

Übungen zu Systemnahe Programmierung in C (SPiC)

Moritz Strübe, Rainer Müller
(Lehrstuhl Informatik 4)



Sommersemester 2014



Inhalt

Aufgabe 2: Snake

Hinweise

Zerlegung in Teilprobleme

Verwendung von int

Compileroptimierung

Beispiel

Volatile



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



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



Verwendung von `int`

- Die Größe von `int` ist nicht genau definiert (> 16 bit)
⇒ Gerade auf μC 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`, ...
- ~> Vorlesung 6-4



Compileroptimierung: Hintergrund

- AVR-Mikrocontroller, sowie die allermeisten 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
⇒ Detaillierte Behandlung in der Vorlesung
- Der Compiler darf den Code nach Belieben ändern, solange der "globale" Zustand beim Verlassen der Funktion (auch Aufruf einer anderen Funktion) gleich bleibt



■ 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:
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 }
```



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 und wird deswegen komplett wegoptimiert.



- 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 geschrieben werden
 - Der Wertebereich der Variable darf nicht geändert werden
- Einsatzmöglichkeiten von volatile:
 - Warteschleifen
 - Zugriff auf Hardware (z. B. Pins): Wird in einer der nächsten TÜ besprochen
 - Debuggen; der Wert wird nicht wegoptimiert

