U3 Grundlagen der AVR-Programmierung

- Datentypen mit fest definierter Größe
- Register
- I/O-Ports
- Aufgabe 2: LED-Modul

U3-1 Datentypen fester Größe

- Die Größe der primitiven Datentypen in C ist architekturabhängig
- Beispiel für drei Architekturen (Angaben in Bits)

	8-bit AVR	Intel x86-32	Intel x86-64
char	8	8	8
short	16	16	16
int	16	32	32
long	32	32	64

- Probleme
  - Portabilität des Quellcodes eingeschränkt
  - Insbesondere in der hardwarenahen Programmierung braucht man oft Datentypen einer bekannten, festen Größe (I/O-Register)

U3-1 Datentypen fester Größe

- C99 definiert neue Datentypen mit definierter Größe
- Auszug der wichtigsten Typen:

`int8_t`	`8-bit signed`
`uint8_t`	`8-bit unsigned`
`int16_t`	`16-bit signed`
`uint16_t`	`16-bit unsigned`
`int32_t`	`32-bit signed`
`uint32_t`	`32-bit unsigned`

- verfügbar durch Einbinden von **stdint.h**
  ☞ wird auch mit vielen nicht C99-konformen Compilern geliefert
  ☞ kann ansonsten auch relativ einfach selbst erstellt werden
- die Größe von Zeigertypen ist immer architekturabhängig (Adressbusbreite)

U3-2 Register beim AVR-µC

- Beim AVR-µC sind die Register
  - in den Daten-Adressraum eingeblendet
  - am Anfang des Adressbereichs angeordnet
- Adressen sind der Dokumentation zu entnehmen
- vollständige Dokumentation für unseren Mikrokontroller ATMega32:
  `http://www4.informatik.uni-erlangen.de/Lehre/SS09/V_SPIC/Uebung/doc/mega32.pdf`
- Für die Aufgaben benötigte Register sind auf den Folien erwähnt
- Die Bibliothek (avr-libc), die wir verwenden, definiert bereits sinnvolle Makros für alle Register des AVR µC
  (`#include <avr/io.h>`)

1 Makros für Register-Zugriffe

- Makros mit aussagekräftigen Namen können den Umgang mit Registern deutlich vereinfachen
- Beispiel:
  - Makro für Register an Adresse 0x3b (Port A beim ATmega32):

```
#define PORTA (*(volatile uint8_t *)0x3b)
```

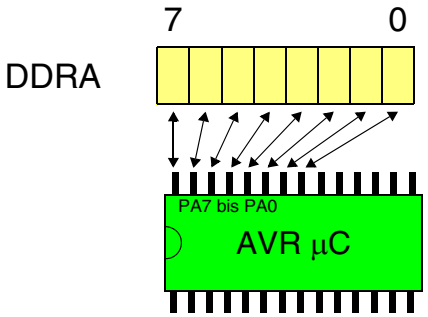
  - Verwenden dieses Registers:

```
volatile uint8_t *portPtr = &PORTA;
PORTA = 0;              /* schreibender Zugriff */
...
if (PORTA == 0x04)      /* lesender Zugriff */
    PORTA &= ~4;        /* lesender und schreibender Zugriff */
*portPtr |= 1;          /* Zugriff über Zeiger */
```

- Das `volatile`-Schlüsselwort wird später erläutert, im Moment ist es bei sämtlichen Zugriffen auf Hardwareregister zu verwenden.

U3-3 I/O-Ports des AVR-µC

- 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:



1 I/O-Port-Register

- **DDRx**: hier konfiguriert man einen Pin *i* von Port *x* als Ein- oder Ausgang
  - 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
- **PINx**: Bit *i* gibt den aktuellen Wert des Pin *i* von Port *x* an (nur lesbar)

2 Beispiel: Initialisierung eines Ports

- Pin 3 von Port B (PB3) als Ausgang konfigurieren und auf V_{cc} schalten:

```
DDRB |= 0x08;  /* PB3 als Ausgang nutzen... */
PORTB |= 0x08; /* ...und auf 1 (=high) setzen */
```

- Pin 0 von Port D (PD0) als Eingang nutzen, pull-up-Widerstand aktivieren und prüfen ob ein low-Pegel anliegt:

```
DDRD &= ~0x01; /* PD0 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 */
}
```

- Die Initialisierung der Hardware wird in der Regel **einmalig** zum Programmstart durchgeführt

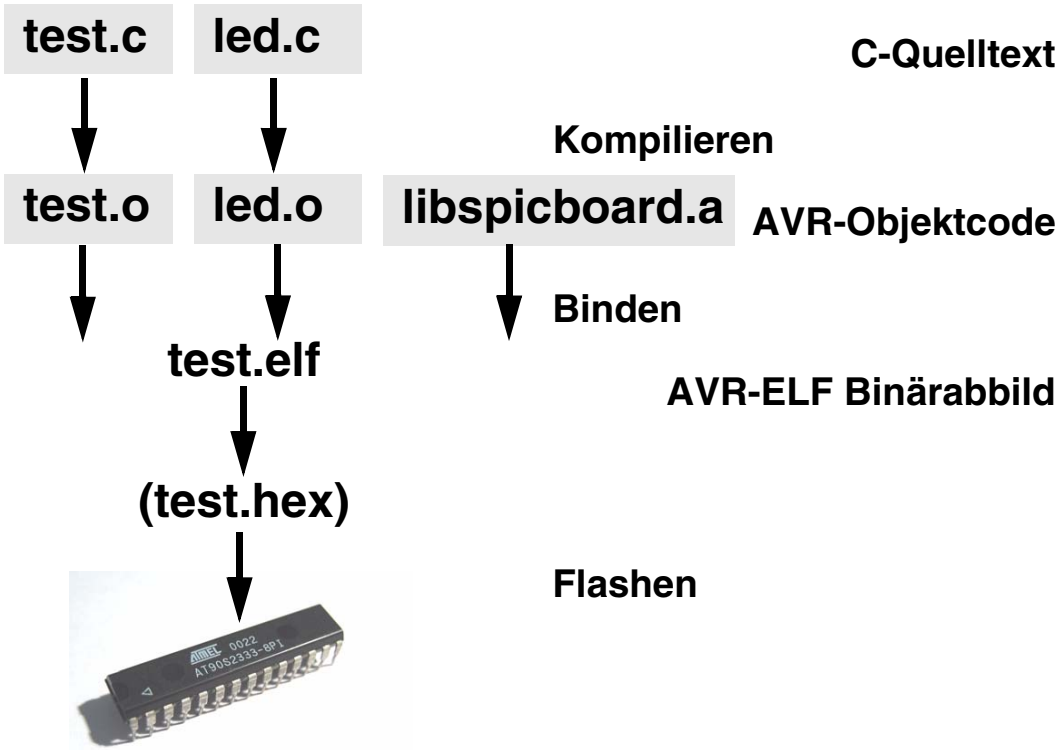
U3-4 Aufgabe 2: LED-Modul

- Das LED-Modul der SPiCboard-Bibliothek selbst implementieren
- Das eigene Modul dann mit einem Testprogramm linken
- Andere Teile der Bibliothek können für den Test benutzt werden

1 LEDs des SPiCboard

- Die Anschlüsse und Namen der einzelnen LEDs können dem Übersichtsbildchen entnommen werden
- Alle LEDs sind *active low*, d.h. leuchten wenn ein Low-Pegel auf dem Pin angelegt wird.
- `PD7` = Port D, Pin 7

2 Toolchain



3 AVR-Studio Projekteinstellungen

- Projekt wie gehabt anlegen
  - Projektname/Verzeichnis: `aufgabe2`
  - Initiale Quelldatei: `test.c`
  - alle sonstigen Einstellungen wie bisher
- Dann weitere Quelldatei `led.c` hinzufügen
- Wenn nun übersetzt wird, wird das eigene LED-Modul verwendet
- Andere Teile der Bibliothek werden nach Bedarf hinzugebunden
- Zum Test mit der Referenzimplementierung `led.c` aus der Liste der Quelldateien löschen

4 Übersetzen unter Linux

- Das vorgegebene Makefile kann nur Programme aus einem Modul bauen
  - geeignet zum Testen mit der Referenzimplementierung
  - ein evtl. vorhandenes eigenes LED-Modul wird nicht verwendet
  - `make -f /proj/i4spic/pub/i4/debug.mk test.hex.flash`
- Zum Testen des eigenen Moduls muss dieses vor Hinzubinden der Bibliothek zum Testprogramm gebunden werden
  - das entsprechende Kommando findet sich in einem gesonderten Makefile
  - `make -f /proj/i4spic/pub/aufgabe2/Makefile test.hex.flash`
- Achtung: Beim Wechsel zwischen obigen Varianten müssen zunächst die erzeugten Dateien gelöscht werden
  - `make -f /proj/i4spic/pub/aufgabe2/Makefile clean`