

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben 5 – 21. November 2024

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) $AFFE_{16} \gg 1 = 1010\ 1111\ 1111\ 1110_2 \gg 1 = 0101\ 0111\ 1111\ 1111_2 = 57FF_{16}$

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

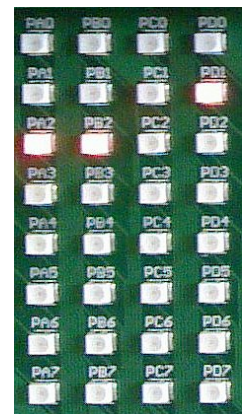
An die vier Ports eines ATmega16-Mikrocontrollers sind Leuchtdioden angeschlossen:

- 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.

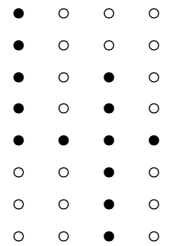
(Das Zeichen • steht für eine leuchtende, ○ für eine nicht leuchtende LED.)

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: LED-Blinkmuster

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 ();
    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 initialisiert sie die benötigten Bits an Port C als Output-Ports. Sie selbst brauchen die Funktion `init()` nicht weiter zu erklären.

Lösung

- 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