U5 4. Übungsaufgabe

- Grundlegendes zur Übung mit dem AVR-µC
- Register
- I/O Ports
- Interrupts
- AVR-Umgebung

U5-1 Grundlegendes zur Übung mit dem AVR-µC

- Die Übungsaufgaben werden wir mit Hilfe eines Simulators testen und entwickeln
  - ➤ der Simulator simuliert den Mikrocontroller **ATmega128**
- Wer möchte kann sein Programm auch auf einer echten Hardware testen
  - ➤ hier wird ein **ATmega8** eingesetzt
  - ➤ wurde für uns von Studenten der AGEE gebaut (DANKE!)
- Die beiden Plattformen unterscheiden sich geringfügig.
  - ➤ daher sind kleine Anpassungen nötig, um ein Programm welches auf dem Simulator läuft auf der echten Hardware einzusetzen
- Im Folgenden werden immer beide Varianten des AVR-µC angesprochen. Auf Unterschiede wird gelegentlich hingewiesen

U5-2 Register beim AVR-µC

1 Überblick

- Beim AVR µC sind die Register:
  - ◆ in den Speicher eingebettet
  - ◆ am Anfang des Adressbereichs angeordnet
- Adressen sind der Dokumentation zu entnehmen
- vollständige Dokumentation für "unsere" Mikrocontroller: `/proj/i4gdi/tools/doc/`
- Die für die Übungsaufgabe benötigten 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>`)

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)
```

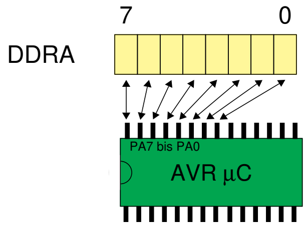
  - ◆ Verwenden dieses Registers:

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

U5-3 I/O-Ports des AVR µC

1 Überblick

- Jeder I/O-Port des AVR µC wird durch 3 (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:



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
    // ja ein low Pegel liegt an...
}
```


2 I/O Port Register

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

U5-4 Externe Interrupts des AVR µC

1 Flanken-/Pegel-Steuerung

- Externe Interrupts durch Pegeländerung an bestimmten I/O-Pins
  - ◆ ATmega128: 8 Quellen an den Pins PD0-PD3 und PE4 - PE7
  - ◆ ATmega8: 2 Quellen an den Pins PD2 und PD3
- Für jeden Interrupt kann bestimmt werden, ob er Pegel- oder Flanken-gesteuert ist
  - ➤ Steuerung für Interrupt *n* durch Interrupt Sense Control Bits (`ISCn0` und `ISCn1`)

ISC*n*1	ISC*n*0	IRQ bei:
0	0	low-Pegel
0	1	jedem Wechsel
1	0	fallender Flanke
1	1	steigender Flanke

1 Flanken-/Pegel-Steuerung (2)

- Je nach Variante des AVR-µC befinden sich die ISC-Bits in unterschiedlichen Registern:
  - ATmega128: External Interrupt Control Register A bzw. B (`EICRA` für INT 0-3, `EICRB` für INT 4-7).
  - ATmega8: MCU Control Register (`MCUCR`)
- Position der ISC-Bits in den Registern durch Makros definiert `ISCn0` und `ISCn1`
- Beispiel: `INT0` bei ATmega128 für fallende Flanke konfigurieren

```
// die ISCs für INT0 befinden sich im EICR A
EICRA &= ~(1<<ISC00);     // ISC00 löschen
EICRA |= (1<<ISC01);      // ISC01 setzen
```


2 Maskieren

- Alle Interruptquellen können separat ausgeschaltet (=maskiert) werden
- Für externe Interrupts sind folgende Register zuständig:
  - ATmega128: External Interrupt Mask Register (`EIMSK`)
  - ATmega8: General Interrupt Control Register (`GICR`)
- Die Bits in diesen Registern sind durch Makros `INTn` definiert
- Ein gesetztes Bit aktiviert den jeweiligen Interrupt
- Beispiel: Interrupt 0 aktivieren

```
EIMSK |= (1<<INT0);      // unmask Interrupt 0
```


2 Maskieren (2)

- Alle Interrupts können nochmals bei der CPU direkt abgeschaltet werden
  - ➤ durch speziellen Maschinenbefehl
- Die Bibliothek avr-libc bietet hierfür Funktionen an: (`#include <avr/interrupt.h>`)
  - `sei()` - lässt Interrupts zu
  - `cli()` - blockiert alle Interrupts
- Beispiel

```
#include <avr/interrupt.h>

sei();                  // IRQs zulassen
```

- Innerhalb eines Interrupt-Handlers sind automatisch alle Interrupts maskiert, beim Verlassen werden sie wieder demaskiert
- Nach dem Starten des µC sind die Interrupts bei der CPU abgeschaltet

3 Interrupt-Handler

- Installieren eines Interrupt-Handlers wird durch die verwendete Bibliothek unterstützt
- Makro `ISR` (Interrupt Service Routine) zur Definition einer Handler-Funktion (`#include <avr/interrupt.h>`)
- Als Parameter gibt man dem Makro den gewünschten Vektor an, z. B. `INT0_vect` für den externen Interrupt 0
- Beispiel: Handler für Interrupt 0 implementieren

```
#include <avr/interrupt.h>
volatile int zaehler;

ISR (INT0_vect) {
  zaehler++;
}
```

U5-5 AVR Umgebung

1 Compiler

- Um auf einem PC Programme für den AVR-Mikrocontroller zu erstellen, wird ein ***Cross-Compiler*** benötigt
  - ◆ AVR-Studio + WinAVR (Windows)
  - ◆ GNU gcc (Linux)
- Wir verwenden gcc: Alle benötigten Programme, Werkzeuge und Bibliotheken befinden sich in: `/proj/i4gdi/tools/`
  - ➤ z. B. der Cross-Compiler: `/proj/i4gdi/tools/bin/avr-gcc`
- Wichtiger Parameter: `-mmcu`, um den Werkzeugen die genaue Zielplattform mitzuteilen
  - ➤ Beispiel: ein Programm für den ATmega128 erstellen

```
/proj/i4gdi/tools/bin/avr-gcc -mmcu=atmega128 ampel.c -o ampel
```


2 AVR-Simulator

- Simuliert die AVR-CPU und die integrierte Peripherie auf der Entwicklungsplattform
- Einfaches Testen und Debuggen, ohne das Programm auf der echten Hardware installieren zu müssen
- Bietet Möglichkeiten den Zustand der Pins zu verändern und zu beobachten
- In Verbindung mit einem Debugger ist auch das zeilenweise Ausführen des Programmes möglich
- Mögliche Varianten:
  - ➤ integriert in AVR-Studio (Windows)
  - ➤ `simulavr` bzw. `simulavrxx` (Linux)

3 AVR-Simulator simulavrxx

- Simulator für den ATmega128
- Der Simulator befindet sich in: `/proj/i4gdi/tools/bin/`
- Aufruf des Simulators mit einer kleinen graphischen Anzeige der Peripherie für unsere Übungsaufgabe:
  `/proj/i4gdi/tools/bin/simulavr_gui` Programm-Name
- An den Simulator kann man einen Debugger koppeln
  - ◆ Simulator und Debugger starten:
    `/proj/i4gdi/tools/bin/simulavr_gui_debug` Programm-Name

4 Makefile

- Zur Vereinfachung wird unter
  `/proj/i4gdi/pub/aufgabe4/Makefile`
  ein kleines Makefile zur Verfügung gestellt
- Ein Makefile beschreibt wie ein Programm gebaut und ggf. gestartet wird
- Das Programm `make` liest dieses Makefile und baut das Programm entsprechend
- Kopieren sie sich das Makefile in ihr Verzeichnis
  `cp /proj/i4gdi/pub/aufgabe4/Makefile /proj/i4gdi/`loginname`/aufgabe4/`
- Ein Aufruf von `make` erstellt das Programm `ampel` für den ATmega128 falls ein `ampel.c` im selben Verzeichnis liegt
- `make sim` erstellt das Programm und startet es mit Hilfe des Simulators

U5-6 Ein einfaches Beispiel

- `bsp.c`: Die grüne LED einschalten.

```
#include <avr/io.h>

int main(void){

    DDRB |= 0xff;  //alle Pins von Port B als Ausgang nutzen
    PORTB = 0x01;  // ... und Pin 0 auf high setzen

    for (;;);
}
```

- Compilieren:

```
/proj/i4gdi/tools/bin/avr-gcc -mmcu=atmega128 bsp.c -o bsp
```

- Programm im Simulator starten

```
/proj/i4gdi/tools/bin/simulavr_gui bsp
```


U5-7 Übungsaufgabe: Peripherie

- Als externe Peripherie verwenden wir 4 LEDs und einen Taster die wie folgt angeschlossen sind:

	ATmega128 (Simulator)	ATmega8
Taster (Eingang)	Port D Pin 0	Port D Pin 2
LED grün	Port B Pin 0	
LED gelb	Port B Pin 1	
LED rot	Port B Pin 2	
LED blau	Port D Pin 7	

- Adressen der Port-Register im I/O-Adressbereich (identisch für ATmega128 und ATmega8)

Register	Adresse
DDRB	0x37
PORTB	0x38
PINB	0x36

Register	Adresse
DDRD	0x31
PORTD	0x32
PIND	0x30

1 Taster-Details

- Für den Taster muss der pull-up-Widerstand aktiviert werden
- Taster zieht den Pegel auf GND
- Durch den Taster kann Interrupt 0 ausgelöst werden