

Übungen zu Grundlagen der systemnahen Programmierung in C (GSPIC) im Sommersemester 2018

2018-04-23

Bernhard Heinloth

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



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



Variablen

Verwendung von int

- 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 & Lebensdauer

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 Variable, nicht static: 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 ≠ lokale Variable in f()
 - Globale static Variablen: Sichtbarkeit auf Modul beschränken
- wo möglich, static für Funktionen und Variablen verwenden

Bits & Bytes

```

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
15     ↪ sb_button_getState(0)
16     /* ... */
17 }
    
```

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

3

4

Bitoperationen

- 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

■ Übersicht:

&	0	1		0	1	^	0	1	~	
0	0	0	0	0	1	0	0	1	0	1
1	0	1	1	1	1	1	1	0	1	0

■ Beispiel:

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

Aufgabe: snake

■ Beispiel:

```
uint8_t x = 0x9d;
x <<= 2;
x >>= 2;
```

1	0	0	1	1	1	0	1
0	1	1	1	0	1	0	0
0	0	0	1	1	1	0	1

■ Setzen von Bits:

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

0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	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 geschiftet.

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.

8

- 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

9

- 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

10

- 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 }

```

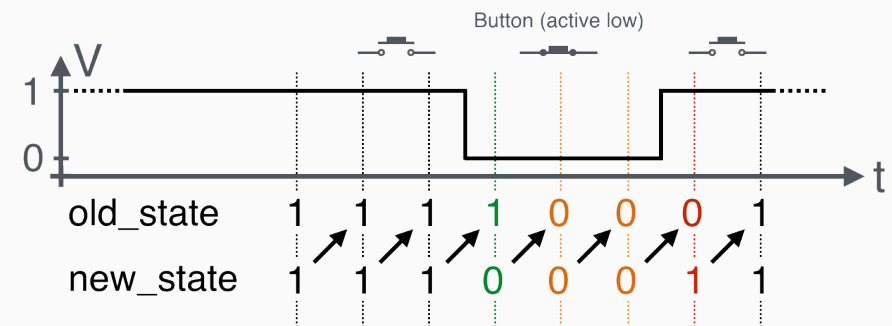
11

- 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
- Wartezeit 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