

# Hardwarenahe Programmierung

## Musterlösung zu den Übungsaufgaben 8 – 12. Dezember 2024

### Aufgabe 1: Trickprogrammierung

Wir betrachten das folgende Programm (Datei: `aufgabe-1.c`):

```
#include <stdio.h>
#include <stdint.h>

int main (void)
{
    uint64_t x = 4262939000843297096;
    char *s = &x;
    printf ("%s\n", s);
    return 0;
}
```

Das Programm wird kompiliert und auf einem 64-Bit-Little-Endian-Computer ausgeführt:

```
$ gcc -Wall -O aufgabe-1.c -o aufgabe-1
aufgabe-1.c: In function 'main':
aufgabe-1.c:7:13: warning: initialization from incompatible pointer type [...]
$ ./aufgabe-1
Hallo
```

- (a) Erklären Sie die Warnung beim Compilieren. (2 Punkte)
- (b) Erklären Sie die Ausgabe des Programms. (5 Punkte)
- (c) Wie würde die Ausgabe des Programms auf einem 64-Bit-Big-Endian-Computer lauten? (3 Punkte)

Hinweis: Modifizieren Sie das Programm und lassen Sie sich Speicherinhalte ausgeben.

### Lösung

- (a) **Erklären Sie die Warnung beim Compilieren.**

Zeile 7 des Programms enthält eine Zuweisung von `&x` an die Variable `s`. Der Ausdruck `&x` steht für die Speicheradresse der Variablen `x`, ist also ein Zeiger auf `x`, also ein Zeiger auf eine `uint64_t`. Die Variable `s` hingegen ist ein Zeiger auf `char`, also ein Zeiger auf eine viel kleinere Zahl, also ein anderer Zeigertyp.

- (b) **Erklären Sie die Ausgabe des Programms.**

Die 64-Bit-Zahl (`uint64_t`) `x` belegt 8 Speicherzellen (Bytes) von jeweils 8 Bit. Um herauszufinden, was diese enthalten, lassen wir uns `x` als Hexadezimalzahl ausgeben, z. B. mittels `printf ("%lx\n", x)` (auf 32-Bit-Rechnern: `printf ("%llx\n")`) oder mittels `printf ("%PRIx64\n", x)` (erfordert `#include <inttypes.h>` – siehe die Datei `loesung-1-1.c`). Das Ergebnis lautet:

```
3b29006f6c6c6148
```

Auf einzelne Bytes verteilt:

```
3b 29 00 6f 6c 6c 61 48
```

Auf einem Little-Endian-Rechner ist die Reihenfolge der Bytes in den Speicherzellen genau umgekehrt:

```
48 61 6c 6c 6f 00 29 3b
```

Wenn wir uns diese Bytes als Zeichen ausgeben lassen (`printf()` mit `%c` – siehe die Datei `loesung-1-2.c`), erhalten wir:

```
H a l l o ) ;
```

Das Zeichen hinter „Hallo“ ist ein Null-Symbol (Zahlenwert 0) und wird von `printf ("%s")` als Ende des Strings erkannt. Damit ist die Ausgabe `Hallo` des Programms erklärt.

(c) **Wie würde die Ausgabe des Programms auf einem 64-Bit-Big-Endian-Computer lauten?**

Auf einem Big-Endian-Computer (egal, wieviele Bits die Prozessorregister haben) ist die Reihenfolge der Bytes in den Speicherzellen genau umgekehrt wie auf einem Little-Endian-Computer, hier also:

3b 29 00 6f 6c 6c 61 48

`printf ("%s")` gibt in diesem Fall die Hexadezimalzahlen 3b und 29 als Zeichen aus. Danach steht das String-Ende-Symbol mit Zahlenwert 0, und die Ausgabe bricht ab. Da, wie oben ermittelt, die Hexadezimalzahl 3b für das Zeichen ; und 29 für das Zeichen ) steht, lautet somit die Ausgabe:

; )

Um die Aufgabe zu lösen, können Sie übrigens auch auf einem Little-Endian-Computer (Standard-Notebook) einen Big-Endian-Computer simulieren, indem Sie die Reihenfolge der Bytes in der Zahl `x` umdrehen – siehe die Datei [loesung-1-3.c](#).

## Aufgabe 2: Thermometer-Baustein an I<sup>2</sup>C-Bus

Eine Firma stellt einen elektronischen Thermometer-Baustein her, den man über die serielle Schnittstelle (RS-232) an einen PC anschließen kann, um die Temperatur auszulesen. Nun wird eine Variante des Thermometer-Bausteins entwickelt, die die Temperatur zusätzlich über einen I<sup>2</sup>C-Bus bereitstellt.

Um das neue Thermometer zu testen, wird es in ein Gefäß mit heißem Wasser gelegt, das langsam auf Zimmertemperatur abkühlt. Alle 10 Minuten liest ein Programm, das auf dem PC läuft, die gemessene Temperatur über beide Schnittstellen aus und erzeugt daraus die folgende Tabelle:

Zeit / min.	Temperatur per RS-232 / °C	Temperatur per I <sup>2</sup> C / °C
0	94	122
10	47	244
20	30	120
30	24	24
40	21	168

- (a) Aus dem Vergleich der Meßdaten läßt sich auf einen Fehler bei der I<sup>2</sup>C-Übertragung schließen. Um welchen Fehler handelt es sich, und wie ergibt sich dies aus den Meßdaten? (5 Punkte)
- (b) Schreiben Sie eine C-Funktion `uint8_t repair(uint8_t data)`, die eine über den I<sup>2</sup>C-Bus empfangene fehlerhafte Temperatur `data` korrigiert. (5 Punkte)

## Lösung

- (a) **Aus dem Vergleich der Meßdaten läßt sich auf einen Fehler bei der I<sup>2</sup>C-Übertragung schließen. Um welchen Fehler handelt es sich, und wie ergibt sich dies aus den Meßdaten?**

Sowohl RS-232 als auch I<sup>2</sup>C übertragen die Daten Bit für Bit. Für die Fehlersuche ist es daher sinnvoll, die Meßwerte als Binärzahlen zu betrachten:

Zeit / min.	Temperatur per RS-232 / °C	Temperatur per I <sup>2</sup> C / °C
0	94 <sub>10</sub> = 01011110 <sub>2</sub>	122 <sub>10</sub> = 01111010 <sub>2</sub>
10	47 <sub>10</sub> = 00101111 <sub>2</sub>	244 <sub>10</sub> = 11110100 <sub>2</sub>
20	30 <sub>10</sub> = 00011110 <sub>2</sub>	120 <sub>10</sub> = 01111000 <sub>2</sub>
30	24 <sub>10</sub> = 00011000 <sub>2</sub>	24 <sub>10</sub> = 00011000 <sub>2</sub>
40	21 <sub>10</sub> = 00010101 <sub>2</sub>	168 <sub>10</sub> = 10101000 <sub>2</sub>

Man erkennt, daß die Reihenfolge der Bits in den (fehlerhaften) I<sup>2</sup>C-Meßwerten genau die umgekehrte Reihenfolge der Bits in den (korrekten) RS-232-Meßwerten ist. Der Übertragungsfehler besteht also darin, daß die Bits in der falschen Reihenfolge übertragen wurden.

Dies paßt gut damit zusammen, daß die Bit-Reihenfolge von I<sup>2</sup>C *MSB First*, die von RS-232 hingegen *LSB First* ist. Offenbar haben die Entwickler der I<sup>2</sup>C-Schnittstelle dies übersehen und die I<sup>2</sup>C-Daten ebenfalls *LSB First* übertragen.

- (b) Schreiben Sie eine C-Funktion `uint8_t repair (uint8_t data)`, die eine über den I<sup>2</sup>C-Bus empfangene fehlerhafte Temperatur `data` korrigiert.

Die Aufgabe der Funktion besteht darin, eine 8-Bit-Zahl `data` entgegenzunehmen, die Reihenfolge der 8 Bits genau umzudrehen und das Ergebnis mittels `return` zurückzugeben.

Zu diesem Zweck gehen wir die 8 Bits in einer Schleife durch – siehe die Datei `loesung-2.c`. Wir lassen eine Lese-Maske `mask_data` von rechts nach links und gleichzeitig eine Schreib-Maske `mask_result` von links nach rechts wandern. Immer wenn die Lese-Maske in `data` eine 1 findet, schreibt die Schreib-Maske diese in die Ergebnisvariable `result`.

Da `result` auf 0 initialisiert wurde, brauchen wir Nullen nicht hineinzuschreiben. Ansonsten wäre dies mit `result &= ~mask_result` möglich.

Um die Schleife bis 8 zählen zu lassen, könnte man eine weitere Zähler-Variable von 0 bis 7 zählen lassen, z. B. `for (int i = 0; i < 8; i++)`. Dies ist jedoch nicht nötig, wenn man beachtet, daß die Masken den Wert 0 annehmen, sobald das Bit aus der 8-Bit-Variablen herausgeschoben wurde. In `loesung-2.c` wird `mask_data` auf 0 geprüft; genausogut könnte man auch `mask_result` prüfen.

Das `return result` ist notwendig. Eine Ausgabe des Ergebnisses per `printf()` o. ä. erfüllt *nicht* die Aufgabenstellung. (In `loesung-2.c` erfolgt entsprechend `printf()` nur im Testprogramm `main()`.)

### Aufgabe 3: Kondensator

Ein Kondensator der Kapazität  $C = 100 \mu\text{F}$  ist auf die Spannung  $U_0 = 5 \text{ V}$  aufgeladen und wird über einen Widerstand  $R = 33 \text{ k}\Omega$  entladen.

- (a) Schreiben Sie ein C-Programm, das den zeitlichen Spannungsverlauf in einer Tabelle darstellt. (5 Punkte)
- (b) Schreiben Sie ein C-Programm, das ermittelt, wie lange es dauert, bis die Spannung unter  $0.1 \text{ V}$  gefallen ist. (4 Punkte)
- (c) Vergleichen Sie die berechneten Werte mit der exakten theoretischen Entladekurve:  $U(t) = U_0 \cdot e^{-\frac{t}{RC}}$  (3 Punkte)

Hinweise:

- Für die Simulation zerlegen wir den Entladevorgang in kurze Zeitintervalle  $dt$ . Innerhalb jedes Zeitintervalls betrachten wir den Strom  $I$  als konstant und berechnen, wieviel Ladung  $Q$  innerhalb des Zeitintervalls aus dem Kondensator herausfließt. Aus der neuen Ladung berechnen wir die Spannung am Ende des Zeitintervalls.
- Für den Vergleich mit der exakten theoretischen Entladekurve benötigen Sie die Exponentialfunktion `exp()`. Diese finden Sie in der Mathematik-Bibliothek: `#include <math.h>` im Quelltext, beim `gcc`-Aufruf `-lm` mit angeben.
- $Q = C \cdot U$ ,  $U = R \cdot I$ ,  $I = \frac{dQ}{dt}$

### Lösung

- Schreiben Sie ein C-Programm, das den zeitlichen Spannungsverlauf in einer Tabelle darstellt.**

In dem Programm `loesung-3a.c` arbeiten wir, dem ersten Hinweis folgend, mit einem Zeitintervall von `dt = 0.01`. Mit dieser Schrittweite lassen wir uns eine Tabelle ausgeben, die jeweils die Zeit und durch die Simulation berechnete Spannung ausgibt.

Wir simulieren, wie die Ladung  $Q = C \cdot U$  des Kondensators im Laufe der Zeit abfließt. Dazu berechnen wir in jedem Zeitschritt zunächst den Strom  $I = U/R$ , der aus dem Kondensator fließt. Dieser Strom bewirkt, daß innerhalb des Zeitintervalls  $dt$  die Ladung  $dQ = I \cdot dt$  aus dem Kondensator abfließt. Am Ende des Zeitintervalls berechnen wir die zur neuen Ladung  $Q$  gehörende neue Spannung  $U = Q/C$ .

Für eine einfache Ausgabe der Tabelle verwenden wir die Formatspezifikationen `%10.3lf` für drei bzw. `%15.8lf` für acht Nachkommastellen Genauigkeit. Damit schreiben wir jeweils eine *lange Fließkommazahl* (`%lf`) rechtsbündig in ein Feld der Breite 10 bzw. 15 und lassen uns 3 bzw. 8 Nachkommastellen ausgeben.

Wir compilieren das Programm mit: `gcc -Wall -O loesung-1a.c -o loesung-1a`

- **Schreiben Sie ein C-Programm, das ermittelt, wie lange es dauert, bis die Spannung unter 0.1 V gefallen ist.**

Wir ändern das Programm [loesung-3a.c](#) so ab, daß zum einen die Schleife abbricht, sobald die Spannung den Wert 0.1 V unterschreitet ([loesung-3b.c](#)), und daß zum anderen nicht jedesmal eine Zeile für die Tabelle ausgegeben wird, sondern erst am Ende die Zeit (und die Spannung).

Der Ausgabe entnehmen wir, daß die Spannung bei etwa  $t = 12.90\text{ s}$  den Wert 0.1 V unterschreitet.

- **Vergleichen Sie die berechneten Werte mit der exakten theoretischen Entladekurve:**

$$U(t) = U_0 \cdot e^{-\frac{t}{RC}}$$

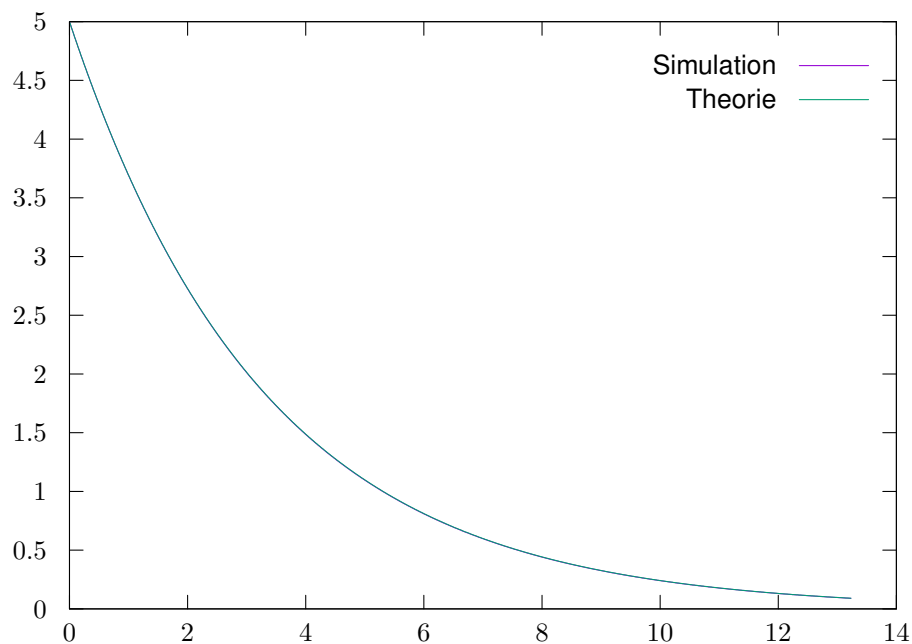
Wir ändern das Programm [loesung-3a.c](#) so ab, daß es zusätzlich zur Zeit und zur simulierten Spannung die exakte Spannung  $U_0 \cdot e^{-\frac{t}{RC}}$  gemäß der theoretischen Entladekurve ausgibt ([loesung-3c.c](#)),

Da dieses Programm die Exponentialfunktion verwendet, müssen wir nun beim Compilieren zusätzlich `-lm` für das Einbinden der Mathematik-Bibliothek angeben.

Der erweiterten Tabelle können wir entnehmen, daß die durch die Simulation berechnete Spannung mit der Spannung  $U_0 \cdot e^{-\frac{t}{RC}}$  gemäß der theoretischen Entladekurve bis auf wenige Prozent übereinstimmt. Dies ist für viele praktische Anwendungen ausreichend, wenn auch nicht für Präzisionsmessungen.

Wenn Sie die Ausgabe des Programms, z. B. mit `./loesung-1c > loesung-1c.dat`, in einer Datei [loesung-3c.dat](#) speichern, können Sie sich die beiden Kurven graphisch darstellen lassen, z. B. mit [gnuplot](#) und dem folgenden Befehl:

```
plot "loesung-1c.dat" using 1:2 with lines title "Simulation",
     "loesung-1c.dat" using 1:3 with lines title "Theorie"
```



Der Unterschied zwischen der simulierten und der theoretischen Entladungskurve ist mit bloßem Auge nicht sichtbar.