

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 7. November 2022

Aufgabe 1: 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-1-1.c` setzt diesen Zwischenschritt um.

Die Ausgabe der Ziffern erfolgt in `loesung-1-1.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-1-2.c`). Da wir wissen, daß `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-1-2.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-1-3.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-1-4.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-1-5.c` ist eine in dieser Weise abgewandelte Variante von `loesung-1-3.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-1-6.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-1-4.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 2: Länge von Strings

Strings werden in der Programmiersprache C durch Zeiger auf **char**-Variable realisiert.

Beispiel: **char** *hello_world = "Hello,_world!\n"

Die Systembibliothek stellt eine Funktion **strlen()** zur Ermittlung der Länge von Strings zur Verfügung (**#include <string.h>**).

- (a) Auf welche Weise ist die Länge eines Strings gekennzeichnet? (1 Punkt)
- (b) Wie lang ist die Beispiel-String-Konstante "Hello,_world!\n", und wieviel Speicherplatz belegt sie? (2 Punkte)
- (c) Schreiben Sie eine eigene Funktion **int strlen (char *s)**, die die Länge eines Strings zurückgibt. (3 Punkte)

Wir betrachten nun die folgenden Funktionen (Datei: **aufgabe-2.c**):

```
int fun_1 (char *s)
{
    int x = 0;
    for (int i = 0; i < strlen (s); i++)
        x += s[i];
    return x;
}
```

```
int fun_2 (char *s)
{
    int i = 0, x = 0;
    int len = strlen (s);
    while (i < len)
        x += s[i++];
    return x;
}
```

- (d) Was bewirken die beiden Funktionen? (2 Punkte)
- (e) Schreiben Sie eine eigene Funktion, die dieselbe Aufgabe erledigt wie **fun_2()**, nur effizienter. (4 Punkte)

Lösung

- (a) **Auf welche Weise ist die Länge eines Strings gekennzeichnet?**

Ein String ist ein Array von **chars**. Nach den eigentlichen Zeichen des Strings enthält das Array ein **Null-Symbol** (Zeichen mit Zahlenwert 0, nicht zu verwechseln mit der Ziffer '0') als Ende-Markierung. Die Länge eines Strings ist die Anzahl der Zeichen *vor* diesem Symbol.

- (b) **Wie lang ist die Beispiel-String-Konstante "Hello,_world!\n", und wieviel Speicherplatz belegt sie?**

Sie ist 14 Zeichen lang ('\n' ist nur 1 Zeichen; das Null-Symbol, das das Ende markiert, zählt hier nicht mit) und belegt Speicherplatz für 15 Zeichen (15 Bytes – einschließlich Null-Symbol / Ende-Markierung).

- (c) **Schreiben Sie eine eigene Funktion **int strlen (char *s)**, die die Länge eines Strings zurückgibt.**

Siehe die Dateien **loesung-2c-1.c** (mit Array-Index) und **loesung-2c-2.c** (mit Zeiger-Arithmetik). Beide Lösungen sind korrekt und arbeiten gleich schnell.

Die Warnung **conflicting types for built-in function "strlen"** kann normalerweise ignoriert werden; auf manchen Systemen (z. B. MinGW) hat jedoch die eingebaute Funktion **strlen()** beim Linken Vorrang vor der selbstgeschriebenen, so daß die selbstgeschriebene Funktion nie aufgerufen wird. In solchen Fällen ist es zulässig, die selbstgeschriebene Funktion anders zu nennen (z. B. **my_strlen()**).

- (d) **Was bewirken die beiden Funktionen?**

Beide addieren die Zahlenwerte der im String enthaltenen Zeichen und geben die Summe als Funktionsergebnis zurück.

Im Falle des Test-Strings "Hello,_world!\n" lautet der Rückgabewert 1171 (siehe **loesung-2d-1.c** und **loesung-2d-2.c**).

- (e) **Schreiben Sie eine eigene Funktion, die dieselbe Aufgabe erledigt wie `fun_2()`, nur effizienter.**

Die Funktion wird effizienter, wenn man auf den Aufruf von `strlen()` verzichtet und stattdessen die Ende-Prüfung in derselben Schleife vornimmt, in der man auch die Zahlenwerte der Zeichen des Strings aufsummiert.

Die Funktion `fun_3()` in der Datei `loesung-2e-1.c` realisiert dies mit einem Array-Index, Die Funktion `fun_4()` in der Datei `loesung-2e-2.c` mit Zeiger-Arithmetik. Beide Lösungen sind korrekt und arbeiten gleich schnell.

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

- (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