RUHR-UNIVERSITÄT BOCHUM

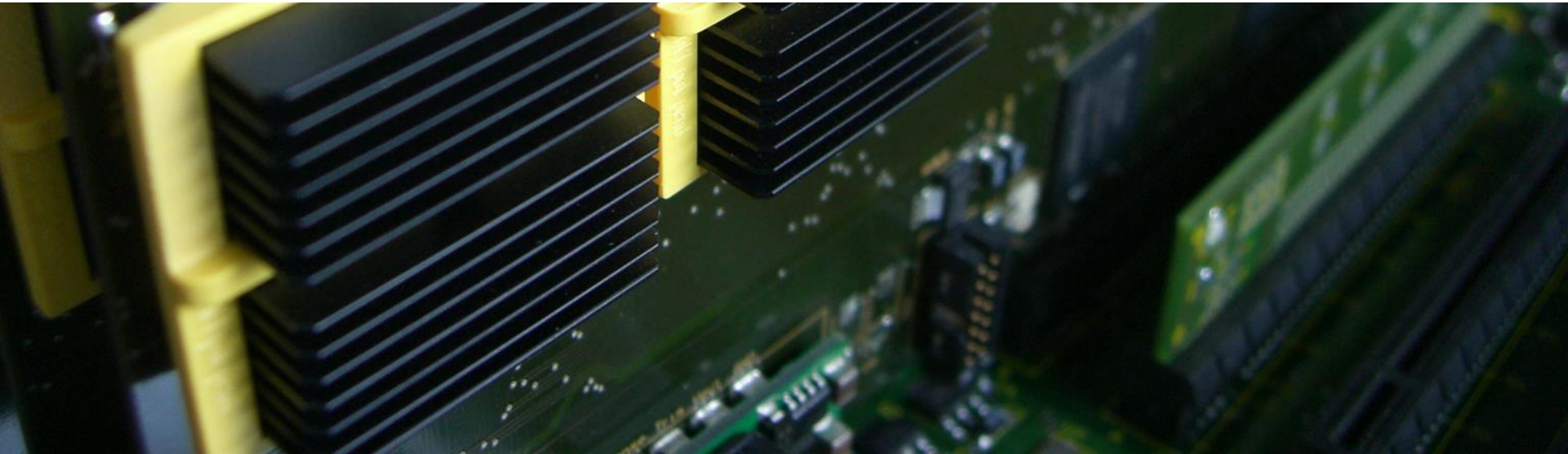
RUB

Eingebettete Prozessoren

SS 2014

Übung 8: EEPROM und I/O Ports

Dipl.-Ing. Thomas Pöppelmann
Arbeitsgruppe Sichere Hardware
Horst Görtz Institut für IT-Sicherheit



Agenda

1. Besprechung Übung 7
2. EEPROM
3. I/O Ports

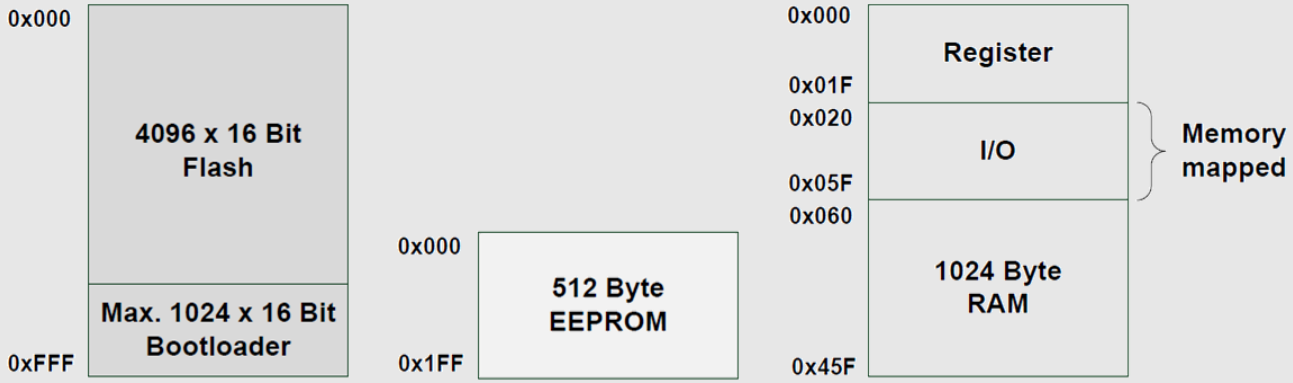
Besprechung Übung 7

Besprechung Übung 7

- Genereller Hinweis:
  - Speicherorganisation ist eine wichtige Grundlage auf der die folgenden Übungen aufbauen
  - Jeder muss in der Lage sein diese Übung zu verstehen und ggf. nachholen!
  - Bei Fragen/Problemen bitte melden

Besprechung Übung 7

- Der AVR besitzt drei Adressräume:
  - Flash/Program Memory: Programm Code und Kostanten
  - EEPROM: Persistenter Speicher
  - RAM: Nicht-persistenter Speicherbereich für temporäre Daten
    - In diesen Adressbereich sind auch die Register (0x000-0x01F) und die I/O Register (0x020-0x05F) „gemappt“
- Im AVR gibt es keine Memory Management Einheit (MMU) – unbeabsichtigtes Überschreiben von Speicherbereichen ist möglich und führt zu schwer zu behebenden Fehlern
- Konstanten und Programm Code teilen sich den Flash: Es kann passieren, dass Konstanten als Programm Code interpretiert werden



Korrektur/Hinweis zur S-Box Aufgabe

- Text der Übung
  - Falsch: „Die Present S-Box erhält 8 Eingangsbits und ersetzt diese durch 8 Ausgangsbits“
  - Richtig: „Die Present S-Box erhält **4** Eingangsbits und ersetzt diese durch **4** Ausgangsbits“
- Quelle der Sbox
  - http://perso.uclouvain.be/fstandae/source_codes/lightweight_ciphers/source/8BitSBox.asm
- Paper zur Implementierung
  - http://perso.uclouvain.be/fstandae/source_codes/lightweight_ciphers/paper.pdf
- Warum 256 Einträge?
  - “The S-boxes are stored as … 256-byte table[s], ... This allows for two S-box lookups in parallel.“
  - Ein Time-Memory trade-off
- Welche bekannte Chiffe hat eine 8-bit S-Box? Antwort: AES
- Warum hat Present eine 4-bit S-Box?
  - Antwort: Look-up Tabelle kleiner bzw. Resourcenverbrauch geringer als bei 8-bit S-Box
  - Andere Perspektive: Zwei Look-ups gleichzeitig bei gleichen Kosten möglich

1. Besprechung Übung 7

Laden Sie die 10 Byte Input der S-Box aus dem Flash/Program Memory (Label sbox_input) in den RAM (Label sbox_input_ram).

```
copy_from_flash:

        ldi ZH, high(sbox_input*2) //Flash Adresse
        ldi ZL, low(sbox_input*2) //niederwertigstes Bit auf 0 setzen

        ldi YH, high(sbox_input_ram) //RAM Adress
        ldi YL, low(sbox_input_ram)

        ldi r16, 10 //Schleifenzähler
CFF_LOOP:
        lpm r17, Z++ //Aus dem Flash laden, Pointer inkrementieren
        st Y++, r17 //In den RAM speichern , Pointer inkrementieren
        dec r16 //Schleifenzähler dekrementieren
        brne CFF_LOOP //Branch
ret
```

1. Besprechung Übung 7

Wenden Sie die S-Box auf die 10 Byte Input Daten an, welche Sie zuvor in den RAM geladen haben. Dabei soll der Input mit dem jeweiligen Output der S-Box überschrieben werden.

```
eval_sbox_flash:
        ldi YH, high(sbox_input_ram) //Pointer laden
        ldi YL, low(sbox_input_ram)

        ldi r16, 10 //Schleifenzähler
        clr r0 //Null-Register
ESF_LOOP:
        ldi ZH, high(sbox*2)  //Adresse der S-Box laden
        ldi ZL, low(sbox*2)
        ld r17, Y //Input aus dem RAM laden
        add ZL, r17 //Input aus dem RAM auf die Adresse der S-Box addieren
        adc ZH, r0
        lpm r18, Z //Aus dem Flash laden
        st Y++, r18 //Input ersetzen und Zähler inkrementieren
        dec r16 //Schleifenzähler dekrementieren
        brne ESF_LOOP
ret
```

1. Besprechung Übung 7

- Vergleichen Sie die Geschwindigkeiten der beiden Ansätze (S-Box im Flash vs. S-Box im RAM)
  - LPM benötigt 3 Zyklen
  - LD benötigt 2 Zyklen
  - Wenn viele Zugriffe auf Konstanten im Flash notwendig sind und genügend RAM vorhanden ist, können diese in den RAM kopiert werden
  - Außerdem interessant, da das Kopieren nicht zeitkritisch zum Programmstart erfolgen kann und damit eventuell zeitkritische Komponenten schneller zugreifen können

EPROM

Speicherzugriff: EEPROM

- Nicht-flüchtiger, Byte adressierbarer Speicher
- Separater Speicherbereich
- Lesezugriff in 5 Takten, Schreiben bis zu 8448 Takte
- Typische Nutzung für Daten, die den Reboot des Systems überdauern (Konfiguration, Status)
- Begrenzte Lebensdauer, spezieller Schutz vor ungewolltem Überschreiben
- “at least 100,000 write/erase cycles”

Speicherzugriff: EEPROM

- Die 4 Register zur Steuerung des EEPROM:
  - **EEARH/EEARL (**EEPROM Address Register)**:** Enthält die Adresse in die geschrieben oder von der gelesen werden soll. Für 512 bytes EEPROM würde ein 8-bit Register nicht ausreichen.
  - **EEDR** (EEPROM Data Register): Enthält Daten die geschrieben werden sollen oder bereits gelesen wurden.
  - **EECR** (EEPROM Control Register):
    - **EEMWE** (Master Write Enable)**:** Muss gesetzt sein, damit eine Schreiboperation ausgeführt wird. Wird von der Hardware nach 4 Zyklen wieder auf 0 gesetzt.
    - **EEWE** (Write Enable): Triggert das Schreiben in den EEPROM. Wird von der Hardware auf 0 gesetzt, nachdem der EEPROM geschrieben wurde (mehrere hundert Zyklen)
      - If (EEMWE=1 and EEWE=1) then *EEPROM(EEAR) = EEDR*
    - **EERE** (Read Enable): EEPROM lesen:
      - If (EERE=1) then *EEDR= EEPROM(EEAR)*
- Das **EEWE** Bit zeigt an, ob eine EEPROM Schreiboperation bereits abgeschlossen wurde (EEWE=0) oder noch läuft (EEWE=1). Dies sollte vor einer erneuten Schreiboperation überprüft werden.

Speicherzugriff: EEPROM

```
EEPROM_WRITE:          ; Warten, bis der vorige Schreibzugriff beendet ist
        SBIC EECR,EEWE
        RJMP EEPROM_WRITE

        ; Schreibe die EEPROM Adresse aus den Registern (R18:17) in den EEPROM I/O Bereich
        OUT EEARH, R18
        OUT EEARL, R17

        ; Schreibe R16 in das Datenregister
        OUT EEDR, R16

        ; Setze das Master Write Enable-Bit (EEMWE) im EEPROM Control Register (EECR)
        SBI EECR, EEMWE

        ; Starte Schreiboperation durch Setzen des Write Enable-Bit (EEWE) im EEPROM Control Register (EECR)
        SBI EECR, EEWE
```

Speicherzugriff: EEPROM

- Test: *u8_prasenz_eeprom.asm*
  - 0x01 an Adresse 0x0000 in den EEPROM schreiben
  - 0x02 an Adresse 0x0001 in den EEPROM schreiben
  - Sicherstellen, dass die vorhergehende Schreiboperation abgeschlossen ist
- Bitte in der Speicheransicht des AVR Studio den Programmablauf verfolgen

I/O Ports

I/O Ports

- I/O Ports dienen zur Kommunikation mit der Außenwelt
  - LED aktivieren
  - Taster abfragen
  - Kommunikation

- Wichtige Definitionen:
  - **Pin**
    - Einzelne I/O Leitung (1 bit) die entweder als Ausgang oder Eingang geschaltet werden kann
  - **Port**
    - Jeweils 8 bzw. 7 Pins sind in einem Port zusammengefasst
    - Jeder Pin kann individuell konfiguriert werden
    - Vorhandene Ports: PORTB (8 Bit), PORTC (7Bit) und PORTD (8 Bit)

I/O Ports

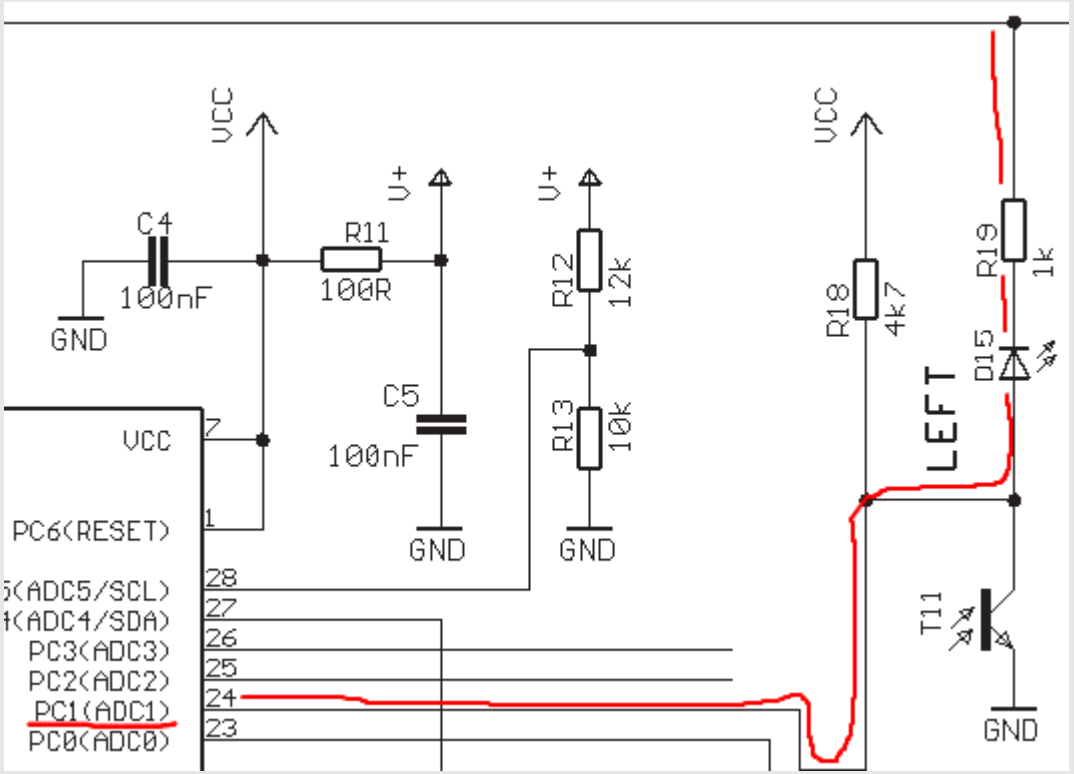
- An den Ports *x={B,C,D}* kann jeder Pin *n={0..7}* bezeichnet als P*xn* durch setzten des Bits *n* in den drei Steuerregistern kontrolliert werden:
  - **DDR*x* (Data Direction Register):** Konfiguration der Ein/Ausgaberichtung der zugeordneten Pins (‘1’ = Ausgabe/’0’=Eingang)
  - **PORT*x* (Data Register)**:
    - Fall 1: P*xn* ist Eingang : PORT*x* konfiguriert die Pull-Up Wiederstände (‘1’ = Pull-Up aktivieren/’0’=Pull-Up ausschalten)
    - Fall 2: P*xn* ist Ausgang: Legt den Ausgang des Pins fest (‘1’ = high/’0’= low)
  - **PIN*x* (Input Pins)**: Auslesen eines Pins

I/O Ports

- Beispielaufgabe: LED D15 (links/hinten) am Asuro leuchten lassen
  1. Port identifizieren (Datenblatt)
  2. Port konfigurieren (Ausgang)
  3. Port beschreiben (Ausgang auf 1)

I/O Ports – Port identifizieren

- LED D15 wird durch Pin PC1 an Port PORTC gesteuert



Asuro Manual
S. 74

I/O Ports - Port konfigurieren

- Setze Bit 1 in DDRC (Pin als Ausgabe Port konfigurieren)
  - Keine anderen Bits überschreiben (führt zu Problemen bei größeren Programmen)
  - Zwei Möglichkeiten:
    - Auslesen und maskieren
    - SBI Befehl (Set Bit in I/O Register) um einzelne Bits zu setzen

```
;Pin PC1 als Ausgabe Port konfigurieren
in  r16, DDRC          ; Aktuellen Inhalt einlesen
ldi r17, (1<<PC1)      ; 0b00000010
or  r16, r17           ; Bit PC1 auf Null setzen
out DDRC, r16          ;Port konfigurieren

;Alternative
sbi DDRC, PC1          ;Setze bit
```

I/O Ports - Port beschreiben

- Setze Bit 1 in PORTC (Pin auf high ziehen)

```
;Pin PC1 auf High setzen
in  r16, PORTC
ldi r17, (1<<PC1)      ; 0b00000010
or  r16, r17
out PORTC, r16

;Alternative
sbi PORTC, PC1
```

I/O Ports

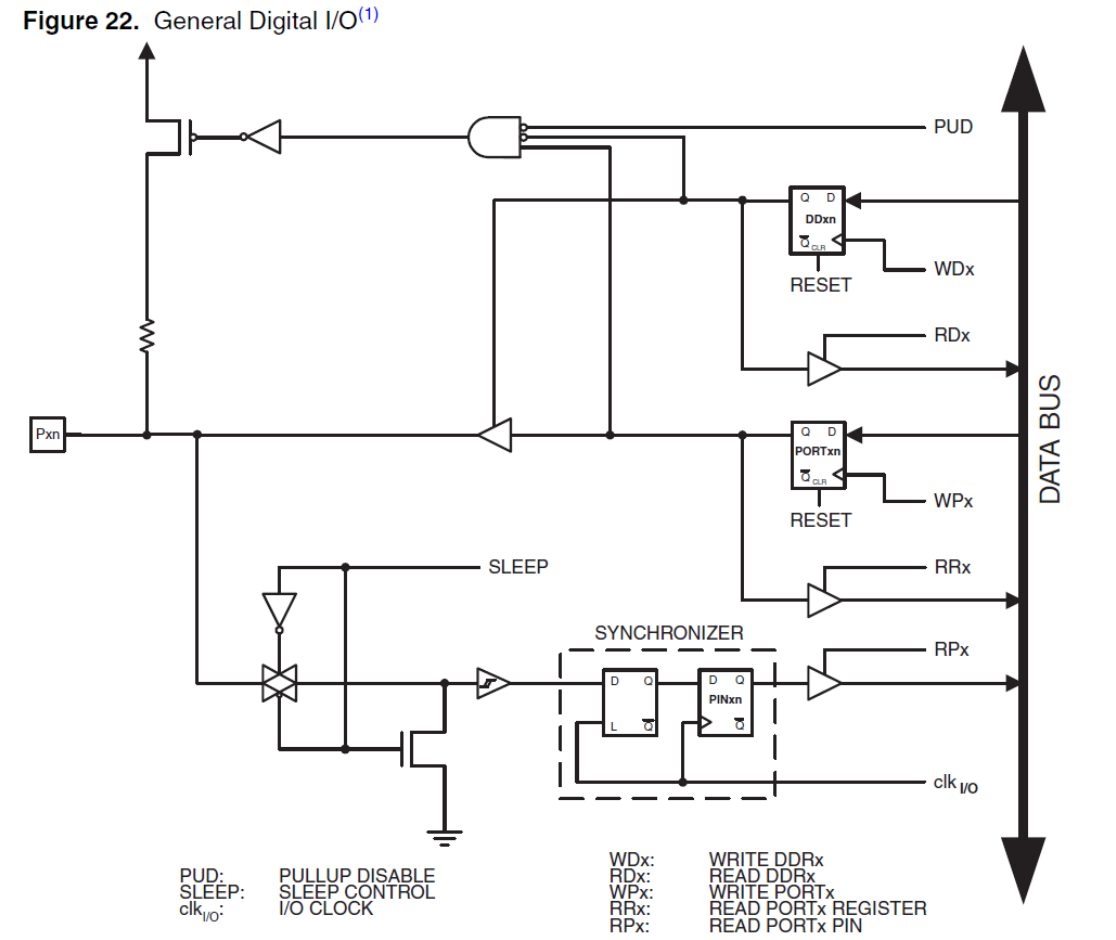
- Ports setzen:
  - Auslesen/Maskieren/Setzen:
    - Vorteil: (Atomare) Konfiguration mehrerer Pins
    - Nachteil: Aufwändiger
  - SBI/CBI
    - Vorteil: Sehr einfach und weniger fehleranfällig
    - Nachteil: Nur ein Pin kann gesetzt werden
- Zusätzlich nützlich:
  - SBIS (Skip if Bit in I/O Register Set)
  - SBIC (Skip if Bit in I/O Register Cleared)

```
;Pin PC1 auf High setzen
in  r16, PORTC
ldi r17, (1<<PC1)      ; 0b00000010
or  r16, r17
out PORTC, r16


;Alternative
sbi PORTC, PC1
```

Siehe auch digitalIO.asm unter Kursunterlagen->Assemblerprogramme aus der Vorlesung

I/O Ports



- Ein Pin kann als Ausgabe Pin betrieben werden
- Ein Zurücklesen über das PINx Register ist trotzdem möglich
- Antwort auf eine Frage in der Übung befindet sich in diesem Bild

- **Datenblatt: “Reading the Pin Value” bzw. im Skript Abschnitt 4.2.2**

I/O Ports

- Test: *bsp_u8.asm*
  - Pin PC1 als Ausgabeport konfigurieren
  - Pin PC1 auf High setzen