

Ausarbeitung

Core und Speicher des ATmega16

Verfasst von: Daniel Dünker

Quellen: http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf

Inhaltsverzeichnis

1. Allgemeines (S. 3)
2. Die Alu (S. 4)
3. Das Statusregister (S. 4-6)
4. Der Stack (S. 7)
5. Interrupts (S. 8)
6. Die Register (S. 9)
7. Der Speicher
 - Flash Speicher (S. 10)
 - SRAM (S. 11)
 - EEPROM (S. 12)
 - Register (S. 12-13)
 - Schreibvorgang (S. 14)

1. Allgemeines

Die Harvard Architektur

Der Mikrocontroller verfügt über eine so genannte Harvard-Architektur, was bedeutet, dass der Speicher für den Programmcode und der Arbeitsspeicher getrennt sind. Dadurch entstehen deutliche Vorteile, wie die Tatsache, dass ein Programm sich nicht selber verändern kann und Programmcode und Daten gleichzeitig gelesen werden können.

Ein anderer Vorteil, der bei dem Controller auch genutzt wird ist die Möglichkeit, die Befehlswortbreite und Datenwortbreite voneinander unabhängig zu gestalten.

Allerdings ist man durch diese Aufteilung des Speichers in seiner Nutzung etwas eingeschränkter, was aber keineswegs so viel gewichtet wird wie die oben genannten Vorteile.

Die Arbeitsregister

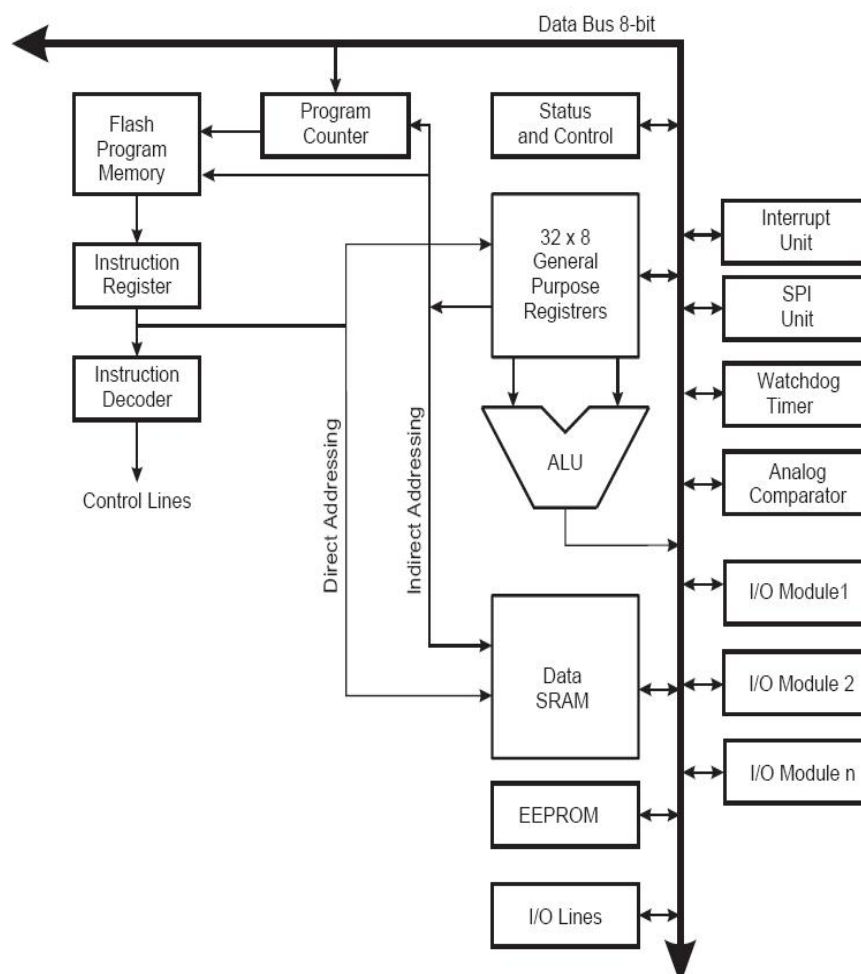
Dem Programmierer stehen insgesamt 32 8-Bit Arbeitsregister zur Verfügung, mit denen er frei arbeiten kann.

Speicher

Insgesamt stehen die folgenden drei Speicherarten zur Verfügung:

- Flash Speicher
- SRAM
- EEPROM

Blockschaltbild der AVR MCU Architektur:



2. Die ALU

Die ALU des Mikrocontrollers bietet 3 verschiedene Arten von Operationen: arithmetische, logische und bit-funktionen.

Sie verfügt über eine direkte Verbindung zu allen 32 Arbeitsregistern der CPU. Arithmetische Operationen zwischen Registern oder zwischen einem Register und einem Absolutwert werden innerhalb eines Taktzyklus ausgeführt.

3. Das Statusregister (SREG)

Das Statusregister ist ein 8-Bit Register, das Informationen zu vorhergegangenen arithmetischen Operationen enthält. Anhand dieser Informationen ist es möglich bedingte Sprünge abhängig von diesen Operationen durchzuführen.

Beispieloperationen:

Mnemonic	Bedeutung	Abhängigkeit
BREQ	Branch if Equal	Falls Zeroflag=1
BRLO	Branch if Lower	Falls Carryflag=1
BRCS	Branch if Carry set	Falls Carryflag=1

Beispielcode:

```
ldi r21, 0xA ;0xA in r21 laden
loop: dec r21 ;r21 um eins verringern
breq ende ;Falls ZeroFlag=1 zu ende springen
rjmp loop ;springe zu loop
ende:
```

Aufbau von SREG:

7	I	Global Interrupt Enable
6	T	Bit Copy Storage
5	H	Half Carry Flag
4	S	Sign Bit
3	V	Two's Complement Overflow Flag
2	N	Negative Flag
1	Z	Zero Flag
0	C	Carry Flag

Bit 7 - I: Global Interrupt Enable

Das I-Bit wird genutzt um Interrupts global abzuschalten. Damit Interrupts stattfinden können muss dieses Bit auf 1 gesetzt werden. Nachdem ein Interrupt einsetzt setzt die Hardware das Bit auf 0, um innerhalb der Interruptroutine keine weiteren Interrupts auszuführen.

Falls der Programmierer dies dennoch wünscht muss er innerhalb der Interruptroutine das Bit mittels des Befehls RETI wieder auf 1 setzen.

Bit 6 - T: Bit Copy Storage

Die beiden Instruktionen BLD (Bit LoaD) und BST (Bit STore) nutzen das T-Bit als Zwischenspeicher für das Kopieren von Bits.

Beispielcode:

```
bst r20,1    ;Bit 1 von r20 im T-Bit speichern  
bld r21,1    ;Bit 1 in r21 mit dem Wert des T-Bits belegen
```

Bit 5 - H: Half Carry Flag

Das Half Carry Flag wird im Falle eines halben Carries in einigen arithmetischen Operationen gesetzt und wird für BCD Arithmetik benutzt.

Bit 4 - S: Sign Bit

Das Sign Bit stellt immer das Ergebnis eines exklusiven Oders das auf das V-Bit und das N-Bit angewandt wird.

Bit 3 - V: Two's Complement Overflow Flag

Wird im Falle eines Überlaufs bei der Zweierkomplementarithmetik gesetzt, beispielsweise von der Instruktion NEG

Beispielcode:

```
ldi r20,0x80 ;0x80 nach r20  
neg r20      ;zweierkomplement von r20 bilden  
brvs ziel   ;springe falls overflowflag gesetzt  
neg r20      ;wird nie ausgeführt  
ziel:
```

Bit 2 - N: Negative Flag

Wird gesetzt, falls das Ergebnis einer arithmetischen oder logischen Operation negativ ist.

Beispielcode:

```
ldi r20,0x80 ;0x80 nach r20  
subi r20,0x81 ;0x81 von r20 abziehen  
brmi ziel    ;springe falls negativflag gesetzt  
subi r20,0x01 ;wird nie ausgeführt  
ziel:
```

Bit 1 - Z: Zero-Flag

Das Zero-Flag wird gesetzt, falls das Ergebnis einer arithmetischen oder logischen Operation 0 beträgt.

Beispielcode:

```
ldi r21, 0xA ;0xA in r21 laden
loop: dec r21 ;r21 um eins verringern
breq ende ;Falls ZeroFlag=1 zu ende springen
rjmp loop ;springe zu loop
ende:
```

Bit 0 - C: Carry-Flag

Das Carry-Flag wird gesetzt, falls es bei einer arithmetischen oder logischen Operation zu einem Überlauf kommt.

Beispielcode:

```
ldi r21, 0xFF ;0xFF in r21 laden
ldi r20, 0x01 ;0x01 in r20 laden
add r20,r21 ;r20 und r21 addieren
brcs ziel ;springe falls überlauf
inc r20 ;wird nie ausgeführt
ziel:
```

4. Der Stack

Der Stack wird über den SP (Stack-Pointer) im SRAM initialisiert. Er wird genutzt um Daten zwischenspeichern und als Möglichkeit bei dem Aufruf von Subroutinen die Übergabeparameter sowie die Rücksprungadresse zu speichern.

Bei der Initialisierung muss darauf geachtet werden, dass der Wert in SP größer als \$60 sein muss.

Für die Arbeit mit dem Stack gibt es grundsätzlich zwei Operationen: PUSH und POP. Push schiebt den Inhalt eines Registers auf den Stack, der SP wird dabei um 1 verringert. Pop holt einen Wert vom Stack und speichert ihn in einem Register, der Sp wird dabei um 1 vergrößert, der Stack wächst also während des Speicherns nach unten.

Beispielcode:

```
ldi temp,low(RAMEND)
out SPL,temp
ldi temp,high(RAMEND)
out SPH,temp           ;SP initialisieren
ldi r20,0x0F
push r20
pop r21                ;r21 ist nun 0x0F
```

Wie man im obigen Beispielcode erkennen kann ist der SP ein aus zwei Registern zusammengesetztes 16-Bit Register, das im IO-Space zu finden ist.

5. Interrupts

Interrupts sind Unterbrechungen des normalen Programmflusses, ausgelöst beispielsweise durch einen Benutzer, der eine Tastatur betätigt.

Das Hauptprogramm wird in diesem Falle unterbrochen und die Interruptroutine abgearbeitet.

Interrupts werden generell über das Global Interrupt Enable bit an- und abgeschaltet.

Zudem besitzt jeder Interrupt selber nochmal ein eigenes Interrupt Enable bit, das gesetzt sein muss, damit der Interrupt stattfindet.

Im Falle eines Interrupts wird zunächst die aktuelle Operation durchgeführt, dann zur Interruptroutine gesprungen.

Daraufhin wird das I-Bit auf 0 gesetzt, um weitere Interrupts während der Abarbeitung der Routine zu verhindern.

Bei Interrupts, die während einer bereits laufenden Interruptroutine ausgelöst werden wird das jeweilige Flag gesetzt und diese ausgeführt, nachdem die aktuelle Routine abgearbeitet ist, dies geschieht in Reihenfolge je nach Priorität.

Nachdem die Routine abgearbeitet wurde und das Hauptprogramm weiterläuft wird zunächst mindestens ein Befehl des Hauptprogramms abgearbeitet, bevor wieder in eine Interruptroutine gesprungen wird.

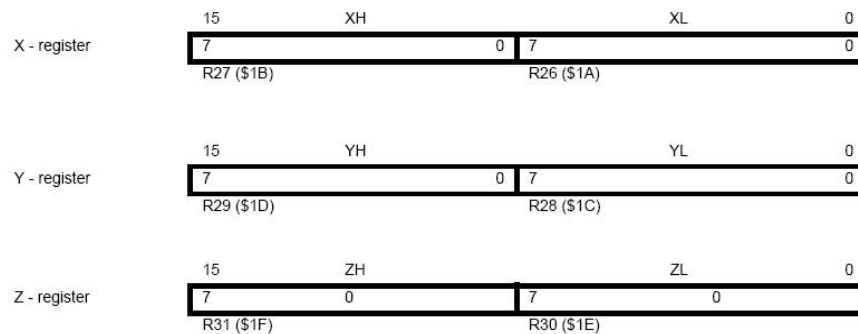
6. Die Register

Der AVR-Mikrocontroller verfügt über 32 8-Bit Arbeitsregister, wobei die oberen 6 nochmal eine gesonderte Rolle spielen.

Diese Arbeitsregister zeichnen sich durch einen Taktzyklus Zugriffszeit aus und sind nicht nur über ihren jeweiligen Namen ansprechbar, sondern zudem noch als die ersten 32 Adressen im SRAM verfügbar.

	7	0	Addr.	
General Purpose Working Registers	R0		\$00	
	R1		\$01	
	R2		\$02	
	...			
	R13		\$0D	
	R14		\$0E	
	R15		\$0F	
	R16		\$10	
	R17		\$11	
	...			
	R26		\$1A	X-register Low Byte
	R27		\$1B	X-register High Byte
	R28		\$1C	Y-register Low Byte
	R29		\$1D	Y-register High Byte
	R30		\$1E	Z-register Low Byte
	R31		\$1F	Z-register High Byte

Die Register R26 bis R31 bilden jeweils zu zweit ein 16bit Registerpaar, die genutzt werden können um Speicher im SRAM zu adressieren, also auch die dort abgebildeten 32 Register (obwohl diese nicht wirklich im SRAM vorhanden sind).



Beispielcode:

```
ldi r16,0x01           ; 0x01 nach r16
ldi XL,0x10
ldi XH,0x00           ; X zeigt jetzt auf r16 ($10)
ld r17,X              ; r17 = 0x01
```

7. Der Speicher

Flash Speicher

Der ATmega16 verfügt über einen 16k großen Flashspeicher, in dem die Programme abgelegt werden.

Da alle AVR-Instruktionen entweder 16 oder 32 Bit lang sind, ist dieser Speicher in 8192 Zellen von jeweils 16Bit aufgeteilt.

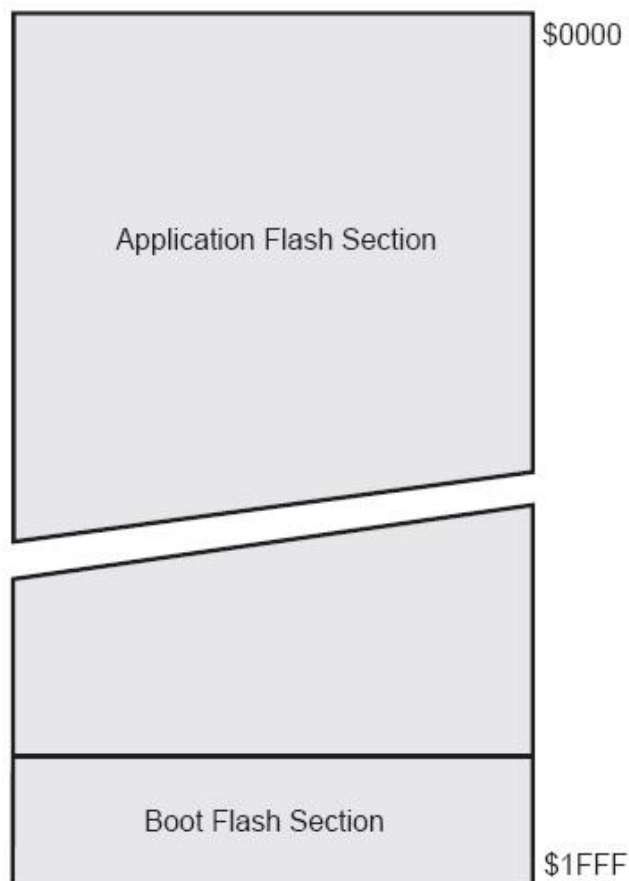
Somit beträgt die höchste zu Adresse innerhalb dieses Speichers $2^{13}-1$, worauf begründet der PC (Programmcounter) eine Größe von genau 13 Bit hat.

(8191d = 1111111111111b)

Um die Sicherheit der Software zu erhöhen ist der Speicher in eine Boot Section und eine Application Section aufgeteilt.

Lediglich der Programmcode, welcher in der Boot Section abgelegt ist hat die Befähigung den Flashspeicher zu beschreiben.

Dieser Speicher hat eine Lebensdauer von mindestens 10000 Schreib- und Löschrzyklen.



SRAM

Der SRAM des ATmega16 ist sozusagen sein Arbeitsspeicher, sollte man den Mikrocontroller mit einem Personal Computer vergleichen wollen. Die Größe des SRAM beträgt 1 Kilobyte wobei mehr als 1024 Byte adressiert werden können, da die ersten 96 Adressen nicht wirklich zum SRAM gehören, sondern lediglich darin abgebildet werden.

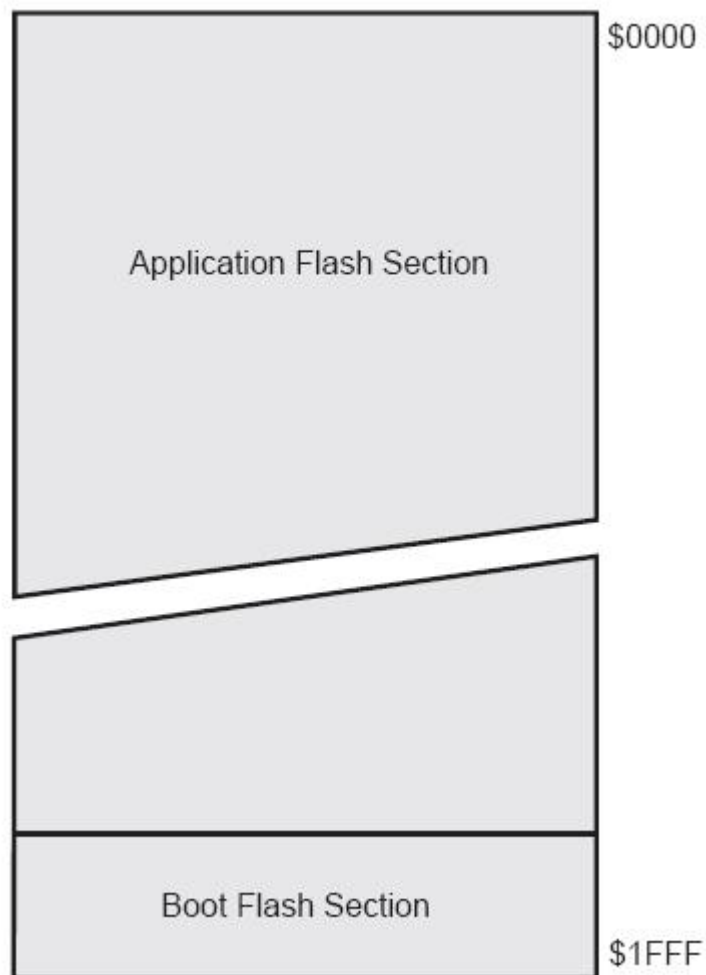
Die ersten 32 Byte bilden die Register File, die darauffolgenden 64 sind die I/O Register.

Ab der Adresse \$60 beginnt der interne SRAM, in den dann auch der zuvor erwähnte Stackpointer initialisiert werden kann.

\$0000-\$001F: Register File

\$0020-\$005F: I/O Register

\$0060-\$045F: Interner SRAM



EEPROM

Der EEPROM (Electrically Erasable Programmable Read Only Memory) des ATmega16 ist 512 Byte groß und befindet sich in einem separaten Datenbereich.

Der EEPROM kann von Programmen genutzt werden, um Daten auch über den Neustart des Controllers hinaus zu speichern.

Der Prozess des Speicherns ist allerdings etwas fehleranfällig, weshalb während dieses Vorgangs die Interrupts abgeschaltet werden müssen.

Insgesamt hat dieser Speicher eine Mindestlebensdauer von 100000 Schreib- und Löschzyklen.

EEPROM-Register

Für den Zugriff auf das EEPROM werden spezielle Register zur Verfügung gestellt, über die sich das Lesen und Schreiben des EEPROMs steuern lässt.

Die EEPROM-Register

EEARH/EEARL

Address Register

EEDR

Data Register

EECR

Control Register

Bits 7..4: Reserved (0)

Bit 3 : Ready Interrupt Enable (EERIE)

Bit 2: Master Write Enable (EEMWE)

Bit 1: Write Enable (EWE)

Bit 0: Read Enable (EERE)

Das EEPROM Adressregister (EEARH/EEARL)

Bit	15	14	13	12	11	10	9	8	
	-	-	-	-	-	-	-	EEAR8	EEARH
	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	X	
	X	X	X	X	X	X	X	X	

Bits	Bedeutung
15..9	Diese Bits werden für die Adressierung im EEPROM nicht benötigt und sind daher immer auf 0 gesetzt.
8..0	Diese Bits werden genutzt um die 512 Byte des EEPROM zu adressieren.

Das EEPROM Datenregister (EEDR)

Bit	7	6	5	4	3	2	1	0	
	MSB							LSB	EEDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Beim Schreibvorgang wird der Inhalt dieses Registers an das durch EEARH:EEARL adressierte Byte geschrieben.

Beim Lesevorgang hingegen wird das Byte an EEARH:EEARL im EEPROM in dieses Register geladen.

Das EEPROM Controlregister (EECR)

Bit	7	6	5	4	3	2	1	0	
	-				EERIE	EEMWE	EEWE	EERE	EECR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	X	0	

Bits	Bedeutung
7..4	Diese Bits sind im ATmega16 immer 0
3 - EERIE	EEPROM Ready Interrupt Enable Wenn dieses Bit auf 1 gesetzt ist, sowie Interrupts global eingeschaltet wird durchgängig der Interrupt ausgelöst solange EEWE 0 ist.
2 - EEMWE	EEPROM Master Write Enable Dieses Bit muss auf 1 gesetzt werden, damit das Schreiben durch das Setzen von EEWE auf 1 durchgeführt wird. Nach 4 Taktzyklen wird EEMWE automatisch zurück auf 0 gesetzt.
1 - EEWE	EEPROM Write Enable Wenn EEMWE 1 ist und dieses Bit auf 1 gesetzt wird, wird der Schreibvorgang in den EEPROM gestartet.
0 - EERE	EEPROM Read Enable Das setzen dieses Bits auf 1 bewirkt, dass das durch EEAR adressierte Byte im EEPROM in EEDR eingelesen wird.

Der EEPROM Schreibvorgang

Das EEPROM kann nicht programmiert werden, während die CPU auf den Flash schreibt!

- 1.) Zunächst muss das Byte, an das im EEPROM geschrieben werden soll über EEAR adressiert werden und das zu schreibende Datum in EEDR geladen sein.
- 2.) Um den Schreibvorgang möglichst sicher zu gestalten müssen Interrupts zuvor abgeschaltet werden.
- 3.) Bevor EEMWE auf 1 gesetzt wird muss zunächst geprüft werden, ob EEWE auf 0 gesetzt ist und somit ein möglicher bereits vorhergegangener Schreibvorgang bereits abgeschlossen.
- 4.) Innerhalb von 4 Taktzyklen muss nach EEMWE das EEWE Bit auf 1 gesetzt werden, da EEMWE nach 4 Taktzyklen automatisch gecleared wird. Aus diesem Grund müssen Interrupts abgeschaltet werden, da ein dazwischen einsetzender Interrupt bewirken würde, dass EEMWE gecleared würde bevor EEWE gesetzt werden kann und somit das Schreiben des EEPROMS nicht stattfinden kann.

Nachdem EEWE auf 1 gesetzt wurde wird die CPU für 2 Taktzyklen angehalten.

Der folgende Beispielcode kopiert 10 Byte vom SRAM, beginnend ab Adresse \$60 in den EEPROM, beginnend ab Adresse \$0000

```
cli                ; Interrupts abschalten
ldi r19,0x0a
ldi XL,0x60
ldi XH,0x00
ldi r17,0x00
ldi r18,0x00      ; Adressinitialisierung
eeprom_write:
sbic EECR,EEWE    ; warten auf Abschluss eines vorhergehenden Schreibvorgangs
rjmp eeprom_write
ld r16,X          ; Datum aus dem SRAM laden
out EEARH, r18
out EEARL, r17    ; Adressierung im EEPROM
out EEDR, r16     ; Byte zur Ausgabe schreiben.
sbi EECR,EEMWE   ; Start des Schreibvorgangs
sbi EECR,EEWE
inc XL            ; Adressen anpassen
inc r17
dec r19           ; Laufvariable um eins verringern
brne eeprom_write ; Falls Zeroflag nicht gesetzt Schleife fortsetzen
sei              ; Interrupts einschalten
```