

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 31. Oktober 2022

Aufgabe 1: Zahlensysteme

Wandeln Sie ohne Hilfsmittel

- | | | |
|--------------------|----------------------------|-----------------|
| • nach Dezimal: | • nach Hexadezimal: | • nach Binär: |
| (a) $0010\ 0000_2$ | (d) $0010\ 0000_2$ | (g) 750_8 |
| (b) 42_{16} | (e) 42_{10} | (h) 42_{10} |
| (c) 17_8 | (f) $192.168.20.254_{256}$ | (i) $AFFE_{16}$ |

Berechnen Sie ohne Hilfsmittel:

- (j) $750_8 \& 666_8$
(k) $A380_{16} + B747_{16}$
(l) $AFFE_{16} >> 1$

Die tiefgestellte Zahl steht für die Basis des Zahlensystems. Jede Teilaufgabe zählt 1 Punkt.
(In der Klausur sind Hilfsmittel zugelassen, daher ist dies *keine* typische Klausuraufgabe.)

Lösung

Wandeln Sie ohne Hilfsmittel

- nach Dezimal:
 - (a) $0010\ 0000_2 = 32_{10}$
Eine Eins mit fünf Nullen dahinter steht binär für $2^5 = 32$:
mit 1 anfangen und fünfmal verdoppeln.
 - (b) $42_{16} = 4 \cdot 16 + 2 \cdot 1 = 64 + 2 = 66$
 - (c) $17_8 = 1 \cdot 8 + 7 \cdot 1 = 8 + 7 = 15$

Umwandlung von und nach Dezimal ist immer rechenaufwendig. Umwandlungen zwischen Binär, Oktal und Hexadezimal gehen ziffernweise und sind daher wesentlich einfacher.

- nach Hexadezimal:
 - (d) $0010\ 0000_2 = 20_{16}$
Umwandlung von Binär nach Hexadezimal geht ziffernweise:
Vier Binärziffern werden zu einer Hex-Ziffer.
 - (e) $42_{10} = 32_{10} + 10_{10} = 20_{16} + A_{16} = 2A_{16}$
 - (f) $192.168.20.254_{256} = C0\ A8\ 14\ FE_{16}$
Umwandlung von der Basis 256 nach Hexadezimal geht ziffernweise:
Eine 256er-Ziffer wird zu zwei Hex-Ziffern.
Da die 256er-Ziffern dezimal angegeben sind, müssen wir viermal Dezimal nach Hexadezimal umwandeln. Hierfür bieten sich unterschiedliche Wege an.
 $192_{10} = 128_{10} + 64_{10} = 1100\ 0000_2 = C0_{16}$
 $168_{10} = 10_{10} \cdot 16_{10} + 8_{10} = A_{16} \cdot 10_{16} + 8_{16} = A8_{16}$
 $20_{10} = 16_{10} + 4_{10} = 10_{16} + 4_{16} = 14$
 $254_{10} = 255_{10} - 1_{10} = FF_{16} - 1_{16} = FE_{16}$
- nach Binär:
 - (g) $750_8 = 111\ 101\ 000_2$
Umwandlung von Oktal nach Binär geht ziffernweise:
Eine Oktalziffer wird zu drei Binärziffern.

- (h) $42_{10} = 2A_{16}$ (siehe oben) = 0010 1010₁₆
 Umwandlung von Hexadezimal nach Binär geht ziffernweise:
 Eine Hex-Ziffer wird zu vier Binärziffern.
- (i) $AFFE_{16} = 1010\ 1111\ 1111\ 1110_2$
 Umwandlung von Hexadezimal nach Binär geht ziffernweise:
 Eine Hex-Ziffer wird zu vier Binärziffern.

Berechnen Sie ohne Hilfsmittel:

- (j) $750_8 \& 666_8 = 111\ 101\ 000_2 \& 110\ 110\ 110_2 = 110\ 100\ 000_2 = 640_8$
 Binäre Und-Operationen lassen sich am leichtesten in binärer Schreibweise durchführen. Umwandlung zwischen Oktal und Binär geht ziffernweise: Eine Oktalziffer wird zu drei Binärziffern und umgekehrt.
 Mit etwas Übung funktionieren diese Operationen auch direkt mit Oktalzahlen im Kopf.
- (k)
$$\begin{array}{r} A380_{16} \\ + B747_{16} \\ \hline 15AC7_{16} \end{array}$$

 Mit Hexadezimalzahlen (und Binär- und Oktal- und sonstigen Zahlen) kann man genau wie mit Dezimalzahlen schriftlich rechnen. Man muß nur daran denken, daß der „Zehner“-Überlauf nicht bei 10_{10} stattfindet, sondern erst bei $10_{16} = 16_{10}$ (hier: $8_{16} + 4_{16} = C_{16}$ und $3_{16} + 7_{16} = A_{16}$, aber $A_{16} + B_{16} = 10_{10} + 11_{10} = 21_{10} = 16_{10} + 5_{10} = 10_{16} + 5_{16} = 15_{16}$).
- (l) $AFFE_{16} >> 1 = 1010\ 1111\ 1111\ 1110_2 >> 1 = 0101\ 0111\ 1111\ 1111_2 = 57FF_{16}$
 Bit-Verschiebungen lassen sich am leichtesten in binärer Schreibweise durchführen. Umwandlung zwischen Hexadezimal und Binär geht ziffernweise: Eine Hex-Ziffer wird zu vier Binärziffern und umgekehrt.
 Mit etwas Übung funktionieren diese Operationen auch direkt mit Hexadezimalzahlen im Kopf.

Aufgabe 2: Mikrocontroller

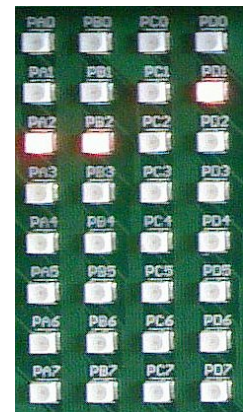
An die vier Ports eines ATmega16-Mikrocontrollers sind Leuchtdioden angeschlossen:

- 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 Programm ([aufgabe-2.c](#)):

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

int main (void)
{
    DDRA = 0xff;
    DDRB = 0xff;
    DDRC = 0xff;
    DDRD = 0xff;
    PORTA = 0x1f;
    PORTB = 0x10;
    PORTD = 0x10;
    PORTC = 0xfc;
    while (1);
    return 0;
}
```



- (a) Was bewirkt dieses Programm? (4 Punkte)
- (b) Wozu dienen die ersten vier Zeilen des Hauptprogramms? (2 Punkte)
- (c) Was würde stattdessen die Zeile `DDRA, DDRB, DDRC, DDRD = 0xff;` bewirken? (2 Punkte)
- (d) Schreiben Sie das Programm so um, daß die durch das Programm dargestellte Figur spiegelverkehrt erscheint. (3 Punkte)
- (e) Wozu dient das `while (1)`? (2 Punkte)
- Alle Antworten bitte mit Begründung.

Lösung

(a) Was bewirkt dieses Programm?

Es läßt die LEDs in dem rechts abgebildeten Muster aufleuchten, das z. B. als die Ziffer 4 gelesen werden kann.

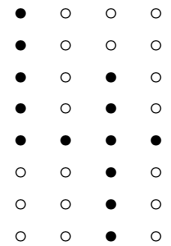
(Das Zeichen • steht für eine leuchtende, ○ für eine nicht leuchtende LED.)

Die erste Spalte (Port A) von unten nach oben gelesen (Bit 7 bis 0) entspricht der Binärdarstellung von **0x1f**: 0001 1111.

Die dritte Spalte (Port C) von unten nach oben gelesen (Bit 7 bis 0) entspricht der Binärdarstellung von **0xfc**: 1111 1100.

Die zweite und vierte Spalte (Port B und D) von unten nach oben gelesen (Bit 7 bis 0) entsprechen der Binärdarstellung von **0x10**: 0001 0000.

Achtung: Die Zuweisung der Werte an die Ports erfolgt im Programm *nicht* in der Reihenfolge A B C D, sondern in der Reihenfolge A B D C.



(b) Wozu dienen die ersten vier Zeilen des Hauptprogramms?

Mit diesen Zeilen werden alle jeweils 8 Bits aller 4 Ports als Output-Ports konfiguriert.

(c) Was würde stattdessen die Zeile **DDRA, DDRB, DDRC, DDRD = 0xff;** bewirken?

Der Komma-Operator in C bewirkt, daß der erste Wert berechnet und wieder verworfen wird und stattdessen der zweite Wert weiterverarbeitet wird. Konkret hier hätte das zur Folge, daß **DDRA, DDRB** und **DDRC** gelesen und die gelesenen Werte ignoriert werden; anschließend wird **DDRD** der Wert **0xff** zugewiesen. Damit würde also nur einer von vier Ports überhaupt konfiguriert.

Da es sich bei den **DDR**-Variablen um **volatile**-Variable handelt, nimmt der Compiler an, daß der Lesezugriff schon irgendeinen Sinn hätte. Der Fehler bliebe also unbemerkt.

(d) Schreiben Sie das Programm so um, daß die durch das Programm dargestellte Figur spiegelverkehrt erscheint.

Hierzu vertauschen wir die Zuweisungen an **PORTA** und **PORTD** sowie die Zuweisungen an **PORTB** und **PORTC**:

```
PORTD = 0x1f;  
PORTC = 0x10;  
PORTA = 0x10;  
PORTB = 0xfc;
```

Damit ergibt sich eine Spiegelung an der vertikalen Achse.

Alternativ kann man auch an der horizontalen Achse spiegeln. Dafür muß man die Bits in den Hexadezimalzahlen umdrehen:

```
PORTA = 0xf8;  
PORTB = 0x08;  
PORTD = 0x08;  
PORTC = 0x3f;
```

Die Frage, welche der beiden Spiegelungen gewünscht ist, wäre übrigens *auch in der Klausur zulässig*.

(e) Wozu dient das **while (1)**?

Mit dem **return**-Befehl am Ende des Hauptprogramms gibt das Programm die Kontrolle an das Betriebssystem zurück.

Dieses Programm jedoch läuft auf einem Mikrocontroller, auf dem es kein Betriebssystem gibt. Wenn das **return** ausgeführt würde, hätte es ein undefiniertes Verhalten zur Folge.

Um dies zu verhindern, endet das Programm in einer Endlosschleife, mit der wir den Mikrocontroller anweisen, nach der Ausführung des Programms *nichts mehr* zu tun (im Gegensatz zu: *irgendetwas Undefiniertes* zu tun).

Aufgabe 3: Einfügen in Strings

Wir betrachten das folgende Programm (aufgabe-2.c):

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

void insert_into_string (char src, char *target, int pos)
{
    int len = strlen (target);
    for (int i = pos; i < len; i++)
        target[i+1] = target[i];
    target[pos] = src;
}

int main (void)
{
    char test[100] = "Hochshule_Bochum";
    insert_into_string ('c', test, 5);
    printf ("%s\n", test);
    return 0;
}
```

Die Ausgabe des Programms lautet: `Hochschhhhhhhhhhh`

- (a) Erklären Sie, wie die Ausgabe zustandekommt. (3 Punkte)
- (b) Schreiben Sie die Funktion `insert_into_string()` so um, daß sie den Buchstaben `src` an der Stelle `pos` in den String `target` einfügt.
Die Ausgabe des Programms müßte dann `Hochschule Bochum` lauten. (2 Punkte)
- (c) Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char test[] = "Hochshule_Bochum";` ersetzen? Begründen Sie Ihre Antwort. (2 Punkte)
- (d) Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char *test = "Hochshule_Bochum";` ersetzen? Begründen Sie Ihre Antwort. (2 Punkte)

Lösung

- (a) **Erklären Sie, wie die Ausgabe zustandekommt.**
In der Schleife wird *zuerst* der nächste Buchstabe `target[i + 1]` gleich dem aktuellen gesetzt und *danach* der Zähler `i` erhöht. Dadurch wird im nächsten Schleifendurchlauf der bereits verschobene Buchstabe noch weiter geschoben und letztlich alle Buchstaben in `target[]` durch den an der Stelle `pos` ersetzt.
- (b) **Schreiben Sie die Funktion `insert_into_string()` so um, daß sie den Buchstben `src` an der Stelle `pos` in den String `target` einfügt.**
Die Ausgabe des Programms müßte dann `Hochschule Bochum` lauten.
Um im String „Platz zu schaffen“, muß man von hinten beginnen, also die Schleife umdrehen (siehe: `loesung-3.c`):

```
for (int i = len; i >= pos; i--)
    target[i + 1] = target[i];
```
- (c) **Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char test[] = "Hochshule_Bochum";` ersetzen und warum?**
Die Schreibweise `test[]` bedeutet, daß der Compiler selbst zählt, wieviel Speicherplatz der String benötigt, un dann genau die richtige Menge Speicher reserviert (anstatt, wie wir es manuell getan haben, pauschal Platz für 100 Zeichen).

Wenn wir nun in den String ein zusätzliches Zeichen einfügen, ist dafür kein Speicherplatz reserviert worden, und wir **überschreiben** dann Speicher, an dem sich andere Variable befinden, was zu einem **Absturz** führen kann.

Da wir hier nur ein einziges Zeichen schreiben, wird dieser Fehler nicht sofort auffallen. Dies ist schlimmer, als wenn das Programm direkt beim ersten Test abstürzt, denn dadurch entsteht bei uns der Eindruck, es sei in Ordnung. Wenn danach der Fehler in einer Produktivumgebung auftritt, kann dadurch Schaden entstehen – je nach Einsatzgebiet der Software u. U. erheblicher Vermögens-, Sach- und/oder Personenschaden (z. B. Absturz eines Raumflugkörpers).

- (d) **Was kann passieren, wenn Sie `char test[100] = "Hochshule_Bochum";` durch `char *test = "Hochshule_Bochum";` ersetzen und warum?**

In diesem Fall wird der Speicher für den eigentlichen String in einem unbenannten, **nicht schreibbaren** Teil des Speichers reserviert. Unser Versuch, dorthin ein zusätzliches Zeichen zu schreiben, führt dann normalerweise zu einem **Absturz**.

In manchen Systemen (Betriebssystem, Compiler, ...) ist der Speicherbereich tatsächlich sehr wohl schreibbar. In diesem Fall tritt der Absturz nicht immer und nicht immer sofort auf – genau wie in Aufgabenteil (c).