# U3 3. Übung

- Wiederholung: Zugriff auf AVR-Prozessor-Register
- Wiederholung: I/O-Ports
- Zahlensysteme
- Überblick: Modulare Softwareentwicklung
- Aufgabe 3: 7seg-Modul der SPiCboard-Bibliothek


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.



# U3-1 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`


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

## 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?


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

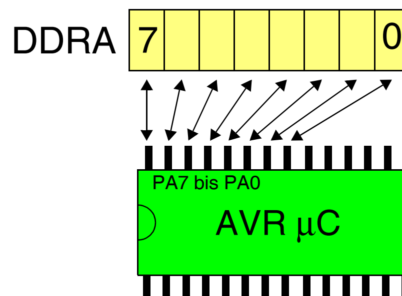


# U3-2 I/O-Ports des AVR-µC

## 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:

DDRA




- Ausführliche Beschreibung: ATMega32-Datenblatt


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

## 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


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.



## 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 */
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

# U3-3 Zahlensysteme

- 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
- Lesbarkeit Versus Aufwand


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.



## 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


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

# 2 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


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.



# 2 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)


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

## 3 Schreibweise

- Verschiedene Schreibweisen der selben Zuweisung

```
i =       42; //Dezimal
i =     0x2a; //Hexadezimal
i =      052; //Oktal
i = 0b101010; //Binär
```

## 4 Lesbarkeit versus Aufwand

- Setzen des ersten und dritten Bits, löschen des zweiten und siebten Bits

```
REG |= (1 << 1);
REG |= (1 << 3);
REG &= ~(1 << 2);
REG &= ~(1 << 7);
```

```
REG |= (1 << 1) | (1 << 3);
REG &= ~((1 << 2) | (1 << 7));
```

```
REG |= 0x0A;
REG &= ~0x84;
```

```
REG |= 10;
REG &= ~132;
```

```
REG |= 0x0A;
REG &= 0x7B;
```

```
REG = REG | (1 << 1) & ~(1 << 2) | (1 << 3) & ~(1 << 7);
```

```
REG = REG |    (1 << FUNC1) | (1 << FUNC3)
          & ~((1 << FUNC2) | (1 << FUNC7));
```

```
REG = REG | 0x0A & ~0x84;
```


© Moritz Strübe • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.



# U3-4 Überblick: Modulare Softwareentwicklung

- Ausführlichere Behandlung: Vorlesung
- 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


© Moritz Strübe • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

# 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


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.



# 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);

/* Irgendeine Variable */
extern uint8_t einevariable;

...
#endif
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

- 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


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.



## 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!


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

# U3-5 Aufgabe 3

- Teile des 7seg-Modul der SPiCboard-Bibliothek selbst implementieren
  - Konfiguration der Ports
  - LED-Elemente ein-, aus- und umschalten
- Beliebiges Test-Programm, das alle Funktionen des 7seg-Moduls testet
- Die notwendige Schnittstelle liegt im Pub-Verzeichnis
- Die Anschlusspins der LEDs sowie deren Namen sind auf dem Board-Übersichtsbild gekennzeichnet (Dokumentation)
- Hilfsfunktionen/-variablen dürfen nicht nach außen sichtbar sein!
- Abgabe von **7seg.c** und **test.c**
- **7seg.h** muss nicht abgegeben werden (warum?)


© Moritz Strübe • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.



## 1 Hinweise

- Die beiden sieben Segmentanzeigen sind parallel geschaltet um Pins zu sparen.
- Durch die Pins PD0 und PD1 kann zwischen den beiden Anzeigen hin und hergeschaltet werden.
- Die libspicboard verwendet für die Funktionen der timer.h einen Timer-Interrupt. Dieser sollte so selten wie möglich aufgerufen werden und so kurz wie möglich sein, damit das ’Hauptprogramm’ möglichst wenig beeinflusst wird.
- Trotzdem soll die Anzeige nicht flimmern.


© Moritz Strübe • Universität Erlangen-Nürnberg • Informatik 4, 2010
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.