

Übungen zu Systemnahe Programmierung in C

Abschnitt 2.1: Compileroptimierung

27.04.2020

Tim Rheinfels
Benedict Herzog
Bernhard Heinloth

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



- 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)
 - Der Wertebereich von automatic Variablen wird geändert:



```
01 void wait(void) {  
02     uint8_t u8 = 0;  
03     while(u8 < 16) {  
04         u8++;  
05     }  
06 }
```

- Inkrementieren der Variable u8 bis 16
- Verwendung z.B. für aktive Warteschleifen



■ Assembler ohne Optimierung

```
01 ; void wait(void){
02 ; uint8_t u8;
03 ; [Prolog (Register sichern, Y initialisieren, etc.)]
04 rjmp while      ; Springe zu while
05 ; u8++;
06 addone:
07 ldd r24, Y+1     ; Lade Daten aus Y+1 in Register 24
08 subi r24, 0xFF   ; Ziehe 255 ab (addiere 1)
09 std Y+1, r24     ; Schreibe Daten aus Register 24 in Y+1
10 ; while(u8 < 16)
11 while:
12 ldd r24, Y+1     ; Lade Daten aus Y+1 in Register 24
13 cpi r24, 0x10    ; Vergleiche Register 24 mit 16
14 brcs addone     ; Wenn kleiner, dann springe zu addone
15 ;[Epilog (Register wiederherstellen)]
16 ret             ; Kehre aus der Funktion zurück
17 ;}
```



■ Assembler mit Optimierung

```
01 ; void wait(void){  
02 ret          ; Kehre aus der Funktion zurück  
03 ; }
```



■ Assembler mit Optimierung

```
01 ; void wait(void){  
02 ret          ; Kehre aus der Funktion zurück  
03 ; }
```

- Die Schleife hat keine Auswirkung auf den Zustand
- ↪ Die Schleife wird 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 ggf. 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 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)