

Übung zum C/C++-Praktikum

Embedded C



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Übungsblatt 5.2

 Die folgendenden Aufgaben sind nur zur Bearbeitung mit dem ausleihbaren Evaluationsboard gedacht.

Aufgabe 5.2.1: [C] Testprogramm auf den Microcontroller laden

5.2.1a) Überblick

Für die Arbeit mit dem Evaluationsboard nutzen wir die Entwicklungsumgebung *WinIDEA Open*¹. Eine Installationsanleitung für Windows kann im Moodle gefunden werden. Im Vergleich zu CodeLite ist diese Umgebung speziell auf die Entwicklung von Embedded C zugeschnitten. Der Bauprozess für Embedded-C-Programme sieht teilweise anders aus als bei C++-Programmen:

1. Das Ergebnis der **Link-Phase** ist kein auf dem PC ausführbares Programm, sondern ein sogenanntes *Image*. Dieses Image wird in den statischen Speicher des Microcontrollers geladen.
2. Nach der Link-Phase folgt die **Flash-Phase** (in WinIDEA auch „Download“ genannt). Während dieser Phase wird das Image auf den Microcontroller übertragen.
3. Anschließend beginnt die **Ausführung** direkt oder man muss den *Reset*-Knopf des Boards drücken, um den Programmzähler zurückzusetzen.
4. Standardmäßig geht WinIDEA hierbei direkt in den **Debug-Modus**. Das bedeutet, dass die Ausführung des Programms am Beginn der `main`-Funktion angehalten wird.

Die weiteren Besonderheiten vom Embedded C sehen wir uns anhand eines einfachen Testprogramms an.

5.2.1b) Testprogramm

Für diese und alle weiteren Aufgaben stellen wir dir ein Codetemplate zur Verfügung, das von dir ergänzt wird. Wir beginnen mit einem kleinen fertigen Programm, das die RGB-LED des Evaluationsboards periodisch blinken lässt. Dies ist das „Hello World“-Programm der Embedded-C-Welt.

1. Entpacke und kopiere zunächst den **vorbereiteten WinIDEA-Workspace** aus dem Moodle-Kurs in ein Verzeichnis **außerhalb** deines Benutzerverzeichnisses (bspw. nach `C:\tmp`).² Falls du deinen eigenen PC verwendest, ist es sehr wichtig, dass der WinIDEA-Workspace und die verwendeten Bibliotheken (bspw. `C:\PortableApps\Cypress\PDLL`) auf dem gleichen Laufwerk liegen.

¹<http://www.isystem.com/download/winideaopen>

²Leider ist es aus technischen Gründen nicht möglich, mit WinIDEA im Benutzerverzeichnis zu arbeiten, da dieses auf ein Netzlaufwerk abgebildet wird.

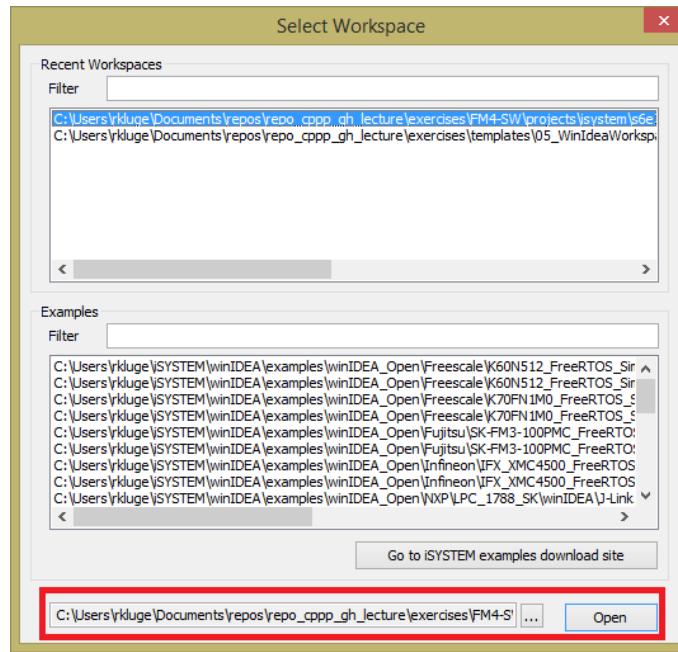


Abbildung 1: Workspace-Auswahl in WinIDEA

2. **Öffne WinIDEA:** Das Programm liegt unter `C:\PortableApps\iSYSTEM\winIDEAOpen9\winIDEA.exe`. Bei Bedarf kannst du dir eine Desktop-Verknüpfung erstellen.
3. **Wähle** in dem erscheinenden Dialogfenster den soeben **kopierten WinIDEA-Workspace** aus wie in Abbildung 1 gezeigt (roter Rahmen).
4. Nun öffnet sich der eigentliche Workspace. Du findest links einen baumartig aufgebauten Datei-Browser, der die Datei-Gruppen `lib`, `src` und `Dependencies` enthält.
 - Die Gruppe `lib` enthält manuell konfigurierte Abhängigkeiten deines Projekts, die du normalerweise nicht anpassen musst.
 - Die Gruppe `src` enthält den Quelltext, mit dem du arbeiten wirst.
 - Die Gruppe `Dependencies` enthält automatisch von WinIDEA aufgelöste Abhängigkeiten und muss nicht angepasst werden.

Öffne nun die Datei `main.c` in der Gruppe `src`.

5. Hier findest du die **main-Funktion**, deren Inhalt du im Folgenden immer wieder anpassen wirst. Im Moment delegiert sie an die Funktion `BlinkMain`. Diese ist in der Datei `blink.h` deklariert und in `blink.c` definiert.
Öffne nun die Datei `blink.c`.
6. Du findest den folgenden Quelltext (Listing 1) vor:

```

1 #include "blink.h"
2
3 #include <stdint.h>
4 #include "delay.h"
5 #include "s6e2ccxj.h"
6
7 #include "pins.h"
8
9 int BlinkMain() {
10     LED_BLUE_DDR |= (1 << LED_BLUE_PIN); // Configure blue LED pin as output.
11     LED_BLUE_DOR |= (1 << LED_BLUE_PIN); // Turn LED off.

```

```

12
13     const uint32_t sleepTime = 1000000;
14
15     // Main loop
16     while (1) {
17         // Clear bit -> Switch LED on
18         LED_BLUE_DOR &= ~(1 << LED_BLUE_PIN);
19         cppp_microDelay(sleepTime);
20
21         // Set bit -> Switch LED off
22         LED_BLUE_DOR |= (1 << LED_BLUE_PIN);
23         cppp_microDelay(sleepTime);
24     }
25 }
```

Listing 1: blink.c

- Zu Beginn wird der Pin, an den die blaue LED angeschlossen ist, als Ausgang konfiguriert (Zeile 10).
 - Dann wird der Pin auf 1 gesetzt, was die LED ausschaltet (Zeile 11).
 - In einer Endlosschleife wird die LED dann immer wieder ein- und ausgeschaltet. Zwischen den Schaltvorgängen wird eine Pause von einer Sekunde eingelegt.
7. Um das Projekt jetzt zu **compilieren** und zu **linken**, wähle im Menü *Project* den Befehl *Make* (oder drücke F7). Dieser Befehl wird – seinem Namen entsprechend – nur diejenigen Teile neu kompilieren, die sich seit dem letzten Aufruf verändert haben. Mit dem Befehl *Project → Rebuild* wird das Projekt von Grund auf neu gebaut. Der Vorgang sollte im *Output*-Fenster³ mit 0 Error(s) 0 Warning(s) enden.
 8. Um das erzeugte Image auf den Microcontroller zu flashen, verbinde den **Micro-USB-Anschluss CN2** des Boards mit dem **Data-USB-Port** des PCs. Die grüne LED in der Nähe des 2x5-Pin-Multicon-Blocks (CN12) sollte nun dauerhaft leuchten. Wähle anschließend in WinIDEA **Debug → Download** (oder Ctrl + F3).
 9. In WinIDEA wird nun ein **gelber Pfeil** neben der Signatur der **main-Funktion** erscheinen. Dies deutet an, dass du jetzt auf dem Microcontroller debuggen kannst. Setze die Ausführung einfach fort, indem du **Debug → Run Control → Run** wählst (oder F5). Die blaue LED sollte nun blinken.
 10. Falls dein Programm einmal „unsauber“ startet oder du es neu starten möchtest, kannst du den Programmzähler mithilfe des **Reset Buttons** zurücksetzen. Er befindet sich links oberhalb der MCU und ist auch mit „Reset“ beschriftet. Ganz in der Nähe befindet sich auch der **User Button**, den wir im Folgenden noch kennenlernen werden.

Herzlichen Glückwunsch – du bist nun bereit, selber Hand an die Aufgaben zu legen!

5.2.1c) LED bunt blinken lassen

Als erste eigenständige Aufgabe erweiterst du die Funktion `BlinkRainbowMain` so, dass abwechselnd alle drei möglichen LEDs angesteuert werden. Führe dazu die folgenden Schritte aus:

1. Öffne die Datei `blinkrainbow.c` (in der Gruppe `src`).
2. Nutze die Dateien `blink.h` und `blink.c` als Ausgangspunkt, um diese Aufgabe zu lösen.
3. Erweitere nun den Code so, dass du auch auf die Ports der roten und grünen LED zugreifst und die LED abwechselnd rot, grün und blau leuchtet.

³Kann über *View → Output* oder Alt+2 geöffnet werden.

Listing 2 zeigt am Beispiel der blauen LED und des linken Joystick-Buttons wie man auf IO-Pins zugreifen kann. Die Bit-Operatoren wurden bereits in der vorangegangenen Übung behandelt. Die Namen der verschiedenen Register für die LEDs und Buttons sind in der Datei `pins.h` definiert. Ausdrücke wie `LED_BLUE_DDR |= (1 << LED_BLUE_PIN)` oder `BUTTON_LEFT_DDR &= ~(1 << BUTTON_LEFT_PIN)` werden verwendet, um ein einzelnes Bit in einem Register zu setzen oder zu löschen.

```
1 #include "s6e2ccxj.h"
2 #include "pins.h"
3
4 int io_example(void) {
5     /*
6     Configure the pin of the blue LED as output by setting the corresponding
7     bit in the Data Direction Register (DDR).
8     */
9     LED_BLUE_DDR |= (1 << LED_BLUE_PIN);
10    /*
11     Set the pin to 1 by setting the corresponding bit in the Data Output
12     Register (DOR). This turns the LED off.
13     */
14     LED_BLUE_DOR |= (1 << LED_BLUE_PIN);
15
16    /*
17     Configure the pin of the left Joystick Button as input by clearing the corresponding
18     bit in the Data Direction Register (DDR).
19     */
20     BUTTON_LEFT_DDR &= ~(1 << BUTTON_LEFT_PIN);
21
22    /*
23     Enable the pull-up resistor in the pin by setting the corresponding bit
24     in the Pullup Configuration Register (PCR).
25     */
26     BUTTON_LEFT_PCR |= (1 << BUTTON_LEFT_PIN);
27
28    /*
29     Check the pin status by combining the Data Input Register (DIR) with the
30     corresponding bitmask.
31     The expression is inverted because the pin is 0 when the button is pressed.
32     */
33     if (!(BUTTON_LEFT_DIR & (1 << BUTTON_LEFT_PIN))) {
34         /*
35         Toggle the LED when the button is pressed. Toggling is done by XORing
36         the current pin state with 1.
37         */
38         LED_BLUE_DOR ^= (1 << LED_BLUE_PIN);
39     }
}
```

Listing 2: Beispiele zur Ansteuerung von Ein-/Ausgabepins

Hinweise

- Für diese und die folgenden Kennenlernaufgaben stellen wir dir Musterlösungen direkt im WinIDEA-Workspace bereit. Du findest sie in der Gruppe `solution`. Beachte, dass die (nicht-statischen) Funktionen der Musterlösung und deren entsprechende Implementierungsdateien stets auf `_s` enden, um Namenskonflikte mit den Vorlagefunktionen zu vermeiden.

Aufgabe 5.2.2: [C] Taster abfragen

In dieser Aufgabe erweiterst du die vorherige Aufgabe um eine Benutzerinteraktion über den Taster (**Switch 2**) oder einen der Analog Sticks. Ziel dieser Aufgabe ist es, mithilfe einer Tasteneingabe die RGB-LED in zwei verschiedenen Szenarien zu kontrollieren. Im ersten Szenario soll die Tasteneingabe als Lichtschalter arbeiten: Wird ein Taster einmal betätigt, schaltet sich die blaue LED ein; wird ein Taster erneut betätigt, schaltet sie sich wieder ab. Im zweiten Szenario soll die blaue LED solange leuchten, wie ein Taster gedrückt gehalten wird.

Falls der Taster auf dem Board (**Switch 2**) zum Einsatz kommen soll: Verwende die Konstanten/Register USER_BUTTON_PIN (Pin Number), USER_BUTTON_DATA (Input Register), USER_BUTTON_IO (Data Direction Register) und FM4_GPIO->PCR2_f.P0 (Pull-Up Configuration Register). Im Falle des linken Analog Sticks: Verwende die Konstanten/Register BUTTON_LEFT_PIN (Pin Number), BUTTON_LEFT_DIR (Input Register), BUTTON_LEFT_DDR (Data Direction Register) und BUTTON_LEFT_PCR (Pull-Up Configuration Register).

Anmerkung: Es muss nur eine Eingabemethode implementiert werden (Taster oder Stick). In der Musterlösung verwenden wir beispielsweise den linken Analog Stick.

Achtung: In der Mitte unten auf dem Board, direkt über dem Display, befinden sich zwei Taster. Der Taster links oben ist der richtige Taster. Der Taster rechts unterhalb ist der Reset-Button!

1. In dieser Aufgabe wirst du mit den Dateien `button.c` und `button.h` arbeiten.
2. Zunächst werden wir die nötigen Variablen in `button.c` deklarieren: Um den aktuellen Zustand der LED zu speichern wird eine vorzeichenlose 8-Bit-Integer-Variablen `ledStatus` angelegt.
3. Implementiere zunächst die Funktion `initLED()`. Diese soll `ledStatus` und den Pin der blauen LED initialisieren.
 - Der LED-Status soll zu Beginn 0 (= „aus“) sein.
 - Der Pin der blauen LED muss als Ausgang konfiguriert werden.
 - Das Daten-Register der blauen LED soll entsprechend initialisiert sein, dass die LED ausgeschaltet ist.
4. Implementiere nun die Funktion `toggleBlueLED()`. Diese soll den Status von `ledStatus` umkehren: War der Wert zuvor 1, soll er danach 0 sein und umgekehrt. Der aktuelle Status der LED soll mit `setBlueLED(uint8_t status)` gesetzt werden.
5. Implementiere nun die Funktion `isButtonPressed`, die zurück gibt, ob der Taster gerade gedrückt ist. Beachte, dass der Taster durch den Pull-Up-Widerstand genau dann gedrückt ist, wenn am Pin ein niedriger Pegel (0) anliegt. Ein Beispiel für die Abfrage des Tasters findest du in Listing 2.
6. Implementiere nun mithilfe der zuvor erstellten Hilfsfunktionen die Hauptfunktionen `ButtonToggleBlueLED()` und `ButtonHoldBlueLEDOn()`. Die erste soll dem Button die Funktion eines Lichtschalters geben und die zweite soll die LED zum Leuchten bringen, solange der Taster gedrückt gehalten wird.

Aufgabe 5.2.3: [C] Display ansteuern

In diesem Abschnitt lernst du die Ansteuerung des Displays kennen. Das Display hat eine Auflösung von 480 * 320 Pixeln. Zunächst wirst du den Bildschirm in verschiedenen Farben ausfüllen und die dafür benötigte Farbcodierung kennenlernen. Anschließend wirst du Funktionen implementieren, um auf dem Bildschirm regelmäßige Muster und Texte auszugeben. Alle in dieser Aufgabe verwendeten Funktionen müssen in der Datei `display.c` implementiert werden. Tabelle 1 dokumentiert Funktionen aus `gfx.h`, die du für diese Aufgaben nutzen kannst.

5.2.3a) Bildschirm umfärben

Das Display verwendet eine RGB565-Codierung für Farben (auch als „High Color“ bekannt⁴). Bei der RGB565-Codierung werden 5 Bits für Rot, 6 Bits für Grün und 5 Bits für Blau verwendet (siehe Abbildung 2).

Um auch eigene Farben nutzen zu können, implementierst du zunächst die Funktion `cppp_color565(uint8_t r, uint8_t g, uint8_t b)`, welche RGB888-Farben in RGB565-Farben umwandelt. Der Umwandlungsalgorithmus arbeitet für drei gegebene Bytes (`uint8_t` oder `unsigned char`) `r`, `g`, `b` wie folgt:

⁴https://en.wikipedia.org/wiki/High_color

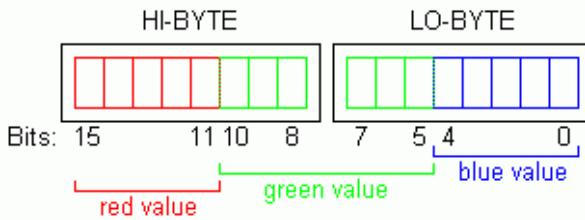


Abbildung 2: RGB565 Codierung

1. Extrahiere die höchstwertigen 5 Bits von `r`, die höchstwertigen 6 Bits von `g` und die höchstwertigen 5 Bits von `b`. Nutze den Bit-Und-Operator mit einer passenden Bitmaske, um nur die entsprechenden Bits beizubehalten und alle anderen Bits auf 0 zu setzen. Verwende anschließend den Shift-Operator, um die extrahierten Bits ans untere Ende des Bytes zu schieben. Es folgt eine Skizze für den Rot-Kanal.

```
uint8_t r = 0xA7;           // --> 0b10100111
uint8_t hiR = r & ____;     // --> 0xA0; 0b10100000
uint8_t hiRShifted = r ____; // --> 0x14; 0b00010100
```

2. Füge die extrahierten Bits mittels des Bit-Oder-Operators zusammen, um den RGB565-Wert zu erhalten: Beachte dabei, dass du mithilfe des Links-Shift-Operators die einzelnen Teile korrekt ausrichtest. Beispielsweise muss `hiRShifted` vor der Veroderung um 11 Bits nach links verschoben werden. Beachte, dass das Ergebnis 16 Bit lang sein muss und wir deshalb den Typ `uint16_t` verwenden.

```
uint16_t result = (hiRShifted ____) | (hiGShifted ____ ) | (hiBShifted ____);
```

Zum Testen deiner Implementation kannst du die Vergleichswerte in Tabelle 2 nutzen. Gehe dazu wie folgt vor:

1. Lege dir eine leere `main`-Funktion an und binde den Header `display.h` ein. Lösche oder kommentiere den Inhalt aus vorherigen Aufgaben aus.

2. Füge die folgende Zeile ein, um deinen Code für die Farbe Cyan zu testen:

```
uint16_t result = color565(0x00, 0xEA, 0xFF);
```

3. Kompiliere das Projekt wie gewohnt, aber, bevor du das Programm auf den Microcontroller lädst, setze einen *Breakpoint* in `color565`. Öffne dazu die Datei `display.c`. Rechtsklicke in die Zeile, in der die Variable `result` zurückgegeben wird. Wähle *Set Breakpoint* (oder drücke F9). Links neben der Zeile erscheint ein rotes Rechteck.

4. Lade nun den Code auf den Microcontroller. Die Ausführung sollte an dem eingestellten Breakpoint anhalten (gelber Pfeil).

5. Bewege nun den Mauszeiger über die Variable `result`. Es erscheint ein kleiner Tooltip mit dem Inhalt `result = 1887`. Dies ist das richtige Ergebnis, aber leider in einer nicht-hexadezimalen Darstellung.

6. Um dies zu ändern, öffne die *Watch View* mittels *View → Watch* (oder Alt + 3).

7. Betätige die dritte Schaltfläche von links („Toggle Hex Mode“) am oberen Rand des *Watch*-Fensters.

8. Wenn du mit dem Mauszeiger nun erneut auf `result` zeigst, erfährst du, dass `result=0x075F`.

9. Um das Programm weiterlaufen zu lassen, wähle *Debug → Run Control → Run* (oder F5).

Die Datei `display.h` (bzw. `gfx.h`) enthält einige vordefinierte Farben, die du ebenfalls bei den folgenden Aufgaben nutzen kannst: BLACK (0x0000), BLUE (0x001F), RED (0xF800), GREEN (0x07E0), YELLOW (0xFFE0), WHITE (0xFFFF).

Im letzten Teil dieser Aufgabe färbst du den Bildschirm mit der Farbe deiner Wahl.

1. Beginne wieder mit einer leeren main-Funktion.
2. Binde die Header init.h (aus include) ein und füge den folgenden Initialisierungscode für das Board in main ein:

```
initBoard();
```

3. Um das Display in Cyan einzufärben, füge folgende Zeile ein:

```
cppp_fillScreen(color565(0x00, 0xEA, 0xFF));
```

Hinweis: Die Funktion void cppp_fillScreen(int16_t color) ist in gfx.h deklariert.

5.2.3b) Regelmäßiges Muster ausgeben

In diesem Abschnitt implementierst du die Funktion `void printPattern(backgroundColor, foregroundColor)`. Diese Funktion ordnet auf dem Display Quadrate (4 x 4 Pixel) als Schachbrettmuster an.

Hinweise

- Nutze die Funktion `cppp_fillScreen`, um zunächst den Bildschirm in der Farbe `backgroundColor` zu füllen. Nutze anschließend die Funktion `cppp_fillRect`, um die einzelnen Quadrate in der Farbe `foregroundColor` zu erzeugen.
- Eine einfache Möglichkeit ist, zwei geschachtelte `for`-Schleifen zu verwenden, die den Schleifenzähler in X-Richtung jeweils um die Blockgröße und in Y-Richtung um die doppelte Blockgröße weiterschalten.

5.2.3c) Text ausgeben

In diesem Aufgabenteil wirst du einen Cursor implementieren, der es erlaubt, auf einfache Art und Weise Zeichen auf dem Bildschirm auszugeben. Zum Anzeigen von Buchstaben auf dem Bildschirm stellen wir dir eine kleine Fontbibliothek

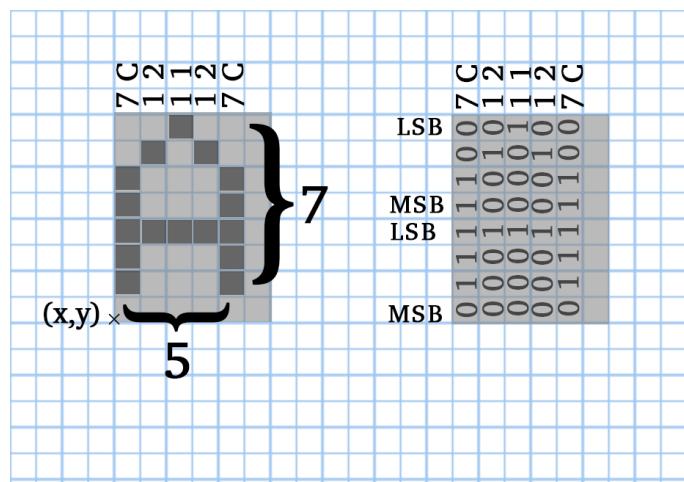


Abbildung 3: Schematische Darstellung des Zeichens 'A' in einer ASCII-5*7-Schriftart

im Extended-ASCII 5*7-Format zur Verfügung. Du findest diese in der Datei glcdfont.h (Gruppe include). Die

Bibliothek besteht aus einem Array mit 255 Buchstaben, die jeweils als 5 Bytes abgespeichert sind (siehe Abbildung 3). Das 5*7-Format eignet sich für den 480 * 320 großen Bildschirm, da die Darstellung dieser Schriftart mit einer Fläche von 35 Pixeln je Buchstaben immer noch gut lesbar ist. Wenn man von einem Pixel Abstand zwischen den Buchstaben ausgeht, passen auf das Display bis zu 3200 Zeichen.

Du wirst in dieser Aufgabe mehrere Funktionen in der Datei `display.c` implementieren. Der Koordinatenursprung (0, 0) des Bildschirms ist links unten. In der Vorlagendatei `display.c` ist die Fontbibliothek `glcdfont.h` bereits eingebunden. Um globale Grundeinstellungen für die Schriftart festzulegen, wurden bereits die Variablen `cursorX`, `cursorY`, `textColor`, `textSize` und `textBackground` deklariert.

1. Implementiere die Setter für den Cursor (`setCursor(int16_t x, int16_t y)`), die Textfarbe (`setTextColor(uint16_t c)`), die Textgröße (`setTextSize(uint8_t s)`) sowie die Hintergrundfarbe (`setBackgroundColors(int bg)`).

Zudem sollen in der Funktion `initCursor` folgende Grundeinstellungen vorgenommen werden: Der Cursor ist im Koordinatenursprung, die Textfarbe ist Weiß, die Hintergrundfarbe ist Schwarz und die Textgröße ist 2.

2. Die Funktion `drawChar(x, y, c, color, bg, size)` soll einen Buchstaben `c` in ASCII-Schriftart auf dem Display an der Position `x, y` in der Farbe `color` mit der Hintergrundfarbe `bg` und in der Größe `size` abbilden.

- Zunächst muss überprüft werden, ob die angegebene Position gültig ist. Sofern die Werte die Grenzen des Bildschirms überschreiten, soll die Funktion `drawChar` beendet werden.
- Als nächster Schritt soll auf die richtige Stelle des Arrays `font` zugegriffen werden und die gesetzten Bits der Hexadezimalwerte als farbiges Pixel interpretiert werden. Abbildung 3 zeigt den Aufbau der Daten im Array `font` am Beispiel des Buchstabens 'A'. Ein beliebiger Buchstabe `c` ist in `font` an der Position `c * 5` gespeichert. Für diesen Buchstaben werden 5 Byte/40 Bits durchliefert. Für jedes gesetzte Bit wird an der entsprechenden Stelle ein Pixel (für `size == 1`) oder Rechteck (für `size > 1`) auf dem Display ausgegeben.
- Zusätzlich soll nach dem Buchstaben ein Leerraum gesetzt werden. Der Leerraum soll die Breite `size` haben.

3. Um die Textausgabe auf dem Display für Strings zu ermöglichen, müssen weitere Funktionen implementiert werden, die einen automatischen Cursor verwenden. Dieser Cursor speichert die Position des letzten geschriebenen Buchstabens. Die Funktion `writeAuto(char c)` soll die Aufgabe übernehmen, einen Buchstaben `c` auf dem Display mithilfe von `drawChar` zu schreiben und die Cursorposition zu verändern. Hierbei sind Display-Grenzen und Zeilenumbrüche zu beachten.
4. Implementiere die Funktion `writeText(const char *text)`, welche einen C-String als Parameter erhält und diesen mithilfe von `writeAuto` auf das Display schreibt.
5. Implementiere abschließend die Funktion `writeTextln(const char *text)`, die sich ähnlich wie `writeText` verhält, am Ende der Textausgabe jedoch zusätzlich einen Zeilenumbruch einfügt.
6. Implementiere zur Ausgabe von Zahlen die Funktionen `writeNumberOnDisplay(const uint8_t *value)` und `writeNumberOnDisplayRight(const uint8_t *value)`. Erstere soll die per Pointer übergebene Zahl linksbündig ausgeben, Zweitere soll die Zahl rechtsbündig ausgeben. Rechtsbündig bedeutet hier, dass die Ausgabe immer die gleiche Breite hat, egal ob der Wert 0 oder 255 ist (also: 3 Ziffern). Verwende die Funktion `sprintf` (aus `stdio.h`⁵), um den per Zeiger übergebenen Wert in ein `char`-Array zu schreiben und anschließend mittels `writeText` auf dem Display aus.
7. Implementiere die Funktion `write16BitNumberOnDisplay(const uint16_t *value, uint8_t mode)` die eine 16 Bit Zahl auf dem Display ausgeben soll. Hierbei soll zwischen linksbündig und rechtsbündig Schreibweise unterschieden werden. Gilt `mode == 1`, dann soll die Zahl linksbündig angezeigt werden, sonst rechtsbündig.

⁵<https://cplusplus.com/reference/cstdio/sprintf/>

Hinweise

- Falls du Sonderzeichen darstellen möchtest, kannst du dich an folgender Tabelle orientieren: <http://www.theasciicode.com.ar/american-standard-code-information-interchange/ascii-codes-table.png>.

Um beispielsweise ein 'ß' auszugeben, definierst du es einfach als `char ss = '\xE1';`

Tabelle 1: Wichtige Funktionen und Variablen für das Display (aus gfx.h)

| Funktionen/Variablen | Beschreibung |
|---|--|
| <code>void cppo_drawPixel(int16_t x, int16_t y, uint16_t color)</code> | Zeichnet ein Pixel an x,y mit der Farbe color. |
| <code>void cppo_drawLine(int16_t x0, int16_t y0, int16_t x1, int16_t y1, uint16_t color)</code> | Zeichnet eine Linie von x0,y0 nach x1,y1 mit der Farbe color. |
| <code>void cppo_drawRect(int16_t x, int16_t y, int16_t w, int16_t h, uint16_t color)</code> | Zeichnet die Umrandung eines Rechtecks in der Farbe color mit der Breite w und der Höhe h an die Stelle x,y. |
| <code>void cppo_fillRect(int x1, int y1, int w, int h, uint16_t fillcolor)</code> | Zeichnet ein ausgefülltes Rechteck in der Farbe fillcolor mit der Breite w und der Höhe h an die Stelle x,y. |
| <code>void cppo_fillCircle(int x0, int y0, int r, unsigned int color)</code> | Zeichnet einen ausgefüllten Kreis in der Farbe fillcolor mit dem Radius r an die Stelle x,y. |
| <code>void cppo_drawTriangle(int16_t x0, int16_t y0, int16_t x1, int16_t y1, int16_t x2, int16_t y2, uint16_t color)</code> | Zeichnet die Umrandung eines Dreiecks in der Farbe color mit den Eckpunkten x0,y0; x1,y1; x2,y2. |
| <code>void cppo_fillScreen(uint16_t color)</code> | Füllt den gesamten Bildschirm mit der Farbe color. |

Tabelle 2: RGB565-Farbwerthe

| Farbe | RGB888 | RGB565 |
|--------|----------|--------|
| Cyan | 0x00EAFF | 0x075F |
| Rosa | 0xFC00FF | 0xF81F |
| Orange | 0xFFB400 | 0xFDA0 |

Aufgabe 5.2.4: [C] Joysticks abfragen

In diesem Abschnitt nutzen wir die Funktionen der vorigen Aufgabe, um die analogen Werte der Joysticks auszugeben. Darauf aufbauend entwickelst du eine Aufgabe, die mithilfe des Joysticks die Farbe der RGB-LED verändert. Die zu implementierenden Funktionen befinden sich in der Datei `joystick.c`, deren Deklarationen befinden sich in `joystick.h`. Solltest du die vorherige Aufgabe nicht (vollständig) bearbeitet haben, kannst du auf die Musterlösung in der Datei `display_s.c` zurückgreifen. Um diese statt deiner eigenen Lösung zu nutzen, hängst du an jeden Funktionsnamen das Suffix `_s` an (bspw. `writeChar_s` statt `writeChar`).

5.2.4a) Analoge Werte auf dem Display anzeigen

Jeder Joystick besitzt zwei analoge Leitungen, welche die X- oder Y-Position des Steuerknüppels auslesen. Die analogen Werte entsprechen dabei der Spannung des jeweiligen Drehpotentiometers der Achse, welche zwischen 0 V und 5 V liegt. Die Spannungswerte der Joysticks werden durch den Analog-Digital-Wandler des Microcontrollers auf einen 8-Bit-Wert abgebildet (Wertebereich: 0 bis 255). Die Aufteilung der Wertebereiche sowie die Orientierung der X- und Y-Richtung sind in Abbildung 4 dargestellt. Die Aufteilung ist nicht gleichmäßig. Stattdessen ergibt sich in Neutralstellung der Wert

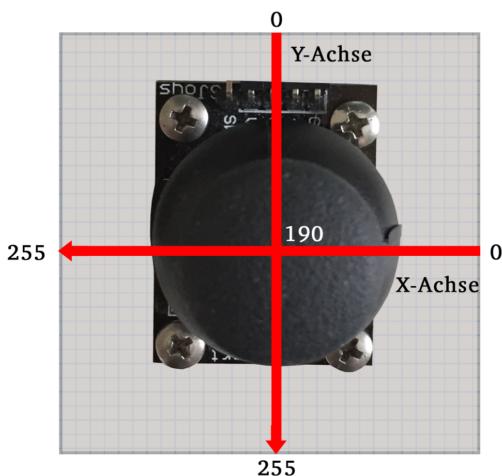


Abbildung 4: Wertebereich der Joysticks

(190, 190). Joystick 1 ist mit den analogen Anschlüssen AN16 (X) und AN19 (Y) und Joystick 2 ist mit AN13 (X) und AN23 (Y) verbunden.

1. Implementiere die Funktion `printValues`, welche über Zeiger auf die analogen Leitungen AN13, AN16, AN19 und AN23 die Werte der Joysticks ausliest und diese auf dem Bildschirm ausgibt. Nutze dazu die Funktionen in Tabelle 3 und folgendes Codefragment, mithilfe dessen man die Analog-Kanäle ausliest:

```
// #include "analog.h"
uint8_t analog11;
uint8_t analog12;
uint8_t analog13;
uint8_t analog16;
uint8_t analog19;
uint8_t analog23;
uint8_t analog17;
cppp_getAnalogValues(&analog11, &analog12, &analog13, &analog16, &analog17, &analog19, &analog23);
```

2. Um fortlaufend die Positionsdaten des Joysticks auszugeben, rufe `printValues` in einer Schleife auf:

```

#include "init.h"

int main() {
    initBoard();
    while (1) {
        printValues();
        cppp_microDelay(1000);
    }
}

```

Tabelle 3: Wichtige Funktionen und Variablen für die Verwendung der Joysticks

| Funktionen/Variablen | Beschreibung |
|---|--|
| setCursor(0, 319) | Setzt den Cursor auf die linke obere Ecke |
| void writeTextln(char *text) | Schreibt text auf das Display und verschiebt den Cursor mit Zeilensprung |
| void writeText(char *text) | Schreibt text auf das Display und verschiebt den Cursor ohne Zeilensprung |
| void writeNumberOnDisplayRight(const uint8_t *number) | Schreibt die per Pointer referenzierte Zahl number an die Position des Cursors und verschiebt den Cursor |

5.2.4b) LED mit Joystick 1 kontrollieren

In dieser Aufgabe soll die Funktion `controlLEDs` geschrieben werden, um die Farbe der RGB-LED durch die Bewegung des Joysticks 1 nach links oder rechts zu verändern. Tabelle 4 zeigt die verschiedenen möglichen Wertebereiche mit den anzusteuernden Ports und Pins der LEDs (wie in `pins.h` definiert).

Tabelle 4: Vorschläge für die Anzeigebereiche der LEDs. Diese können von Board zu Board variieren (Verschmutzung, schlechte Kontakte, defekte Kabel, etc.).

| Position des Joystick | LED-Farbe | Werbereich AN16 | Daten-Port | Ausgabe-Pin |
|-----------------------|-----------|-----------------|---------------|---------------|
| Links | Grün | 255 ... 180 | LED_GREEN_DOR | LED_GREEN_PIN |
| Mitte | Blau | 180 ... 140 | LED_BLUE_DOR | LED_BLUE_PIN |
| Rechts | Rot | 140 ... 0 | LED_RED_DOR | LED_RED_PIN |

1. Implementiere zunächst die Hilfsfunktion `controlLEDsInit`, welche die Leitungen der RGB-LEDs initialisiert. Die analogen Kanäle der LEDs sollen ausgeschaltet werden. Definiere die Pins der LEDs als Ausgänge und initialisiere sie mit `1u` (= „aus“).
2. Implementiere nun die Funktion `controlLEDs`, welche die Position der X-Achse des Joystick über den analogen Kanal AN16 ausliest und die Farbe der RGB-LED gemäß Tabelle 4 verändert.
3. Dein Testcode sollte in etwa wie folgt aussehen:

```

#include "init.h"

int main() {
    initBoard();
    controlLedsInit();
    while (1) {
        controlLeds();
        delay(1000);
    }
}

```

Aufgabe 5.2.5: [C] Touchscreen ansteuern

Eingebaut im mit dem Microcontroller verbundenen Bildschirm ist eine resistive 4-Wire Touchschicht. Der Name setzt sich zusammen aus den Eigenschaften dieser Schaltung. Resistiv steht für die Messung von Widerständen zum Erkennen von Touch-Gesten und 4-Wire bedeutet, dass für diese Technik vier Datenleitungen gebraucht werden. Ein resistiver 4-Wire-Touchscreen ist wie in Abbildung 5 aufgebaut.

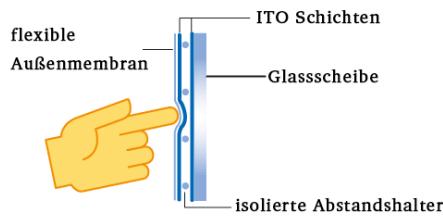


Abbildung 5: Aufbau eines resistiven Touchscreens

Dieser besteht aus 1. einer Glas- oder Acryl-Schicht, 2. einer äußeren resistiven Schicht, die mit Indium-Zinn-Oxid („indium tin oxide“, ITO) beschichtet ist, 3. isolierenden Punkten, 4. der inneren resistiven Schicht aus ITO und 5. einem Polyester-Film. Die beiden resistiven Schichten sind jeweils an 2 Polen angeschlossen (Abbildung 6).

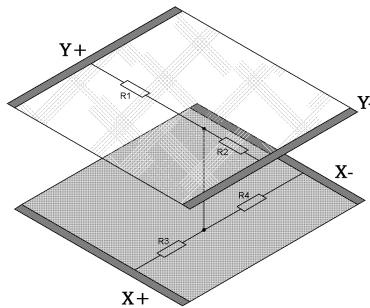


Abbildung 6: Aufbau eines 4-Wire resistiven Touchscreens

Die Schichten sind in Bezug auf ihre Pole um 90 Grad zueinander gedreht. Dies ist wichtig, um später die X- und Y-Koordinaten des Druckpunkts zu lesen. Sobald ein Objekt die oberste Glas- oder Acryl-Schicht berührt und genügend Druck ausübt, wird sich die oberste ITO-Schicht mit der unteren verbinden. Die X- und Y-Koordinate des Druckpunkts wird bestimmt, indem die Spannungen an den Polen gemessen werden. Zur Messung des X-Wertes werden X+ und X- über Gleichspannung geschaltet. Das heißt, X+ ist beispielsweise auf Vcc und X- ist mit GND verbunden. Durch die Verbindung der beiden ITO-Schichten entsteht ein Stromfluss durch beide Schichten und es kommt zu einem Spannungsteiler in der X-Schicht. Die Spannungen zwischen X+ und dem Druckpunkt sowie dem Druckpunkt und X- lassen sich durch das Ausmessen von Y- und Y+ bestimmen. Diese Information wird vom Microcontroller ausgelesen, der die gemessene Spannung in Relation zur Auflösung des Displays setzt. Die Y-Koordinate wird gemessen, indem Y+ und Y- an eine Gleichspannung gelegt und die Spannungen an X+ und X- ausgelesen werden.

Für dein Projekt stellen wir dir die Funktionen `cppo_readTouchX()`, `cppo_readTouchY()` und `cppo_readTouchZ()` zur Verfügung, die X-, Y- und Z-Werte eines Druckpunkts auf dem Touchscreen auslesen können (*analog.h*). Ist der Z-Wert größer als ein bestimmter Grenzwert, kann von einer Berührung des Touchscreens ausgegangen werden.

5.2.5a) Werte des Touchscreens debuggen

Implementiere zunächst die Funktion `debugTouch()`, die kontinuierlich die X-, Y- und Z-Werte des Touchscreens auf

dem Bildschirm ausgibt. Zur Ausgabe empfehlen wir die Funktion `cppp_write3Digits16Bit(...)` aus `gfx.h`.

5.2.5b) Zeichnen auf dem Touchscreen

In diesem Abschnitt implementierst du eine kleine Mal-Anwendung für den Touchscreen. Mit dieser soll es möglich sein, verschiedene Farben auszuwählen und mithilfe des Fingers auf dem Bildschirm zu zeichnen. Vervollständige hierfür die Funktion `paintTouch()` sowie `loopPaintTouch()`. Bereits implementiert sind die Farbpaletten und der Lösch-Button auf der unteren Seite des Bildschirms. Die Funktion `loopPaintTouch()` muss noch um eine Touch-Logik ergänzt werden, die Berührungs punkte auf dem Bildschirm erkennt und diese korrekt interpretiert. Hierbei gibt es folgende Szenarien:

- Die Farbpalette wird berührt und somit verändert sich die aktuelle Malfarbe.
- Bild erneuern wurde gedrückt und der Malbereich wird zurückgesetzt.
- Wird der freie Zeichen-Bereich berührt, so soll an dieser Stelle in der ausgewählten Farbe ein ausgefüllter Kreis mit dem Radius `PENRADIUS` gezeichnet werden.

Abbildung 7 zeigt, wie die fertige Anwendung aussehen kann.

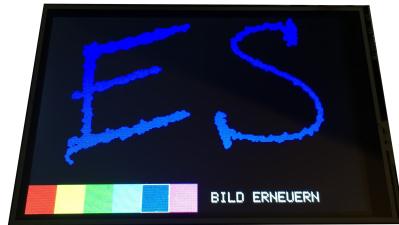


Abbildung 7: Mal-Anwendung auf dem Touchscreen

Aufgabe 5.2.6: [C] Beschleunigungssensor (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C und ist nicht notwendig, um die Klausur zu bestehen.

Auf dem Evaluationsboard ist ein Beschleunigungssensor integriert, den du verwenden kannst um die Orientierung des Entwicklungsbords auszulesen. Der Beschleunigungssensor wird von einem eigenen Microcontroller ausgelesen. Die Kommunikation zwischen FM4 und diesem Prozessor findet über die I2C-Schnittstelle statt.

In der Gruppe *lib* findest du die Datei `acceleration_app.h`, mit der du den Beschleunigungssensor für deine eigene Applikation verwenden kannst. In dieser Aufgabe wirst du mithilfe des Beschleunigungssensors eine digitale Wasserwaage simulieren.

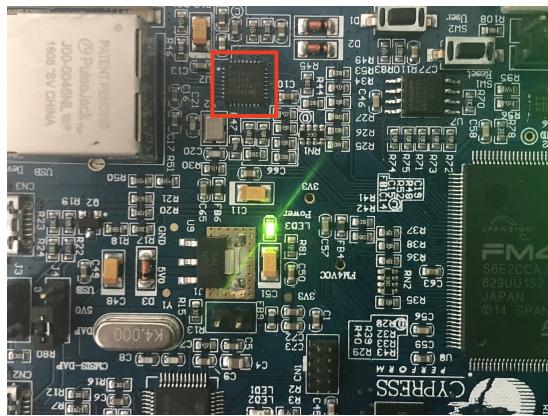


Abbildung 8: Der Beschleunigungssensor des FM4-Boards (rot umrandet)

In der Datei `acceleration.c` (Gruppe *src*) findest du eine Vorlage für diese Aufgabe. In dieser fehlt noch die fertige Implementierung der Funktion `cppp_rgbLEDAcceleration`. Bei der Initialisierung des Boards wird eine Interrupt-Routine gestartet, sobald neue Messdaten des Beschleunigungssensor des Chips vorliegen. Sofern die Routine ausgelöst wurde, wird die globale Variable `cppp_accelerationDataAvailable` auf 1 gesetzt. In dem Array `float cппp_orientationValues[3]` werden die aktuellen X-,Y- und Z-Achsen-Orientierungen des Boards in alphabetischer Reihenfolge gespeichert. Gehe nun wie folgt vor:

1. Gib auf dem Display kontinuierlich die aktuelle Orientierung des Boards aus. Setze hierzu den Cursor per `setCursor` an die Position `(0, 319)` (linke obere Ecke des Displays) und gib die Daten in folgendem Format aus:

```
/** Beschleunigungssensor ***
//Orientierung X:
//Orientierung Y:
//Orientierung Z:

//Die Ebene ist (nicht) waagrecht.
```

Hinweise

- Verwende zur Ausgabe von Variablen des Typs `float` auf den Display die Funktion `cppp_writeFloat` (im Header `gfx.h`).
2. Nutze die Ergebnisse aus Teilaufgabe a um ein Schema zu erkennen, wann das Board sich nicht im Gleichgewicht befindet. Setze die RGB-LED auf Rot sofern sich das Board nicht im Gleichgewicht befindet und auf Grün falls ja. Gib ebenfalls auf dem Display aus, ob sich das Board im Gleichgewicht befindet (siehe Beispielformat in Teilaufgabe a).

Hinweise

- Zum Ansteuern der RGB-LED kannst du in dieser Aufgabe als Vereinfachung den Header `rgbled.h` (Gruppe: `lib`) verwenden. Hierzu muss zunächst die Funktion `cppp_initLEDs` einmalig aufgerufen werden. Beispielsweise kannst du mithilfe der Funktionen `cppp_redLEDOn` und `cppp_redLEDOff` die rote LED ein- bzw. ausschalten.



Dieses Werk ist unter einer Creative Commons Lizenz vom Typ Namensnennung - Nicht kommerziell - Keine Bearbeitungen 4.0 International zugänglich. Um eine Kopie dieser Lizenz einzusehen, konsultieren Sie <http://creativecommons.org/licenses/by-nc-nd/4.0/> oder wenden Sie sich brieflich an Creative Commons, Postfach 1866, Mountain View, California, 94042 USA.