Übungen zu Systemnahe Programmierung in C (SPiC)

Peter Wägemann, Heiko Janker, Moritz Strübe, Rainer Müller
(Lehrstuhl Informatik 4)

Wintersemester 2014/2015

Inhalt

Inhalt

Aufgabe: Interrupt-Zähler

Besprechung einer Abgabe

Aufgabe: Interrupt-Zähler

- Verwendung von volatile
- Lost-Update: Zugriff auf 16-Bit Variable
- Lost-Wakeup: Position vom sei()
- Siehe später ⇝ Minimalbeispiel
- Speicherverwaltung für Alarme:
  - Verwaltung von auf Heap allozierten Variablen
  - Intern verwendet: malloc()/ free()

```
1 ALARM* a = sb_timer_setAlarm(my_callback_function,
2                               time,
3                               repeat);
4 ...
5 sb_timer_cancelAlarm(a);
```

- **Ressourcen wieder freigegeben, sonst Stackoverflow!**
- my_callback_function wird aus dem **Interrupt-Kontext** aufgerufen

Inhalt

PORT- und PIN-Array

- Port Definition

```
1 #define PORTD (* (volatile unsigned char *) 0x2B)
```

- Adressoperator: &
- Dereferenzierungsoperator: *
- Port Array:

```
1 static volatile uint8_t * const ports[] = { &PORTD,
2                                             &PORTC,
3                                             ... };
```

- Pin Array:

```
1 static uint8_t const pins[] = { PD7, PC0, PC1, ... };
```

- Zugriff:

```
1 * (ports[led]) &= ~(1<< pins[led]);
```

const uint8_t* vs. uint8_t* const

- const uint8_t*:
  - ein Pointer auf einen uint8_t-**Wert**, der konstant ist
  - Wert nicht über den Pointer veränderbar
- uint8_t* const
  - ein **konstanter Pointer** auf einen (beliebigen) uint8_t-Wert
  - Pointer darf nicht mehr auf eine andere Speicheradresse zeigen

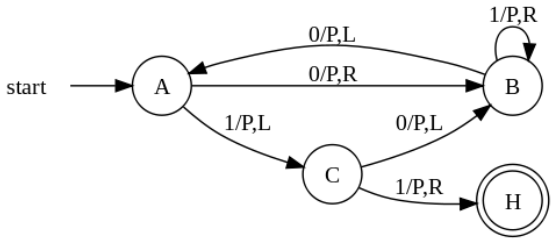
Inhalt

Zerlegung in Teilprobleme

- Basisablauf: Welche Schritte wiederholen sich immer wieder?
- Teilprobleme können in eigene Funktionen ausgelagert werden
- Wiederkehrende Teilprobleme sollten in Funktionen ausgelagert werden
- Welcher Zustand muss über Basisabläufe hinweg erhalten bleiben?
  - Ist der Zustand gegebenenfalls nuRessourcenr für ein Teilproblem relevant?
  - Sichtbarkeit dann auf das Teilproblem einschränken (Kapselung)

Zustandsmaschine

- Beispiel einer Zustandsmaschine:



- Jedem Zustand in einen Block kapseln
- switch-Anweisung
  - case-Block bearbeitet einen Zustand
  - break-Anweisung nicht vergessen

Aufgabe: Ampel

- Keine Verwendung von sb_timer_delay(uint16_t waittime)
- Alarme explizit setzten:
  ALARM* sb_timer_setAlarm(alarmcallback_t callback, uint16_t alarm
- Alarm wird auf dem Heap alloziert:
  ⇝ void sb_timer_cancelAlarm(ALARM*) nicht vergessen

Inhalt

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`, ...

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
- Der Compiler darf den Code nach Belieben ändern, solange der "globale" Zustand beim Verlassen der Funktion (auch Aufruf einer anderen Funktion) gleich bleibt

Compileroptimierung: Beispiele

- 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.

Volatile

- Variable können als volatile (engl. unbeständig, flüchtig) deklariert werden

⇝ Der Compiler (C) darf die Variable nicht Optimieren:

- C. muss für die Variable **Speicher reservieren**
- C. darf **Lebensdauer nicht verkürzen**
- Die Variable muss vor jeder Operation **aus dem Speicher geladen** und danach gegebenenfalls wieder in diesen **zurück geschrieben** werden
- Der **Wertebereich darf nicht geändert werden**

- Einsatzmöglichkeiten von volatile:
  - Warteschleifen
  - Zugriff auf Hardware (z. B. Pins)
  - Debuggen; der **Wert wird nicht wegoptimiert**
  - **Nebenläufige** Verwendung von Variablen

Wiederholung: 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