

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 9. November 2023

Aufgabe 1: Text-Grafik-Bibliothek

Schreiben Sie eine Bibliothek für „Text-Grafik“ mit folgenden Funktionen:

- **void clear (char c)**
Bildschirm auf Zeichen `c` löschen,
also komplett mit diesem Zeichen (z. B.: Leerzeichen) füllen
- **void put_point (int x, int y, char c)**
Punkt setzen (z. B. einen Stern (*) an die Stelle (x, y) „malen“)
- **char get_point (int x, int y)**
Punkt lesen
- **void display (void)**
das Gezeichnete auf dem Bildschirm ausgeben

Hinweise:

- Eine C-Bibliothek besteht aus (mindestens) einer `.h`-Datei und einer `.c`-Datei.
- Verwenden Sie ein Array als „Bildschirm“.
Vor dem Aufruf der Funktion `display()` ist nichts zu sehen;
alle Grafikoperationen erfolgen auf dem Array.
- Verwenden Sie Präprozessor-Konstante, z. B. `WIDTH` und `HEIGHT`,
um Höhe und Breite des „Bildschirms“ festzulegen:

```
#define WIDTH 72
#define HEIGHT 24
```
- Schreiben Sie zusätzlich ein Test-Programm, das alle Funktionen der Bibliothek benutzt,
um ein hübsches Bild (z. B. ein stilisiertes Gesicht – „Smiley“) auszugeben.

(8 Punkte)

Lösung

Siehe die Dateien `textgraph.c` und `textgraph.h` (Bibliothek) sowie `test-textgraph.c` (Test-Programm).

Diese Lösung erfüllt zusätzlich die Aufgabe, bei fehlerhafter Benutzung (Koordinaten außerhalb des Zeichenbereichs) eine sinnvolle Fehlermeldung auszugeben, anstatt unkontrolliert Speicher zu überschreiben und abzustürzen.

Das Schlüsselwort **static** bei der Deklaration der Funktion `check_coordinates()` bedeutet, daß diese Funktion nur lokal (d. h. innerhalb der Bibliothek) verwendet und insbesondere nicht nach außen (d. h. für die Benutzung durch das Hauptprogramm) exportiert wird. Dies dient dazu, nicht unnötig Bezeichner zu reservieren (Vermeidung von „Namensraumverschmutzung“).

Man beachte die Verwendung einfacher Anführungszeichen (Apostrophe) bei der Angabe von **char**-Konstanten (`'*'`) im Gegensatz zur Verwendung doppelter Anführungszeichen bei der Angabe von String-Konstanten (String = Array von **chars**, abgeschlossen mit Null-Symbol). Um das einfache Anführungszeichen selbst als **char**-Konstante anzugeben, ist ein vorangestellter Backslash erforderlich: `'\''` („Escape-Sequenz“). Entsprechendes gilt für die Verwendung doppelter Anführungszeichen innerhalb von String-Konstanten:
`printf("Your_name_is:_%s'", name);`

Aufgabe 2: Mikrocontroller

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

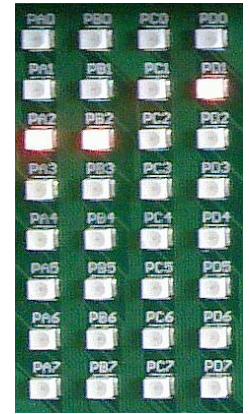
- von links nach rechts an die Ports A, B, C und D,
- von oben nach unten an die Bits Nr. 0 bis 7.

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;
}
```



- Was bewirkt dieses Programm? (4 Punkte)
- Wozu dienen die ersten vier Zeilen des Hauptprogramms? (2 Punkte)
- Was würde stattdessen die Zeile `DDRA, DDRB, DDRC, DDRD = 0xff;` bewirken? (2 Punkte)
- Schreiben Sie das Programm so um, daß die durch das Programm dargestellte Figur spiegelverkehrt erscheint. (3 Punkte)
- 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:

- (a) Wieso macht das Programm überhaupt etwas, wenn doch das Hauptprogramm nach dem Initialisieren lediglich eine Endlosschleife ausführt, in der *nichts* passiert? (1 Punkt)
- (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? (2 Punkte)
- (c) Wie kommt das oben beschriebene Blinkmuster zustande? (2 Punkte)
- (d) Wieso dauert eine Phase ungefähr eine halbe Sekunde? (2 Punkte)
- (e) 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