

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben 4 – 14. November 2024

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

(8 Punkte)

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, z. B.:

```
#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.

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: Datum-Bibliothek

Schreiben Sie eine Bibliothek (**.c**-Datei und **.h**-Datei) zur Behandlung von Datumsangaben.

Diese soll enthalten:

- einen **struct**-Datentyp **date**, der eine Datumsangabe speichert,
- eine Funktion **void date_print (date *d)**, die ein Datum ausgibt,
- eine Funktion **int date_set (date *d, int day, int month, int year)**, die ein Datum auf einen gegebenen Tag setzt und zurückgibt, ob es sich um ein gültiges Datum handelt (0 = nein, 1 = ja),
- eine Funktion **void date_next (date *d)**, die ein Datum auf den nächsten Tag vorrückt.

Schreiben Sie auch ein Programm, das die o. a. Funktionen testet.

(8 Punkte)

Lösung

Die Dateien [loesung-2.c](#), [loesung-2.h](#) und [loesung-2-test.c](#) enthalten die Bibliothek und das Test-Programm. Eine detaillierte Anleitung, wie man auf die Funktion [date_next\(\)](#) kommt, finden Sie im Skript zur Lehrveranstaltung, Datei [hp-2024ws.pdf](#), ab Seite 29.

Aufgabe 3: 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-3-1.c](#) setzt diesen Zwischenschritt um.

Die Ausgabe der Ziffern erfolgt in [loesung-3-1.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-3-2.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-3-2.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-3-3.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.