

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 9. November 2023

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: `"%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 I²C-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 I²C-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 ² 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 I²C-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 I²C-Bus empfangene fehlerhafte Temperatur `data` korrigiert. (5 Punkte)

Lösung

- (a) **Aus dem Vergleich der Meßdaten läßt sich auf einen Fehler bei der I²C-Übertragung schließen. Um welchen Fehler handelt es sich, und wie ergibt sich dies aus den Meßdaten?**

Sowohl RS-232 als auch I²C 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 ² 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) I²C-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 I²C *MSB First*, die von RS-232 hingegen *LSB First* ist. Offenbar haben die Entwickler der I²C-Schnittstelle dies übersehen und die I²C-Daten ebenfalls *LSB First* übertragen.

- (b) Schreiben Sie eine C-Funktion `uint8_t repair (uint8_t data)`, die eine über den I²C-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: Speicherformate von Zahlen

Wir betrachten das folgende Programm (`aufgabe-3.c`):

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

typedef struct
{
    uint32_t a;
    uint64_t b;
    uint8_t c;
} three_numbers;

int main (void)
{
    three_numbers xyz = { 1819042120, 2410670883059281007, 0 };
    printf ("%s\n", &xyz);
    return 0;
}
```

Das Programm wird für einen 32-Bit-Rechner compiliert und ausgeführt.

(Die `gcc`-Option `-m32` sorgt dafür, daß `gcc` Code für einen 32-Bit-Prozessor erzeugt.)

```
$ gcc -Wall -m32 aufgabe-2.c -o aufgabe-2
aufgabe-2.c: In function "main":
aufgabe-2.c:14:13: warning: format "%s" expects argument of type "char *", but
argument 2 has type "three_numbers * {aka struct <anonymous> *}" [-Wformat=]
    printf ("%s\n", &xyz);
                ^
$ ./aufgabe-2
Hallo, Welt!
```

- (a) Erklären Sie die beim Compilieren auftretende Warnung. (2 Punkte)
- (b) Erklären Sie die Ausgabe des Programms. (4 Punkte)
- (c) Welche Endianness hat der verwendete Rechner? Wie sähe die Ausgabe auf einem Rechner mit entgegengesetzter Endianness aus? (2 Punkte)
- (d) Dasselbe Programm wird nun für einen 64-Bit-Rechner compiliert und ausgeführt. (Die `gcc`-Option `-m64` sorgt dafür, daß `gcc` Code für einen 64-Bit-Prozessor erzeugt.)

```
$ gcc -Wall -m64 aufgabe-2.c -o aufgabe-2
aufgabe-2.c: In function "main":
aufgabe-2.c:14:13: warning: format "%s" expects argument of type "char *",
but argument 2 has type "three_numbers * {aka struct <anonymous> *}"
[-Wformat=]
    printf ("%s\n", &xyz);
                   ^
$ ./aufgabe-2
Hall15V
```

(Es ist möglich, daß die konkrete Ausgabe auf Ihrem Rechner anders aussieht.)

Erklären Sie die geänderte Ausgabe des Programms. (3 Punkte)

Lösung

(a) Erklären Sie die beim Compilieren auftretende Warnung.

Die Funktion `printf()` mit der Formatspezifikation `%s` erwartet als Parameter einen String, d. h. einen Zeiger auf `char`. Die Adresse (`&`) der Variablen `xyz` ist zwar ein Zeiger, aber nicht auf `char`, sondern auf einen `struct` vom Typ `three_numbers`. Eine implizite Umwandlung des Zeigertyps ist zwar möglich, aber normalerweise nicht das, was man beabsichtigt.

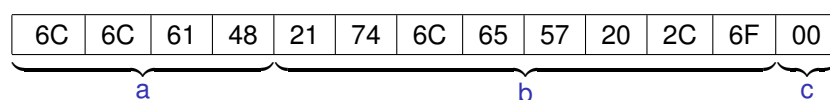
(b) Erklären Sie die Ausgabe des Programms.

Ein String in C ist ein Array von `chars` bzw. ein Zeiger auf `char`. Da die Funktion `printf()` mit der Formatspezifikation `%s` einen String erwartet, wird sie das, worauf der übergebene Zeiger zeigt, als ein Array von `chars` interpretieren. Ein `char` entspricht einer 8-Bit-Speicherzelle. Um die Ausgabe des Programms zu erklären, müssen wir daher die Speicherung der Zahlen in den einzelnen 8-Bit-Speicherzellen betrachten.

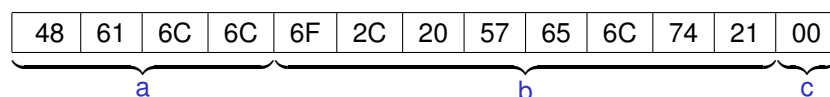
Hierfür wandeln wir zunächst die Zahlen von Dezimal nach Hexadezimal um. Sofern nötig (hier nicht der Fall) füllen wir von links mit Nullen auf, um den gesamten von der Variablen belegten Speicherplatz zu füllen (hier: 32 Bit, 64 Bit, 8 Bit). Jeweils 2 Hex-Ziffern stehen für 8 Bit.

dezimal		hexadezimal
1 819 042 120	=	6C 6C 61 48
2 410 670 883 059 281 007	=	21 74 6C 65 57 20 2C 6F
0	=	00

Die Anordnung dieser 8-Bit-Zellen im Speicher lautet **auf einem Big-Endian-Rechner** wie folgt:



Auf einem **Little-Endian-Rechner** lautet sie hingegen:



Anhand einer ASCII-Tabelle erkennt man, daß die Big-Endian-Variante dem String `"llaH!tleW ,o"` und die Little-Endian-Variante dem String `"Hallo, Welt!"` entspricht – jeweils mit einem Null-Symbol am Ende, das von der Variablen `c` herrührt.

Auf einem Little-Endian-Rechner wird daher `Hallo, Welt!` ausgegeben.

(c) Welche Endianness hat der verwendete Rechner?

Little-Endian (Begründung siehe oben)

Wie sähe die Ausgabe auf einem Rechner mit entgegengesetzter Endianness aus?

`llaH!tleW ,o` (Begründung siehe oben)

(d) Dasselbe Programm wird nun für einen 64-Bit-Rechner compiliert und ausgeführt.

(Die `gcc`-Option `-m64` sorgt dafür, daß `gcc` Code für einen 64-Bit-Prozessor erzeugt.)

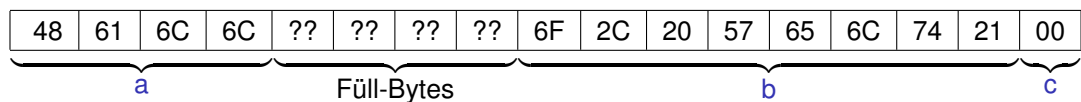
```
$ gcc -Wall -m64 aufgabe-2.c -o aufgabe-2
aufgabe-2.c: In function "main":
aufgabe-2.c:14:13: warning: format "%s" expects argument of type "char *",
but argument 2 has type "three_numbers * {aka struct <anonymous> *}"
[-Wformat=]
    printf ("%s\n", &xyz);
                  ^
$ ./aufgabe-2
Hall15V
```

(Es ist möglich, daß die konkrete Ausgabe auf Ihrem Rechner anders aussieht.)

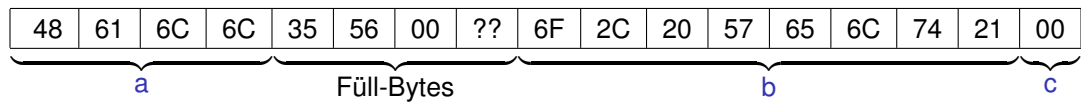
Erklären Sie die geänderte Ausgabe des Programms.

Auf einem 64-Bit-Rechner hat eine 64-Bit-Variable ein **64-Bit-Alignment**, d. h. ihre Speicheradresse muß durch 8 teilbar sein.

Der Compiler legt die Variablen daher wie folgt im Speicher ab (Little-Endian):



Der Inhalt der Füll-Bytes ist undefiniert. Im Beispiel aus der Aufgabenstellung entsteht hier die Ausgabe `5V`, was den (zufälligen) hexadezimalen Werten `35 56` entspricht:



Da danach die Ausgabe aufhört, muß an der nächsten Stelle ein Null-Symbol stehen, das das Ende des Strings anzeigt. Der Inhalt der darauf folgenden Speicherzelle bleibt unbekannt.