

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 3. Dezember 2018

Aufgabe 1: XBM-Grafik

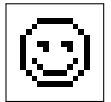
Bei einer XBM-Grafikdatei handelt es sich um ein als C-Quelltext abgespeichertes Array, das die Bildinformationen enthält:

- Jedes Bit entspricht einem Pixel.
- Nullen stehen für Weiß, Einsen für Schwarz.
- Das Bit mit Zahlenwert 1 steht für einen Bildpunkt ganz links im Byte, das Bit mit Zahlenwert 128 für einen Bildpunkt ganz rechts. (Diese Bit-Reihenfolge heißt *LSB first*.)
- Jede Zeile des Bildes wird auf ganze Bytes aufgefüllt.
- Breite und Höhe des Bildes sind als Konstantendefinitionen (**#define**) in der Datei enthalten.

Sie können eine XBM-Datei sowohl mit einem Texteditor als auch mit vielen Grafikprogrammen öffnen und bearbeiten.

Beispiel (`aufgabe-1.xbm`):

```
#define aufgabe_1_width 14
#define aufgabe_1_height 14
static unsigned char aufgabe_1_bits[] = {
    0x00, 0x00, 0xf0, 0x03, 0x08, 0x04, 0x04, 0x08, 0x02, 0x10, 0x32, 0x13,
    0x22, 0x12, 0x02, 0x10, 0x0a, 0x14, 0x12, 0x12, 0xe4, 0x09, 0x08, 0x04,
    0xf0, 0x03, 0x00, 0x00 };
```



Ein C-Programm, das eine XBM-Grafik nutzen will, kann die `.xbm`-Datei mit **#include "..."** direkt einbinden.

Schreiben Sie ein Programm, das die XBM-Datei als ASCII-Grafik ausgibt, z. B.:

```

  * * * * *
 *         *
*         *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
  * * * * *
```

(8 Punkte)

(Hinweis für die Klausur: Abgabe auf Datenträger ist erlaubt und erwünscht, aber nicht zwingend.)

Lösung

Siehe die Datei `loesung-1.c`.

Der Ausdruck $(\text{aufgabe_3_width} + 7) / 8$ ist die auf ganze Bytes aufgerundete Breite des Bildes in Bytes. Ohne das „+ 7“ in der Klammer würde stets abgerundet; die Breite von Bildern, deren Breite kein Vielfaches von 8 ist, wäre dann immer um 1 zu klein. Dadurch daß wir $n - 1$ addieren, bevor wir durch n dividieren, machen wir aus dem Abrunden ein Aufrunden.

In jedem Durchlauf der äußeren Schleife wird der Zeiger `p` auf den Anfang der Zeile `i` des Bildes gesetzt.

Innerhalb der Zeile gehen wir mit einer Bit-Maske `mask` die Bits innerhalb des Bytes `*p` von rechts nach links durch (LSB first). Wenn das Bit aus der Maske links herausgeschoben wird, setzen wir die Maske auf `0x01` zurück und setzen den Zeiger `p` auf das nächste Byte.

(Der Datentyp **unsigned char** ist eine vorzeichenlose ganze Zahl von der Größe einer Speicherzelle und normalerweise identisch mit `uint8_t`. Da das XBM-Dateiformat den Datentyp **unsigned char** verwendet, geschieht dies auch im Programm `loesung-1.c`; ansonsten wäre `uint8_t` eine bessere Wahl.)

Aufgabe 2: LED-Blinkmuster

Wir betrachten das folgende Programm für einen ATmega32-Mikro-Controller (Datei: [aufgabe-2.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