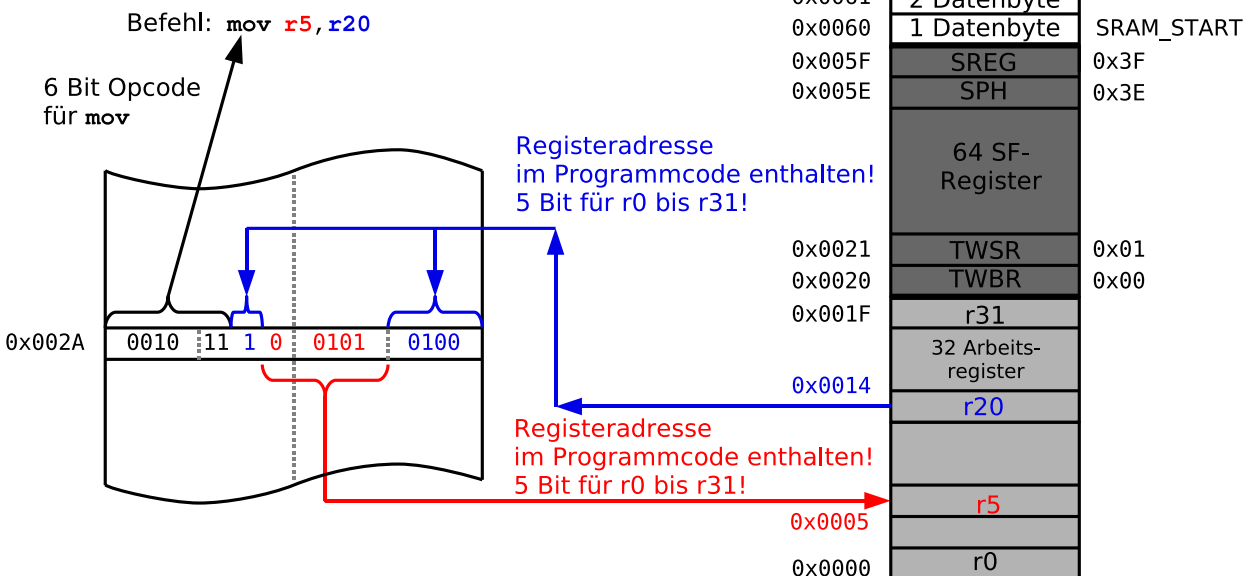
## Programmspeicher (Flash)

16 Bit breit

## Datenspeicher (SRAM)

8 Bit breit

### Registeradressierung 2 Register:



- ✎ **A402**
- Zeichne ein Flussdiagramm für ein Programm das folgende Berechnungen durchführt:  
Zuerst wird die Dezimalzahl **100** in die Variable (Register) **Tmp1** geladen. Dann wird das Register invertiert. Als zweite Zahl wird **90** addiert. Vom Resultat wird **55** mittels Registeradressierung subtrahiert um dann schlussendlich noch einmal **90** zu addieren.
  - Führe die Berechnung zuerst von Hand aus (Subtraktion durch Addition des Zweierkomplements!).
  - Schreibe das Programm und teste es im Studio 4 im Step-Modus.  
Gib dem Programm den Namen "**A402\_register.asm**".
  - Betrachte das Statusregister (Flagregister) **SREG** und die Resultate der Berechnungen im "I/O-View"-Fenster und interpretiere die Ergebnisse.

**add Rd,Rr**

Addition des Arbeitsregisters Rd mit dem Arbeitsregister Rr (*add without carry*).

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

Resultat in Rd. Carry-Flag wird nicht dazuaddiert (siehe adc)  
Beeinflusste Flags: H, S, V, N, Z, C    Taktzyklen: 1

**sub Rd,Rr**

Subtrahiere das Arbeitsregisters Rr vom Arbeitsregister Rd (*subtract without carry*).

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

Resultat in Rd. Carry-Flag wird nicht subtrahiert (siehe sbc)  
Beeinflusste Flags: H, S, V, N, Z, C    Taktzyklen: 1

## Die direkte Adressierung der SF-Register (SonderFunktions-Register)

Es gibt sechs Befehle für die direkte Adressierung der SF-Register:

```

in    r19,0x10      ;Adresse von PIND (Definitionsdatei)
out   PORTD,Tmp1
sbi   PORTD,4
cbi   0x18,PB7      ;Adresse von PORTB (Definitionsdatei)
sbic  PORTA,2
sbis  0x18,PB3
    
```

Die Befehle "sbi", "cbi", "sbic" und "sbis" können nur die unteren 32 SF-Register (0-31) ansteuern. Bitmanipulationen der oberen 32 Register müssen also mit Maskierungen erfolgen.

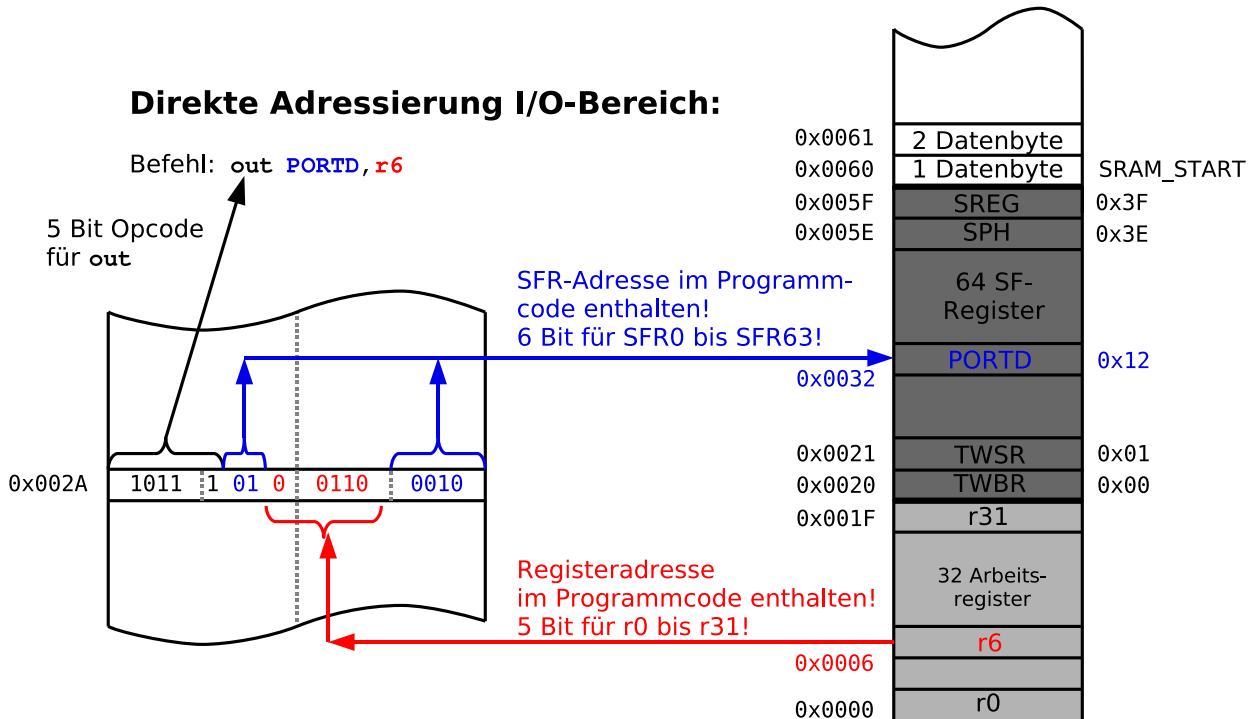
## Programmspeicher (Flash)

16 Bit breit

## Datenspeicher (SRAM)

8 Bit breit

### Direkte Adressierung I/O-Bereich:



## Die direkte Adressierung des Datenspeichers (SRAM)

Für die direkte Adressierung des Datenspeichers existieren nur 2 Befehle ("**lds**" und "**sts**"). Es sind die einzigen Befehle die 2 Befehlswoorte (32 Bit) benötigen. Es sind zwei Taktzyklen nötig. Das höherwertige Befehlswoort enthält den Opcode und die Adresse des Arbeitsregisters. Das niederwertige Befehlswoort enthält die SRAM-Adresse, die also unveränderbar im Programmcode (Flash) enthalten ist.

Es gibt zwei Befehle für die direkte Adressierung des SRAM:

```

sts    0x60, Tmp1    ;Speichere den Inhalt des Registers Tmp1
                    ;in den Datenspeicher auf die
                    ;Anfangsadresse 0x0060 (engl. store)
lds    r16, 0x0AAA  ;Lade den Inhalt der SRAM-Adresse 0x0AAA
                    ;in das Arbeitsregister r16 (engl. load)
    
```

Der ganze Adressbereich des SRAM kann adressiert werden. Es ist also auch möglich die Arbeitsregister und die SF-Register so zu adressieren. Wegen der größeren und langsameren Befehle macht das aber wenig Sinn, da bessere Befehle zur Verfügung stehen.

Mit zusätzlichem externen Speicher ist eine Adressierung bis 64 KiB möglich (0xFFFF).

**sts k,Rr**

**Kopiert den Registerinhalt von Rr (8 Bit) direkt ins SRAM (*store direct to data space (sram)*).**

1	0	0	1	0	0	1	r	r	r	r	r	0	0	0	0
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

Direkte Adressierung! Die 16 Bit-Adresse ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).  
**Beeinflusste Flags: keine    Taktzyklen: 2**

**lds Rd,k**

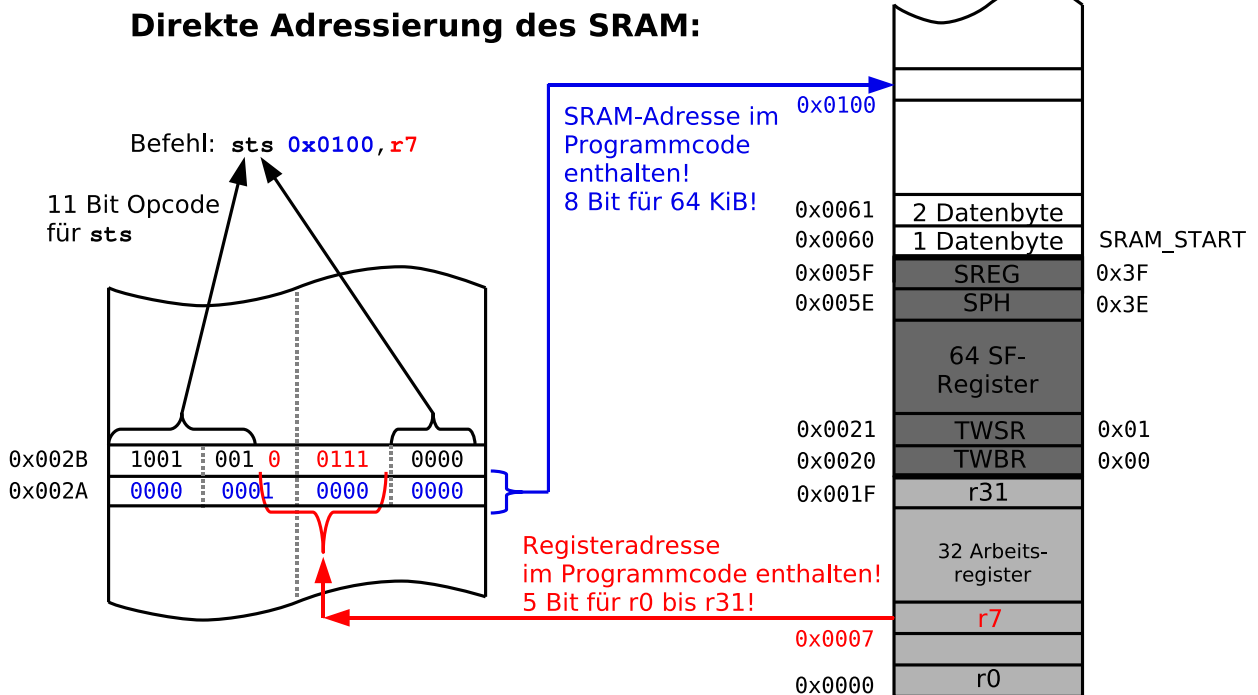
**Kopiert eine Speicherzeile (8 Bit) aus dem SRAM ins Register Rd (*load direct from data space (sram)*).**

1	0	0	1	0	0	0	d	d	d	d	d	0	0	0	0
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k

Direkte Adressierung! Die 16 Bit-Adresse ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).  
**Beeinflusste Flags: keine    Taktzyklen: 2**

## Programmspeicher (Flash) 16 Bit breit

## Datenspeicher (SRAM) 8 Bit breit



Die direkte Adressierung des Datenspeichers ist unflexibel, da die Adresse nicht durch das Programm verändert werden kann. Bei der Bearbeitung mehrerer Daten ist für jeden Zugriff eine Zeile Programmcode nötig. Für die Adressierung größerer Datenbestände wird man die indirekte Adressierung verwenden.

- 🔪 **A403**
- Schreibe das vorige Programm so um, dass zuerst alle vier Zahlen in den Datenspeicher geladen werden (ab Adresse **0x0100**). Auch sollen alle Resultate der Berechnungen im Datenspeicher abgelegt werden. Es sollen nur zwei Arbeitsregister verwendet werden. Versuche mit einem Minimum an Programmzeilen auszukommen.  
Gib dem Programm den Namen "**A403\_sram\_direct.asm**".
  - Teste das Programm im Studio 4 im Step-Modus. Beobachte die Veränderungen im Speicher indem du das Speicherfenster einschaltest ("View/Memory" oder "Alt+4").

## 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.

Dies bietet den großen Vorteil, dass die Adresse veränderbar ist und somit zum Beispiel in einer Schleife erhöht werden kann um große Datensätze oder Tabellen zu adressieren. Da die Adresse 16 Bit groß ist, werden für sie zwei Arbeitsregister benötigt. Hierfür sind die Arbeitsregister **r26-r31** vorgesehen welche als Doppelregister **X (r27:r26)**, **Y (r29:r28)** und **Z (r31:r30)** angesprochen werden können.

**Vor der indirekten Adressierung muss der Adresszeiger initialisiert werden!**

### Beispiel:

```
ldi XL,0x00 ;Adresszeiger X mit 0x0100 initialisieren
ldi XH,0x01 ;
```

Mit der indirekten Adressierung kann ebenfalls der gesamte SRAM-Bereich adressiert werden, also im Notfall auch die Arbeits- und SF-Register sowie auch eventuell vorhandener externer Speicher (64 KiB).

Einige Befehle für die indirekte Adressierung des SRAM (es existieren je ein "**ld**" (*load*) und ein "**st**" (*store*) Befehl pro Doppelregister (6 mögliche Befehle)):

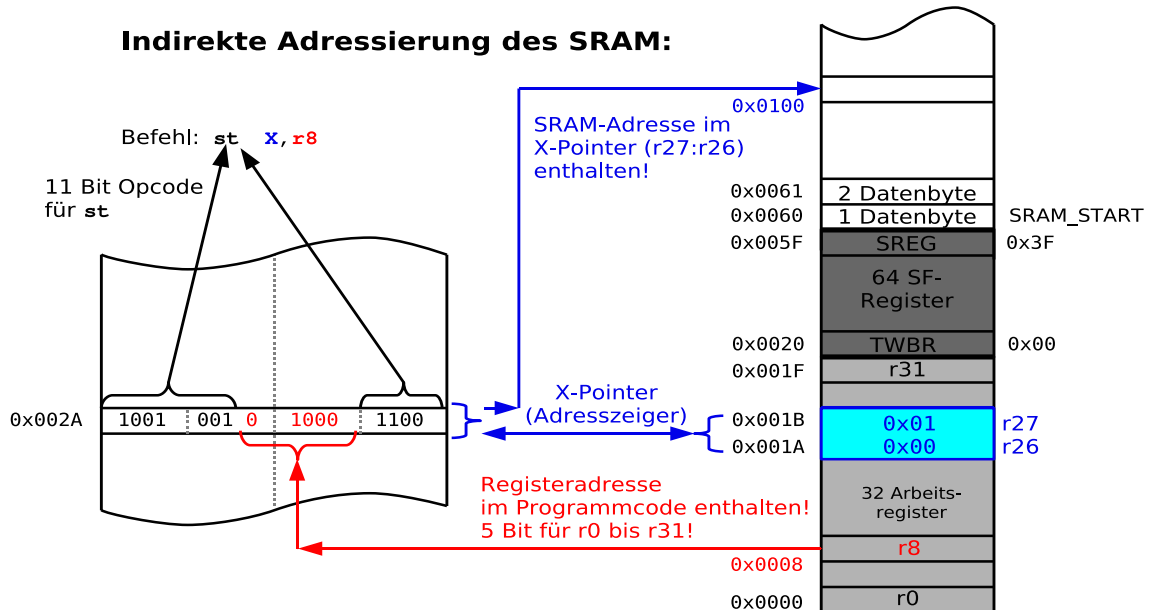
```
st X,Tmp1 ;Speichere den Inhalt des Registers Tmp1 in
           ;den Datenspeicher. Die Adresse befindet sich
           ;im Doppelregister X (engl. store)
ld r16,Y ;Lade den Inhalt der SRAM-Adresse die sich im
          ;Doppelregister Y befindet in das Arbeits-
          ;register r16 (engl. load)
```

## Programmspeicher (Flash)

16 Bit breit

## Datenspeicher (SRAM)

8 Bit breit



### A404

- Entwickle das Flussdiagramm eines Programms, das den SRAM-Speicher ab der Adresse **0x0100** mit den Dezimalzahlen **0** bis **255** auffüllt.
- Schreibe das Assemblerprogramm. Benutze das **X**-Register um den Adresszähler zu initialisieren (Das niederwertige Byte kann mit "**XL**" adressiert werden, das höherwertige Byte mit "**XH**" (siehe Definitionsdatei)). Nenne das Programm "**A404\_sram\_indirect\_1.asm**".
- Teste das Programm im Studio 4 zuerst im Step-Modus. Lass das Programm dann ganz durchlaufen ("Run" ("F5")) dann "Break" ("Ctrl+F5") und beobachte das Resultat im Speicherfenster ("View/Memory" oder "Alt+4").

### st X,Rr

Kopiert den Registerinhalt von Rr (8 Bit) indirekt mittels Adresszeiger X ins SRAM (*store indirect from register to data space (sram) using index X*).

1	0	0	1	0	0	1	r	r	r	r	r	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Indirekte Adressierung! Die 16 Bit-Adresse im Adresszeiger ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).

**Beeinflusste Flags: keine**    **Taktzyklen: 2**

## ld Rd,X

Kopiert eine Speicherzelle (8 Bit) aus dem SRAM indirekt mittels Adresszeiger X ins Register Rd (*load indirect from data space (sram) to register using index X*).

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

Indirekte Adressierung! Die 16 Bit-Adresse im Adresszeiger ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).

**Beeinflusste Flags:** keine    **Taktzyklen:** 2

## inc Rd

Inkrementiere Rd (*increment*).

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

$Rd \leftarrow Rd + 1$ . Resultat in Rd. Das Carry-Flag wird nicht beeinflusst!

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

## dec Rd

Dekrementiere Rd (*decrement*).

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

$Rd \leftarrow Rd - 1$ . Resultat in Rd. Das Carry-Flag wird nicht beeinflusst!

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

**Bemerkung:** Bei der indirekten Adressierung muss praktisch immer der Adresszeiger laufend erhöht oder erniedrigt werden. Werden mehr als 256 Speicherzeilen adressiert, so reicht der einfache Inkrement bzw. Dekrement-Befehl der ein Byte adressiert auch nicht mehr aus und es muss mit 16 Bit gerechnet werden (Befehle "**adiw**" bzw. "**sbiw**"). Um dies zu vereinfachen bietet die ATmega-Serie die indirekte Adressierung mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers.

## Indirekte Adressierung mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers

Einige Befehle für die indirekte Adressierung des SRAM mit automatischem Erhöhen bzw. Erniedrigen des Adresszeigers (aus 12 möglichen Befehlen):

```

st  Y+,Tmp1    ;Speichere den Inhalt des Registers Tmp1 in
                ;den Datenspeicher. Die Adresse befindet sich
                ;im Doppelregister Y. Erhoehe danach den
                ;Adresszeiger um Eins (Y=Y+1)
st  -Z,Tmp1    ;Erniedrige zuerst den Adresszeiger um Eins
                ;(Z=Z-1). Speichere danach den Inhalt des
                ;Registers Tmp1 in den Datenspeicher (Adresse
                ;in Z)
ld  r16,Z+     ;Lade den Inhalt der SRAM-Adresse die sich im
                ;Doppelregister Z befindet in das Arbeits-
                ;register r16 und erhoehe dann den
                ;Adresszeiger um Eins (Z = Z+1)
ld  r16,-X     ;Erniedrige zuerst den Adresszeiger um Eins
                ;(X=X-1) und lade dann den Inhalt der SRAM-
                ;Adresse die sich im Doppelregister X befindet
                ;in das Arbeitsregister r16.
    
```

Beim Inkrementieren (**Post-Inkrement**) wird immer zuerst gespeichert bzw. geladen und dann erst inkrementiert. **Inkrementiert** wird **nach der Operation!** Im Befehl befindet sich das **Pluszeichen** hinter dem Adresszeiger.

Beim Dekrementieren ist es umgekehrt (**Pre-Dekrement**)! Der Adresszeiger wird zuerst dekrementiert ehe die Daten gespeichert oder geladen werden. **Dekrementiert** wird **vor der Operation!** Im Befehl befindet sich das **Minuszeichen** immer vor dem Adresszeiger.

**Bemerkung:** Der folgende Post-Inkrement-Befehl:

```
st  Y+,Tmp1
```

spart eine Befehlszeile und 2 Taktzyklen. Er entspricht den beiden Befehlen:

```
st  Y,Tmp1
adiw YL,1
```

Der folgende Pre-Dekrement-Befehl:

```
st  -Z,Tmp1
```

spart ebenfalls eine Befehlszeile und 2 Taktzyklen. Er entspricht den beiden Befehlen:

```
sbiw ZL,1
st  Z,Tmp1
```

Es existieren keine Post-Dekrement oder Pre-Inkrement-Befehle.

- A405** Ändere das vorige Programm so um, dass der Speicher ab der Adresse **0x0060** bis zur Adresse **0x03FF** mit dem Buchstaben 'B' auffüllt wird (siehe ASCII-Tabelle im Anhang). Es soll ein Post-Inkrement-Befehl mit dem **Y**-Register verwendet werden.  
 Nenne das Programm: "**A405\_sram\_indirect\_postincrement.asm**".  
 Teste das Programm im Studio 4.

**st Y+,Rr**

Kopiert den Registerinhalt von Rr (8 Bit) indirekt mittels Adresszeiger Y (Post-inkrementiert) ins SRAM (*store indirect from register to data space (sram) using index Y*).

1	0	0	1	0	0	1	r	r	r	r	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Indirekte Adressierung! Nach dem Kopieren wird der Adresszeiger automatisch erhöht! Die 16 Bit-Adresse im Adresszeiger ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren (bis 64 KiB).  
**Beeinflusste Flags: keine Taktzyklen: 2**

## Indirekte Adressierung mit festem (konstantem) Abstand

Die ATmega-Serie bietet eine weitere Annehmlichkeit bei der indirekten Adressierung. Es besteht die Möglichkeit mehrere Datenbytes mit einem festen Abstand zur momentanen Adresse (Adresszeiger) zu adressieren. Dies bietet eine Vereinfachung bei der Adressierung von Tabellen mit festen Datensätzen.

Es können nur die Adresszeiger **Y** und **Z** benutzt werden! Der Abstand (**displacement**) kann 5 Bit betragen (0-63). Der Abstand wird für die Operation zum Adresszeiger dazu addiert, wobei dessen Inhalt nicht verändert wird.

Es gibt vier Befehle für die indirekte Adressierung des SRAM mit festem Abstand:

```

std  Y+q,Tmp1    ;Speichere den Inhalt des Registers Tmp1 in
                ;den Datenspeicher. Die Adresse ist die Summe
                ;aus dem Doppelregister Y und dem Abstand q
std  Z+q,r20     ;Speichere den Inhalt des Registers r20 in
                ;den Datenspeicher. Die Adresse ist die Summe
                ;aus dem Doppelregister Z und dem Abstand q
ldd  r16,Y+q     ;Lade den Inhalt der SRAM-Adresse (Y+q) in das
                ;Arbeitsregister r16
ldd  Tmp1,Z+q    ;Lade den Inhalt der SRAM-Adresse (Z+q) in das
                ;Arbeitsregister Tmp1
    
```

- ✎ **A406** Ein Datenloggerprogramm hat im SRAM des ATmega-Controllers ab der Adresse **0x0100** jede Stunde 8 Mittelwerte von 8 Sensoren (Sensor 1 bis Sensor 8) abgespeichert. Jeweils nach einer Woche (168 Stunden) sollen die Werte der Sensoren drei, eins und fünf in dieser Reihenfolge auf den LEDs (**PORTD**) ausgegeben werden.
- a) Zeichne das Flussdiagramm und schreibe ein Programm um diese Aufgabe zu erledigen.  
 Nenne das Programm: "**A406\_sram\_indirect\_displacement.asm**".
- b) Teste das Programm im Studio 4. Gib dazu vor dem schrittweisen "Debugging" manuell 20 Werte im Speicherfenster ab Adresse **0x0100** ein. Beobachte die Adresszeiger, die Register und Port D im Ein-/Ausgabefenster (I/O-View).

## ldd Rd,Z+q

Kopiert eine Speicherzelle (8 Bit) aus dem SRAM indirekt mittels Adresszeiger Z (+ Abstand) ins Register Rd (*load indirect from data space (sram) to register using index Z*).

1	0	q	0	q	q	0	d	d	d	d	0	q	q	q
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Indirekte Adressierung! Zum Adresszeiger wird ein fester (konstanter) Abstand q vor der Adressierung hinzu addiert. Die 16 Bit-Adresse im Adresszeiger ermöglicht es den gesamten SRAM (ATmega32A) zu adressieren.

**Beeinflusste Flags: keine Taktzyklen: 2**

## adiw Rdl,K

Addiert die Konstante K (0-63) zu einem Doppelregister (*add immediate to word*).

1	0	0	1	0	1	1	0	K	K	d	d	K	K	K	K
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Resultat im Doppelregister. Neben den Doppelregistern X,Y,Z kann auch das Doppelregister r25:r24 verwendet werden. Obschon das Doppelregister adressiert wird ist im Befehl nur das niederwertige Register anzugeben Bsp.: adiw YL,10. (andere Schreibweisen sind je nach Assembler möglich). Die Konstante kann maximal 63 betragen (6 Bit).

**Beeinflusste Flags: S, V, N, Z, C Taktzyklen: 2**

## Indirekte Adressierung mit "push" und "pop"

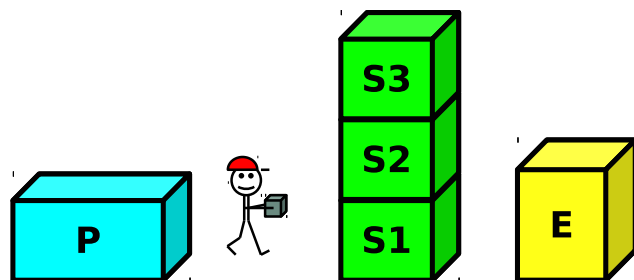
Eine Sonderform der indirekten Adressierung erfolgt mit den Befehlen **"push"** und **"pop"** in Kombination mit dem Stapelzeiger SP als Adresszeiger. In einem späteren Kapitel wird diese Adressierung behandelt.

## Wiederholung

Um die Adressierung besser zu verstehen, wollen wir die Adressierungsarten noch mal anhand eines Beispiels mit Transferbefehlen erläutern. Zu jedem Unterpunkt soll noch einmal die entsprechende Speicher-Grafik betrachtet werden.

### Das kleine Märchen vom Laufburschen.

Ein Laufbursche soll Pakete austragen. Auf dem Gelände der Firma befinden sich drei Gebäude. Das Direktionsgebäude P (Programm), das Verwaltungsgebäude S (SRAM) und ein Lagerraum E (EEPROM).



Die Befehle erhält der Laufbursche im Direktionsgebäude P.

Das Verwaltungsgebäude hat drei Stockwerke: Im ersten Stock des Gebäudes befinden sich die Büros der Administration für gängige Arbeiten (Arbeitsregister). Im zweiten Stock die Büros für Im- und Export (SF-Register). Im dritten Stock das Archiv (Speicher). Der Lagerraum E kann nur mit einer speziellen Genehmigung betreten werden.

- Bei einer **unmittelbaren Adressierung** erhält der Laufbursche sein Paket (**0xA3**) bereits im Direktionsgebäude und liefert es einfach an ein Administrationsbüro (**r18**) im Gebäude S (erster Stock S1) aus (**ldi r18,0xA3**).
- Bei den **direkten Adressierungen** holt er das Paket (**0xA3**) in einem Administrationsbüro (**r20**) im Gebäude S (erster Stock S1) ab, und trägt es in ein anderes Büro oder in das Archiv im selben Gebäude. Ziel- und Quelladresse der Büros (bzw. des Archivs) hat er zuvor im Direktionsgebäude erfragt. Je nach Stockwerk unterscheiden wir die:

**Registeradressierung:** Er trägt das Paket in ein anderes Administrationsbüro (**r5**) im gleichen Stockwerk (**mov r5,r20**).

**Adressierung des Ein- Ausgabebereichs:** Er trägt das Paket vom ersten (S1) in das zweite Stockwerk (S2), in ein Büro (**PORTD**) der Abteilung Im- und Export (**out PORTD,r20**).

**Adressierung des SRAM-Speichers:** Er trägt das Paket vom ersten in das dritte Stockwerk (S3) ins Archiv. Hier wird eine größere Adresse (16 Bit) benötigt, da das Archiv sehr viele Kisten zum Archivieren besitzt (**sts 0x0AAA,r20**).

- Bei der **indirekten Adressierung** kennt der Laufbursche die Adresse nicht, aber im Direktionsgebäude (P) teilt man ihm mit in welchem der drei Großraumbüros **X**, **Y** oder **Z** er die Adresse zuvor abholen soll. Erst dann kann er das Paket ausliefern (**st X, r20**). Meist ist es eine große Adresse für das Archiv (S3).

**Pre-Dekrement:** Hierbei kann es vorkommen, dass er dem Beamten im Großraumbüro mitteilen muss, dieser soll die Adresse um Eins erniedrigen, bevor er sie dem Laufburschen aushändigt (**st -X, r20**).

**Post-Inkrement:** Es kann auch vorkommen, dass der Laufbursche die Adresse entgegen nimmt und dann dem Beamten mitteilt, er soll die im Großraumbüro aufbewahrte Adresse um Eins erhöhen (**st X+, r20**).

**Displacement:** Eine dritte Möglichkeit ist die, dass man dem Laufburschen im Direktionsgebäude eine Zahl (0-63) nennt, welche er zur Adresse (nachdem er diese im Großraumbüro erhalten hat) addieren soll (**std X+3, r20**).

- 🔗 **A407**
- a) Zeichne ein Flussdiagramm eines Programms, das den Speicherbereich von der Adresse **0x0060** bis **0x01FF** (erste Tabelle) nach **0x0300** (zweite Tabelle) verschiebt (Cut and Paste). Anstelle der verschobenen Daten soll nachher das Null-Byte (siehe ASCII-Tabelle) in der ersten Tabelle stehen.
  - b) Schreibe das entsprechende Programm und gib dem Programm den Namen **"A407\_sram\_indirect\_cut\_paste.asm"**.
  - c) Teste das Programm im Studio 4. Gib dazu vor dem schrittweisen "Debugging" manuell 20 Werte im Speicherfenster ab Adresse **0x0060** ein. Beobachte die Adresszeiger und Register im Ein-/Ausgabefenster (I/O-View).

## Adressierung des Programmbereichs

Beim Programmbereich unterscheiden wir folgende Adressierungsarten:

- **Relative Adressierung des Programmspeichers**  
(Befehle **"rjmp"** und **"rcall"**)
- **Direkte Adressierung des Programmspeichers**  
(Befehle **"jmp"** und **"call"**)
- **Indirekte Adressierung des Programmspeichers**  
(Befehle **"ijmp"** und **"icall"**)
- **Indirekte Adressierung von Konstanten im Programmspeicher**  
(Befehle **"lpm"**)

### Relative Adressierung des Programmspeichers mit "rjmp" und "rcall"

Meist werden relative Sprünge benutzt. Hierbei wird ein relativer positiver (Vorwärtssprung) oder negativer (Rückwärtssprung) Abstand **k** zur momentanen Adresse

hinzu addiert. Der Assembler berechnet diesen Abstand mit Hilfe der Labels. Hierbei ist zu beachten, dass nach der Addition des Abstandes zum Programmzähler (*Program Counter* PC) dieser nochmals um Eins erhöht wird ( $PC = PC+k+1$ ).

Bei relativen Sprüngen stehen im Opcode für den Abstand 12 Bit zur Verfügung. Es kann also maximal über 2k Worte vorwärts und rückwärts gesprungen werden. Soll im Programm noch weiter gesprungen werden, so muss die direkte Adressierung verwendet werden. Der Assembler überwacht ob die Grenze überschritten wird und meldet in diesem Fall einen Fehler.

Die relative Adressierung ist schneller und Platz sparender als die direkte Adressierung. Man soll also, falls möglich, diese Adressierung verwenden.

Hier ein kurzer Ausschnitt aus einer List-Datei:

```
00002e 9985     MAIN:  sbic  PIND,PButt  ;
00002f cffe           rjmp  MAIN
000030 e420           ldi   Mask,0x40  ;
000031 2702           eor   Tmp1,Mask  ;
000032 bb02           out  PORTD,Tmp1  ;
000033 cffa           rjmp  MAIN      ;
```

Der erste "rjmp"-Befehl (Opcode: **0xc**) springt ein Wort rückwärts. Der Abstand muss also -2 oder **0xfffffe** (Zweierkomplement!) betragen, da der PC nach der Addition noch um Eins erhöht wird!. Der zweite "rjmp"-Befehl springt 5 Worte rückwärts ( $k = -6$  entspricht **0xfffffa**).

## Direkte Adressierung des Programmspeichers mit "jmp" und "call"

Bei der direkten Adressierung des Programmspeichers stehen 22 Bit! (4M Worte) für eine feste Adresse zur Verfügung. Die Befehle benötigen zwei Worte und drei oder vier Taktzyklen!

## Indirekte Adressierung des Programmspeichers mit "ijmp" und "icall"

Für spezielle Anwendungen (z.B. Umsetzung von "switch"-Konstruktionen in C) kann eine indirekte Adressierung des Programmspeichers benutzt werden. Die Sprungadresse muss dazu zuerst im **Z**-Adresszeiger initialisiert werden.

## Indirekte Adressierung von Konstanten im Programmspeicher mit "lpm"

Der Ladebefehl "lpm" (*load program memory*) ermöglicht es ein beliebiges Datenbyte aus dem Programmspeicher (Flash) in das Arbeitsregister **r0** zu laden. Die indirekte Adresse muss sich dazu im **Z**-Pointer (Adresszeiger) befinden.

Bei der ATmega-Familie ist es besser die beiden erweiterten "lpm" Befehle benutzen, da diese alle Arbeitsregister adressieren können und die Syntax mitteilt welches Register als Adresszeiger benutzt wird.

```

lpm    r20,Z    ;Speichere den Inhalt der Programmzeile deren
                ;Adresse im Z-Pointer steht in das Arbeits-
                ;register r20.

lpm    r20,Z+   ;Speichere den Inhalt der Programmzeile deren
                ;Adresse im Z-Pointer steht in das Arbeits-
                ;register r20. Inkrementiere danach den Adresszeiger Z
    
```

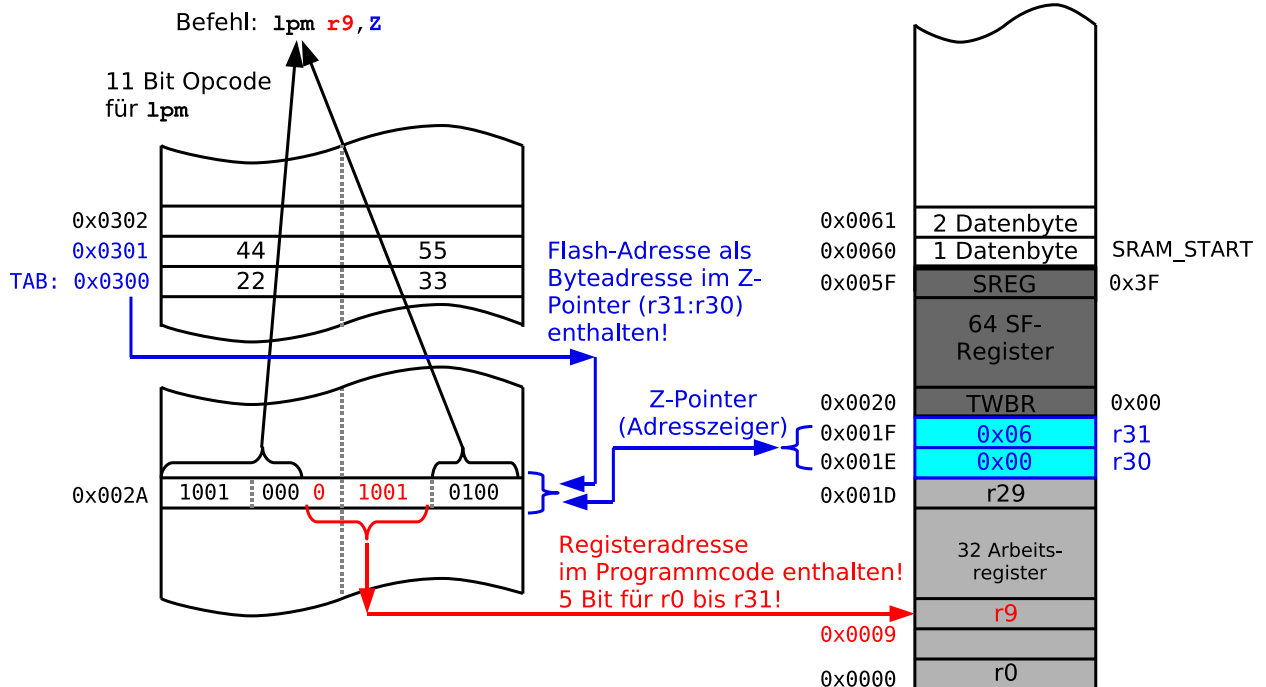
## Programmspeicher (Flash)

16 Bit breit (Wortadressen!)

## Datenspeicher (SRAM)

8 Bit breit

### Indirekte Adressierung von Konstanten im Flash:



### lpm Rd, Z

Kopiert eine Speicherzelle (8 Bit) aus dem Flash indirekt mittels Adresszeiger Z ins Register Rd (*load program memory*).

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

Indirekte Adressierung! Der gesamte Programmspeicher (Flash) des ATmega32A kann mittels Adresszeiger Z adressiert werden. Achtung! In Z muss die Byteadresse und nicht die Wortadresse enthalten sein (Byteadresse = 2 \* Wortadresse).

**Beeinflusste Flags: keine Taktzyklen: 3**

Diese Adressierungsart wird meist in Kombination mit der **.DB** (bzw. **.DW**) und eventuell der **.ORG** Direktive verwendet. Damit können Tabellen mit Konstanten (zum Beispiel Zeichenketten (Strings)) gleich beim Programmieren im Programmspeicher abgelegt werden. Die recht aufwändige Programmierung des EEPROM kann so vermieden werden.

<b>.ORG</b>	Adresse	Legt eine Anfangsadresse fest ab der der folgende Code abgespeichert wird. Hiermit kann der Speicher organisiert werden. Beispiel: <b>.ORG = 0xA00</b>
<b>.DB</b>	Liste mit Bytekonstanten	(engl.: „define Byte“) Fügt konstante Bytes ein. Dabei ist es die Bedeutung der Bytes egal (Zahl von 0..255, ASCII-Zeichen 'b', eine Zeichenkette "Hallo"; alle Bytes werden durch Kommas getrennt). Im Flash muss eine gerade Zahl von Bytes eingefügt werden (16 Bit-Worte), sonst hängt der Assembler ein Nullbyte an.
<b>.DW</b>	Liste mit Wortkonstanten	(engl.: „define Word“) Fügt konstantes binäres Wort (16 Bit) ein. Im EEPROM und Flash zuerst das niederwertige Byte, dann das höherwertige Byte.

Beispiel für die Programmierung einer Tabelle:

```

-----
;
;   Tabelle      (Konstantenbereich im Flash (hinter dem Programm))
;
-----
.ORG   0x0300           ;ab Adresse 0x0300 im Programmspeicher
TAB:
.DB    0x22,0x33,0x44,0x55 ;Tabelle mit 4 Byte
    
```

Auf diese Tabellen kann dann natürlich nur Lesend (load) zugegriffen werden.

**Bemerkung:** Die Speicherorganisation mit der der **.ORG** Direktive kann Programme vereinfachen. Bei großen Projekten mit unterschiedlichen Bibliotheken (bzw. Unterprogrammen) kann es allerdings zu Überschneidungen und damit zu Fehlern führen. Hier ist es besser die Speicherorganisation ausschließlich mit Labeln vorzunehmen.

Da der Flash-Speicher in Worten (2 Byte) organisiert ist und intern in Worten adressiert wird, ist es nötig die Wortadresse mit dem Faktor zwei zu multiplizieren, da der Adresszeiger (hier **Z**) mit einer byteweisen Adressierung arbeitet<sup>21</sup>.

Bei der Initialisierung des Z-Adresszeigers muss in unserem Beispiel also **0x0600** (Byteadresse) statt **0x0300** (Wortadresse) als Anfangsadresse verwendet werden.

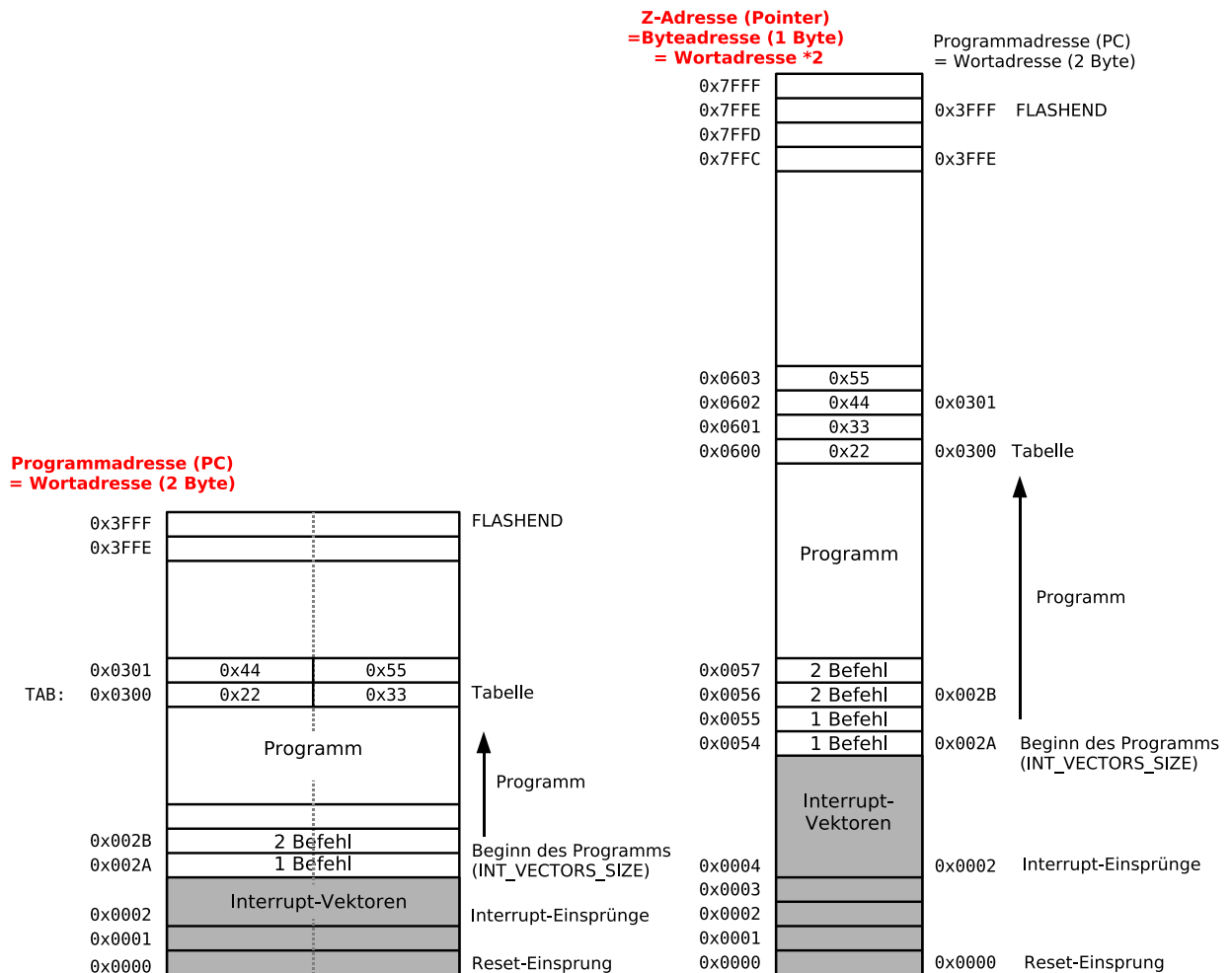
<sup>21</sup> Eine wortweise Adressierung macht keinen Sinn, da der Ladebefehl nur Werte in 1 Byte großen Arbeitsregistern ablegen kann. Jedes zweite Byte könnte so nicht genutzt werden.

## Programmspeicher (32KiB) FLASH

nichtflüchtiger Speicher, 16 Bit breit

### 16-Bit Darstellung

### 8-Bit Darstellung



Die Berechnung kann man getrost dem Assembler überlassen.

```
ldi    ZL,LOW(TAB*2)    ;Adresszeiger mit der Adresse der Tabelle
ldi    ZH,HIGH(TAB*2)  ;* 2 (Worte statt Bytes) initialisieren
```

Der verwendete Label **Tab** steht dabei für die durch Anfangsadresse der Tabelle.

Im Folgenden ein Beispielprogramm zur Verwendung von Tabellen im Programmspeicher.

```
-----
;
;   Initialisierungen und eigene Definitionen
;
-----
.ORG   INT_VECTORS_SIZE      ;Platz fuer ISR Vektoren lassen
INIT:
.DEF   Tmp1 = r16            ;Register 16 dient als erster Zwischenspeicher
      ;Adresszeiger initialisieren
      ldi    ZL,LOW(TXTSTR*2) ;Adresszeiger mit der Adresse der Texttabelle
      ldi    ZH,HIGH(TXTSTR*2);* 2 (Worte statt Bytes) initialisieren
      ;alle Pins von PortD als Ausgang initialisieren
      ldi    Tmp1,0xFF        ;DDR_D = 11111111b
      out    DDRD,Tmp1       ;alle Bits im Datenrichtungsregister auf Eins
```

```

;-----
;
;   Hauptprogramm
;-----
MAIN:  lpm      Tmp1,Z+      ;Speichere das Zeichen der Speicherzeile deren
;Adresse im Z-Pointer steht in das Arbeits-
;register Tmp1. Inkrementiere danach den
;Adresszeiger Z
      tst      Tmp1        ;Letztes Zeichen (Nullbyte) erreicht?
      breq     END         ;Falls ja Ende
      out     PortD,Tmp1   ;Gib das Zeichen an PortD aus
      rjmp    MAIN        ;Hole nächstes Zeichen

;-----
;   Ende des Hauptprogramms (falls keine Endlosschleife im Hauptprogramm)
END:   rjmp    END         ;Endlosschleife
;-----
;
;   Tabellen im Programmspeicher (Flash)
;-----
.ORG   0x0300             ;ab Adresse 0x0300
TXTSTR: .DB      "MICEL",10,13,0 ;Text mit "carriage return" und "line feed"
; sowie abschliessendem Nullbyte (ASCII-Tab)

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

**Bemerkung:** Die Tabelle im Beispiel besteht aus acht ASCII-Zeichen. Nach den fünf ASCII Buchstaben "MICEL", kommen drei ASCII-Steuerzeichen, die einfach durch Komma getrennt in Dezimal angeschrieben sind. Das erste Steuerzeichen ist ein Wagenrücklauf und das zweite Steuerzeichen ein Zeilenvorschub (siehe ASCII-Tabelle im Anhang). Das letzte Steuerzeichen (Nullbyte) wird verwendet um das Ende der Tabelle zu erkennen.

- ✎ **A408**
  - a) Teste das obige Programm im Studio 4. Stell das Speicherfenster beim Betrachten des Programmspeichers auf 2 Kolonnen ein (wortweise Adressierung, 16 Bit).  
Nenne das Programm: "**A408\_flash\_indirect\_textstring.asm**".
  - b) Was ist die Aufgabe des Programms?
- ✎ **A409**
  - a) Zu welcher Befehlsgruppe gehören die folgenden Befehle?
  - b) Nenne ebenfalls die jeweilige Adressierungsart:

<b>nop</b>	.....
<b>lsl r5</b>	.....
<b>ldd X+3, r20</b>	.....
<b>lpm</b>	.....
<b>swap r20</b>	.....
<b>rjmp NOL00P</b>	.....
<b>sbic 0x11,3</b>	.....
<b>adiw r28,4</b>	.....
<b>com Tmp1</b>	.....

```
ld    r22, -Z .....  
movw r16, r18 .....  
icall .....  
and  r1, r2  .....
```

