

# Hardwarenahe Programmierung / Angewandte Informatik

## Musterlösung zu den Übungsaufgaben – 5. Dezember 2016

### Aufgabe 1: Trickprogrammierung

Wir betrachten das folgende Programm `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: Ausgabe von Hexadezimalzahlen

Schreiben Sie eine Funktion `void print_hex (uint32_t x)`, die eine gegebene vorzeichenlose 32-Bit-Ganzzahl `x` als Hexadezimalzahl ausgibt. (Der Datentyp `uint32_t` ist mit `#include <stdint.h>` verfügbar.)

Verwenden Sie dafür *nicht* `printf()` mit der Formatspezifikation `%x` als fertige Lösung, sondern programmieren Sie die nötige Ausgabe selbst. (Für Tests ist `%x` hingegen erlaubt und sicherlich nützlich.)

Die Verwendung von `printf()` mit anderen Formatspezifikationen wie z. B. `%d` oder `%c` oder `%s` ist hingegen zulässig.

(8 Punkte)

(Hinweis für die Klausur: Abgabe auf Datenträger ist erlaubt und erwünscht, aber nicht zwingend.)

## Lösung

Um die Ziffern von `x` zur Basis 16 zu isolieren, berechnen wir `x % 16` (modulo 16 = Rest bei Division durch 16) und dividieren anschließend `x` durch 16, solange bis `x` den Wert 0 erreicht.

Wenn wir die auf diese Weise ermittelten Ziffern direkt ausgeben, sind sie *Little-Endian*, erscheinen also in umgekehrter Reihenfolge. Die Datei [loesung-2-1.c](#) setzt diesen Zwischenschritt um.

Die Ausgabe der Ziffern erfolgt in [loesung-2-1.c](#) über `printf ("%d")` für die Ziffern 0 bis 9. Für die darüberliegenden Ziffern wird der Buchstabe `a` um die Ziffer abzüglich 10 inkrementiert und der erhaltene Wert mit `printf ("%c")` als Zeichen ausgegeben.

Um die umgekehrte Reihenfolge zu beheben, speichern wir die Ziffern von `x` in einem Array `digits[]` zwischen und geben sie anschließend in einer zweiten Schleife in umgekehrter Reihenfolge aus (siehe [loesung-2-2.c](#)). Da wir wissen, `x` eine 32-Bit-Zahl ist und daher höchstens 8 Hexadezimalziffern haben kann, ist 8 eine sinnvolle Länge für das Ziffern-Array `digits[8]`.

Nun sind die Ziffern in der richtigen Reihenfolge, aber wir erhalten zusätzlich zu den eigentlichen Ziffern führende Nullen. Da in der Aufgabenstellung nicht von führenden Nullen die Rede war, sind diese nicht verboten; [loesung-2-2.c](#) ist daher eine richtige Lösung der Aufgabe.

Wenn wir die führenden Nullen vermeiden wollen, können wir die `for`-Schleifen durch `while`-Schleifen ersetzen. Die erste Schleife zählt hoch, solange `x` ungleich 0 ist; die zweite zählt von dem erreichten Wert aus wieder herunter – siehe [loesung-2-3.c](#). Da wir wissen, daß die Zahl `x` höchstens 32 Bit, also höchstens 8 Hexadezimalziffern hat, wissen wir, daß `i` höchstens den Wert 8 erreichen kann, das Array also nicht überlaufen wird.

Man beachte, daß der Array-Index nach der ersten Schleife „um einen zu hoch“ ist. In der zweiten Schleife muß daher *zuerst* der Index dekrementiert werden. Erst danach darf ein Zugriff auf `digit[i]` erfolgen.

Alternativ können wir auch mitschreiben, ob bereits eine Ziffer ungleich Null ausgegeben wurde, und andernfalls die Ausgabe von Null-Ziffern unterdrücken – siehe [loesung-2-4.c](#).

Weitere Möglichkeiten ergeben sich, wenn man bedenkt, daß eine Hexadezimalziffer genau einer Gruppe von vier Binärziffern entspricht. Eine Bitverschiebung um 4 nach rechts ist daher dasselbe wie eine Division durch 16, und eine Und-Verknüpfung mit  $15_{10} = f_{16} = 1111_2$  ist dasselbe wie die Operation Modulo 16. Die Datei [loesung-2-5.c](#) ist eine in dieser Weise abgewandelte Variante von [loesung-2-3.c](#).

Mit dieser Methode kann man nicht nur auf die jeweils unterste Ziffer, sondern auf alle Ziffern direkt zugreifen. Damit ist kein Array als zusätzlicher Speicher mehr nötig. Die Datei [loesung-2-6.c](#) setzt dies auf einfache Weise um. Sie gibt wieder führende Nullen mit aus, ist aber trotzdem eine weitere richtige Lösung der Aufgabe.

Die führenden Nullen ließen sich auf die gleiche Weise vermeiden wie in [loesung-2-4.c](#).

Die Bitverschiebungsmethode hat den Vorteil, daß kein zusätzliches Array benötigt wird. Auch wird die als Parameter übergebene Zahl `x` nicht verändert, was bei größeren Zahlen, die über Zeiger übergeben werden, von Vorteil sein kann. Demgegenüber steht der Nachteil, daß diese Methode nur für eine ganze Anzahl von Bits funktioniert, also für Basen, die Zweierpotenzen sind (z. B. 2, 8, 16, 256). Für alle anderen Basen (z. B. 10) eignet sich nur die Methode mit Division und Modulo-Operation.

### Aufgabe 3: 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                                  |

- 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)
- 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

- 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           | $94_{10} = 01011110_2$     | $122_{10} = 01111010_2$              |
| 10          | $47_{10} = 00101111_2$     | $244_{10} = 11110100_2$              |
| 20          | $30_{10} = 00011110_2$     | $120_{10} = 01111000_2$              |
| 30          | $24_{10} = 00011000_2$     | $24_{10} = 00011000_2$               |
| 40          | $21_{10} = 00010101_2$     | $168_{10} = 10101000_2$              |

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-3.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-3.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-3.c` erfolgt entsprechend `printf()` nur im Testprogramm `main()`.)