U2 2. Übung

- Wiederholung
- Aufgabe 2

U2-1 Wiederholung

1 Variablentypen

- Standardtypen sind in C nicht genau definiert:
  - `char` ist immer 8 Bit (bis auf einige Ausnahmen)
  - `short int` ≤ 16 Bit ≤ `int` ≤ `long int` ≤ `long long int`
  - 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

- Der genaue Wertebereich steht in der Header-Datei `limits.h`
  - z. B. `INT_MIN`, `INT_MAX` oder `UINT_MIN`, `UINT_MAX`

1 Variablentypen (2)

- Besser: Verwendung der Header-Datei `stdint.h`
  - `int16_t` 16 Bit mit Vorzeichen (*signed*):
    `INT16_MIN` (–32768) bis `INT16_MAX` (32767)
  - `uint16_t` 16 Bit ohne Vorzeichen (*unsigned*):
    `UINT16_MIN` (0) bis `UINT16_MAX` (65535)
  - Zeiger sind immer vom Adressbus der Architektur abhängig
    - AVR: 16 Bit
    - Aber: 8-Bitter -> Es kann immer nur ein Byte aus dem Speicher gelesen bzw. geschrieben werden
- Vorteile:
  - Wertebereich ist bekannt
  - Insbesondere in der hardwarenahen Programmierung braucht man oft Datentypen einer bekannten, festen Größe (I/O-Register)
  - Quellcode kann leichter auf andere µC portiert werden

2 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird
- Zwei Arten
  - Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
    - ➤ statische (`static`) Variablen
  - Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben
    - ➤ dynamische (`auto`) Variablen

2 Lebensdauer von Variablen (2)

auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
  - ➤ der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
    - ➥ der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
  - ➤ die Initialisierung wird bei jedem Eintritt in den Block wiederholt
  - !!! **Wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert** (= irgendwas)

2 Lebensdauer von Variablen (3)

static-Variablen

- Der Speicher für alle globalen Variablen ist generell von Programmstart bis Programmende reserviert
- Lokale Variablen erhalten bei Definition mit dem Schlüsselwort `static` eine **Lebensdauer über die gesamte Programmausführung** hinweg
  - ➥ der Inhalt bleibt bei Verlassen des Blocks erhalten und ist bei einem erneuten Eintreten in den Block noch verfügbar
  - !!! Das Schlüsselwort `static` hat bei globalen Variablen eine völlig andere Bedeutung (Einschränkung des Zugriffs auf das Modul)
- Static-Variablen können nur durch konstante Ausdrücke initialisiert werden
  - ➤ die Initialisierung wird nur einmal beim Programmstart vorgenommen (auch bei lokalen Variablen!)
    - erfolgt keine explizite Initialisierung, wird automatisch mit 0 vorbelegt

U2-2 Deklaration und Definition

- Compiler arbeiten den Quelltext von oben nach unten ab
- Deklaration
  - ◆ Das "Versprechen", dass es eine bestimmte Variable bzw. eine Funktion geben wird, die einen bestimmten Rückgabewert hat und bestimmte Parameter übergeben bekommt.

```
uint8_t meineFunktion(uint8_t w1, uint16_t w2);
```

- Definition
  - ◆ Die eigentliche Funktion

```
uint8_t meineFunktion(uint8_t w1, uint16_t w2) {
    /* Hier passiert was */
}
```

- Die Funktionen der libspicboard werden in Headerdateien deklariert.

U2-3 Optimierung durch den Compiler

- AVR-Mikrocontroller und verwandte CPUs können ihre Rechenoperationen nicht direkt auf Variablen ausführen, die im Speicher liegen
  - ◆ Stattdessen:
  - (1) Laden der Operanden aus dem Speicher in Prozessorregister
  - (2) Abarbeiten der Operationen in den Registern
  - (3) Zurückschreiben des Ergebnisses in den Speicher
- Der Compiler macht Annahmen, um den Code zu optimieren. Beispiele:
  - ◆ Variableninhalte sind beständig. Sie ändern sich nicht "von alleine".
  - ◆ Operationen, die den Zustand nicht ändern, können entfernt werden.

U2-3 Optimierung durch den Compiler (2)

- Typische Optimierungen:
  - Redundanter und "toter" Code wird weggelassen.
  - Die Reihenfolge des Codes wird umgestellt.
  - Für lokale Variablen wird kein Speicher reserviert; es werden stattdessen Prozessorregister verwendet.
  - Wenn möglich, übernimmt der Compiler die Berechnung:
    `a = 3 + 5;` wird zu `a = 8;`
  - Der Wertebereich von Auto-Variablen wird geändert:
    Statt von 0 bis 10 wird von 246 bis 256 (= 0 für `uint8_t`) gezählt und dann getestet, ob ein Überlauf stattgefunden hat.

- Codebeispiel

```
void wait(void) {
    uint8_t u8;
    while(u8 < 200) {
        u8++;
    }
}
```


U2-3 Code ohne Optimierung

- Codebeispiel ohne Optimierung:

```
;void wait(void){
; uint8_t u8;
; [Prolog (Register sichern, etc)]
    rjmp while;      Springe zu while
; u8++;
addone:
    ldd  r24, Y+1;  Lade Daten aus Y+1 in Register 24
    subi r24, 0xFF; Ziehe 255 ab (addiere 1)
    std  Y+1, r24;  Schreibe Daten aus Register 24 in Y+1
; while(u8 < 200)
while:
    ldd  r24, Y+1;  Lade Daten aus Y+1 in Register 24
    cpi  r24, 0xC8; Vergleiche Register 24 mit 200
    brcs addone;    Wenn kleiner, dann springe zu addone
;[Epilog (Register wiederherstellen)]
    ret;            Kehre aus der Funktion zurück
}
```


U2-3 Code mit Optimierung

- Codebeispiel mit Optimierung:

```
; void wait(void){
    ret;            Kehre aus der Funktion zurück
; }
```

- Die Schleife hat keine Auswirkung auf den Zustand und wird deswegen komplett wegoptimiert.
- Und wenn wir die Schleife eigentlich als Warteschleife einsetzen wollten...?
- Lösung: Variable als *volatile* (engl. unbeständig, flüchtig) deklarieren
  - Für Variablen bedeutet dies: Sie müssen immer in den Speicher gelegt und vor und nach jeder Operation mit diesem synchronisiert werden; ihr Wertebereich darf nicht geändert werden.

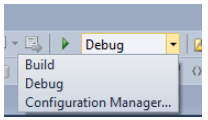
- Einsatzmöglichkeiten von *volatile*:
  - Warteschleifen

```
void wait(void){
    volatile uint8_t u8;
    while(u8 < 200){
        u8++;
    }
}
```

  - Zugriff auf Hardware (z. B. Pins): Wird in einer der nächsten TÜ besprochen.
  - Debuggen; der Wert wird nicht wegoptimiert.

U2-4 Neues SPiC-Template

- Importieren des I4-Projekts (einmalig)
  - File -> Import Project Template
  - Q:\tools\AVRStmpl_mO.zip (mit Optimierung)
  - Add to Folder: <Root>
  - OK
- Ab sofort bei neuen Projekten immer "SPiC mit Optimierung" wählen
- Es gibt jetzt zwei "Build-Targets"
  - Debug: Wie bisher ohne Optimierung
  - Build: Mit Optimierung



- Alle Abgaben müssen auch mit Optimierung funktionieren.

U2-5 Aufgabe 2: snake

- Schlange bestehend aus benachbarten LEDs
- Länge 1 bis 5 LEDs, regelbar mit Potentiometer (POTI)
- Geschwindigkeit abhängig von der Umgebungshelligkeit (je heller, desto schneller)
- Bewegungsrichtung umschaltbar mit Taster

U2-6 Parameter der Schlange

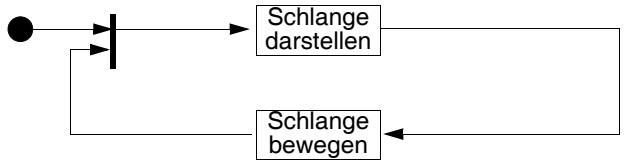
- Position des Kopfes
  - Nummer einer LED
  - Wertebereich [0; 7]
- Länge der Schlange
  - Ganzzahl im Bereich [1;5]
- Richtung der Schlange
  - aufwärts oder abwärts
  - z. B. 0 oder 1
- Geschwindigkeit der Schlange
  - hier: Durchlaufzahl der Warteschleife

U2-7 Zerlegung in Teilprobleme

- Basisablauf: Welche Schritte wiederholen sich immer wieder?
- Teilprobleme *können* in eigene Funktionen ausgelagert werden
- Wiederkehrende Teilprobleme *sollten* in Funktionen ausgelagert werden
- Welcher Zustand muss über Basisabläufe hinweg erhalten bleiben?
  - Ist der Zustand gegebenenfalls nur für ein Teilprobleme relevant?
  - Sichtbarkeit dann auf das Teilproblem einschränken (Kapselung)

U2-7 Basisablauf

- Darstellung der Schlange
- Bewegung der Schlange



SPiC - Ü
Systemnahe Programmierung in C — Übungen
© Moritz Strübe, Michael Stilkerich • Universität Erlangen-Nürnberg • Informatik 4, 2011
U2.fm 2011-05-17 17.23
U2.1
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

1 Darstellung der Schlange

- Bestimmung der Darstellungsparameter
  - Kopfposition
  - Länge
  - Richtung
- Anzeige der Schlange abhängig von den Parametern
  - Aktivieren der zur Schlange gehörenden LEDs
  - Deaktivieren der restlichen LEDs

SPiC - Ü
Systemnahe Programmierung in C — Übungen
© Moritz Strübe, Michael Stilkerich • Universität Erlangen-Nürnberg • Informatik 4, 2011
U2.fm 2011-05-17 17.23
U2.1
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Bewegung der Schlange

- Bestimmung der Bewegungsparameter
  - Geschwindigkeit
  - Richtung
- Bewegen der Schlange
  - Anpassen der Kopfposition abhängig von der Richtung
- Wartepause abhängig von der Geschwindigkeit
- gegebenenfalls Richtungsänderung
  - bisheriger Schlangenschwanz wird zum Schlangenkopf

SPiC - Ü
Systemnahe Programmierung in C — Übungen
© Moritz Strübe, Michael Stilkerich • Universität Erlangen-Nürnberg • Informatik 4, 2011
U2.fm 2011-05-17 17.23
U2.2
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

U2-8 Ablaufplan für Teilproblem Schlangenanzeige

- Einen Wertebereich reduzieren mit dem Modulo-Operator: %
- *Modulo* ist der Divisionsrest einer Ganzzahldivision.
- b = a % 4;

a	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
b	-1	0	-3	-2	-1	0	1	2	3	0	1	2

U2-9 Bestimmung der Richtungsänderung

- Der Taster muss periodisch abgefragt werden
  - ◆ man bekommt nur eine Momentaufnahme des Tasters
- eventuell wird ein Tastendruck verpasst
  - ◆ Hochfrequente Abfrage zur Minimierung des Risikos
  - ◆ Wo im Programm ist dies sinnvoll möglich?
    - in der Warteschleife (jedem Durchlauf)
    - hier wird die meiste Zeit verbracht
- mehrfache Abfrage des Tasterzustands während eines Tasterdrucks
  - ◆ Mehrfachinterpretation eines Tasterdrucks vermeiden
  - ◆ Loslassen des Tasters muss explizit registriert werden
    - Historie über den vorigen Tasterzustand mitführen (Flankenerkennung)