

# Übungen zu Systemnahe Programmierung in C (SPiC)

Peter Wägemann, Sebastian Maier, Heiko Janker  
(Lehrstuhl Informatik 4)



Sommersemester 2015



## Inhalt

---

Verwendung von int

Compileroptimierung

- Nutzen

- Beispiel

- Schlüsselwort volatile

Sichtbarkeit & Lebensdauer

Typdefs & Enums

Aufgabe 2: Snake

- Hinweise

- Beschreibung der Schlange

- Zerlegung in Teilprobleme

- Flankendetektion ohne Interrupts



Verwendung von int

Compileroptimierung

Sichtbarkeit & Lebensdauer

Typdefs & Enums

Aufgabe 2: Snake



## Verwendung von int

---

- Die Größe von `int` ist nicht genau definiert (ATmega32: 16 bit)  
⇒ Gerade auf  $\mu\text{C}$  führt dies zu Fehlern und/oder langsameren Code
- Für die Übung:
  - Verwendung von `int` ist ein “Fehler”
  - Stattdessen: Verwendung der in der `stdint.h` definierten Typen:  
`int8_t`, `uint8_t`, `int16_t`, `uint16_t`, etc.
- Wertebereich:
  - `limits.h`: `INT8_MAX`, `INT8_MIN`, ...
- Speicherplatz ist sehr teuer auf  $\mu\text{C}$   
⇒ Nur so viel Speicher verwenden, wie tatsächlich benötigt wird!



Verwendung von int

Compileroptimierung

Nutzen

Beispiel

Schlüsselwort volatile

Sichtbarkeit & Lebensdauer

Typdefs & Enums

Aufgabe 2: Snake



## Compileroptimierung: Hintergrund

---

- AVR-Mikrocontroller, sowie die allermeisten CPUs, können ihre Rechenoperationen nicht direkt auf Variablen ausführen, die im Speicher liegen
  - Ablauf von Operationen:
    1. **Laden** der Operanden aus dem Speicher in Prozessorregister
    2. **Ausführen** der Operationen in den Registern
    3. **Zurückschreiben** des Ergebnisses in den Speicher
- ⇒ Detaillierte Behandlung in der Vorlesung
- Der Compiler darf den Code nach Belieben ändern, solange der "globale" Zustand beim Verlassen der Funktion gleich bleibt
  - Optimierungen können zu drastisch schnellerem Code führen



## ■ Typische Optimierungen:

- Beim Betreten der Funktion wird die Variable in ein Register geladen und beim Verlassen in den Speicher zurückgeschrieben
- Redundanter und "toter" Code wird weggelassen
- Die Reihenfolge des Codes wird umgestellt
- Für automatic Variablen wird kein Speicher reserviert; es werden stattdessen Prozessorregister verwendet
- Wenn möglich, übernimmt der Compiler die Berechnung (Konstantenfaltung):  
a = 3 + 5; wird zu a = 8;
- Der Wertebereich von automatic Variablen wird geändert:  
Statt von 0 bis 10 wird von 246 bis 256 ( = 0 für uint8\_t ) gezählt und dann geprüft, ob ein Überlauf stattgefunden hat



## Compileroptimierung: Beispiel (1)

```
1 void wait(void) {  
2     uint8_t u8 = 0;  
3     while(u8 < 200) {  
4         u8++;  
5     }  
6 }
```

- Inkrementieren der Variable u8 bis 200
- Verwendung z.B. für aktive Warteschleifen



# Compileroptimierung: Beispiel (2)

## ■ Assembler ohne Optimierung

```
1 ; void wait(void){
2 ; uint8_t u8;
3 ; [Prolog (Register sichern, Y initialisieren, etc)]
4 rjmp while ; Springe zu while
5 ; u8++;
6 addone:
7 ldd r24, Y+1 ; Lade Daten aus Y+1 in Register 24
8 subi r24, 0xFF ; Ziehe 255 ab (addiere 1)
9 std Y+1, r24 ; Schreibe Daten aus Register 24 in Y+1
10 ; while(u8 < 200)
11 while:
12 ldd r24, Y+1 ; Lade Daten aus Y+1 in Register 24
13 cpi r24, 0xC8 ; Vergleiche Register 24 mit 200
14 brcs addone ; Wenn kleiner, dann springe zu addone
15 ;[Epilog (Register wiederherstellen)]
16 ret ; Kehre aus der Funktion zurück
17 ;}
```



# Compileroptimierung: Beispiel (3)

## ■ Assembler mit Optimierung

```
1 ; void wait(void){
2 ret ; Kehre aus der Funktion zurück
3 ; }
```

- Die Schleife hat keine Auswirkung auf den Zustand

~> Die Schleife wird komplett wegoptimiert



# Schlüsselwort volatile

- Variable können als `volatile` (engl. unbeständig, flüchtig) deklariert werden
- ↪ Der Compiler darf die Variable nicht optimieren:
  - Für die Variable muss **Speicher reserviert** werden
  - Die **Lebensdauer** darf nicht verkürzt werden
  - Die Variable muss vor jeder Operation aus dem **Speicher geladen** und danach gegebenenfalls wieder in diesen zurückgeschrieben werden
  - Der **Wertebereich** der Variable darf nicht geändert werden
- Einsatzmöglichkeiten von `volatile`:
  - Warteschleifen: Verhinderung der Optimierung der Schleife
  - nebenläufigen Ausführungen (später in der Vorlesung)
    - Variable wird im Interrupthandler und in der Hauptschleife verwendet
    - Änderungen an der Variable müssen "bekannt gegeben werden"
  - Zugriff auf Hardware (z. B. Pins) ↪ wichtig für das LED Modul
  - Debuggen: der Wert wird nicht wegoptimiert



## Übersicht: Sichtbarkeit & Lebensdauer

		SB Sichtbarkeit LD Lebensdauer	nicht static	static
Variablen	lokal (auto)		Block SB / LD Block	Block SB / LD Programm
	global		Programm SB / LD Programm	Modul SB / LD Programm
Funktionen			SB Programm	SB Modul

- Lokale Variable = auto Variable (automatisch allokiert & freigegeben)
- Funktionen als `static`, wenn kein Export notwendig



# Globale Variablen

```
1 static uint8_t state; // global static
2 uint8_t event_counter; // global
3
4 void main(void) {
5     ...
6 }
7
8 static void f(uint8_t a) {
9     static uint8_t call_counter = 0; // auto static
10    uint8_t num_leds; // auto
11    ...
12 }
```

- Sichtbarkeit/Gültigkeit möglichst weit **einschränken**
  - Globale Variable  $\neq$  lokale Variable in der main()
  - Globale static Variablen: Sichtbarkeit auf Modul beschränken
- ↪ static bei Funktionen und globalen Variablen verwenden, wo möglich



# Typdefs & Enums

```
1 #define PD3 3
2 typedef enum { BUTTON0 = 4, BUTTON1 = 8
3 } BUTTON;
4 #define MAX_COUNTER 900
5 ...
6 void main(void) {
7     ...
8     PORTB |= (1 << PB3); // nicht (1 << 3)
9     ...
10    BUTTONEVENT old, new; // nicht uint8_t old, new;
11    ...
12    // Deklaration: BUTTONEVENT sb_button_getState(BUTTON btn);
13    old = sb_button_getState(BUTTON0); // nicht sb_button_getState(4)
14    ...
15 }
```

- Vordefinierte Typen verwenden
- Explizite Zahlenwerte nur verwenden, wenn notwendig



Verwendung von int

Compileroptimierung

Sichtbarkeit & Lebensdauer

Typdefs & Enums

Aufgabe 2: Snake

Hinweise

Beschreibung der Schlange

Zerlegung in Teilprobleme

Flankendetektion ohne Interrupts



## Aufgabe 2: Snake

---

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



- Variablen in Funktionen verhalten sich weitgehend wie in Java
  - ↪ Zur Lösung der Aufgabe sind lokale Variablen ausreichend
- Der C-Compiler liest Dateien von oben nach unten
  - ↪ Legen Sie die Funktionen in der folgenden Reihenfolge an:
    1. wait()
    2. drawsnake()
    3. main()

⇒ Details zum Kompilieren werden in der Vorlesung besprochen.



## Beschreibung 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



# Zerlegung in Teilprobleme

- Basisablauf: Welche Schritte wiederholen sich immer wieder?
  - Wiederkehrende Teilprobleme sollten in eigene Funktionen ausgelagert werden
  - Vermeidung der Duplikation von Code
  - Welcher Zustand muss über Basisabläufe hinweg erhalten bleiben?
    - Ist der Zustand nur für ein Teilproblem relevant?
    - Sichtbarkeit auf das Teilproblem einschränken
- **Kapselung** soweit wie möglich



## Basisablauf

- Darstellung der Schlange
- Bewegung der Schlange
- Pseudocode:

```
1 void main(void) {
2   while(1) {
3     // berechne Laenge
4     laenge = ...
5
6     // zeichne Schlange
7     drawSnake(kopf, laenge, richtung);
8
9     // veraendere Schlangenkopf, abhaengig von Richtung
10    ...
11
12  } // Ende der Hauptschleife
13 }
```



# Darstellung der Schlange

---

- Darstellungsparameter

- Kopfposition
- Länge
- Richtung

- Funktionssignatur:

```
void drawSnake(uint8_t head, uint8_t length, uint8_t direction)
```

- Anzeige der Schlange abhängig von den Parametern

- Aktivieren der zur Schlange gehörenden LEDs
- Deaktivieren der restlichen LEDs



# 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



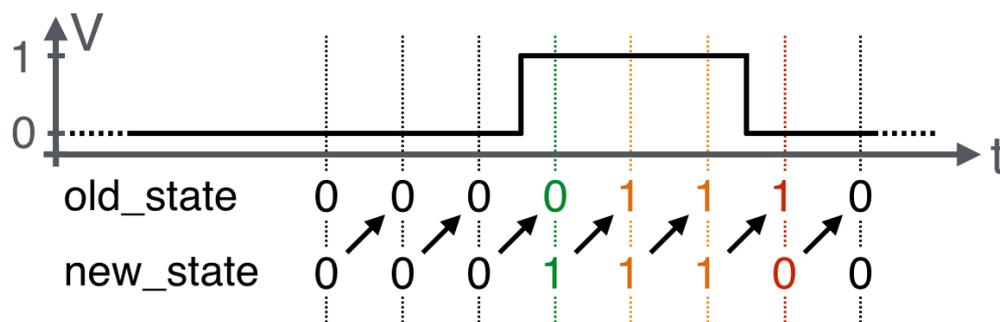
# Verwendung von Modulo

- Modulo ist der Divisionsrest einer Ganzzahldivision
- **Achtung:** In C ist das Ergebnis im negativen Bereich auch negativ
- Beispiel:  $b = a \% 4;$

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



# Flankendetektion ohne Interrupts



- Detektion der Flanke durch aktives, **zyklisches Abfragen** (engl. Polling) eines Pegels
- Später im Geschicklichkeitsspiel: Realisierung durch Interrupts

