AVR-8-bit-Mikrocontroller
Gruppe 300 - Arbeiten mit AVR-Assembler
Teil 302 - Syntax der Assembler-Sprache

Teil 301 - Einführung

1 Eine Einführung in die Assembler-Sprache
- 1.1 Warum Assembler ?
- 1.2 In der Kürze liegt die Würze - Schnell wie der Wind
- 1.3 Grundausstattung und Installation der Werkzeuge

Teil 302 - Syntax der Assembler-Sprache

2 Struktur und Syntax der Assembler-Sprache
- 2.1 Struktur der Assembler-Zeilen
- 2.2 Struktur der Programm-Datei
- 2.3 Kommentare
- 2.4 Kopf des Programms
- 2.5 Assembler-Direktive (Directivs)
- 2.6 Assembler-Ausdrücke (Expressions)
  - 2.6.1 Operanden
  - 2.6.2 Operatoren
  - 2.6.3 Funktionen
  - 2.6.4 Operationen
- 2.7 Daten- und Adress-Struktur
  - 2.7.1 Nomenklatur: Abkürzungen für Register und Operanden
  - 2.7.2 AVR-Speicher-Räume aus Assembler-Sicht
    - 2.7.2.1 Programm-Speicher
    - 2.7.2.2 Daten-Speicher
    - 2.7.2.3 EEPROM-Speicher
  - 2.7.3 Adress-Modi
    - 2.7.3.1 Direkte Adressierung eines einzelnen Arbeits-Registers
    - 2.7.3.2 Direkte Adressierung von 2 Arbeits-Registern
    - 2.7.3.3 Direkte Adressierung eines I/O-Registers
    - 2.7.3.4 Direkte Daten-Adressierung im SRAM
    - 2.7.3.5 Indirekte Daten-Adressierung mit Displacement
    - 2.7.3.6 Indirekte Daten-Adressierung
    - 2.7.3.7 Indirekte Daten-Adressierung mit Pre-Decrement
    - 2.7.3.8 Indirekte Daten-Adressierung mit Post-Decrement
    - 2.7.3.9 Adressierung des Programm-Speichers
    - 2.7.3.10 Adressierung des Programm-Speichers mit Post-Increment
    - 2.7.3.11 Direkte Adressierung des Programm-Speichers
    - 2.7.3.12 Indirekte Adressierung des Programm-Speichers
    - 2.7.3.13 Relative Adressierung des Programm-Speichers

Teil 303 - Instruktionen

3 Mnemotechnische Abkürzungen der Instruktionen
- 3.1 Arithmetische und logische Instruktionen
- 3.2 Verzweigungs-Instruktionen
- 3.3 Instruktionen für den Daten-Transfer
- 3.4 Bit- und Bit-Test-Instruktionen

Teil 304 - Register

4 Register-Datei
- 4.1 Arbeits-Register
- 4.2 Unterschiede der Arbeits-Register
- 4.3 Pointer-Register
- 4.4 Empfehlungen zu Registerwahl

Teil 305 - I/O-Ports

5 Die Ein-/Ausgabe-Ports der AVR's
- 5.1 Die Ports der AVR's
- 5.2 Die Ports als I/O-Register
- 5.3 Das Status-Register SREG

Teil 306 - SRAM

6 Statisches RAM - SRAM
- 6.1 Beschreibung
- 6.2 Verwendung von SRAM
- 6.3 Zugriff auf das SRAM
- 6.4 Verwendung des SRAM als Stack
- 6.5 Einrichtung des Stapels
- 6.6 Verwendung des Stapels
- 6.7 Fehlermöglichkeiten beim (Hoch-)Stapeln

Teil 307 - Programmablauf

7 Steuerung des Programmablaufs
- 7.1 Der Reset
- 7.2 Linearer Programmablauf und Verzweigungen
- 7.3 Zeitzusammenhänge beim Programmablauf
- 7.4 Makros im Programmablauf
- 7.5 Unterprogramme (Subroutines)
- 7.6 Interrupts im Programmablauf

Teil 308 - Zahlendarstellung

8 Zahlendarstellungen im Assembler
- 8.1 Dual-Zahlen
- 8.2 Hexadezimal-Zahlen im ASCII-Format
- 8.3 Oktal-Zahlen
- 8.4 BCD-Zahlen (binär kodierte Dezimal-Zahlen)
- 8.5 Zahlen im ASCII-Format
- 8.6 Gleitkomma-Zahlen

Teil 309 - Bits, Bytes und Zahlen

9 Umwandlungen von Bits, Bytes und Zahlen
- 9.1 Bitmanipulationen, Schieben und Rotieren
- 9.2 Umwandlung einer Dual-Zahl in eine ungepackte BCD-Zahl
- 9.3 Umwandlung einer BCD-Zahl in eine Dual-Zahl
- 9.4 Umwandlung einer Dual-Zahl in eine Dezimal-Zahl im ASCII-Format
  - 9.4.1 Lösung 1
  - 9.4.2 Lösung 2
  - 9.4.3 Lösung 3
- 9.5 Umwandlung einer Dezimal-Zahl im ASCII-Format in eine Dual-Zahl
- 9.6 Umwandlung einer Dual-Zahl in eine Hexadezimal-Zahl im ASCII-Format
- 9.7 Umwandlung einer Hexadezimal-Zahl im ASCII-Format in eine Dual-Zahl
- 9.8 Lineare Umwandlung

Teil 310 - Rechnen

10 Rechnen in Assembler
- 10.1 Addition, Subtraktion und Vergleich
- 10.2 Multiplikation
- 10.3 Division

IN BEARBEITUNG

2 Struktur und Syntax der Assembler-Sprache

Es gibt es eine sehr umfangreiche Hilfefunktion für den Assembler, die mit AVRASM.chm aufgerufen werden kann (siehe Bild auf der nächsten Seite).

Hier werden für jeden AVR-Mikrocontroller die benutzbaren Instruktionen individuell aufgeführt und zusätzlich alle Instruktionen mit ihren Beschreibungen gesondert dargestellt.

Am Ende der Hilfe-Datei wird auch noch der AVR-Assembler 2 (ebenfalls Bestandteil des AVR Studios) beschrieben und darauf verwiesen, dass dieser auch für ältere Versionen anwendbar ist:

Ab der Freigabe von AVR Studio Vers. 4.11 ist der AVR Assembler 2 (AVRASM2) der Standard-Assembler (default). Er ist rückwärts kompatibel mit dem AVRASM und eröffnet viele neue Möglichkeiten. Der alte Assembler ist zwar noch gültig, wird aber nicht mehr gepflegt und neue AVRs unterstützen ihn nicht mehr.

Gleich ein Hinweis: Der "AVR Assembler 2" geht dazu über, die C-Syntax für Preprozessor-Anweisungen zu verwenden. Diese haben dann als Kennzeichen ein "#" vorangestellt.

Die Include-Datei (partfile) `m88def.inc` für den einzusetzenden Mikrocontroller (hier: ATmega88) wird automatisch aus dem bei der Installation von AVR Studio vorgesehen Programm-Ordner zugeladen und braucht nicht mehr im selben Ordner wie das Assembler-Quellprogramme aufgeführt zu werden. Für den aktuellen Assembler (in den Programm-Ordnern steht **AvrAssembler2 mit der Ordnungszahl 2 !** im Gegensatz zum ebenfalls noch vorhandenen älteren Assembler **ohne Ordnungszahl**) stehen die Include-Dateien für alle AVR-Mikrocontroller, die von AVR Studio bedient werden können, im Ordner

`C:\Program Files (x86)\Atmel\AVR Tools\AvrAssembler2\Appnotes`

Die umständliche vollständige Pfadangabe wird weder in **Additional Include path** des Menüs für die Assembler-Optionen noch in der `.INCUDE`-Direktive im Quellprogramm angegeben.

In den Assembler-Optionen wird noch der Haken gesetzt, um ein Listen-File zu erzeugen. Die Assembler-Optionen werden aufgerufen durch:

Project => Assembler Options

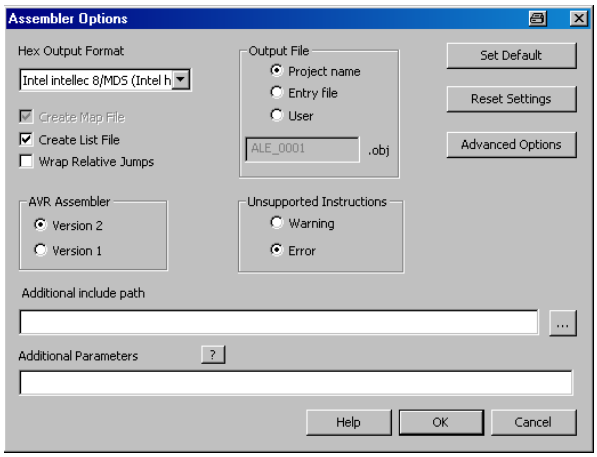


Bild 2-01: Assembler-Optionen aufrufen

Create Map File ist Standard-Einstellung und

Create List File erhält einen zusätzlichen Haken

Mit der Einstellung **Version 2** (Default-Einstellung) wird der neueste Assembler eingestellt. Sollte ein älteres ASM-Programm assembliert werden, welches nicht mit dem AVR-Assembler 2 kompatibel ist, so hilft ggf. die Markierung bei **Version 1**

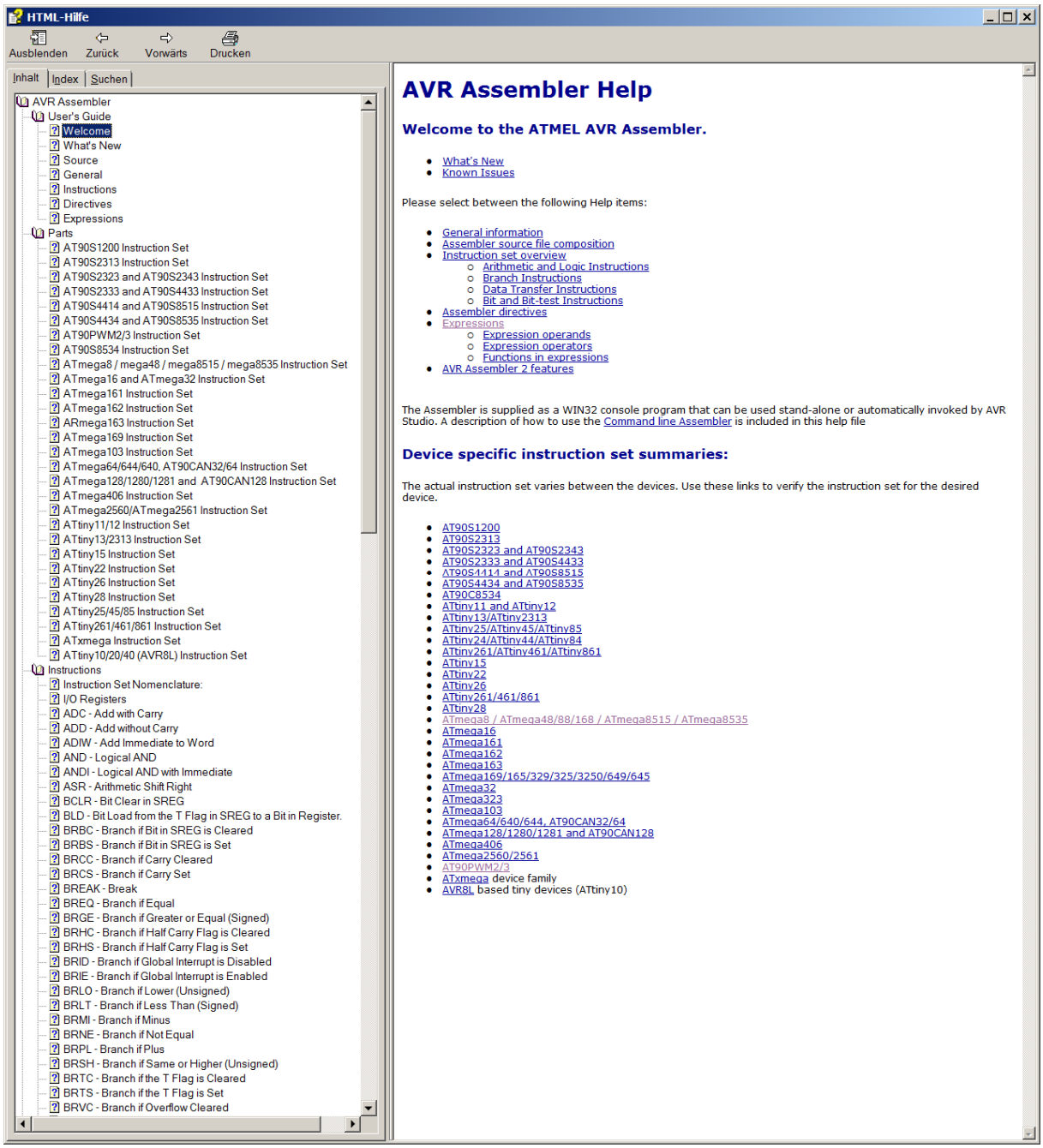


Bild 2-02: Assembler-Hilfe - Aufruf des Hilfspramms AVRASM.chm

2.1 Struktur der Assembler-Zeilen

Die Zeilen der Assembler-Sprache bestehen aus mnemonischen **Instruktionen**, **Labels** und **Direktiven** und werden gemäß der Programmstruktur der Reihe nach in einer Quelldatei (Source File) in Form von **ASCII-Zeichen** gem. der ASCII-Tabelle (siehe **Abschnitt 8.5 Zahlen im ASCII-Format** im **Teil 308**) abgelegt und mit **`CR/LF (EOL)`** abgeschlossen. Instruktionen und Direktiven sind darüber hinaus noch meistens mit **Operanden** versehen. Eine Zeile darf höchstens 120 Zeichen beinhalten. Sinnvoll ist es jedoch, die Zeilen auf maximal 80 Zeichen (Zeichengröße im Listing 10 Pkt.) zu beschränken, da diese vom Editor-Programm beim Ausdrucken gerade noch in einer Zeile dargestellt werden können. Alle darüber hinaus in der Zeile angeordneten Zeichen werden beim Ausdrucken an den Anfang der nächsten Zeile gesetzt, so dass das Gesamtbild sehr "flatterhaft" aussieht.

Jede Assembler-Zeile kann mit einem sog. Label beginnen. Ein Label besteht aus einer alphanumerischen Zeichenfolge (String), die durch einen Doppelpunkt abgeschlossen wird. Das Label ist sozusagen die "Hausnummer" (Adresse) der nachfolgenden Instruktion und kann auch in der der Instruktion vorangegangenen Zeile allein aufgeführt werden, so dass die Auflistung des Programms strukturierter aussieht. Labels werden gebraucht, um Programmsprüngen und Verzweigungs-Instruktionen das genaue Ziel anzugeben.

Ebenfalls werden variable Namen von Speicheradressen mittels alphanumerischer Zeichenfolgen dargestellt, um das Programm anwendungsfreundlicher zu gestalten.

Eine Assembler-Zeile kann eine der folgenden Formen besitzen:

```
[Label:]
[Label:] Direktive [Operanden] [;Kommentar]
[Label:] Instruktion [Operanden] [;Kommentar]
[;Kommentar]
[Leerzeile]
```

Ausdrücke in eckigen Klammern sind optional, d.h. sie können - müssen aber nicht - angegeben werden. Die eckigen Klammern **[ ]** stellen dabei lediglich Begrenzungen und kein Bestandteil des Ausdrucks dar; sie sind bei der Codierung **nicht einzugeben**. Der Text zwischen dem Kommentar-Abgrenzer (Semikolon **;**) und dem Ende der Zeile (EOL) wird vom Assembler ignoriert.

2.2 Struktur der Programm-Datei

Zunächst: Es gibt keine Einschränkungen im Bezug auf Spaltenanordnungen der Labels, Direktiven, Instruktionen und Kommentare. Lediglich sind die Reihenfolgen und die Trennungen der Ausdrücke in den Assembler-Zeilen nach den oben beschriebenen Formen einzuhalten.

Beispiel einer Struktur ohne Design:

```
label: .EQU var1=100 ;Setze var1 auf 100 (Direktive)
.EQU var2=200 ;Setze var2 auf 200
test: rjmp test ;Unendliche Schleife (Instruktion)
;Reine Kommentarlinie
;Eine weitere Kommentarlinie
```

Aber: Wenn man sich beim Codieren (d.h. beim Eingeben) der Assembler-Zeilen nicht selbst gewisse Ordnungskriterien auferlegt, wird man später bei Änderungen - und auch schon bei der anfänglichen Fehlersuche - sehr leicht die Übersicht verlieren!

Daher werden hier ein paar praktische Tipps für den Aufbau (Design) einer Programm-Datei gegeben, d.h. die nachfolgenden Assembler-Programme werden nach selbst festgelegten Regeln strukturiert. Und da es die eigenen Regeln sind, können sie - wenn es zweckmäßig erscheint - natürlich auch nach anderen Mustern verändert werden.

Hier noch einmal das Beispiel in einer etwas strukturierteren Form, aus der einige Regeln für die Assembler-Zeilen abgeleitet werden sollen. Der Inhalt der Assembler-Zeilen ist hier noch ohne Bedeutung.

Beispiel:

```
; Tabulatoren:
;       10        20        30        40
LABEL:
.EQU    VAR1      = 100               ; Setze VAR1 auf 100 (Direktive)
.EQU    VAR2      = 200               ; Setze VAR2 auf 200
;
TEST:
        rjmp      TEST                ; Unendliche Schleife (Instruktion)
;
; Reine Kommentarlinie
```

```
; Eine weitere Kommentarlinie
```

1. **Label** als Sprungadressen müssen stets am Anfang und sollten allein in der Zeile erscheinen. Sie sollten möglichst mit Großbuchstaben geschrieben werden und müssen durch einen Doppelpunkt abgeschlossen werden (z.B. `TEST:`).
2. **Direktiven** (beginnen immer mit einem Punkt und sollten in Großbuchstaben geschrieben werden) stehen ebenfalls am Anfang einer Zeile (z.B. `.EQU`).
3. Der erste **Operand einer Direktiven** steht an der ersten Tabulatorstellung (ca. Spalte 10).
4. Der nächste **Operand der Direktive** steht an der zweiten Tabulatorstellung (ca. Spalte 20).
5. **Instruktionen** stehen an der ersten Tabulatorstellung (ca. Spalte 10) und werden mit Kleinbuchstaben dargestellt (z.B. `rjmp`).
6. Der **Operand einer Instruktion** steht an der zweiten Tabulatorstellung (ca. Spalte 20). Bei Instruktionen mit 2 Operanden werden die Operanden durch Komma und Leerzeichen getrennt.

Der Operand verweist auf ein Label	möglichst in Großbuchstaben (z.B. `TEST`)
Der Operand ist ein Arbeitsregister	Kleinbuchstaben (z.B. `r16`)
Der Operand ist ein I/O-Register	Großschreibung entspr. der Register-Liste (z.B. `PORTB`)
Der Operand ist ein Register-Bit	Großschreibung entspr. der Register-Liste (z.B. `PB4`)
Der Operand ist eine Variable	Großschreibung (z.B. `VAR1`)

7. **Kommentare** zu den Assembler-Zeilen beginnen an der letzten Tabulatorstellung (ab ca. Spalte 30 oder 40).
8. **Kommentare** zu den Assembler-Zeilen sollten so häufig wie möglich eingefügt werden.
9. **Kommentare** am Anfang einer Zeile sollten alle logischen Abschnitte einleiten und beenden.

Die Tabulatorstellungen sollen den "Flattersatz" vermeiden helfen und hängen deshalb primär von der Länge der Ausdrücke ab. Wenn man z.B. lange Ausdrücke wählt, um möglichst prägnante Beziehungen zu den Variablen usw. herstellen zu können, werden die Tabulatoren entsprechend weiter auseinander liegen müssen.

Nicht nur die einzelnen Assembler-Zeilen sollten strukturiert sein, sondern auch die gesamte Programm-Datei. Nachfolgend wird das Muster einer Struktur dargestellt, die neben des geordneten und "immer" wiederkehrenden Aufbaus auch als Checkliste für das Vorhandensein aller logischen Abschnitte dienen soll.

Struktur einer Programm-Datei:

```
;       10        20        30        40
; ******************************************************************************
; *                 Name des Programms; hier: aaa-vorlage.asm                  *
; *                       Beschreibung des Programms                            *
; *                       Beschreibung des Programms                            *
; *                       Beschreibung des Programms                            *
; *                              Funktionen                                     *
; *                               Eingaben                                      *
; *                               Ausgaben                                      *
; * (C)2006 by Alwin J. Lenck                    Letzte Aenderung: 15.11.2007   *
; ******************************************************************************
;
; Hardware-Anforderungen:
;
; Software-Funktionen:
;
.NOLIST                              ; Auflisten der Datei abstellen
.INCLUDE          "m8def.inc"        ; Include-Datei des Chip-Typs einbinden
.LIST                                ; Auflisten wieder anstellen
;
; Konstanten:
```

```
;
.EQU    ZYKLEN    = 12345             ; z.B. Anzahl ZYKLEN
;
; Benutzte Register:
;
.DEF    UniReg1   = r16               ; Register r16 mit Namen UniReg1 versehen
;
; Es folgt der Code des Segmentes
;
.CSEG                           ; Beginn des Segmentes
.ORG    $0000                   ; Segment-Anfangsadresse festlegen
;
; Reset und Interrupt-Vektoren (relative Spruenge) speziell fuer den ATmega8:
;
        rjmp      START         ; PC $0000 Hardware-Reset
        rjmp      EXT_INT0      ; PC $0001 Pegelwechsel am INT0-Pin
        rjmp      EXT_INT1      ; PC $0002 Pegelwechsel am INT1-Pin
        rjmp      TIM2_COMP     ; PC $0003 Timer2 Output-Vergleich
        rjmp      TIM2_OVF      ; PC $0004 Timer2 Ueberlauf
        rjmp      TIM1_CAPT     ; PC $0005 Timer1 Input-Fangschaltung
        rjmp      TIM1_COMPA    ; PC $0006 Timer1 Output-Vergleich A
        rjmp      TIM1_COMPB    ; PC $0007 Timer1 Output-Vergleich B
        rjmp      TIM1_OVF      ; PC $0008 Timer1 Ueberlauf
        rjmp      TIM0_OVF      ; PC $0009 Timer0 Ueberlauf
        rjmp      SPI_STC       ; PC $000A Serielle Uebertragung abgeschlossen
        rjmp      USART_RXC     ; PC $000B USART Zeichen im Empfangspuffer
        rjmp      USART_UDRE    ; PC $000C USART/UDr0 Senderegister ist leer
        rjmp      USART_TXC     ; PC $000D USART Uebertragung abgeschlossen
        rjmp      AD_C          ; PC $000E Analog/Digital-Wandler
        rjmp      EE_RDY        ; PC $000F EEPROM Ready
        rjmp      ANA_COMP      ; PC $0010 Analog-Komparator
        rjmp      TWSI          ; PC $0011 2-Draht-Interface
        rjmp      SPM_RDY       ; PC $0012 Store Program Memory Ready
;
; *** Interrupt-Service-Routines ***********************************************
;
; $0001 Pegelwechsel am INT0-Pin
EXT_INT0:
; [........]
        reti                          ; Ruecksprung vom Interrupt
;
; $0002 Pegelwechsel am INT1-Pin
EXT_INT1:
; [........]
        reti
;
; $0003 Timer2 Output-Vergleich
TIM2_COMP:
; [........]
        reti
;
; $0004 Timer2 Ueberlauf
TIM2_OVF:
; [........]
        reti
;
; $0005 Timer1 Input-Fangschaltung
TIM1_CAPT:
; [........]
        reti
;
; $0006 Timer1 Output-Vergleich A
TIM1_COMPA:
; [........]
        reti
;
; $0007 Timer1 Output-Vergleich B
TIM1_COMPB:
; [........]
        reti
```

```
;
; $0008 Timer1 Ueberlauf
TIM1_OVF:
; [........]
        reti
;
; $0009 Timer0 Ueberlauf
TIM0_OVF:
; [........]
        reti
;
; $000A Serielle Uebertragung abgeschlossen
SPI_STC:
; [........]
        reti
;
; $000B USART Zeichen im Empfangspuffer
USART_RXC:
; [........]
        reti
;
; $000C USART/UDr0 Senderegister ist leer
USART_UDRE:
; [........]
        reti
; $000D USART Uebertragung abgeschlossen
USART_TXC:
; [........]
        reti
;
; $000E Analog/Digital-Wandler
AD_C:
; [........]
        reti
;
; $000F EEPROM Ready
EE_RDY:
; [........]
        reti
;
; $0010 Analog-Komparator
ANA_COMP:
; [........]
        reti
;
; $0011 2-Draht-Interface
TWSI:
; [........]
        reti
;
; $0012 Store Program Memory Ready
SPM_RDY:
; [........]
        reti
;
; ******************************************************************************
; ************************************** Ende der Interrupt-Service-Routinen ***
; ******************************************************************************
;
; ******************************************************************************
; *** Es folgt das Hauptprogramm ************************************************
; ******************************************************************************
START:
        ldi     r16, HIGH(RAMEND)   ; Setze Stack-Pointer auf die
        out     SPH, r16            ; hoechste RAM-Adresse HIGH
        ldi     r16, LOW(RAMEND)    ; Setze Stack-Pointer auf die
        out     SPL, r16            ; hoechste RAM-Adresse LOW
        sei                         ; Setzte das Interrupt-Enable/Disable-Bit
; [........]
; [........]
```

```
; usw.
; [........]
; [........]
ENDE:
; ******************************************************************************
; ************************************************** Ende des Hauptprogramms ***
; ******************************************************************************
;
; ******************************************************************************
; *** Es folgen verschiedene Unterprogramme ***********************************
; ******************************************************************************
; [........]
; [........]
; Hier werden ggf. die Unterprogramme abgelegt
; [........]
; [........]
; ******************************************************************************
; ****************************************** Ende des Unterprogramm-Bereichs ***

; ******************************************************************************
```

2.3 Kommentare

Sehr wichtig bei den Assembler-Programmen sind die Kommentare. Ohne Kommentierung des geschriebenen Codes blickt man schon nach wenigen Tagen oft nicht mehr durch, wofür das Programm gut war oder was an dieser Stelle des Programms eigentlich warum getan wird. Man kann natürlich auch ohne Kommentare Programme schreiben, vor allem wenn man sie vor anderen und vor sich selbst geheim halten will.

Ein Kommentar beginnt mit einem Semikolon. Alles, was danach in dieser Zeile folgt, wird vom Übersetzungs-Programm, dem Assembler, einfach ignoriert. Wenn man mehrere Zeilen lange Kommentare schreiben möchte, muss man eben jede weitere Zeile mit einem Semikolon beginnen. Kommentieren kann und soll man aber auch einzelne Abschnitte eines Programms, wie z.B. eine abgeschlossene Routine oder eine Tabelle. Kommentare über Randbedingungen (z.B. Erklärungen über die verwendeten Register), erwartete Register-Inhalte oder das Ergebnis nach der Bearbeitung eines Teilabschnittes erleichtern das spätere Aufsuchen von Fehlern und vereinfachen nachträgliche Änderungen.

Man kann aber auch einzelne Assembler-Zeilen oder Programmteile, die z.B. für Testzwecke ausgeblendet werden sollen, mit einem Semikolon am Zeilenbeginn vor dem Assembler abschirmen und dahinter alles Mögliche anmerken.

2.4 Kopf des Programms

Den Sinn und Zweck des Programms, sein Autor, der Revisionsstand und andere Kommentare sind leicht als Bestandteil des Kopfes identifizierbar. Weitere Angaben, die hier hin gehören, sind der Prozessortyp, für den die Software geschrieben ist, die wichtigsten Konstanten (zur übersichtlichen Änderung) und die Festlegung von errinnerungsfördernden Registernamen.

Der Prozessortyp hat dabei eine besondere Bedeutung. Programme laufen nicht ohne Änderungen auf jedem Prozessortyp. Nicht alle Prozessoren haben den gleichen Instruktions-Satz, jeder Typ hat seine typische Menge an EEPROM und SRAM, usw. Alle diese Besonderheiten sind in einer besonderen **`Include`**-Datei (Part File xxxdef.inc; z.B. **m8def.inc**) für jeden Prozessortyp festgelegt und werden bei der Assemblierung in den Objektkode eingebunden. Damit der Assembler den richtigen Prozessortyp einbinden kann, wird der entsprechende Code aus der **`Include`**-Datei, die vom Hersteller für jeden Prozessortyp zur Verfügung gestellt wird, folgendermaßen eingelesen:

```
.NOLIST                          ; Damit wird das Auflisten
                                 ; der Datei abgestellt
.INCLUDE    "m8def.inc"          ; Import der Include-Datei
.LIST                            ; Auflisten wieder anschalten
```

Der Pfad, an dem sich die **Include**-Datei befindet, kann weggelassen werden, wenn sie sich im automatisch beim Einrichten von AVR Studio vorgesehenen Verzeichnis befindet:

```
C:\Program Files (x86)\Atmel\AVR Tools\AvrAssembler2\Appnotes
```

Andernfalls ist der Pfad entsprechend den eigenen Verhältnissen anzupassen.

Das Auflisten der **Include**-Datei beim Übersetzen kann nervig sein, weil solche Dateien sehr lang sind und beim Auflisten des übersetzten Codes entsprechend lange Listen (List-Dateien mit der Endung .LST) von meist uninteressanten (weil trivialen) Informationen produzieren. Das Abschalten vor dem Einlesen der **Include**-Datei spart jede Menge Papier beim Ausdrucken der List-Datei.

Es lohnt sich aber trotzdem, einen kurzen Blick in solche **Include**-Datei zu werfen. Zu Beginn der Datei wird mit

```
;***** Specify Device
.DEVICE     ATmega8                     ; Festlegung des Ziel-Devices auf ATmega8
```

der Ziel-Chip definiert. Das wiederum bewirkt, dass Instruktionen, die auf dem Ziel-Chip nicht definiert sind, vom Assembler mit einer Fehlermeldung zurückgewiesen werden. Die **.DEVICE**-Direktive an den Assembler darf also beim Einlesen der Definitions-Datei nicht noch einmal in den Quellcode eingegeben werden (das ergäbe eine Fehlermeldung).

In der **Include**-Datei sind z.B. auch die Register **XH**, **XL**, **YH**, **YL**, **ZH** und **ZL** definiert, die in den Register-Paaren der Pointer **X**, **Y** und **Z** den Zugriff auf die Adressen des Daten-Bereiches ermöglichen. Ferner sind darin alle Port-Speicherstellen definiert, z.B. erfolgt hier die Übersetzung von **PORTB** in hex 18 (**$18**). Schließlich sind hier auch alle Port-Bits mit denjenigen Namen registriert, die im Datenblatt des jeweiligen Chips verwendet werden. So wird hier z.B. das Bit3 vom Port beim Einlesen von Port B als **PINB3** übersetzt, exakt so wie es auch im Datenblatt heißt.

Mit anderen Worten: Vergisst man die Einbindung der **Include**-Datei des AVR-Mikrocontrollers zu Beginn des Programms, dann hagelt es Fehlermeldungen, weil der Assembler nur "Bahnhof" versteht. Die resultierenden Fehlermeldungen sind nicht immer sehr aussagekräftig, weil fehlende Labels und Konstanten vom Assembler nicht mit einer Fehlermeldung quittiert werden. Stattdessen nimmt der Assembler einfach an, die fehlende Konstante sei Null und übersetzt einfach weiter. Man kann sich leicht vorstellen, welches Chaos dabei herauskommt. Der arglose Programmierer denkt: Alles in Ordnung. In Wirklichkeit wird ein ziemlicher Käse im Chip ausgeführt.

In den Kopf des Programms gehören insbesondere auch die Register-Definitionen, also z.B.

```
.DEF     MPR        = r16               ; r16 mit dem Namen MPR belegen
```

Das hat den Vorteil, dass man eine vollständige Liste der Register erhält und sofort sehen kann, welche Register verwendet werden und welche noch frei sind. Das Umbenennen vermeidet nicht nur Verwendungskonflikte, die Namen sind auch aussagekräftiger.

Ferner gehört in den Kopf die Definition von Konstanten, die den gesamten Programmablauf beeinflussen können. So eine Konstante wäre z.B. die Oszillatorfrequenz des Chips, wenn im Programm später die serielle Schnittstelle verwendet werden soll. Mit

```
.EQU     FREQUENZ   = 4000000           ; Quarzfrequenz in Hz festlegen
```

zu Beginn der Assemblerdatei sieht man sofort, für welchen Takt das Programm geschrieben ist. Beim Umschreiben auf eine andere Frequenz muss nur diese Zahl geändert werden und man braucht nicht den gesamten Quelltext nach dem Auftauchen von **4000000** zu durchsuchen.

Programmbeginn

Nach dem Kopf sollte der Programm-Code beginnen. Am Beginn jedes Codes stehen die Reset- und Interrupt-Vektoren. Da diese relative Sprünge enthalten müssen, folgen darauf am besten die Interrupt-Service-Routinen. Danach ist ein guter Platz für das Hauptprogramm gefolgt von abgeschlossenen Unterprogrammen.

Das Hauptprogramm beginnt immer mit dem Einstellen der Register-Startwerte, dem Initialisieren des Stack-Pointers und der verwendeten Hardware. Danach geht es programmspezifisch weiter.

2.5 Assembler-Direktive (Directivs)

Der Assembler bietet mit einer Anzahl von Direktiven die Möglichkeit, zum Beginn des Programms Definitionen festzulegen. Sie werden bei der Übersetzung vom Assembler berücksichtigt, ohne dass sie selbst Objekt-Code erzeugen. Stattdessen werden sie benutzt, um Lokationen im Programm-Speicher festzulegen, Makros zu definieren usw.

Directive	Description
`.BYTE`	Reserve Byte to a Variable
`.CSEG`	Code Segment
`.CSEGSIZE`	Program Memory Size
`.DB`	Define Constant Byte(s)
`.DEF`	Define a Symbolic Name on a Register
`.DEVICE`	Define which Device to Assemble for
`.DSEG`	Data Segment
`.DW`	Define Constant Word(s)
`.ENDM, ENDMACRO`	EndMacro
`.EQU`	Set a symbol equal to an expression
`.ESEG`	EEPROM Segment
`.EXIT`	Exit from file
`.INCLUDE`	Read source from another file
`.LIST`	Turn listfile generation on
`.LISTMAC`	Turn Macro expansion in list file on
`.MACRO`	Begin Macro
`.NOLIST`	Turn listfile generation off
`.ORG`	Set program origin
`.SET`	Set a symbol to an expression
New Directives	**Description**
`.ELSE, .ELIF`	Conditional assembly
`.ENDIF`	Conditional assembly
`.ERROR`	Outputs an error message
`.IF, .IFDEF, .IFNDEF`	Conditional assembly
`.MESSAGE`	Outputs a message string

Die folgenden Direktiven sind nur für AVRASM2 verfügbar:

AVRASM2 Directives	Description
`.DD`	Define Doubleword
`.DQ`	Define Quadword
`.UNDEF`	Undefine register symbol
`.WARNING`	Outputs a warning message
`.OVERLAP/NOOVERLAP`	Set up overlapping section (requires AVRASM2.1 or better)

Man beachte, dass die Direktiven mit einem vorangehenden Punkt eingeleitet werden müssen!

Beispiel:

Wie bekommt man konstante Werte in den Programm-Speicher? Dazu gibt es die **.DB**- und **.DW**-Direktiven für den Assembler. Byteweise Listen werden so erzeugt:

```
.DB          123, 45, 67, 78                ; eine Liste mit 4 Byte
.DB          "Das ist ein Text. "           ; eine Liste mit einem Text
```

Auf jeden Fall ist darauf zu achten, dass die Anzahl der einzufügenden Bytes pro Zeile geradzahlig ist. Sonst fügt der Assembler ein Null-Byte am Ende hinzu, das vielleicht gar nicht erwünscht ist. Das Problem gibt es bei wortweise organisierten Tabellen nicht. Die sehen so aus:

```
.DW          12345, 6789                    ; zwei Worte
```

Statt der Konstanten können selbstverständlich auch Labels (Sprungadressen) eingefügt werden, also z.B. so:

```
LABEL1:
[... hier kommen irgendwelche Instruktionen...]
LABEL2:
[... hier kommen noch irgendwelche Instruktionen...]
Sprungtabelle:
.DW          LABEL1,LABEL2                  ; zwei Labelpositionen
```

2.6 Assembler-Ausdrücke (Expressions)

Der Assembler verwendet konstante Ausdrücke, die aus Operanden, Operatoren und Funktionen bestehen können. Alle Ausdrücke belegen intern im Assembler AVRASM 32 Bit und 64 Bit im AVRASM2 (für Instruktions-Zähler und Instruktion).

2.6.1 Operanden

- Benutzerdefinierte Labels, die mit dem Platz an dem sie stehen die Adresse im Programm-Memory-Location-Counter (PC) repräsentieren.
- Variable, vom Benutzer mit der **.SET**-Direktive definiert
- Konstante, vom Benutzer mit der **.EQU**-Direktive definiert
- Integer-Konstante in verschiedenen Formaten:
  - o Dezimal: **10**, **255**
  - o Hexadezimal (2 Notationen): **0x0a**, **$0a**, **0xff**, **$ff**, **$00FF**
  - o Binär: **0b00001010**, **0b11111111**
  - o Oktal (mit führender Null): **010**, **077**
- PC - der laufende Wert des **P**rogram-Memory-Location-**C**ounters
- Gleitkomma-Konstante - nur im Assembler AVRASM2 nutzbar.

2.6.2 Operatoren

Der Assembler ermöglicht einige hier dargestellte Schreibweisen für Operatoren. Die Ausdrücke können durch Klammern eingeschlossen werden, diese werden zunächst ausgeführt, bevor die Ausdrücke außerhalb der Klammer bearbeitet werden. Bei binären Operatoren (z.B. Rechenoperatoren wie *** / + -**) ist auf die genaue Reihenfolge zu achten: Linke Assoziativität bedeutet Interpretation von Links nach Rechts, z.B. **2-3-4** bedeutet **(2-3)-4**, während rechte Assoziativität die Folge **2-3-4** zu **2-(3-4)** interpretiert. Viele Operatoren sind nicht assoziativ, ihre Reihenfolge ist mithin bedeutungslos.

Symbol	Description	Beispiel
!	Logische Verneinung	`ldi r16, !0xF0  ; 0x00 nach r16`
~	Bitweise Verneinung	`ldi r16, ~0xF0  ; 0x0F nach r16`

-	Minuszeichen	`ldi r16, -2 ; -2 d.h. 0xFE nach r16`
*	Multiplikation	`ldi r30, MULTI*2 ; MULTI*2 nach r30`
/	Division	`ldi r30, DIVI/2 ; DIVI/2 nach r30`
%	Modulo	nur im Assembler AVRASM2 nutzbar
+	Addition	`ldi r30, A1+A2 ; A1+A2 nach r30`
-	Subtraktion	`ldi r17, S1-S2 ; S1-S2 nach r17`
<<	Links-Shift	`ldi r17, 1<<ANZ ; 1 um ANZ nach links`
>>	Rechts-Shift	`ldi r17, V1>>ANZ ; V1 um ANZ nach rechts`
<	Kleiner als	`ori r18, BITMASKE*(C1<C2)+1`
<=	Kleiner oder gleich als	`ori r18, BITMASKE*(C1<=C2)+1`
>	Größer als	`ori r18, BITMASKE*(C1>C2)+1`
>=	Größer oder gleich als	`ori r18, BITMASKE*(C1>=C2)+1`
==	Gleich	`andi r19, BITMASKE*(C1==C2)+1`
!=	Ungleich	`.SET FLAG =(C1!=C2) ; Setze FLAG auf 1 oder 0`
&	Bitweises UND	`ldi r18, HIGH(C1&C2) ; Lädt r18 mit einem Ausdr.`
^	Bitweises exkl. ODER	`ldi r18, LOW(C1^C2) ; Lädt r18 mit einem Ausdr.`
|	Bitweises ODER	`ldi r18, LOW(C1|C2) ; Lädt r18 mit einem Ausdr.`
&&	Logisches UND	`ldi r18, LOW(C1&&C2) ; Lädt r18 mit einem Ausdr.`
||	Logisches ODER	`ldi r18, LOW(C1||C2) ; Lädt r18 mit einem Ausdr.`
?	Conditional Operator	nur im Assembler AVRASM2 nutzbar

2.6.3 Funktionen

In jedem Assembler gibt es feststehende Begriffe, d.h. Zeichenfolgen, die nicht anderweitig - z.B. als freie Variable usw. - benutzt werden dürfen. Dazu gehören vor allem die mnemotechnischen Abkürzungen der Instruktionen aber auch die folgenden **Funktionen**. Sie sind ausschließlich für den Assembler reserviert:

- **`LOW(expression)`** liefert das niedere Byte der **`expression`**
- **`HIGH(expression)`** liefert das höhere Byte der **`expression`**
- **`BYTE2(expression)`** liefert das 2. Byte der **`expression`**; identisch mit **`HIGH`**
- **`BYTE3(expression)`** liefert das 3. Byte der **`expression`**
- **`BYTE4(expression)`** liefert das 4. Byte der **`expression`**
- **`LWRD(expression)`** liefert die Bit0 bis Bit15 (15:0) der **`expression`**
- **`HWRD(expression)`** liefert die Bit16 bis Bit31 (31:16)der **`expression`**
- **`PAGE(expression)`** liefert Bit16 bis Bit21 (21:16) der **`expression`**
- **`EXP2(expression)`** liefert den Exponenten 2 zur **`expression`**
- **`LOG2(expression)`** liefert den Integer-Teil von **`log2(expression)`**

2.6.4 Operationen

Um die Auswirkungen einer Instruktion besser verstehen zu können, werden im Handbuch **8-Bit AVR Instruction Set** bei der Beschreibung der einzelnen Instruktionen die jeweils verursachte Operation (ggf. auch in mehreren Schritten) beschrieben. Leider wird dem Laien dabei nicht immer klar, ob es sich bei Operatoren um Inhalte von Adressen oder den Adressen selbst handelt. Deshalb wurde hier folgende Regel für die Beschreibung der Operationen und der Kommentare aufgestellt:

Hinweis:

Wenn Register oder Adressen bei den Operationen in geschweiften Klammern **`{Operand}`** gesetzt sind, bezieht sich die Angabe in den Operationen stets auf den Inhalt des Registers bzw. der Adresse, andernfalls ist stets die Register-Nummer bzw. die Adresse selbst gemeint. Wenn es sich sogar um den Inhalt einer Adresse, die in einem bestimmten Register steht, handelt, wird der "Inhalt des Inhalts der Register-Adresse" mit zwei geschweiften Klammern dargestellt. Zum Beispiel ist mit dem Ausdruck **`{{Z}}`** das Daten-Byte gemeint, welches sich in der Adresse befindet, die im Zeiger **`Z`** (Register-Paar

r31:r30) steht. Ein Doppelpunkt/Gleichheitszeichen **:=** bedeutet "ergibt sich als", "wird gesetzt auf" oder "wird ersetzt durch" und ein allein stehendes Gleichheitszeichen **=** bedeutet "ist gleich".

Beispiel 1: `; {{Z}} := {UniReg}, {Z} := {Z} + 1`

Lesart: Der Inhalt der Adresse, auf die **Z** verweist, wird durch den Inhalt, der in dem durch **UniReg** repräsentierten Register steht, ersetzt. Wie später noch erläutert wird, ist **Z** ein "Zeiger", der im Prinzip auch eine Adresse darstellt.

Beispiel 2: `; Wenn {DUAL1H}={DUAL2H} dann {Z-Bit} := 1`

Lesart: Wenn der Inhalt der durch **DUAL1H** repräsentierten variablen Zelle gleich dem Inhalt der durch **DUAL2H** repräsentierten variablen Zelle ist, dann wird das **Z-Bit** auf **1** gesetzt.

2.7 Daten- und Adress-Struktur

Die Daten- und Adressstruktur und die Methoden, um auf die verschiedenen Speicher zuzugreifen, sind der Dreh- und Angelpunkt beim Verständnis der AVRs. Weiter unten sind deshalb alle von der AVR-Architektur unterstützten Adressierungs-Modi aufgeführt, um auf den Programm-Speicher, den separaten Daten-Speicher (Arbeits-Register, I/O-Memory, SRAM und ggf. erweiterter SRAM) sowie auf das ebenfalls separate EEPROM zuzugreifen.

2.7.1 Nomenklatur: Abkürzungen für Register und Operanden

Rd	Ziel- (und Quell-)Arbeits-Register in der Register-Datei
Rr	Quell-Arbeits-Register in der Register-Datei
R	Resultat (Ergebnis) nach der Instruktionsausführung
K	Konstante allgemein
P	Konstante (0 bis 31 bzw. 63)
K6	Konstante (0 bis 63)
K8	Konstante (0 bis 255)
k	2 Byte lange Adresse im Daten-Speicher
b	Konstante (0 bis 7); Bit-Nummer im Arbeits-Register oder I/O-Register (3 Bit)
s	Konstante (0 bis 7); Bit-Nummer im Status-Register (3 Bit)
X, Y, Z	Indirekt-Adress-Register (Pointer-Register **X** steht für das Register-Paar **r27:r26**; Pointer-Register **Y** steht für das Register-Paar **r29:r28** und Pointer-Register **Z** steht für das Register-Paar **r31:r30**)
RdH:RdL	**r25:r24**, **r27:r26**, **r29:r28**, **r31:r30**. Register-Paare für die Instruktionen **adiw** und **sbiw**
A	Adresse eines I/O-Registers (Adressen von **$00** bis **$3F**)
q	Ersetzung für direkte Adressierung (6 Bit)
PC	Program-Counter (Programm-Zähler = Instruktions-Zähler = Wortzähler)

2.7.2 AVR-Speicher-Räume aus Assembler-Sicht

Zunächst: Externer Speicher, d.h. extern an einen AVR angekoppelter SRAM, soll hier nicht behandelt werden - auch wenn er gelegentlich kurz angesprochen wird. Die verschiedenen internen Speicher der AVRs sind eingeteilt in (alle Angaben beziehen sich primär auf den ATmega8):

- Programm-Speicher (Flash) — eigener Adress-Raum von **$0000** bis **$0FFF** für 4 K Worte
- Daten-Speicher — eigener Adress-Raum von **$0000** bis **$045F** aufgeteilt nach
  - Arbeits-Register — von **$0000** bis **$001F** für 32 Register
  - I/O-Port — von **$0020** bis **$005F** für die 64 Ports
  - SRAM — von **$0060** bis **$045F** für 1024 Byte internem SRAM
- EEPROM — eigener Adress-Raum von **$0000** bis **$01FF** für 512 Byte

3.7.2.1 Programm-Speicher

Der Programm-Speicher (**I**n-**S**ystem Self-**P**rogrammable Flash Program Memory) ist physikalisch ausgeführt als Flash-Speicher, d.h. er kann bis zu 10.000 mal neu beschrieben werden. Er ist in zwei Sektionen aufgeteilt: Der Applikations-Sektion, die uns an dieser Stelle interessieren soll und der Boot-Loader-Sektion, die für das Laden des Programms in die Applikations-Sektion zuständig ist. Geladen wird das Programm im Normalfall mittels eines Computers und seiner Computer-Schnittstelle (z.B. USB) über die **ISP**-Schnittstelle (**I**n-**S**ystem-**P**rogrammierbarkeit) des AVR.

In der Applikations-Sektion des Programm-Speichers werden die Instruktionen abgelegt und vom Program-Counter (PC = Instruktions-Zähler) bei der Verarbeitung abgerufen. Der PC ist 12 Bit lang, um die 4.096 Lokationen (Wort-Adressen für je 2 Instruktions-Byte) im Programm-Speicher des ATmega8 adressieren zu können. Die Instruktionen **`lpm`** und **`spm`**, die als einzige Instruktionen auf den Programm-Adress-Raum zugreifen können, betrachten dagegen den Speicher auf Byte-Ebene, also als einen 8 kByte großen Speicher mit den Adressen **`$0000`** bis **`$1FFF`**.

Die Bit-Anzahl für die Adressierung der Instruktionen im Flash muss also ebenfalls mindestens 12 Bit umfassen. Weiter ist zu beachten, dass es 2-Wort-Instruktionen (nur **`lds`** und **`sds`**) und 1-Wort-Instruktionen (alle übrigen) gibt; sie umfassen mithin 32 Bit oder 16 Bit (4 oder 2 Byte). Der als Wort-Zähler ausgebildete PC wird nach der Abarbeitung einer Instruktion (ausgenommen sind die Branch-Instruktionen) entsprechend um 2 bzw. 1 erhöht.

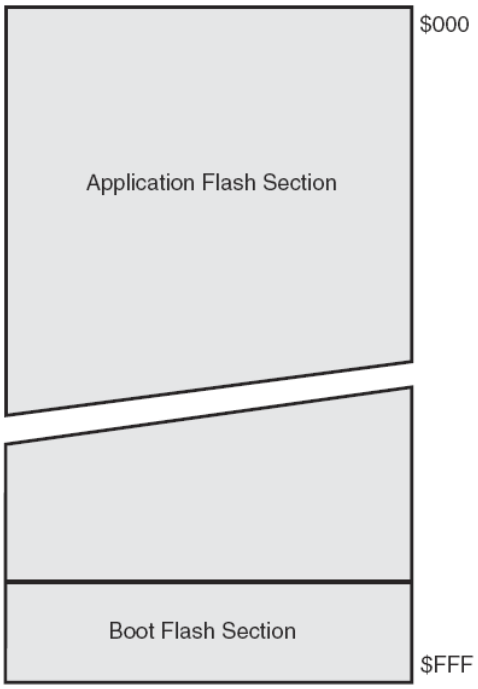


Bild 2.7.2.1 - 01 Flash-Speicher

2.7.2.2 Daten-Speicher

Der Daten-Speicher, der den gesamten I/O-Daten-Adress-Raum (Data Address Space) umfasst, beinhaltet die 32 Arbeits-Register **`r0`** bis **`r31`**, die 64 I/O-Register für periphere CPU-Funktionen und das interne statische RAM (SRAM). Die I/O-Register können direkt mit ihrer Register-Nummer (von **`$00`** bis **`$3F`**) oder mit der Lokation im Daten-Adress-Raum, der nach den Arbeits-Registern (von **`$0020`** bis **`$005F`**) folgt, angesprochen werden.

Beim ATmega8 ist das SRAM 1 kByte groß und umfasst den I/O-Daten-Adress-Raum von Adresse **`$0060`** bis **`$045F`**.

Merke: Es ist zu beachten, dass eine Adresse für eine Lokation im Daten-Adress-Raum 16 Bit bzw. 2 Byte lang ist, während die Adresse für ein Arbeits-Registers in der Register-Datei (Register-File) mit nur

5 Bit und die logische Adresse eines Ports (I/O-Register in der I/O-Memory) ggf. mit 6 Bit auskommen muss.

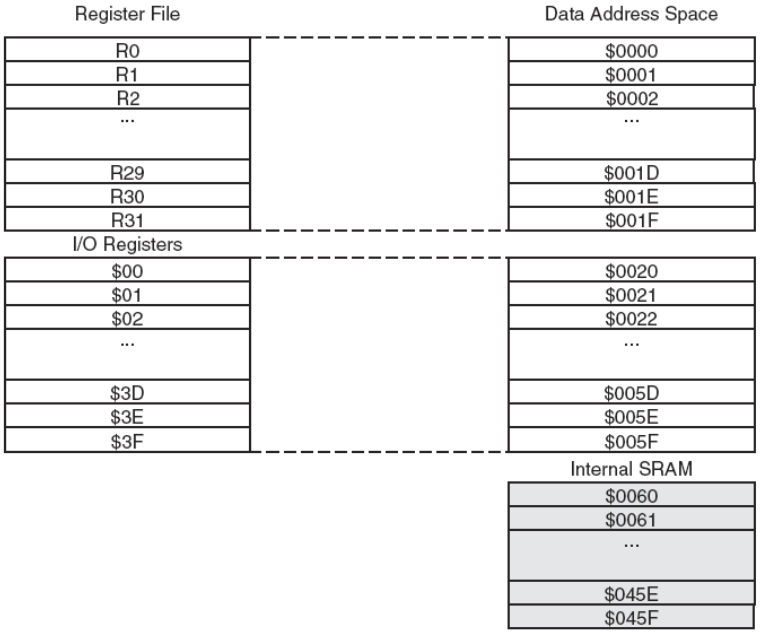


Bild 2.7.2.2 - 01 Daten-Speicher

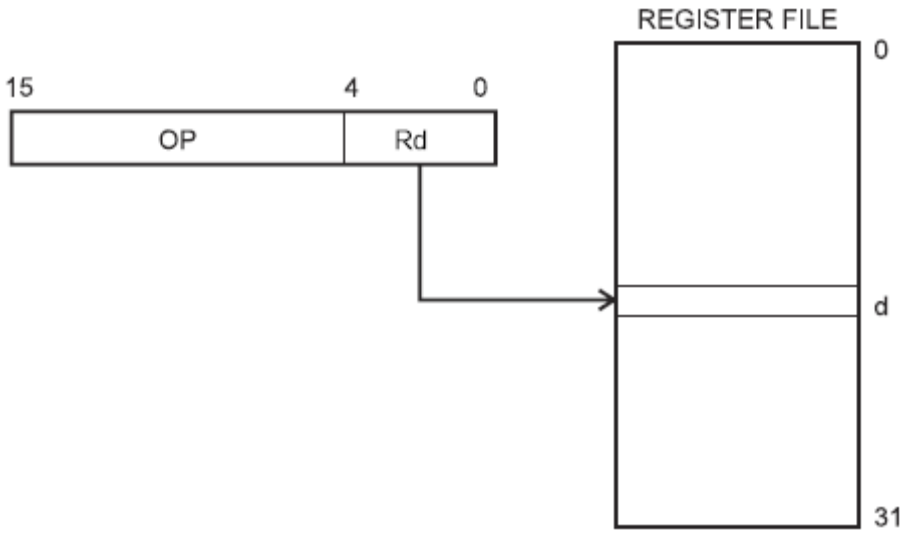
2.7.2.3 EEPROM-Speicher

Das EEPROM ist ebenfalls ein separater Datenspeicher, in dem einzelne Bytes geschrieben und gelesen werden können. Seine Haltbarkeit wird mit bis zu 100.000 Schrei/Lese-Zyklen angegeben. Das EEPROM stellt einen eigenen vom übrigen Speicher unabhängigen Adress-Raum dar. Die einzelnen Bytes werden über das EEPROM-Daten-Register EEDR geschrieben und gelesen; die jeweilige Lese-/Schreibadresse muss sich im EEPROM-Adress-Register EEARH/EEARL befinden.

2.7.3 Adress-Modi

In den folgenden Abbildungen steht **OP** für den Teil der Instruktion, der den Operationscode enthält. Zur exakten Ausrichtung der Adressierungs- und Daten-Bits ist die Darstellung in den Beispielen heranzuziehen. Die abstrakten Ausdrücke **RAMEND** und **FLASHEND** repräsentieren jeweils die höchste absolute Adresse im Daten- bzw. im Programm-Speicher. **Beachte:** Nicht alle Adressierungs-Modi werden auch von allen AVR-Mikrocontrollern unterstützt.

2.7.3.1 Direkte Adressierung eines einzelnen Arbeits-Registers

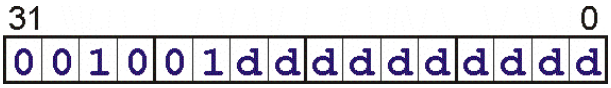


Der Operand **Rd** bezieht sich auf das Register mit der Nummer d. Da **Rd** maximal die Adresse 31 = $1F (11111) annehmen kann, sind die 5 höheren **d**-Bits nicht genutzt.

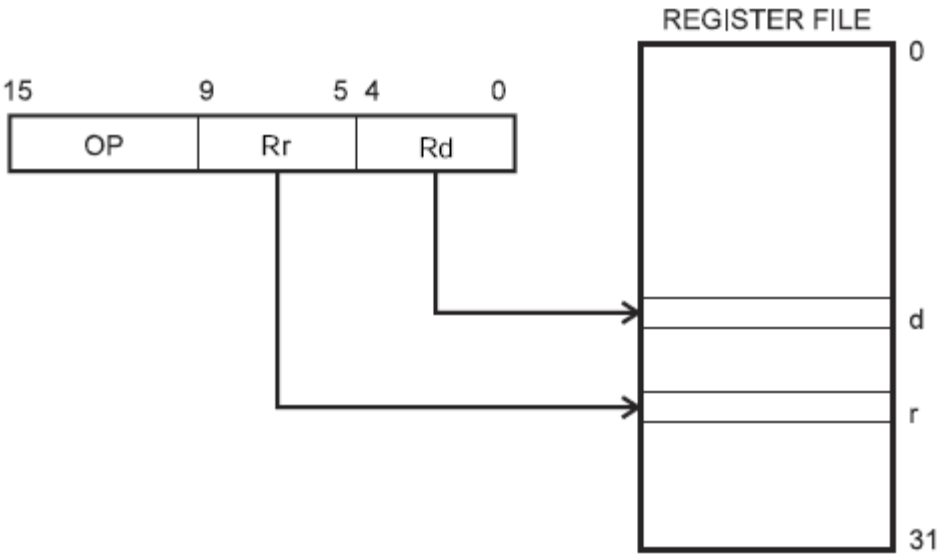
Beispiel:

Clear Register

clr Rd Operation: {Rd} := {Rd} ρ {Rd} (exklusives ODER)



2.7.3.2 Direkte Adressierung von 2 Arbeits-Registern



Die Operanden **Rr** und **Rd** beziehen sich auf die Register mit den Nummern r und d. Das Ergebnis wird im Register **Rd** mit der Nummer d abgelegt. Man beachte die Verteilung der **r**- und **d**-Bits im Instruktionswort.

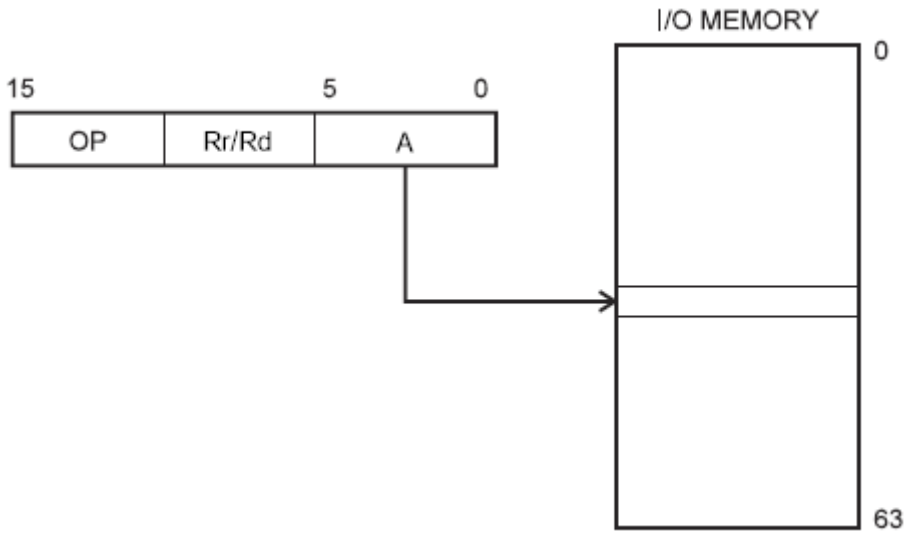
Beispiel:

Add Without Carry

add Rd, Rr Operation: {Rd} := {Rd} + {Rr}

31															0
0	0	0	0	1	1	r	d	d	d	d	d	r	r	r	r

2.7.3.3 Direkte Adressierung eines I/O-Registers



Die Adresse des I/O-Registers befindet sich in den verstreuten 6 Bit, die mit **A** im Instruktionswort gekennzeichnet sind.

Beispiele:

1.) Load Data From an I/O Location to a Register

`in Rd, A` Operation: {Rd} := {A}

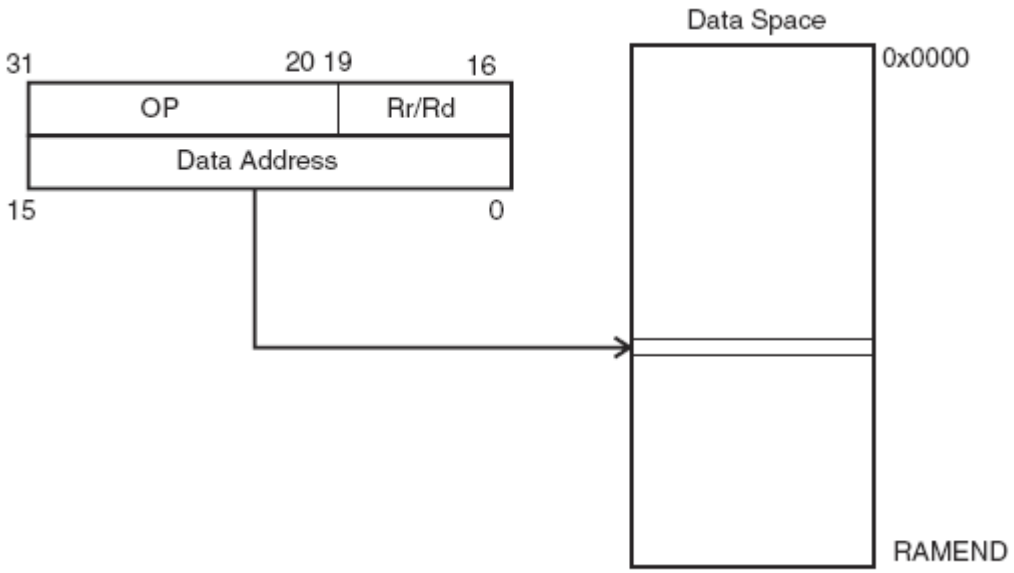
31															0
1	0	1	1	0	A	A	d	d	d	d	d	A	A	A	A

2.) Store Data From a Register to an I/O Location

`out A, Rr` Operation: {A} := {Rr}

31															0
1	0	1	1	1	A	A	r	r	r	r	r	A	A	A	A

2.7.3.4 Direkte Daten-Adressierung im SRAM



Die 16 Bit der Daten-Adresse befinden sich in den 16 niederen **k**-Bits (LSB) der 2-Wort-Instruktion. **Rd** bzw. **Rr** kennzeichnen das Ziel- bzw. das Quell-Register.

Beispiele:

1.) Load Direct From Data Space to Register

lds Rd, k Operation: {Rd} := {k}

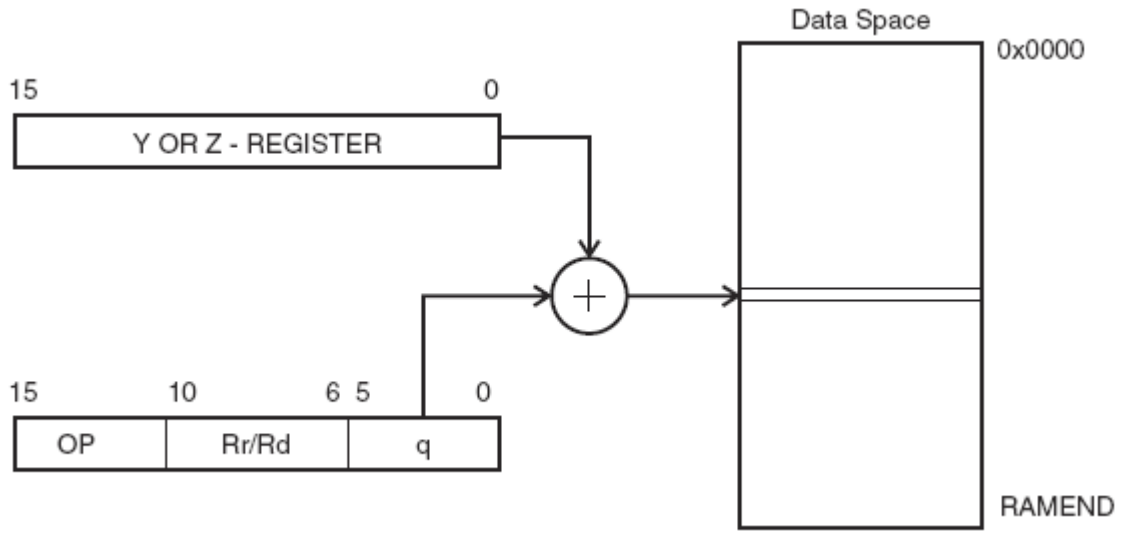
63															32
1	0	0	1	0	0	0	d	d	d	d	d	0	0	0	0
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k
31															0

2.) Store From Register Direct to Data Space

sts k, Rr Operation: {k} := {Rr}

63															32
1	0	0	1	0	0	1	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
31															0

2.7.3.5 Indirekte Daten-Adressierung mit Displacement



Beispiele:

1.) Load Indirect with Displacement

`ldd Rd, Y+q` Operation: {Rd} := {{Y}+q}

31															0
1	0	q	0	q	q	0	d	d	d	d	d	0	q	q	q

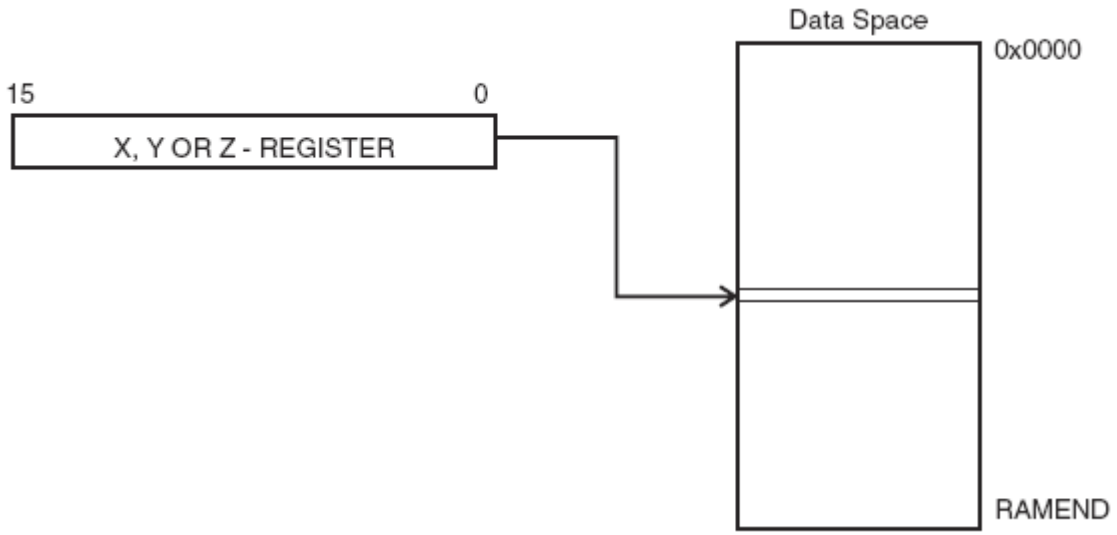
2.) Store Indirect with Displacement

`std Y+q, Rr` Operation: {{Y}+q} := {Rr}

31															0
1	0	q	0	q	q	1	r	r	r	r	r	0	q	q	q

Die Daten-Adresse ergibt sich aus der Summe des Inhalts aus dem Y- (bzw. Z-) Register und dem Wert der im Instruktionswort verteilten 6 **q**-Bit.

2.7.3.6 Indirekte Daten-Adressierung



Die Daten-Adresse steht im X-, Y- bzw. Z-Register (2 Byte lange Adresse). **1.)** Das Daten-Byte, das an dieser Adresse abgespeichert ist, wird in das Register **`Rd`** geladen (**`ld`**) oder **2.)** der Inhalt des Registers **`Rr`** wird in der angegebenen Daten-Adresse abgelegt (**`st`**).

Beispiele:

1.) Load Indirect From Data Space to Register using Index X

`ld Rd, X` Operation: {Rd} := {{X}}

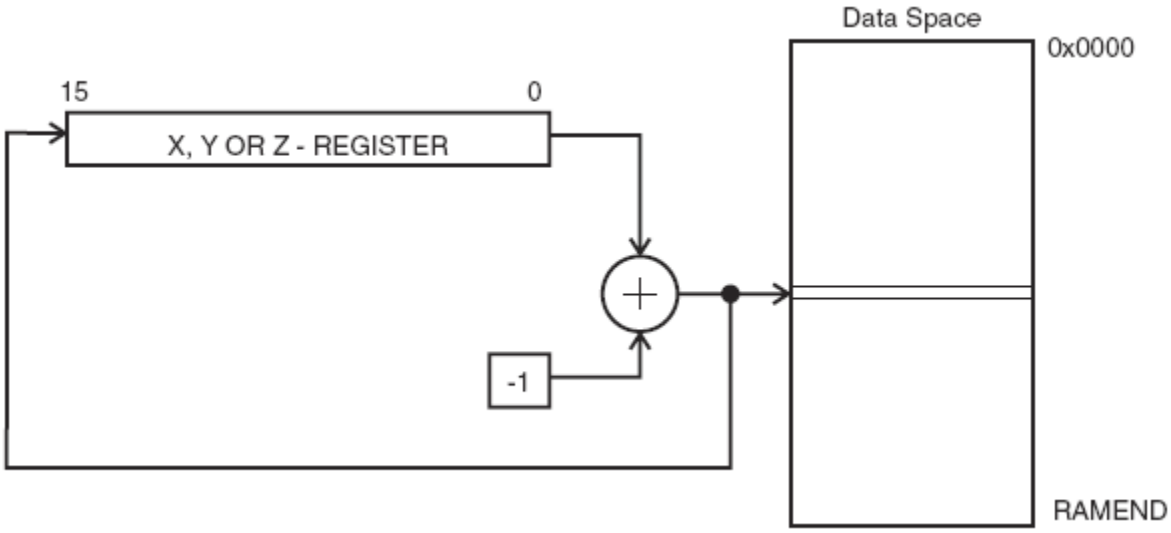
31															0
1	0	0	1	0	0	0	d	d	d	d	d	1	1	0	0

2.) Store Indirect From Register to Data Space using Index X

`st X, Rr` Operation: {{X}} := {Rr}

31															0
1	0	0	1	0	0	1	r	r	r	r	r	1	1	0	0

2.7.3.7 Indirekte Daten-Adressierung mit Pre-Decrement



X, Y- oder Z-Register wird **vor** der Operation um 1 reduziert

Beispiele:

1.) Load Indirect From Data Space to Register Using Index X With Pre-Decrement

`ld Rd, -X` Operationen: {X} := {X} - 1 {Rd} := {{X}}

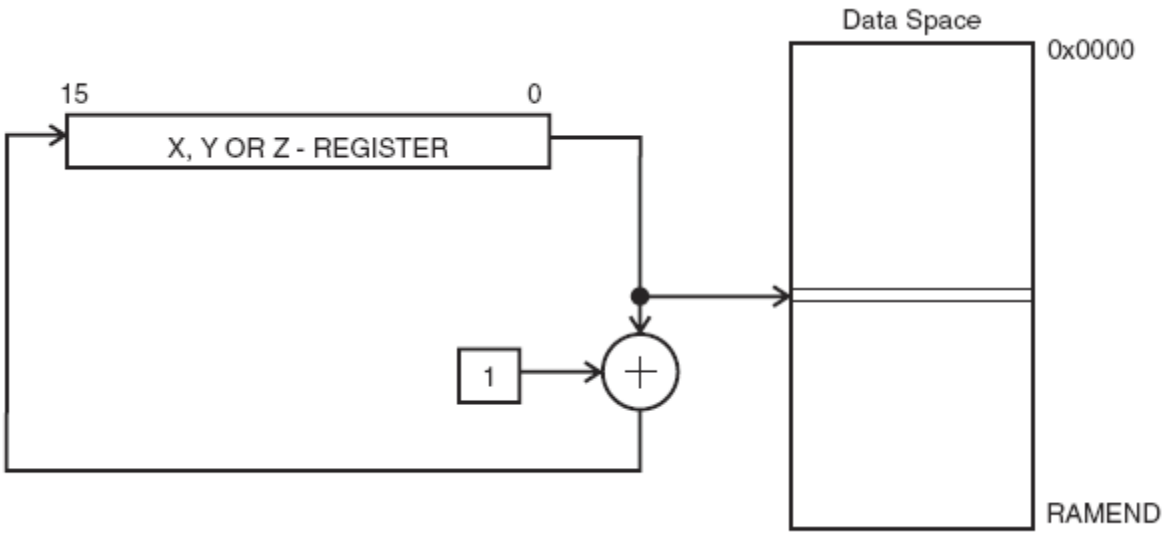
31															0
1	0	0	1	0	0	0	d	d	d	d	d	1	1	1	0

2.) Store Indirect From Register to Data Space Using Index X With Pre-Decrement

`st -X, Rr` Operationen: {X} := {X} - 1 {{X}} := {Rr}

31															0
1	0	0	1	0	0	1	r	r	r	r	r	1	1	1	0

2.7.3.8 Indirekte Daten-Adressierung mit Post-Decrement



X, Y- oder Z-Register wird **nach** der Operation um 1 erhöht

Beispiele:

1.) Load Indirect From Data Space to Register using Index X With Post-Increment

`ld Rd, X+` Operationen: {Rd} := {{X}} {X} := {X} + 1

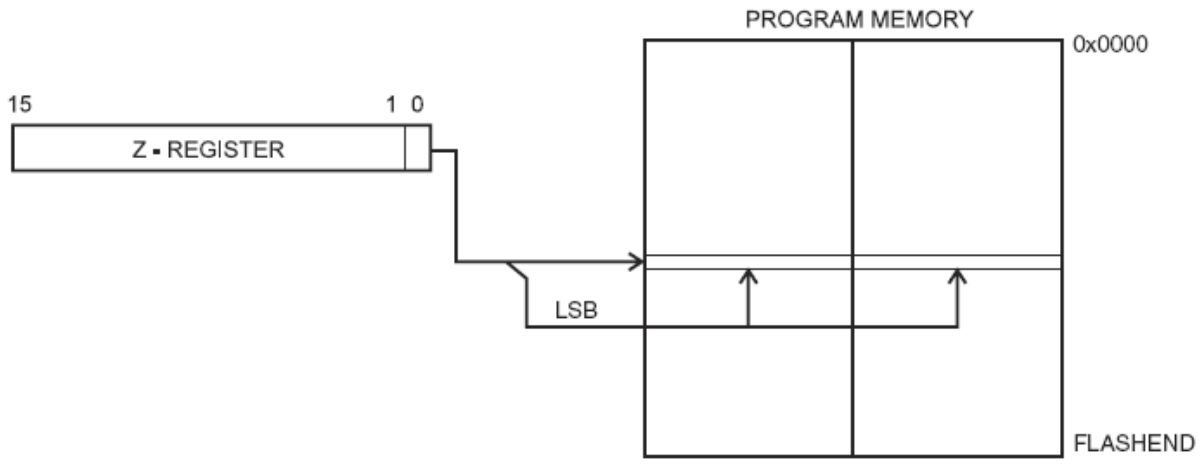
31															0
1	0	0	1	0	0	0	d	d	d	d	d	1	1	0	1

2.) Store Indirect From Register to Data Space using Index X With Post-Increment

`st X+, Rr` Operationen: {{X}} := {Rr} {X} := {X} + 1

31															0
1	0	0	1	0	0	1	r	r	r	r	r	1	1	0	1

2.7.3.9 Adressierung des Programm-Speichers



Die 15 MSBs im Z-Register bestimmen die Wort-Adresse im Programm-Speicher. Es wird nur ein Byte der 2-Byte-Instruktion im Ziel-Register **`r0`** bzw. **`Rd`** geladen. Das LSB bestimmt den oberen (LSB = 1) bzw. den unteren (LSB = 0) Adressteil im Programm-Speicher.

Beispiele:

1.) Load Program Memory (ohne Operand wird **`r0`** unterstellt)

`lpm` Operation: {r0} := {{Z}}

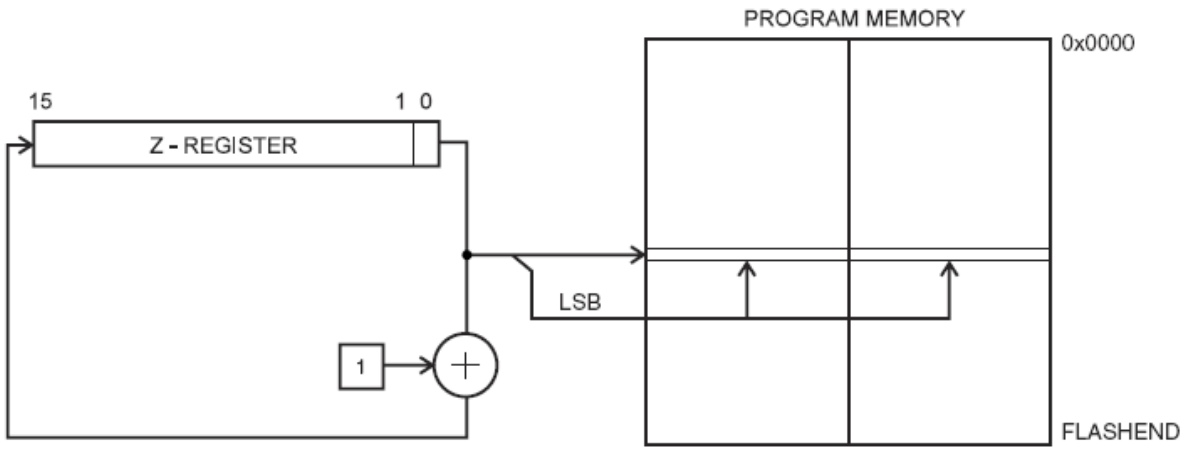
31															0
1	0	0	1	0	1	0	1	1	1	0	0	1	0	0	0

2.) Load Program Memory

`lpm Rd, Z` Operation: {Rd} := {{Z}}

31															0
1	0	0	1	0	0	0	d	d	d	d	d	0	1	0	0

2.7.3.10 Adressierung des Programm-Speichers mit Post-Increment



Beispiele:

1.) Load Program Memory and Post-Increment

`lpm Rd, Z+` Operation: {Rd} := {{Z}} {Z} := {Z} + 1

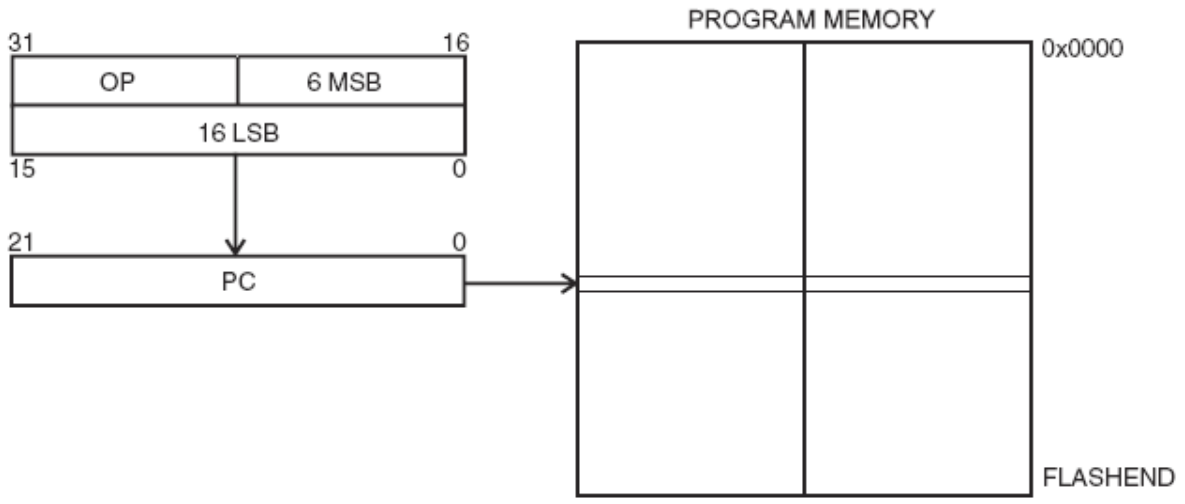
31															0
1	0	0	1	0	0	0	d	d	d	d	d	0	1	0	1

2.) Extended Load Program Memory and Post-Increment (**nur bei Extended SRAM**)

`elpm Rd, Z+` Operationen: {Rd} := {{RAMPZ:Z}} {RAMPZ:Z} := {RAMPZ:Z} + 1

31															0
1	0	0	1	0	0	0	d	d	d	d	d	0	1	1	1

2.7.3.11 Direkte Adressierung des Programm-Speichers (0 < k < 4 MB)



Beispiele:

1.) Jump

`jmp k` Operation: {PC} := k

63															32
1	0	0	1	0	1	0	k	k	k	k	k	1	1	0	k
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k
31															0

2.) Call Subroutine

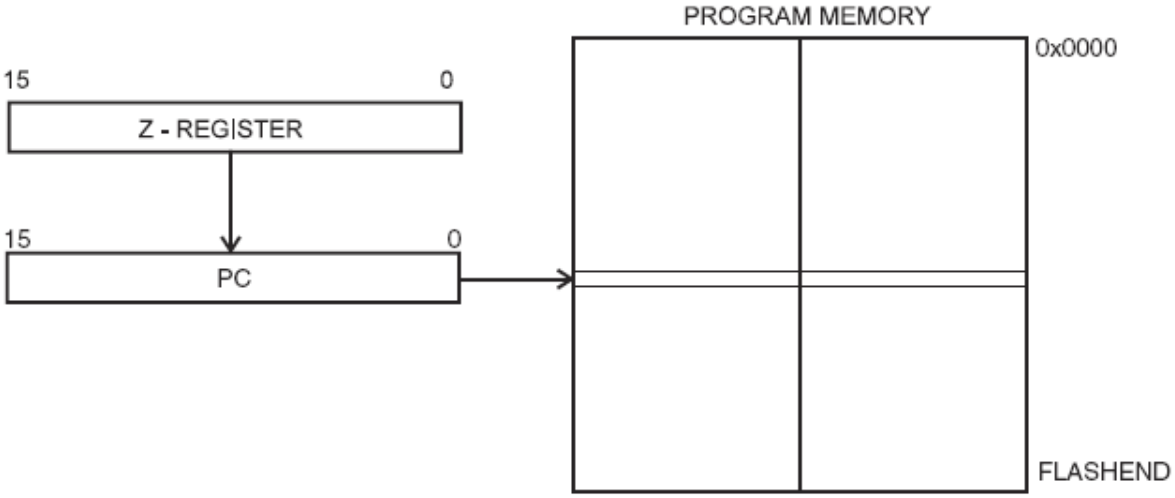
`call k` Operationen: {PC} := k

Stack {STACK} := {PC} + 2

Stack-Pointer {SP} := {SP} - 2

63															32
1	0	0	1	0	1	0	k	k	k	k	k	1	1	1	k
k	k	k	k	k	k	k	k	k	k	k	k	k	k	k	k
31															0

2.7.3.12 Indirekte Adressierung des Programm-Speichers (0 < k < 4 MB)



Beispiele:

1.) Jump

`ijmp` Operation: {PC} := {Z}

31															0
1	0	0	1	0	1	0	0	0	0	0	0	1	0	0	1

2.) Call Subroutine

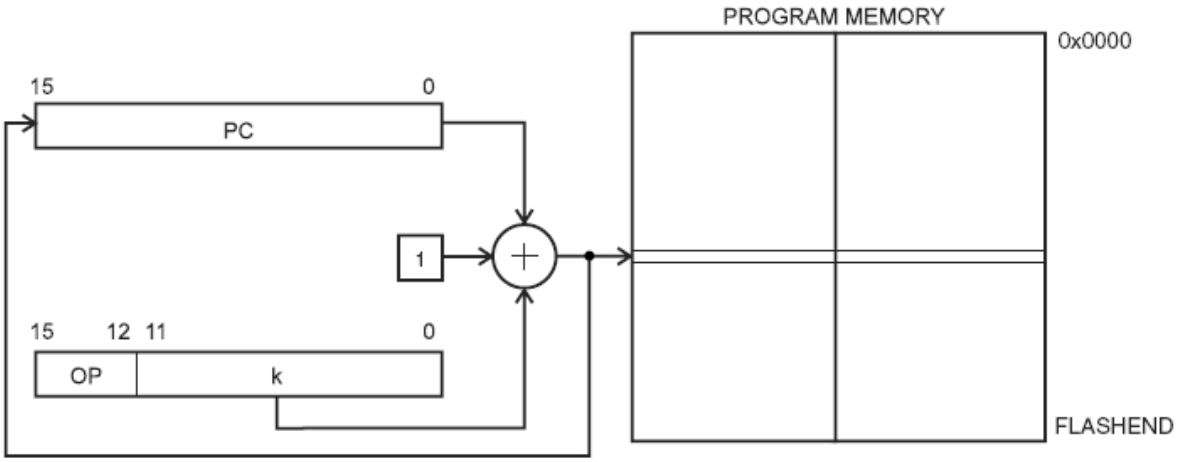
`icall` Operationen: {PC} := {Z}

Stack {STACK} := {PC} + 1

Stack-Pointer {SP} := {SP} - 2

31															0
1	0	0	1	0	1	0	1	0	0	0	0	1	0	0	1

2.7.3.13 Relative Adressierung des Programm-Speichers



Beispiele:

1.) Relative Jump

`rjmp k` Operation: {PC} := {PC} + k + 1

31															0
1	1	0	0	k	k	k	k	k	k	k	k	k	k	k	k

2.) Relative Call Subroutine

`rcall k` Operationen: {PC} := {PC} + k + 1

Stack {STACK} := {PC} + 1

Stack-Pointer {SP} := {SP} - 2

31															0
1	1	0	1	k	k	k	k	k	k	k	k	k	k	k	k