

# Hardwarenahe Programmierung

## Musterlösung zu den Übungsaufgaben – 17. Dezember 2018

### Aufgabe 1: Fakultät

Die Fakultät  $n!$  einer ganzen Zahl  $n \geq 0$  ist definiert als:

$$\begin{aligned} &1 \quad \text{für } n = 0, \\ &n \cdot (n - 1)! \quad \text{für } n > 0. \end{aligned}$$

Mit anderen Worten:  $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ .

Die folgende Funktion `fak()` berechnet die Fakultät *rekursiv* (Datei: [aufgabe-1.c](#)):

```
int fak (int n)
{
    if (n <= 0)
        return 1;
    else
        return n * fak (n - 1);
}
```

- (a) Schreiben Sie eine Funktion, die die Fakultät *iterativ* berechnet, d. h. mit Hilfe einer Schleife anstelle von Rekursion. (3 Punkte)
- (b) Wie viele Multiplikationen (Landau-Symbol) erfordern beide Versionen der Fakultätsfunktion in Abhängigkeit von  $n$ ? Begründen Sie Ihre Antwort. (2 Punkte)
- (c) Wieviel Speicherplatz (Landau-Symbol) erfordern beide Versionen der Fakultätsfunktion in Abhängigkeit von  $n$ ? Begründen Sie Ihre Antwort. (3 Punkte)

### Lösung

- (a) **Schreiben Sie eine Funktion, die die Fakultät *iterativ* berechnet, d. h. mit Hilfe einer Schleife anstelle von Rekursion.**

Datei: [loesung-1.c](#)

```
int fak (int n)
{
    int f = 1;
    for (int i = 2; i <= n; i++)
        f *= i;
    return f;
}
```

- (b) **Wie viele Multiplikationen (Landau-Symbol) erfordern beide Versionen der Fakultätsfunktion?**  
In beiden Fällen werden  $n$  Zahlen miteinander multipliziert – oder  $n - 1$ , wenn man Multiplikationen mit 1 ausspart. In jedem Fall hängt die Anzahl der Multiplikationen linear von  $n$  ab; es sind  $\mathcal{O}(n)$  Multiplikationen. Insbesondere arbeiten also beide Versionen gleich schnell.

- (c) **Wieviel Speicherplatz (Landau-Symbol) erfordern beide Versionen der Fakultätsfunktion?**

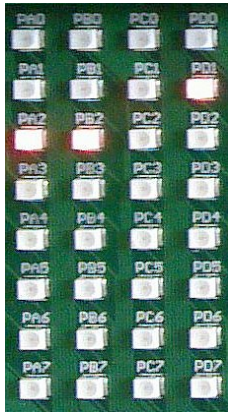
Die iterative Version der Funktion benötigt 2 Variable vom Typ `int`, nämlich  $n$  und  $f$ . Dies ist eine konstante Zahl; der Speicherplatzverbrauch ist daher  $\mathcal{O}(1)$ .

Die rekursive Version der Funktion erzeugt jedesmal, wenn sie sich selbst aufruft, eine zusätzliche Variable  $n$ . Es sind  $n + 1$  Aufrufe; die Anzahl der Variablen  $n$  hängt linear von  $n$  ab; der Speicherplatzverbrauch ist also  $\mathcal{O}(n)$ .

## Aufgabe 2: Lauflicht

An die vier Ports eines ATmega16-Mikrocontrollers sind Leuchtdioden angeschlossen:

- von links nach rechts  
an die Ports A, B, C und D,
- von oben nach unten  
an die Bits Nr. 0 bis 7.



Wir betrachten das folgende C-Programm (Datei: [aufgabe-2.c](#)) für diesen Mikrocontroller:

```
#include <avr/io.h>
#include <avr/interrupt.h>

int counter = 0;

ISR (TIMER0_COMP_vect)
{
    PORTA = 1 << ((counter++ >> 6) & 7);
}

int main (void)
{
    cli ();
    TCCR0 = (1 << CS01) | (1 << CS00);
    TIMSK = 1 << OCIE0;
    sei ();
    DDRA = 0xff;
    while (1);
    return 0;
}
```

Das Programm bewirkt ein periodisches Lauflicht in der linken Spalte von oben nach unten. Eine Animation davon finden Sie in der Datei [aufgabe-2.gif](#).

- Wieso bewirkt das Programm überhaupt etwas, wenn doch das Hauptprogramm nach dem Initialisieren lediglich eine Endlosschleife ausführt, in der *nichts* passiert? (3 Punkte)
- Erklären Sie, wie die Anweisung  
`PORTA = 1 << ((counter++ >> 6) & 7);`  
das LED-Blinkmuster hervorruft. (6 Punkte)  
Hinweis: Zerlegen Sie die eine lange Anweisung in mehrere kürzere.  
Wenn nötig, verwenden Sie zusätzliche Variable für Zwischenergebnisse.
- Was bedeutet „ISR (TIMER0\_COMP\_vect)“? (1 Punkt)
- Wieso leuchten die Leuchtdioden PB2 und PD1? (2 Punkte)

## Lösung

- Wieso bewirkt das Programm überhaupt etwas, wenn doch das Hauptprogramm nach dem Initialisieren lediglich eine Endlosschleife ausführt, in der *nichts* passiert?**  
Die Funktion `ISR (TIMER0_COMP_vect)` ist ein *Interrupt-Handler* (oder *Interrupt Service Routine – ISR*) und wird nicht durch das Hauptprogramm oder andere Funktionen aufgerufen, sondern durch ein Hardware-Ereignis.  
In diesem Fall sorgt die Initialisierung dafür, daß die Funktion regelmäßig durch eine Uhr (*Timer*) aufgerufen wird.
- Erklären Sie, wie die Anweisung**  
`PORTA = 1 << ((counter++ >> 6) & 7);`  
**das LED-Blinkmuster hervorruft.**  
Um die Anweisung zu verstehen, zerlegen wir sie in mehrere kürzere. Dabei ist es wichtig, die Reihenfolge zu erkennen, in der die einzelnen Operationen ausgeführt werden.  
Zunächst stellen wir fest, daß der Post-Inkrement-Operator `++` erst nach der Anweisung – also als letztes – ausgeführt wird. Die Anweisung ist somit äquivalent zu den folgenden zwei Anweisungen:

```
PORTA = 1 << ((counter >> 6) & 7);  
counter++;
```

Von der verbleibenden langen Anweisung wird als erstes die innerste Klammer ausgeführt:

```
int temp1 = counter >> 6;  
PORTA = 1 << (temp1 & 7);  
counter++;
```

Das Verschieben um 6 Binärstellen nach rechts entspricht einer Division durch  $2^6 = 64$ . Die Bedeutung der neu eingeführten Variablen ist somit ein Zähler, der 64mal langsamer zählt als der ursprüngliche:

```
int slow_counter = counter >> 6;  
PORTA = 1 << (slow_counter & 7);  
counter++;
```

(Ohne diese Verlangsamung wäre das Lauflicht zu schnell, um es mit bloßem Auge wahrnehmen zu können.)

Als nächste Operation wird die bitweise Und-Verknüpfung ausgeführt:

```
int slow_counter = counter >> 6;  
int temp2 = slow_counter & 7;  
PORTA = 1 << temp2;  
counter++;
```

Die Binärdarstellung von 7 sind drei Einsen:  $7_{10} = 111_2$ . Eine Und-Verknüpfung mit 7 isoliert also die untersten drei Bits von `slow_counter`.

Mit drei Bits lassen sich Zahlen von 0 bis 7 darstellen. Wenn also `slow_counter` immer größer wird, läuft `slow_counter & 7` immer wieder von 0 bis 7:

```
int slow_counter = counter >> 6;  
int slow_counter_0_to_7 = slow_counter & 7;  
PORTA = 1 << slow_counter_0_to_7;  
counter++;
```

(Bemerkung: Eine bitweise Und-Verknüpfung mit 7 ist gleichbedeutend zur Berechnung des Rests bei Division durch 8: `slow_counter_0_to_7 = slow_counter % 8`. Auch dadurch wird klar, daß das Ergebnis der Operation immer wieder die Zahlen von 0 bis 7 durchläuft, wenn `slow_counter` immer größer wird.)

Die letzte Operation `1 << slow_counter_0_to_7` erzeugt eine Bitmaske, in der ein Bit von Position 0 bis Position 7, also von rechts nach links wandert:

```
00000001  
00000010  
00000100  
00001000  
00010000  
00100000  
01000000  
10000000
```

Die zyklische Zuweisung dieser Bitmasken an `PORTA` erzeugt das Lauflicht.

### Zusammenfassung:

- Das jeweils zum Schluß ausgeführte `++` läßt die Variable `counter` bei jedem Aufruf um 1 hochzählen.
- Durch die Bit-Verschiebung um 6 nach rechts erzeugen wir einen um den Faktor 64 langsameren Zähler.
- Die bitweise Und-Verknüpfung mit 7 isoliert die untersten 3 Bit des langsameren Zählers und erzeugt so einen neuen Zähler, der immer wieder von 0 bis 7 zählt.
- Die Linksverschiebung einer 1 um diesen neuen Zähler erzeugt Bit-Masken mit jeweils einem Bit, das von rechts nach links läuft.
- Die zyklische Zuweisung dieser Bitmasken an `PORTA` erzeugt das Lauflicht.

(c) Was bedeutet „ISR (TIMER0\_COMP\_vect)“?

Es bezeichnet einen *Interrupt-Handler* (*Interrupt Service Routine – ISR*) für einen *Timer-Interrupt*, also eine Funktion, die durch ein von einer Uhr periodisch ausgelöstes Hardware-Ereignis aufgerufen wird.

(d) Wieso leuchten die Leuchtdioden PB2 und PD1?

Der Mikro-Controller enthält kein Betriebssystem. Auf ihm läuft kein Programm außer dem, das wir auf ihn herunterladen.

Beim Start enthalten alle Ports **zufällige Werte**. Wenn wir wünschen, daß eine LED aus ist, muß unser Programm sie explizit ausschalten.

In diesem Fall ließe sich dies durch die Zeilen

```
DDRB = 0xff;
DDRC = 0xff;
DDRD = 0xff;
PORTB = 0;
PORTC = 0;
PORTD = 0;
```

im Hauptprogramm erreichen.

(Wegen der **volatile**-Eigenschaft der **DDR**- und **PORT**-Variablen ist dies übrigens *nicht* äquivalent zu den folgenden Zeilen:

```
DDRB = DDRC = DDRD = 0xff;
PORTB = PORTC = PORTD = 0;
```

Für Details siehe: [http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq\\_assign\\_chain](http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_assign_chain))

### Aufgabe 3: Länge von Strings

Diese Aufgabe ist eine Neuauflage von Aufgabe 3 der Übung vom 5. November 2017, ergänzt um die Teilaufgaben (f) und (g).

Strings werden in der Programmiersprache C durch Zeiger auf **char**-Variable realisiert.

Beispiel: **char** \*hello\_world = "Hello,\_world!\n"

Die Systembibliothek stellt eine Funktion **strlen()** zur Ermittlung der Länge von Strings zur Verfügung (**#include <string.h>**).

- (a) Auf welche Weise ist die Länge eines Strings gekennzeichnet? (1 Punkt)
- (b) Wie lang ist die Beispiel-String-Konstante "Hello,\_world!\n", und wieviel Speicherplatz belegt sie? (2 Punkte)
- (c) Schreiben Sie eine eigene Funktion **int strlen (char \*s)**, die die Länge eines Strings zurückgibt. (3 Punkte)

Wir betrachten nun die folgenden Funktionen (Datei: [aufgabe-3.c](#)):

```
int fun_1 (char *s)
{
    int x = 0;
    for (int i = 0; i < strlen (s); i++)
        x += s[i];
    return x;
}
```

```
int fun_2 (char *s)
{
    int i = 0, x = 0;
    int len = strlen (s);
    while (i < len)
        x += s[i++];
    return x;
}
```

- (d) Was bewirken die beiden Funktionen? (2 Punkte)

- (e) Schreiben Sie eine eigene Funktion, die dieselbe Aufgabe erledigt wie `fun_2()`, nur effizienter. (4 Punkte)
- (f) Von welcher Ordnung (Landau-Symbol) sind die beiden Funktionen hinsichtlich der Anzahl ihrer Zugriffe auf die Zeichen im String? Begründen Sie Ihre Antwort. Sie dürfen für `strlen()` Ihre eigene Version der Funktion voraussetzen. (3 Punkte)
- (g) Von welcher Ordnung (Landau-Symbol) ist Ihre effizientere Funktion? Begründen Sie Ihre Antwort. (1 Punkt)

## Lösung

- (a) **Auf welche Weise ist die Länge eines Strings gekennzeichnet?**

Ein String ist ein Array von `chars`. Nach den eigentlichen Zeichen des Strings enthält das Array ein **Null-Symbol** (Zeichen mit Zahlenwert 0, nicht zu verwechseln mit der Ziffer '0') als Ende-Markierung. Die Länge eines Strings ist die Anzahl der Zeichen vor diesem Symbol.

- (b) **Wie lang ist die Beispiel-String-Konstante "Hello,\_world!\n", und wieviel Speicherplatz belegt sie?**

Sie ist 14 Zeichen lang ('`\n`' ist nur 1 Zeichen; das Null-Symbol, das das Ende markiert, zählt hier nicht mit) und belegt Speicherplatz für 15 Zeichen (15 Bytes – einschließlich Null-Symbol / Ende-Markierung).

- (c) **Schreiben Sie eine eigene Funktion `int strlen(char *s)`, die die Länge eines Strings zurückgibt.**

Siehe die Dateien `loesung-3c-1.c` (mit Array-Index) und `loesung-3c-2.c` (mit Zeiger-Arithmetik). Beide Lösungen sind korrekt und arbeiten gleich schnell.

Die Warnung `conflicting types for built-in function "strlen"` kann normalerweise ignoriert werden; auf manchen Systemen (z. B. MinGW) hat jedoch die eingebaute Funktion `strlen()` beim Linken Vorrang vor der selbstgeschriebenen, so daß die selbstgeschriebene Funktion nie aufgerufen wird. In solchen Fällen ist es zulässig, die selbstgeschriebene Funktion anders zu nennen (z. B. `my_strlen()`).

- (d) **Was bewirken die beiden Funktionen?**

Beide addieren die Zahlenwerte der im String enthaltenen Zeichen und geben die Summe als Funktionsergebnis zurück.

Im Falle des Test-Strings "Hello,\_world!\n" lautet der Rückgabewert 1171 (siehe `loesung-3d-1.c` und `loesung-3d-2.c`).

- (e) **Schreiben Sie eine eigene Funktion, die dieselbe Aufgabe erledigt wie `fun_2()`, nur effizienter.**

Die Funktion wird effizienter, wenn man auf den Aufruf von `strlen()` verzichtet und stattdessen die Ende-Prüfung in derselben Schleife vornimmt, in der man auch die Zahlenwerte der Zeichen des Strings aufsummiert.

Die Funktion `fun_3()` in der Datei `loesung-3e-1.c` realisiert dies mit einem Array-Index, Die Funktion `fun_4()` in der Datei `loesung-3e-2.c` mit Zeiger-Arithmetik. Beide Lösungen sind korrekt und arbeiten gleich schnell.

**Bemerkung:** Die effizientere Version der Funktion arbeitet doppelt so schnell wie die ursprüngliche, hat aber ebenfalls die Ordnung  $\mathcal{O}(n)$  – siehe unten.

- (f) **Von welcher Ordnung (Landau-Symbol) sind die beiden Funktionen hinsichtlich der Anzahl ihrer Zugriffe auf die Zeichen im String? Begründen Sie Ihre Antwort. Sie dürfen für `strlen()` Ihre eigene Version der Funktion voraussetzen.**

Vorüberlegung: `strlen()` greift in einer Schleife auf alle Zeichen des Strings der Länge  $n$  zu, hat also  $\mathcal{O}(n)$ .

`fun_1()` ruft in jedem Schleifendurchlauf (zum Prüfen der `while`-Bedingung) einmal `strlen()` auf und greift anschließend auf ein Zeichen des Strings zu, hat also  $\mathcal{O}(n \cdot (n + 1)) = \mathcal{O}(n^2)$ .

`fun_2()` ruft einmalig `strlen()` auf und greift anschließend in einer Schleife auf alle Zeichen des Strings zu, hat also  $\mathcal{O}(n + n) = \mathcal{O}(n)$ .

- (g) **Von welcher Ordnung (Landau-Symbol) ist Ihre effizientere Funktion?**  
**Begründen Sie Ihre Antwort.**

In beiden o. a. Lösungsvarianten – [loesung-3e-1.c](#) und [loesung-3e-2.c](#) – arbeitet die Funktion mit einer einzigen Schleife, die gleichzeitig die Zahlenwerte addiert und das Ende des Strings sucht.

Mit jeweils einer einzigen Schleife haben beide Funktionen die Ordnung  $\mathcal{O}(n)$ .