

Hardwarenahe Programmierung / Angewandte Informatik

Musterlösung zu den Übungsaufgaben – 19. Dezember 2016

Aufgabe 1: Bürgerentscheid

Bei einem [Bürgerentscheid](#) stimmen 66066 Wahlberechtigte mit „Ja“ und 65104 Wahlberechtigte mit „Nein“. Ein Rechenwerk vergleicht die Zahlen miteinander. Zu welchem Wahlergebnis – jeweils mit Begründung – kommt man

- (a) auf einem vorzeichenbehafteten 32-Bit-Rechenwerk? (2 Punkte)
- (b) auf einem vorzeichenlosen 16-Bit-Rechenwerk? (3 Punkte)
- (c) auf einem vorzeichenbehafteten 16-Bit-Rechenwerk? (3 Punkte)

Lösung

Zu welchem Wahlergebnis – jeweils mit Begründung – kommt man

- (a) **auf einem vorzeichenbehafteten 32-Bit-Rechenwerk?**

Der Wertebereich vorzeichenbehafteter 32-Bit-Zahlen geht von -2147483648 bis 2147483647 . Ein vorzeichenbehaftetes 32-Bit-Rechenwerk kann daher problemlos mit den Zahlen 66066 und 65104 rechnen und kommt zu dem Wahlergebnis „Ja“.

- (b) **auf einem vorzeichenlosen 16-Bit-Rechenwerk?**

Der Wertebereich vorzeichenloser 16-Bit-Zahlen geht von 0 bis 65535. Die Zahl 65104 paßt in diesen Wertebereich; die Zahl $66066 = 0x10212$ paßt jedoch nicht, sondern wird auf 16 Bit abgeschnitten. Übrig bleibt die Zahl $0x0212 = 530$, die kleiner ist als 65104. Das Wahlergebnis lautet somit „Nein“.

- (c) **auf einem vorzeichenbehafteten 16-Bit-Rechenwerk?**

Der Wertebereich vorzeichenbehafteter 16-Bit-Zahlen geht von -32768 bis 32767 . Weder 66066 noch 65104 paßt in diesen Wertebereich; beide Zahlen werden auf 16 Bit abgeschnitten. $66066 = 0x10212$ wird dadurch wie in Teilaufgabe (b) zu $0x0212 = 530$; $65104 = 0xfe50$ hat das höchste Bit gesetzt und wird daher als negative Zahl interpretiert. Über das Zweierkomplement erhalten wir -432 . (Siehe auch: [loesung-1c.c](#))

Da 530 größer ist als -432 , lautet das Wahlergebnis „Ja“.

Aufgabe 2: Lokale Variable im Speicher

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

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

int main (void)
{
    int8_t a = -1;
    uint32_t b = 770;
    uint16_t c = 513;
    printf ("%8x\n", &a);
    printf ("%8x\n", &b);
    printf ("%8x\n", &c);
    return 0;
}
```

Das Programm wird auf einem 32-Bit-Little-Endian-Rechner compiliert (mit drei Warnungen) und gestartet:

```
$ gcc -Wall -O aufgabe-2.c -o aufgabe-2
aufgabe-2.c: In function 'main':
aufgabe-2.c:9:3: warning: format '%x' expects argument of type 'unsigned int',
                but argument 2 has type 'int8_t *' [-Wformat]
aufgabe-2.c:10:3: warning: format '%x' expects argument of type 'unsigned int',
                but argument 2 has type 'uint32_t *' [-Wformat]
aufgabe-2.c:11:3: warning: format '%x' expects argument of type 'unsigned int',
                but argument 2 has type 'uint16_t *' [-Wformat]
$ ./aufgabe-2
ffd148ef
ffd148e8
ffd148e6
```

- Begründen Sie die Warnungen und, soweit möglich, die ausgegebenen Zahlen. (4 Punkte)
- Skizzieren Sie die Anordnung der Variablen – einschließlich Zahlenwerte – in den Speicherzellen (Bytes). (4 Punkte)
- Wie würde die Anordnung der Variablen – einschließlich Zahlenwerte – in den Speicherzellen (Bytes) auf einem 8-Bit-Big-Endian-Rechner lauten? (4 Punkte)

Lösung

- Begründen Sie die Warnungen und, soweit möglich, die ausgegebenen Zahlen.**

Die Funktion `printf()` mit Formatspezifikation `%8x` erwartet Zahlen. (Diese werden dann hexadezimal rechtsbündig in einem Feld der Breite 8 ausgegeben.) Übergeben werden an `printf()` jedoch die *Adressen* der Variablen `a`, `b` und `c`, also keine Zahlen, sondern Zeiger auf Zahlen.

Als Ausgabe sehen wir die Nummern der Speicherzellen, in denen die Variablen `a`, `b` und `c` jeweils beginnen.

Es fällt auf, daß die Speicheradressen immer kleiner statt größer werden. (Dies liegt daran, daß auf Intel-x86-kompatiblen Prozessoren der CPU-Stack, auf dem lokale Variable liegen, von oben nach unten wächst.)

Es fällt weiterhin auf, daß die Variablen nicht lückenlos aufeinander folgen. Dies dient dazu, korrekte *Speicherausrichtung (Alignment)* zu gewährleisten: Die 32-Bit-Variable `b` liegt auf der durch 4 teilbaren Speicheradresse `0xffd148e8`, die 16-Bit-Variable `c` auf der durch 2 teilbaren Speicheradresse `0xffd148e6`.

(Die genauen Zahlen können bei jedem Aufruf des Programms andere sein. Entscheidend für das Alignment ist die Teilbarkeit.)

- Skizzieren Sie die Anordnung der Variablen – einschließlich Zahlenwerte – in den Speicherzellen (Bytes).**

Die vorzeichenlose 8-Bit-Zahl `a` mit dem Wert `-1` belegt eine Speicherzelle, in der alle Bits auf 1 gesetzt sind (`0xff = 255`).

Die vorzeichenlose 32-Bit-Zahl `b` mit dem Wert `770 = 0x00000302` belegt in Little-Endian-Darstellung die Speicherzellen mit den Werten `0x02 = 2`, `0x03 = 3`, `0` und `0`.

Die vorzeichenlose 16-Bit-Zahl `c` mit dem Wert `513 = 0x0201` belegt in Little-Endian-Darstellung die Speicherzellen mit den Werten `0x01 = 1`, `0x02 = 2`.

Die Anordnung der Variablen – einschließlich Zahlenwerte – in den Speicherzellen lautet somit wie rechts dargestellt.

ffd148ef	255	<code>a</code>
ffd148ee	?	unbenutzt
ffd148ed	?	unbenutzt
ffd148ec	?	unbenutzt
ffd148eb	0	} <code>b</code>
ffd148ea	0	
ffd148e9	3	
ffd148e8	2	} <code>c</code>
ffd148e7	2	
ffd148e6	1	

(c) **Wie würde die Anordnung der Variablen – einschließlich Zahlenwerte – in den Speicherzellen (Bytes) auf einem 8-Bit-Big-Endian-Rechner lauten?**

Auf einem 8-Bit-Rechner spielt das Alignment keine Rolle; die Variablen können lückenlos aneinandergereiht werden.

Für die vorzeichenlose 32-Bit-Zahl **b** und die vorzeichenlose 16-Bit-Zahl **c** kehrt sich auf einem Big-Endian-Rechner die Reihenfolge der Bytes in den Speicherzellen genau um.

Die Anordnung der Variablen – einschließlich Zahlenwerte – in den Speicherzellen kann somit z. B. wie rechts dargestellt lauten.

ffd148ef	255	b
ffd148ee	2	
ffd148ed	3	
ffd148ec	0	b
ffd148eb	0	
ffd148ea	1	c
ffd148e9	2	

Aufgabe 3: Blinkende LEDs

Wir betrachten das folgende Programm für einen ATmega32-Mikro-Controller (Datei: [aufgabe-3.c](#)).

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

uint8_t counter = 1;
uint8_t leds = 0;

ISR (TIMER0_COMP_vect)
{
    if (counter == 0)
    {
        leds = (leds + 1) % 8;
        PORTC = leds << 4;
    }
    counter++;
}

void init (void)
{
    cli ();
    TCCR0 = (1 << CS01) | (1 << CS00);
    TIMSK = 1 << OCIE0;
    sei ();
    PORTB = 0;
    DDRC = 0x70;
}

int main (void)
{
    init ();
    while (1)
        ; /* do nothing */
    return 0;
}
```

An die Bits Nr. 4, 5 und 6 des Output-Ports C des Mikro-Controllers sind LEDs angeschlossen. Sobald das Programm läuft, blinken diese in charakteristischer Weise:

Phase	LED oben (rot)	LED Mitte (gelb)	LED unten (grün)
1	aus	aus	an
2	aus	an	aus
3	aus	an	an
4	an	aus	aus
5	an	aus	an
6	an	an	aus
7	an	an	an
8	aus	aus	aus

Jede Phase dauert etwas länger als eine halbe Sekunde. Nach 8 Phasen wiederholt sich das Schema.

Erklären Sie das Verhalten des Programms anhand des Quelltextes:

- Wieso macht das Programm überhaupt etwas, wenn doch das Hauptprogramm nach dem Initialisieren lediglich eine Endlosschleife ausführt, in der *nichts* passiert? (1 Punkt)
- Wieso wird die Zeile `PORTC = leds << 4;` überhaupt aufgerufen, wenn dies doch nur unter der Bedingung `counter == 0` passiert, wobei die Variable `counter` auf 1 initialisiert, fortwährend erhöht und nirgendwo zurückgesetzt wird? (2 Punkte)
- Wie kommt das oben beschriebene Blinkmuster zustande? (2 Punkte)
- Wieso dauert eine Phase ungefähr eine halbe Sekunde? (2 Punkte)
- Was bedeutet „`ISR (TIMER0_COMP_vect)`“? (1 Punkt)

Hinweis:

- Die Funktion `init()` sorgt dafür, daß der Timer-Interrupt Nr. 0 des Mikro-Controllers etwa 488mal pro Sekunde aufgerufen wird. Außerdem schaltet sie eventuell an Port B angeschlossene weitere LEDs aus und initialisiert Port C als Output-Port. Sie selbst brauchen die Funktion `init()` nicht weiter zu erklären.

Lösung

- (a) **Wieso macht das Programm überhaupt etwas, wenn doch das Hauptprogramm nach dem Initialisieren lediglich eine Endlosschleife ausführt, in der *nichts* passiert?**

Das Blinken wird durch einen Interrupt-Handler implementiert. Dieser wird nicht durch das Hauptprogramm, sondern durch ein Hardware-Ereignis (hier: Uhr) aufgerufen.

- (b) **Wieso wird die Zeile `PORTC = leds << 4`; überhaupt aufgerufen, wenn dies doch nur unter der Bedingung `counter == 0` passiert, wobei die Variable `counter` auf 1 initialisiert, fortwährend erhöht und nirgendwo zurückgesetzt wird?**

Die vorzeichenlose 8-Bit-Variable `counter` kann nur Werte von 0 bis 255 annehmen; bei einem weiteren Inkrementieren springt sie wieder auf 0 (Überlauf), und die `if`-Bedingung ist erfüllt.

- (c) **Wie kommt das oben beschriebene Blinkmuster zustande?**

In jedem Aufruf des Interrupt-Handlers wird die Variable `leds` um 1 erhöht und anschließend modulo 8 genommen. Sie durchläuft daher immer wieder die Zahlen von 0 bis 7.

Durch die Schiebeoperation `leds << 4` werden die 3 Bits der Variablen `leds` an diejenigen Stellen im Byte geschoben, an denen die LEDs an den Mikro-Controller angeschlossen sind (Bits 4, 5 und 6).

Entsprechend durchläuft das Blinkmuster immer wieder die Binärdarstellungen der Zahlen von 0 bis 7 (genauer: von 1 bis 7 und danach 0).

- (d) **Wieso dauert eine Phase ungefähr eine halbe Sekunde?**

Der Interrupt-Handler wird gemäß Hinweis 488mal pro Sekunde aufgerufen. Bei jedem 256sten Aufruf ändert sich das LED-Muster. Eine Phase dauert somit $\frac{256}{488} \approx 0.52$ Sekunden.

- (e) **Was bedeutet „`ISR (TIMER0_COMP_vect)`“?**

Deklaration eines Interrupt-Handlers für den Timer-Interrupt Nr. 0

Aufgabe 4: Objektorientierte Tier-Datenbank

```
#include <stdio.h>

#define ANIMAL 0
#define WITH_WINGS 1
#define WITH_LEGS 2

typedef struct animal
{
    int type;
    char *name;
} animal;

typedef struct with_wings
{
    int wings;
} with_wings;

typedef struct with_legs
{
    int legs;
} with_legs;

int main (void)
{
    animal *a[2];

    animal duck;
    a[0] = &duck;
    a[0]—>type = WITH_WINGS;
    a[0]—>name = "duck";
    a[0]—>wings = 2;  ← ((with_wings *) a[0])—>wings = 2;

    animal cow;
    a[1] = &cow;
    a[1]—>type = WITH_LEGS;
    a[1]—>name = "cow";
    a[1]—>legs = 4;  ← ((with_legs *) a[1])—>legs = 4;

    for (int i = 0; i < 2; i++)
        if (a[i]—>type == WITH_LEGS)
            printf ("A_%s_has_%d_legs.\n", a[i]—>name,
                    ((with_legs *) a[i])—>legs);
        else if (a[i]—>type == WITH_WINGS)
            printf ("A_%s_has_%d_wings.\n", a[i]—>name,
                    ((with_wings *) a[i])—>wings);
        else
            printf ("Error_in_animal:_%s\n", a[i]—>name);

    return 0;
}
```

Das oben in Blau dargestellte Programm (Datei: [aufgabe-2a.c](#)) soll Daten von Tieren verwalten.

Beim Compilieren erscheinen die folgende Fehlermeldungen:

```
$ gcc -std=c99 -Wall -O aufgabe-2a.c -o aufgabe-2a
aufgabe-2a.c: In function 'main':
aufgabe-2a.c:31: error: 'animal' has no member named 'wings'
aufgabe-2a.c:37: error: 'animal' has no member named 'legs'
```

Der Programmierer nimmt die oben in Rot dargestellten Ersetzungen vor (Datei: [aufgabe-2b.c](#)).

Daraufhin gelingt das Compilieren, und die Ausgabe des Programms lautet:

```
$ gcc -std=c99 -Wall -O aufgabe-2b.c -o aufgabe-2b
$ ./aufgabe-2b
A duck has 2 legs.
Error in animal: cow
```

- Erklären Sie die o. a. Compiler-Fehlermeldungen. (2 Punkte)
- Wieso verschwinden die Fehlermeldungen nach den o. a. Ersetzungen? (3 Punkte)
- Erklären Sie die Ausgabe des Programms. (5 Punkte)
- Beschreiben Sie – in Worten und/oder als C-Quelltext – einen Weg, das Programm so zu berichtigen, daß es die Eingabedaten ("A duck has 2 wings. A cow has 4 legs.") korrekt speichert und ausgibt. (4 Punkte)

Lösung

(a) **Erklären Sie die o. a. Compiler-Fehlermeldungen.**

`a[0]` und `a[1]` sind gemäß der Deklaration `animal *a[2]` Zeiger auf Variablen vom Typ `animal` (ein `struct`). Wenn man diesen Zeiger dereferenziert (`->`), erhält man eine `animal`-Variable. Diese enthält keine Datenfelder `wings` bzw. `legs`.

(b) **Wieso verschwinden die Fehlermeldungen nach den o. a. Ersetzungen?**

Durch die *explizite Typumwandlung des Zeigers* erhalten wir einen Zeiger auf eine `with_wings`- bzw. auf eine `with_legs`-Variable. Diese enthalten die Datenfelder `wings` bzw. `legs`.

(c) **Erklären Sie die Ausgabe des Programms.**

Durch die explizite Typumwandlung des Zeigers zeigt `a[0]` auf eine `with_wings`-Variable. Diese enthält nur ein einziges Datenfeld `wings`, das an genau derselben Stelle im Speicher liegt wie `a[0]->type`, also das Datenfeld `type` der `animal`-Variable, auf die der Zeiger `a[0]` zeigt. Durch die Zuweisung der Zahl 2 an `((with_wings *) a[0])->wings` überschreiben wir also `a[0]->type`, so daß das `if` in der `for`-Schleife `a[0]` als `WITH_LEGS` erkennt.

Bei der Ausgabe `A duck has 2 legs.` wird das Datenfeld `((with_legs *) a[0])->legs` als Zahl ausgegeben. Dieses Datenfeld befindet sich in denselben Speicherzellen wie `a[0]->type` und `((with_wings *) a[0])->wings` und hat daher ebenfalls den Wert 2.

Auf die gleiche Weise überschreiben wir durch die Zuweisung der Zahl 4 an `((with_legs *) a[1])->legs` das Datenfeld `a[0]->type`, so daß das `if` in der `for`-Schleife `a[1]` als unbekanntes Tier (Nr. 4) erkennt und `Error in animal: cow` ausgibt.

(d) **Beschreiben Sie – in Worten und/oder als C-Quelltext – einen Weg, das Programm so zu berichtigen, daß es die Eingabedaten ("A duck has 2 wings. A cow has 4 legs.") korrekt speichert und ausgibt.**

Damit die *Vererbung* zwischen den Objekten `animal`, `with_wings` und `with_legs` funktioniert, müssen die abgeleiteten Klassen `with_wings` und `with_legs` alle Datenfelder der Basisklasse `animal` erben. In C geschieht dies explizit; die Datenfelder müssen in den abgeleiteten Klassen neu angegeben werden (siehe `loesung-4-1.c`):

```
typedef struct animal
{
    int type;
    char *name;
} animal;

typedef struct with_wings
{
    int type;
    char *name;
    int wings;
} with_wings;

typedef struct with_legs
{
    int type;
    char *name;
    int legs;
} with_legs;
```

Zusätzlich ist es notwendig, die Instanzen `duck` und `cow` der abgeleiteten Klassen `with_wings` und `with_legs` auch als solche zu deklarieren, damit für sie genügend Speicher reserviert wird:

```
animal *a[2];

with_wings duck;
a[0] = (animal *) &duck;
a[0]->type = WITH_WINGS;
a[0]->name = "duck";
((with_wings *) a[0])->wings = 2;
```

```

with_legs cow;
a[1] = (animal *) &cow;
a[1]->type = WITH_LEGS;
a[1]->name = "cow";
((with_legs *) a[1])->legs = 4;

```

Wenn man dies vergißt und sie nur als `animal` deklariert, wird auch nur Speicherplatz für (kleinere) `animal`-Variable angelegt. Dadurch kommt es zu Speicherzugriffen außerhalb der deklarierten Variablen, was letztlich zu einem Absturz führt (siehe [loesung-4f.c](#)).

Für die Zuweisung eines Zeigers auf `duck` an `a[0]`, also an einen Zeiger auf `animal` wird eine weitere explizite Typumwandlung notwendig. Entsprechendes gilt für die Zuweisung eines Zeigers auf `cow` an `a[1]`.

Es ist sinnvoll, explizite Typumwandlungen so weit wie möglich zu vermeiden. Es ist einfacher und gleichzeitig sicherer, direkt in die Variablen `duck` und `cow` zu schreiben, anstatt dies über die Zeiger `a[0]` und `a[1]` zu tun (siehe [loesung-4-2.c](#)):

```

animal *a[2];

with_wings duck;
a[0] = (animal *) &duck;
duck.type = WITH_WINGS;
duck.name = "duck";
duck.wings = 2;

with_legs cow;
a[1] = (animal *) &cow;
cow.type = WITH_LEGS;
cow.name = "cow";
cow.legs = 4;

```