

Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2018

Übung 2

Benedict Herzog
Sebastian Maier

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



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Variablen



- Die Größe von `int` ist nicht genau definiert
 - zum Beispiel beim ATMEGA328PB: 16 bit
 - ⇒ Gerade auf μC führt dies zu Fehlern und/oder langsameren Code
 - Für die Übung gilt
 - 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 μC
(SPICBOARD/ATMEGA328PB hat nur 2048 Byte SRAM)
- ↪ Nur so viel Speicher verwenden, wie tatsächlich benötigt wird!

Sichtbarkeit und Lebensdauer	nicht static	static
lokale Variable	Sichtbarkeit Block Lebensdauer Block	Sichtbarkeit Block Lebensdauer Programm
globale Variable	Sichtbarkeit Programm Lebensdauer Programm	Sichtbarkeit Modul Lebensdauer Programm
Funktion	Sichtbarkeit Programm	Sichtbarkeit Modul

- Lokale Variablen, die **nicht** static deklariert werden:
 - ↪ auto Variable (automatisch allokiert & freigegeben)
- Funktionen als static, wenn kein Export notwendig



```
01 static uint8_t state; // global static
02 uint8_t event_counter; // global
03
04 void main(void) {
05     /* ... */
06 }
07
08 static void f(uint8_t a) {
09     static uint8_t call_counter = 0; // local static
10     uint8_t num_leds; // local (auto)
11     /* ... */
12 }
```

- Sichtbarkeit/Gültigkeit möglichst weit **einschränken**
 - Globale Variable $\neq$ lokale Variable in `f()`
 - Globale `static` Variablen: Sichtbarkeit auf Modul beschränken
- wo möglich, `static` für Funktionen und Variablen verwenden



```
01 #define PB3 3
02 typedef enum {
03     BUTTON0 = 0,
04     BUTTON1 = 1
05 } BUTTON;
06
07 void main(void) {
08     /* ... */
09     PORTB |= (1 << PB3); // nicht (1 << 3)
10
11     BUTTONSTATE old, new; // nicht uint8_t old, new;
12
13     // Deklaration: BUTTONSTATE sb_button_getState(BUTTON btn);
14     old = sb_button_getState(BUTTON0); // nicht
        ↳ sb_button_getState(0)
15     /* ... */
16 }
```

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

Compileroptimierung



- 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

```
01 void wait(void) {  
02     uint8_t u8 = 0;  
03     while(u8 < 200) {  
04         u8++;  
05     }  
06 }
```

- Inkrementieren der Variable u8 bis 200
- 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 < 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 ;}
```

■ 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 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

Bits & Bytes

■ Übersicht:

$\&$	0	1
0	0	0
1	0	1

$ $	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	1
1	1	0

$\sim$	
0	1
1	0

■ Übersicht:

&	0	1
0	0	0
1	0	1

	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

~	
0	1
1	0

■ Beispiel:

	1100	1100	1100
~	&		^
1001	1001	1001	1001
0110	1000	1101	0101

■ Beispiel:

```
uint8_t x = 0x9d;
```

1	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---

```
x <<= 2;
```

0	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---

```
x >>= 2;
```

0	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---

■ Setzen von Bits:

```
(1 << 0)
```

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

```
(1 << 3)
```

0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---

```
(1 << 3) | (1 << 0)
```

0	0	0	0	1	0	0	1
---	---	---	---	---	---	---	---

■ Achtung:

Bei signed-Variablen ist das Verhalten des >>-Operators nicht 100% definiert. Im Normalfall(!) werden bei negativen Werten 1er geshiftet.

Aufgabe: 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
- Modus der Schlange mit Taster (BUTTON0) umschaltbar
 - Normalfall: Helle Schlange auf dunklem Grund
→ leuchtende LEDs repräsentieren die Schlange
 - Invertiert: Dunkle Schlange auf hellem Grund
→ inaktive LEDs repräsentieren die Schlange

⇒ Bearbeitung in Zweier-Gruppen: submit fragt nach Partner



- 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.

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

- 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



- Darstellung der Schlange
- Bewegung der Schlange
- Pseudocode:

```
01 void main(void) {  
02     while(1) {  
03         // berechne Laenge  
04         laenge = ...  
05  
06         // zeichne Schlange  
07         drawSnake(kopf, laenge, modus);  
08  
09         // setze Schlangenkopf weiter  
10         ...  
11  
12     } // Ende der Hauptschleife  
13 }
```

- Darstellungsparameter
 - Kopfposition
 - Länge
 - Modus
- Funktionssignatur: `void drawSnake(uint8_t head, ↪ uint8_t length, uint8_t modus)`
- Anzeige der Schlange abhängig von den Parametern
 - Normaler Modus (Helle Schlange):
 - Aktivieren der zur Schlange gehörenden LEDs
 - Deaktivieren der restlichen LEDs
 - Invertierter Modus (Dunkle Schlange):
 - Deaktivieren der zur Schlange gehörenden LEDs
 - Aktivieren der restlichen LEDs

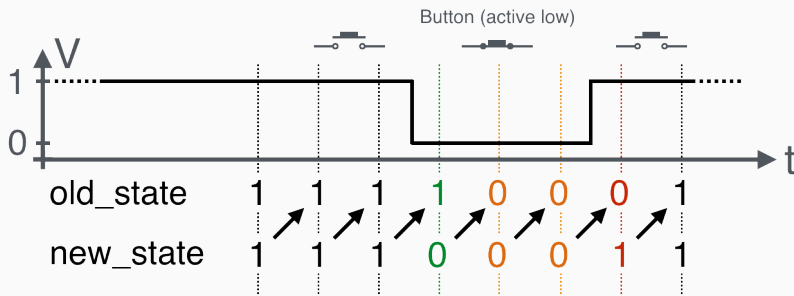


- Bestimmung der Bewegungsparameter
 - Geschwindigkeit
 - Länge
 - Modus
- Bewegen der Schlange
 - Anpassen der Kopfposition
 - Ggf. an den Anfang der LED Leiste zurückspringen
- Wartepause abhängig von der Geschwindigkeit
- gegebenenfalls Modusänderung



- 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



- Detektion der Flanke durch aktives, **zyklisches Abfragen** (engl. Polling) eines Pegels
- Unterscheidung zwischen **active-high** & **active-low** notwendig
- Später: Realisierung durch Interrupts

Hands-on: Signallampe

- Morsesignale über LED o ausgeben
- Steuerung über Taster 1
- Nutzung der Bibliotheksfunktionen für Button und LED
- Dokumentation der Bibliothek:
https://www4.cs.fau.de/Lehre/SS18/V_SPIC/Uebung/doc