U2 4. Übung

- Wiederholung: Zugriff auf AVR-Prozessor-Register
- Wiederholung: I/O-Ports
- Bitoperationen und Hexadezimalzahlen
- Überblick: Modulare Softwareentwicklung
- Aufgabe 3: LED-Modul der SPiCboard-Bibliothek

U2-1 Register beim AVR-µC

1 Überblick

- Beim AVR-µC sind die Register
  - in den Speicher eingeblendet
  - am Anfang des Adressbereichs angeordnet
- Adressen sind der Dokumentation zu entnehmen
  ☞ **ATMega32-Datenblatt** verlinkt im Doku-Bereich der Webseite
  - Die C-Bibliothek (avr-libc), die wir verwenden, definiert bereits sinnvolle Makros für alle Register des AVR µC
    (`#include <avr/io.h>`)
  - Damit die für den jeweiligen µC passenden Adressen verwendet werden, muss man dem Compiler den µC-Typ mitteilen: `-mmcu=atmega32`

2 Makros für Register-Zugriffe

- Makros mit aussagekräftigen Namen können den Umgang mit Registern deutlich vereinfachen
- Beispiel:
  - Makro für Register an Adresse 0x5:

```
#define REG1 (*(volatile unsigned char *)0x5)
```

  - Verwenden dieses Registers:

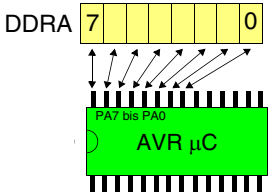
```
REG1 = 0;              /* schreibender Zugriff */
...
if (REG1 == 0x04)      /* lesender Zugriff */
    REG1 &= ~4;        /* lesender und schreibender Zugriff */
```

- Warum ist `volatile` in dieser Makrodefinition notwendig?

U2-2 I/O-Ports des AVR-µC

1 Überblick

- Jeder I/O-Port des AVR-µC wird durch drei 8-bit Register gesteuert:
  - Datenrichtungsregister (`DDRx` = data direction register)
  - Datenregister (`PORTx`)
  - Port Eingabe Register (`PINx` = port input register, nur-lesbar)
- Jedem Anschluss-Pin ist ein Bit in jedem der 3 Register zugeordnet
  ➤ Beispiel: DDR von Port A:



- Ausführliche Beschreibung: ATMega32-Datenblatt

2 I/O-Port-Register

- **`PINx`**: Bit *i* gibt den aktuellen Wert des Pin *i* von Port *x* an (nur lesbar).
- **`DDRx`**: Pin von Port *x* als Ein- oder Ausgang verwenden
  - Bit *i* = 1→ Pin *i* als **Ausgang** verwenden
  - Bit *i* = 0→ Pin *i* als **Eingang** verwenden
- **`PORTx`**: Auswirkung abhängig von `DDRx`:
  - ◆ ist Pin *i* als **Ausgang** konfiguriert, so steuert Bit *i* im `PORTx` Register ob am Pin *i* ein high- oder ein low-Pegel erzeugt werden soll.
    - Bit *i* = 1→ high-Pegel an Pin *i*
    - Bit *i* = 0→ low-Pegel an Pin *i*
  - ◆ ist Pin *i* als **Eingang** konfiguriert, so kann man einen internen pull-up-Widerstand aktivieren
    - Bit *i* = 1→ pull-up-Widerstand an Pin *i* (Pegel wird auf high gezogen)
    - Bit *i* = 0→ Pin *i* als tri-state konfiguriert

3 Beispiel: Aktivieren eines Ports

- Pin 3 von Port B als Ausgang konfigurieren und auf V_{CC} schalten:

```
DDRB |= 0x08;  /* Pin 3 von Port B als Ausgang nutzen... */

PORTB |= 0x08; /* ...und auf 1 (=high) setzen */
```

- Pin 0 von Port D als Eingang nutzen, pull-up-Widerstand aktivieren und prüfen ob ein low-Pegel anliegt:

```
DDRD &= ~0x01; /* Pin 0 von Port D als Eingang nutzen... */
PORTD|= 0x01;  /*...und den pull-up-Widerstand aktivieren*/

if ( (PIND & 0x01) == 0) { /* den Zustand auslesen */
    /* ein low Pegel liegt an, der Taster ist gedrückt */
}
```


U2-3 Bitoperationen und Hexadezimalzahlen

- Hardwarenahe Programmierung erfordert oft das gezielte Setzen bestimmter Bits in Prozessorregistern
- Konfiguration jedes einzelnen Bits ist einfach, aber zeitaufwendig
- Kombination mehrerer gleicher Operationen auf einem Register
- Hexadezimalzahlen erleichtern die Arbeit mit Bitmustern

1 Hexadezimalzahlen

- Dezimalzahlen eignen sich schlecht zur Darstellung von Bitmustern
- Binärzahlen repräsentieren Bitmuster, sind jedoch sehr lang zu schreiben
- Hexadezimalzahlen können Bitmuster kompakt darstellen
  - ◆ Ziffern 0x0 bis 0xf (Werte 0 bis 15)
  - ◆ können jeweils eine Gruppe von 4 Bits darstellen
  - ◆ 8-Bit Zahlen lassen sich also mit 2 Hexadezimalziffern darstellen
- Vom Bitmuster zur Hexziffer
  - ◆ Betrachtung von jeweils 4 Bits des gewünschten Werts
  - ◆ Hexziffer für jedes *Nibble* ermitteln
  - ◆ die Hexziffern zum Gesamtwert zusammensetzen

1 Hexziffern: Beispiele

- 0b 0000 0000 = 0x
- 0b 1111 1111 = 0x
- 0b 1010 0101 = 0x
- 0b 1101 1000 = 0x
- 0b 0010 0111 = 0x
- 0b 1100 1010 1111 1110 1011 1010 1011 1110 = 0x
- 0b 1101 1110 1010 1101 1011 1110 1110 1111 = 0x

1 Hexziffern: Beispiele

- 0b 0000 0000 = 0x00
- 0b 1111 1111 = 0xff
- 0b 1010 0101 = 0xa5
- 0b 1101 1000 = 0xd8
- 0b 0010 0111 = 0x27
- 0b 1100 1010 1111 1110 1011 1010 1011 1110 = 0xcafebabe
  (Magic Number z.B. in Java Classfiles)
- 0b 1101 1110 1010 1101 1011 1110 1110 1111 = 0xdeadbeef
  (Magic Number z.B. zur Freispeichermarkierung im Solaris Kern)

2 Setzen von Bits

- Bitweise ODER-Operation (C-Operator: |)

x	0	0	1	?	?
y	0	1	1	0	1
x|y	0	1	1	?	1

- ODER eines beliebigen Wertes mit 0 erhält den ursprünglichen Wert
- ODER eines beliebigen Wertes mit 1 produziert das Ergebnis 1
- Setzen einer Auswahl von Bits in einem Wert aus mehreren Bits:
  - Konstruktion einer Bitmaske
  - Bits, die gesetzt werden sollen, haben in der Maske eine 1
  - Bits, die unverändert bleiben sollen, haben in der Maske eine 0

2 Beispiel: Setzen von Bits

- Setzen der Bits 2, 5 und 6 eines unbekannten 8-Bit Werts

```
value |= (1 << 2);
value |= (1 << 5);
value |= (1 << 6);
```

- Welche Maske liefert das gewünschte Ergebnis?

Wert	?	?	?	?	?	?	?	?
Maske								
Ergebnis	?	1	1	?	?	1	?	?

2 Beispiel: Setzen von Bits

- Setzen der Bits 2, 5 und 6 eines unbekannten 8-Bit Werts

Wert	?	?	?	?	?	?	?	?
Maske	0	1	1	0	0	1	0	0
Ergebnis	?	1	1	?	?	1	?	?
0x								

- Setzen aller drei Bits in einer Operation

```
value |= 0x__;
```


2 Beispiel: Setzen von Bits

- Setzen der Bits 2, 5 und 6 eines unbekannten 8-Bit Werts

Wert	?	?	?	?	?	?	?	?
Maske	0	1	1	0	0	1	0	0
Ergebnis	?	1	1	?	?	1	?	?
0x	6				4			

- Setzen aller drei Bits in einer Operation

```
value |= 0x64;
```

- Alternative, wenn Bitpositionen nicht fest bekannt

```
value |= ((1<<BP2) | (1<<BP5) | (1<<BP6));
```

  - ◆ höherer Schreibaufwand
  - ◆ dafür lesbarer und portabel

3 Löschen von Bits

- Bitweise UND-Operation (C-Operator: **&**)

x	0	0	1	?	?
y	0	1	1	0	1
x&y	0	0	1	0	?

- UND eines beliebigen Wertes mit 0 produziert das Ergebnis 0
- UND eines beliebigen Wertes mit 1 erhält den Wert
- Löschen einer Auswahl von Bits in einem Wert aus mehreren Bits:
  - ◆ Konstruktion einer Bitmaske
  - ◆ Bits, die gesetzt werden sollen, haben in der Maske eine 0
  - ◆ Bits, die unverändert bleiben sollen, haben in der Maske eine 1
- Alternativ: Invertieren einer Bitmaske (C-Operator: ~)
  - ◆ die zu invertierende Maske ist dann dieselbe wie beim Setzen mit ODER

3 Beispiel: Löschen von Bits

- Löschen der Bits 1, 3, 5 und 7 eines unbekannten 8-Bit Werts

```
value &= ~(1 << 1);
value &= ~(1 << 3);
value &= ~(1 << 5);
value &= ~(1 << 7);
```

- Welche Maske liefert das gewünschte Ergebnis?

Wert	?	?	?	?	?	?	?	?
Maske								
Ergebnis	0	?	0	?	0	?	0	?

3 Beispiel: Löschen von Bits

- Löschen der Bits 1, 3, 5 und 7 eines unbekannten 8-Bit Werts

Wert	?	?	?	?	?	?	?	?
Maske	0	1	0	1	0	1	0	1
Ergebnis	0	?	0	?	0	?	0	?
0x								

- Löschen aller vier Bits in einer Operation

```
value &= 0x__;
```


3 Beispiel: Löschen von Bits

- Löschen der Bits 1, 3, 5 und 7 eines unbekannten 8-Bit Werts

Wert	?	?	?	?	?	?	?	?
Maske	0	1	0	1	0	1	0	1
Ergebnis	0	?	0	?	0	?	0	?
0x	5				5			

- Löschen aller vier Bits in einer Operation

```
value &= 0x55;
```

- Alternative, wenn Bitpositionen nicht fest bekannt

```
value &= ~((1<<BP1) | (1<<BP3) | (1<<BP5) | (1<<BP7));
```

  - ◆ höherer Schreibaufwand
  - ◆ dafür lesbarer und portabel

4 Umschalten von Bits

- Bitweise Exklusiv-ODER-Operation (C-Operator: ^)

x	0	0	1	?	?
y	0	1	1	0	1
x^y	0	1	0	?	not ?

- XOR eines beliebigen Wertes mit 0 erhält den Wert
- XOR eines beliebigen Wertes mit 1 invertiert den Wert
- Invertieren einer Auswahl von Bits in einem Wert aus mehreren Bits:
  - ◆ analog zu ODER, nur mit XOR-Operator
- Klausur MC-Frage 10/2008:
  Welches Ergebnis hat folgender C-Ausdruck?

```
x ^ x
```


U2-4 Überblick: Modulare Softwareentwicklung

- Ausführlichere Behandlung: Vorlesung Folien **D.74 - D.92**
- Bündelung von Daten und darauf operierenden Funktionen zu Modul
- Ein Modul ist eine Blackbox (Kapselung)
  - ◆ Trennung von Schnittstelle und Implementierung
  - ◆ Die Schnittstelle definiert das externe Verhalten des Moduls
  - ◆ Schnittstellenbeschreibung in der Header-Datei
  - ◆ Implementierung in der Quelldatei
- Module sind
  - ◆ austauschbar durch andere Implementierungen der gleichen Schnittstelle
  - ◆ wiederverwendbar
  - ◆ in Programmbibliotheken bereitstellbar

1 Modul-Schnittstelle

- Definiert
  - Funktionsprototypen
  - Typen
  - Daten (globale Variablen, nach Möglichkeit zu vermeiden!)
- Beschreibung der Schnittstelle in einer Header-(**.h**)-Datei
  - verbindliche Vorgabe für Implementierungen
  - darf nicht verändert werden (warum?)
  - Sichtbarkeit von Hilfsdaten/-funktionen auf Modul beschränken (`static`)
- Einbinden der Schnittstellenbeschreibung in anderen Modulen

2 Schnittstellenbeschreibung

- Erstellen einer **.h**-Datei (Konvention: gleicher Name wie **.c**-Datei)

```
#ifndef LED_H
#define LED_H

/* fixed-width Datentypen einbinden (werden im Header verwendet) */
#include <stdint.h>

/* LED-Typ */
typedef enum { RED0=0, YELLOW0=1, GREEN0=2, ... } LED;

/* Funktion zum Aktivieren einer bestimmten LED */
uint8_t sb_led_on(LED led);
...
#endif
```

- Mehrfachinkludierung (evtl. Zyklen!) vermeiden
  - durch Definition und Abfrage eines Präprozessormakros
  - Konvention: das Makro hat den Namen der **.h**-Datei, '**.**' ersetzt durch '_'
  - der Inhalt wird nur eingebunden, wenn das Makro noch nicht definiert ist
- Flacher Namensraum: Wahl möglichst eindeutiger Namen

3 Einbinden von Schnittstellenbeschreibungen

- Einbinden mit `#include` dort, wo diese verwendet werden
  - im Header nur solche Schnittstellen, die auch im Header benötigt werden
    ☞ z.B. `stdint.h` im vorangehenden Beispiel
  - von der Implementierung verwendete Modulschnittstellen sind auch in dieser einzubinden
    ☞ verschiedene Implementierungen verwenden evtl. verschiedene Module

4 Initialisierung eines Moduls

- Module müssen oft Initialisierung durchführen (z.B. Ports konfigurieren)
  - z.B. in Java mit Klassenkonstruktoren möglich
  - C kennt kein solches Konzept
- Workaround: Modul muss bei erstem Aufruf einer seiner Funktionen ggf. die Initialisierung durchführen
  - muss sich merken, ob die Initialisierung schon erfolgt ist
  - Mehrfachinitialisierung vermeiden (Synchronisation!)

```
static uint8_t initDone = 0;
static void init(void) { ... }

void mod_func(void) {
    if(initDone == 0) { /* Sync erforderlich? Warum, wie? */
        initDone = 1;
        init();
    }
```

  - `init` und `initDone` sind nicht Bestandteil der Schnittstelle!
- Initialisierung darf nicht mit anderen Modulen in Konflikt stehen!

U2-5 Aufgabe 3

- LED-Modul der SPiCboard-Bibliothek selbst implementieren
  - Konfiguration der Ports
  - LEDs ein-, aus- und umschalten
  - Pegelanzeige mit den LEDs
- Beliebiges Test-Programm, das alle Funktionen des LED-Moduls testet
- Die notwendige Schnittstelle ist der Online-Dokumentation zu entnehmen
- Die Anschlusspins der LEDs sowie deren Namen sind auf dem Board-Übersichtsbild gekennzeichnet
- Hilfsfunktionen/-variablen dürfen nicht Teil der Schnittstelle sein!
- Abgabe von **led.c** und **test.c**
- **led.h** muss nicht abgegeben werden (warum?)

1 Übersetzung des Programms

- Das Testprogramm kann auf Ihren Wunsch andere Module der SPiCboard-Bibliothek verwenden
- Übersetzung des Programms mit Ihrem eigenen LED-Modul

```
avr-gcc <FLAGS> -o test.elf test.c led.c -lspicboard
```

  - <FLAGS> sind hierbei die notwendigen Compiler-Optionen!
  - Der Linker hat zum Zeitpunkt des Durchsuchens der Bibliothek bereits alle Referenzen auf die Symbole des LED-Moduls aufgelöst
    - ☞ das LED-Modul wird nicht aus der Bibliothek hinzukopiert
    - ☞ andere Module der Bibliothek werden ggf. dazugebunden
- Testen mit unserer Referenzimplementierung wie gehabt

```
avr-gcc <FLAGS> -o test.elf test.c -lspicboard
```

- Ein geeignetes Makefile kann beide Varianten erzeugen (optional)