2.6 Atmega-Peripherie/EEPROM

2.6.1 Wozu das EEPROM benutzen?

Der Benutzer soll mit zwei Tasten einen Wert zwischen 0 und 255 einstellen können (angezeigt am LED-Display). Dieser Wert soll auch nach dem Einschalten sofort wieder verfügbar sein. Welcher Speicherbereich ist für diese Aufgabe geeignet?

- Das Flash-EPROM ist für den Benutzer nur lesbar.
- Der Speicherinhalt des SRAM ist nach einem Neustart verloren.
- Das EEPROM dagegen ist beschreibbar und in seiner Speichereigenschaft unabhängig von der Stromversorgung. Damit ist es geeignet.

2.6.2 Speicherbereich

Das EEPROM hat wie das SRAM eine Speicherbreite von 8 Bit. Seine Speichergröße hängt bei den ATmega-Controllern vom Typ ab; beim ATmega8 und ATmega16 sind es 512, beim ATmega32 1024 Bytes. Der Adressbereich reicht von null bis zum Wert der Konstanten `EEPROMEND`.

2.6.3 Festlegen von Variablen im EEPROM

Das EEPROM wird im Assembler mit der Speicher-Direktiven `eseg` angesprochen. Alle Elemente nach dieser Anweisung werden fortlaufend im EEPROM abgelegt.

Danach kann mit der Direktive `db` ein Speicherplatz mit einem bestimmten Wert vorbelegt werden:

```
1         .eseg         ; ab hier EEPROM
2 merker: .db 20        ; merker hat den Inhalt 20
```

Anders als beim SRAM ist hier eine Initialisierung erlaubt und sinnvoll. Die Initialisierung erfolgt übrigens nicht durch das Programm, sondern über das Programmiergerät.

2.6.4 Lesen des EEPROMs

Zunächst soll ein initialisiertes EEPROM gelesen werden. Hier fällt eine Besonderheit auf: Es gibt für das Lesen und Schreiben des EEPROMs keine Befehle. Das Lesen und Schreiben erfolgt über die drei SFR-Register, deren Namen mit EE beginnen:

- Adressregister: EEARH und EEARL – legen fest, welche Adresse gewählt wird
- Datenregister: EEDR – enthält die Daten
- Steuer- und Statusregister: EECR – Bit0: Lesevorgang starten

Die Vorgehensweise ist wie folgt:

```
1  eewart: sbic EECR, EEWE           ; Warten, bis Schreib-Bit EEWE null ist
2          rjmp eewart               ;
3          ldi r16, low(eebeispiel)  ; Adresse laden
4          out EEARL, r16            ;
5          ldi r16, high(eebeispiel); Adresse laden
6          out EEARH, r16            ;
7          sbi EECR, EERE            ; Lese-Bit EERE setzen
8          in r16, EEDR              ; Wert holen
9          ...
10         .eseg                     ; EEPROM-Begin
11 eebeispiel: .db 20                ; Wert dort
```

Im folgenden Beispiel wird ein Byte im EEPROM beim Aufruf der Programmiersoftware mit dem Zeichen g vorbelegt. Das Programm liest das Zeichen und gibt es sofort aus (`rn32/1bytelesen.asm`):

```
1  .include "/usr/share/avra/m32def.inc"
2  ; liest und sendet erstes EEPROM-Byte
3          ldi r16, 0xff
4          out DDRC, r16
5  warten:
6          sbic EECR, EEWE          ; EEPROM bereit?
7          rjmp warten
8
9          ldi r16, low(eekram)     ; Adresse nach EEAR
10         out EEARL, r16
11         ldi r16, high(eekram)
12         out EEARH, r16
13
14         sbi EECR, EERE           ; Befehl anstossen
15         in r16, EEDR             ; Ergebnis abholen
16
17         out PORTC, r16           ; und anzeigen
18 ende:
19         rjmp ende                ; while(1){}
20 ; =====================
21         .eseg
22 eekram: .db 0x67                 ; 8 bit-char, Inhalt 67 ('g')
23 ; bsp1: .dw 5000                 ; 16 bit-int, Inhalt 5000
24 ; bsp2: .byte 30                 ; = 30 Bytes, Inhalt undef.
```

Wir sagen dem Übertragungsprogramm, dass zusätzlich zum Flash-EPROM noch das EEPROM gesetzt werden soll:

Terminal

```
schueler@debian964:~$ avra 1bytelesen.asm && avrdude -p m8 \
> -c stk500v2 -P /dev/ttyUSB0 -e
> -U flash:w:1bytelesen.hex -U eeprom:w:1bytelesen.eep.hex
```

2.6.5 Schreiben des EEPROMs

Das obige Beispiel hätte man ebenso gut mit Flash-Speicher machen können. Ab jetzt soll das EEPROM durch das Programm selbst geschrieben werden. Anschließend soll es gelesen und per UART zum PC geschickt werden. Das Vorgehen beim Schreiben ist ähnlich wie das Lesen:

```
1          ; cli                    ; Interrupts stoppen
2  _swart: sbic EECR, EEWE          ; Warten, bis EEWE null ist
3          rjmp _swart              ;
4          ldi r16, low(eebeispiel) ; Adresse laden
5          out EEARL, r16           ;
6          ldi r16, high(eebeispiel);
7          out EEARH, r16           ;
8          ldi r16, 0x12            ; zu schreibendes Datenbyte nach EEDR
9          out EEDR, r16            ; ausgeben
10         sbi EECR, EEMWE          ; Schreiben freigeben
11         sbi EECR, EEWE           ; Schreiben anstossen -- fertig
12         ; sei
```

Der einzige Unterschied liegt darin, dass man zuerst mit dem EEMWE-Bit das Schreiben erlaubt und danach mit dem EEWE-Bit den Schreibvorgang ausführt. Das komplette Beispiel sieht so aus (`rn32/2byteschreiben+lesen.asm`):

```
1  .include "/usr/share/avra/m32def.inc"
2  ;
3  ; schreibt, liest und gibt aus: erstes EEPROM-Byte
4  ;
5          ldi r16, 0xff
6          out DDRC, r16
7  ; EEWE-Bit vorsorglich loeschen, da undef.
8          cbi EECR, EEWE
9  ; Byte schreiben:
10 ;_s_warten:             ; nach Reset nicht noetig
11 ;        sbic EECR, EEWE ; nach Reset nicht noetig
12 ;        rjmp _s_warten  ; nach Reset nicht noetig
13         ldi r16, low(eespeicher) ; Adresse nach EEAR
14         out EEARL, r16
15         ldi r16, high(eespeicher)
16         out EEARH, r16
17         ldi r16, 'a'    ; 0x61 soll ins EEPROM geschr. werden
18         out EEDR, r16
19         sbi EECR, EEMWE ; Schreiben freigeben
20         sbi EECR, EEWE  ; Schreiben anstossen
21 ; Byte lesen:
22 _l_warten:
23         sbic EECR, EEWE         ; EEPROM bereit?
24         rjmp _l_warten
25         ldi r16, low(eespeicher) ; Adresse nach EEAR
26         out EEARL, r16
27         ldi r16, high(eespeicher)
28         out EEARH, r16
29
30         sbi EECR, EERE          ; Befehl anstossen
31         in r17, EEDR            ; Ergebnis abholen
32         out PORTC, r17          ; und anzeigen
33 ende:
34         rjmp ende               ; while(1){}
35 ; ======================
36 ; EEPROM:
37         .eseg
38 eespeicher:
39         .db 0           ; 1 Byte
```