

## Aufgabenblatt 06 Termine: KW 24, KW 25

Gruppe	
Name(n)	Matrikelnummer(n)

### 6 Elektrische Ansteuerung eines Displays

Flüssigkristalldisplays (LCDs) sind ein häufig verwendetes Ausgabegerät für eingebettete Systeme. Entsprechend des Bedarfes werden einfache Zeichen-Displays oder auch grafische Displays eingesetzt. Für die späteren Aufgaben der Übungen zu ES werden Sie ein grafisches 1.8" TFT-LCD (Thin-film-transistor liquid-crystal display) verwenden, um darauf grundlegende Funktionalitäten zur visuellen Ausgabe zu implementieren.

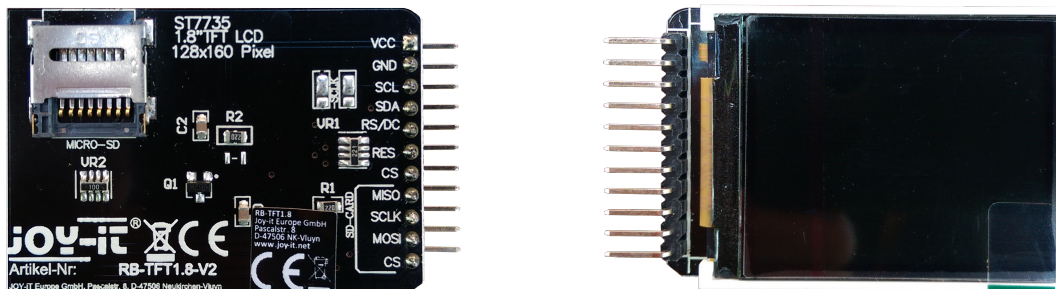


Abbildung 1: 1.8" TFT-Display mit zusätzlichem SD-Card-Slot.

Die Schnittstelle dieses **Displays** wird durch den Displaycontroller (**Sitronix ST 7735R**) bereitgestellt. Über die äußere Beschaltung des Displaycontrollers auf dem Breakout-Board des in den ES Übungen verwendeten Displays (siehe Abb. 1) ist die serielle 4-Draht Schnittstelle hardwareseitig fest konfiguriert.

#### Entwicklung einer Bibliothek für ein ST 7735R TFT-Display

Die Kommunikation mit dem Display kann über den **SPI-Bus** des Arduino realisiert werden, obwohl der Displaytreiber nicht vollständig kompatibel mit dem SPI Protokoll ist. Dadurch ist nur die Kommunikation von Arduino zu Display möglich, nicht anders herum. Für die meisten

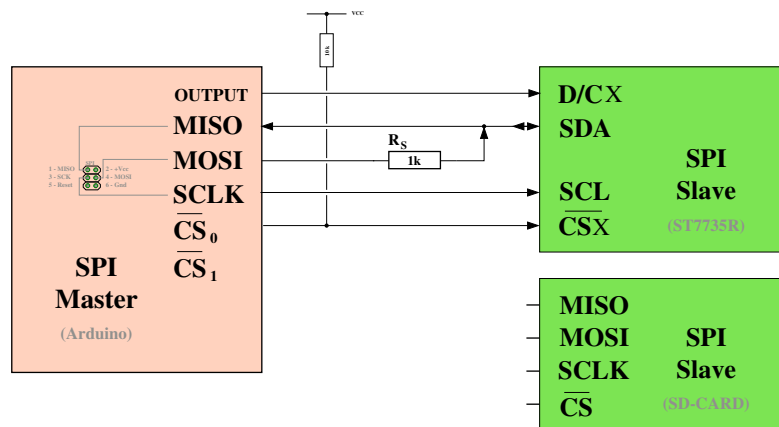


Abbildung 2: Anschlussvorschlag des Displays mit ST7735R-Controller

Aufgaben reicht das allerdings aus. Entsprechend der Abbildung 2 lassen sich die Verbindungen mit dem SPI-Bus realisieren.

Bei einer Umsetzung der Vorschläge zur Verdrahtung beachten Sie bitte folgende Hinweise:

- Verbinden Sie **VCC** mit 3.3 V (DUE) bzw. 5 V (MEGA) und stellen Sie zusätzlich die Masseverbindung **GND** her.
- Verbinden Sie **SCL** (Serial Clock) des Displays mit **SCK** (SPI Clock, Pin 3 des Headers) und **SDA** (Serial Data input/output) des Displays über den Serienwiderstand von  $1\text{ k}\Omega$  mit **MOSI** (SPI Master Out Slave In, Pin 4) des SPI-Headers des Arduino.
- **MISO** (SPI Master In Slave Out) wird für die Display-Ansteuerung (s.o.) nicht zwingend benötigt. (Ein Lesezugriff auf das Display erfordert wegen der Bidirektionalität von **SDA** Änderungen der Arduino SPI-Library).
- Verwenden Sie für die Verbindung der Anschlüsse **RS/DC** (Data/Command des Displays) und **RES** (Hardware-Reset des Displays, active-low) beliebige digitale Anschlusspins des Arduino. Im Beispieldesign wird für RS/DC Pin 8 und für RES Pin 9 verwendet.
- Verbinden Sie den Chip-Select-Pin  $\overline{\text{CS}}$  des Displays mit einem beliebigen digitalen Anschlusspin des Arduino. Üblicherweise wird hierfür der Pin 10 verwendet. Da über den Slave-Select Pin grundsätzlich auch der SPI-Master von außen als SPI-Slave konfiguriert werden kann, die SPI-Library den Arduino aber nur als Master unterstützt, darf dieser Pin nicht unkonfiguriert bleiben oder auf INPUT gesetzt werden.

In Abb. 3 wird als vom Master gesteuerter Slave Select Pin – also das Chip-Select Signal des Displays – der Pin 10 verwendet.

Das Arduino Framework macht Ihnen die Verwendung der seriellen SPI Schnittstellen mit der Bibliothek **SPI** einfach. Vergessen Sie nicht, die Bibliothek explizit in den Quellcode einzubinden (`#include <SPI.h>`).

**Hinweis:** Beim Arduino Due besitzen die Funktionen `SPI.begin`, `SPI.beginTransaction` und

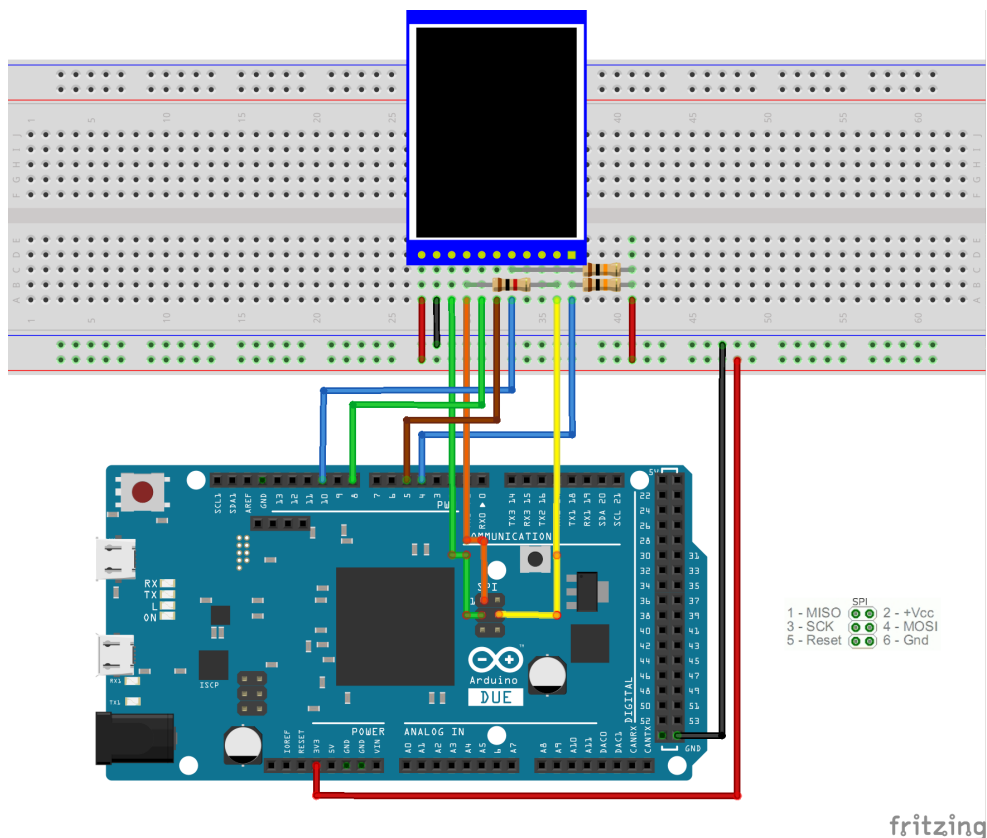


Abbildung 3: Vorschlag für die Verdrahtung des Versuchsaufbaus.

SPI.transfer zusätzlich noch einen weiteren Parameter<sup>1</sup>, den *Slave-Select* bzw. *Chip-Select* Anschlusspin (*Extended SPI library usage with the Due*).

Im Folgenden sind die Funktionen für den DUE aufgelistet, die zur Bearbeitung der Aufgabe benötigt werden:

* SPI.begin(<sce-pin>)	→ SPI.begin
* SPI.beginTransaction(<sce-pin>, <settings>)	→ SPI.beginTransaction
* SPI.endTransaction()	→ SPI.endTransaction
* SPI.transfer(<sce-pin>, <data>)	→ SPI.Transfer

### Aufgabe 6.1 Ansteuerung und Initialisierung des Displays über SPI

Setzen Sie für einen Hardwareaufbau den Verdrahtungsvorschlag aus Abb. 3 um. Beachten Sie dabei, dass der Controller des obigen Displays bereits durch das Design des Breakout-Boards des Displays, also durch das Setzen von Verbindungen beim Entwurf der Platine in Hardware, vorkonfiguriert ist. Dazu zählt beispielsweise die Auswahl der Schnittstelle, Reihenfolge der Farbübermittlung usw.

<sup>1</sup>Eine Erweiterung der grundsätzlichen Arduino der *SPI-Dokumentation*.

Auf der Website des AB TAMS ist ein [Arduino-Sketch](#) für die Initialisierung des Displaycontrollers über den SPI-Port verfügbar. Analysieren Sie diesen Code sowohl aus der Sicht der Nutzung der Arduino SPI-Library, als auch unter dem Aspekt der Kommunikation mit dem Displaycontroller. Die Dokumentation der verwendeten Befehle zur Initialisierung und Ansteuerung des Displays finden Sie im [ST7735-Datenblatt](#).

### Hinweise zur Initialisierung des Displays:

- Zunächst ist es notwendig, ein Reset des Displays durchzuführen. Beachten Sie bitte, dass die RES Leitung **low-active** ist, d.h. der Reset erfolgt nur dann, wenn der Signalpegel auf LOW gesetzt wird.

**Wichtig:** Für das Display ist eine mindestens 10us lange RESET-Phase erforderlich.

- Initialisieren Sie einen Datenübertragungsvorgang wie folgt:

```
SPI.beginTransaction(<scs-pin>, SPISettings(...))
```

SPISettings stellt eine Datenstruktur mit für die Übertragung relevanten Parametern dar, die Sie mit folgendem Konstruktor direkt initialisieren können:

```
SPISettings(<max-speed>, <data-order>, <data-mode>)
```

Geben Sie für *<max-speed>* zunächst **1 MHz** in Hertz an. Der Displaycontroller erwartet das MSB (most significant bit) zuerst; konfigurieren Sie den Parameter *<data-order>* entsprechend. Der Parameter *<data-mode>* spezifiziert die für die Datenübertragung relevanten Zustände des Taktsignals; verwenden Sie hier die entsprechende Konstante für **MODE\_0** (d.h. Bit-Übertragung bei **fallender Flanke**, Inaktivität des Taktsignals bei Signalpegel LOW).

- Übertragen Sie die Daten Byte-weise an das LC-Display:

```
SPI.transfer(<scs-pin>, <data>)
```

- Terminieren Sie den Datenübertragungsvorgang wie folgt:

```
SPI.endTransaction()
```

- Beachten Sie bitte, dass das Display zwischen **LCD-COMMAND** (D/C = LOW) und **LCD-DATA** (D/C = HIGH) unterscheidet. Bevor Sie das Display tatsächlich benutzen können, ist die folgende Initialisierungssequenz zu übertragen. Dabei ist zu beachten, dass die Befehle SWRESET und SLPOUT ca. 120 ms für die Ausführung benötigen (siehe dazu das [Datenblatt](#), ab S. 79):

1. **0x01** : SWRESET (software reset)
2. **0x11** : SLPOUT (Sleep out & booster on)
3. **0x36** : MADCTL (Memory Data Access Control)  
**0xC8** (Row-, Column-, Exchange-, Refresh-, RGB-, Refresh-Order)
4. **0x3A** : COLMOD (RGB-format, 12/16/18bit)  
**0x55** (16-Bit/pixel)
5. **0x2A** : CASET (Column adress set)  
**0x00** (first column, high order byte)

`0x02` (first column, low order byte)  
`0x00` (last column, high order byte)  
`0x81` (last column, low order byte)

6. `0x2B` : RASET (Row adress set)
  - `0x00` (first row, high order byte)
  - `0x01` (first row, low order byte)
  - `0x00` (last row, high order byte)
  - `0x80` (last row, low order byte)
7. `0x13` : NORON (Partial off (Normal))
8. `0x29` : DISPON(Display on)

- Das Display besitzt eine Auflösung von  $128 \times 160$  Pixeln (Zeilen  $\times$  Spalten). Eine Adressierung einzelner Pixel wird durch den Befehlssatz des Displaycontrollers (Befehl: RAMWR) unterstützt. Bevor in den Displayspeicher des Controllers geschrieben werden kann, muss mittels der entsprechender Befehle ein Schreib-Fenster über den Spalten- und Zeilenbereich definiert werden (siehe [Datenblatt](#), S. 61 ff).

### Aufgabe 6.2 RGB-Formate

Mit dem Befehl COLMOD der obigen Initialisierungssequenz wird die Wortbreite für einen RGB-Farbwert auf 16-Bit festgelegt, wobei das Display die Farbwerte im RGB 565 Format erwartet. Machen Sie sich mit dieser Kodierung vertraut. Wie erfolgt die Umkodierung der üblichen 24 Bit RGB-Kodierung in die 16 Bit Kodierung des 565-RGB Formates? Welche Konsequenzen folgen daraus?

### Aufgabe 6.3 Implementierung eines lokalen Grafikspeichers

Um den Kommunikationsaufwand mit dem Display gering zu halten und größere grafische Operationen schneller erledigen zu können, soll ein Pufferspeicher (Framebuffer) für das Display definiert werden. Im Pufferspeicher finden die „Zeichen- bzw. Pixeloperationen“ statt, und nach Abschluss der Operationen wird der Pufferspeicher komplett oder ggf. auch nur der veränderte Bereich an das Display übertragen.

Bedenken Sie beim Entwurf des Pufferspeichers, dass die Variablen des Arduino-Sketches zur Laufzeit im SRAM des Prozessors abgelegt werden. Sollte mehr SRAM benötigt werden, als der Prozessor zur Verfügung stellt, wird zwar der Upload des Programmes problemlos gelingen, sich das Programm aber durch Überschreiben bereits angelegter Variablen (Mehrfacheinblendung von Adressen in den physikalischen Adressraum) unerwartet und scheinbar nichtdeterministisch verhalten (siehe: [Arduino Memory](#)). Der Arduino MEGA2560 verfügt beispielsweise lediglich über ein 8 KB großes SRAM (vgl. [Arduino MEGA 2560](#)).

1. Wie groß darf das durch den Grafikspeicher abgebildete Fenster des Displays (Zeilen $\times$ Spalten) bei einem MEGA2560 mit einer 565-RGB-Kodierung sein? Berücksichtigen Sie für die übrigen Variablen einen Bedarf von 1 KB.
2. Entwerfen Sie in Anlehnung an die 565-Kodierung der Farbwerte ein nur 8 Bit breites Format. Auf diese Art ließe sich die Anzahl der darstellbaren Pixel des Bildspeichers verdoppeln.

Wie gross darf der Grafikspeicher bei einer 16 Bit-Kodierung der Farbwerte bei einem Arduino DUE mit seinem 96 KB großem SRAM und wie groß darf der Grafikspeicher bei einer 8 Bit-Kodierung auf einem Arduino MEGA2560 mit seinem 8 KB großem SRAM ausgelegt werden?

Um grafische Operationen später möglichst flexibel umsetzen zu können, soll der Pufferspeicher Pixel-weise adressierbar sein. Implementieren Sie also eine Funktion, die den gewünschten 8-Bit Farbwert des Pixels an der  $\langle x \rangle, \langle y \rangle$ -Position setzt:

- `setPixel(uint8_t x, uint8_t y, uint8_t value).`

Zur Demonstration der Korrektheit der entwickelten Funktionalität soll folgende Funktion implementiert werden:

1. Definieren Sie eine Vordergrundfarbe (`fgColor`) und eine Hintergrundfarbe (`bgColor`).
2. Initialisieren Sie den Pufferspeicher mit der Hintergrundfarbe.
3. Übertragen Sie den Pufferspeicher an das Display.
4. Beginnen Sie nun mit der ersten Spalte des Displays und **aktivieren** Sie jeden Pixel (setzen der Vordergrundfarbe) dieser Spalte.
5. Aktualisieren Sie die Darstellung des Displays.
6. Warten Sie 20ms ab, fahren Sie mit der nächsten Spalte des Displays fort und aktivieren Sie auch hier jeden Pixel.
7. Wiederholen Sie die Schritte 5-6, bis Sie jede Spalte aktiviert haben.
8. Beginnen Sie erneut mit der ersten Spalte des Displays und **deaktivieren** Sie jeden Pixel (setzen der Hintergrundfarbe) dieser Spalte.
9. Aktualisieren Sie die Darstellung des Displays.
10. Warten Sie 20ms ab, fahren Sie mit der nächsten Spalte des Displays fort und deaktivieren Sie auch hier jeden Pixel.
11. Wiederholen Sie die Schritte 9-10, bis Sie jede Spalte deaktiviert haben.
12. Fahren Sie mit Schritt 4. fort.

#### Aufgabe 6.4 Darstellung des „rotierenden Balkens“

Erweitern Sie Ihr obiges Programm um eine Funktion, die im Pufferspeicher, sowohl horizontal wie auch vertikal zentriert, einen um den Mittelpunkt rotierenden Balken mit folgenden aus dem ASCII-Zeichensatz bekannten Grundelementen erzeugt:

| → / → - → \ → ...

Diese Aktivitätsanzeige war zu DOS- und frühen Linux-Zeiten recht beliebt, da die Ausgabe rein textbasiert geschehen konnte.

Entwickeln Sie hier eine **grafische** Lösung, die sich durch Angabe des Durchmessers der resultierenden Rotationsfläche skalieren lässt. Skalieren Sie die Länge der Zeichen/Balken entsprechend. Verwenden Sie für die Ausführung der Funktion einen Hardware-Timer, so dass das Display mit einer Frequenz von **10 Hz** aktualisiert wird.

**Aufgabe 6.5** Textausgabe

Verwenden Sie den auf der Webseite zur Verfügung gestellten **ASCII-Datensatz** und implementieren Sie mithilfe der bereits existierenden Funktionalität aus der vorherigen Aufgabe folgende Funktion:

- `printChar(uint8_t x, uint8_t y, char value, uint16_t fgColor, uint16_t bgColor)`

Die Funktion soll es Ihnen ermöglichen, ein im Datensatz codiertes Zeichen an einer beliebigen Position des Grafikspeichers zu positionieren.

**ACHTUNG:** Vergessen Sie dabei nicht, die Grenzen des Pufferspeichers zu prüfen! Ihre Funktion soll einen Rückgabewert liefern, z.B. -1, falls die übergebenen Koordinaten es nicht zulassen, das Zeichen vollständig darzustellen.

Entwickeln Sie zur einfachen Ausgabe von Strings folgende weitere Funktion:

- `printString(uint8_t x, uint8_t y, char *c_str, uint16_t fgColor, uint16_t bgColor)`

Die Funktion soll keine Zeilenumbrüche produzieren. Lässt sich ein String nicht vollständig in einer Zeile darstellen, so soll die Funktion dieses mit einem entsprechenden Rückgabewert quittieren.

**Hinweis zum ASCII-Datensatz:** Der Datensatz ist ein zweidimensionales Array, das die gängigsten (nicht alle) ASCII-Zeichen im  $6 \times 8$ -Pixel Format enthält. Jede Zeile des Arrays definiert ein Zeichen durch die Angabe von 6 Bytes. Jedes dieser Bytes spezifiziert eine Spalte des  $6 \times 8$  Blocks (mit MSB = 0). Die letzte Spalte ist immer leer, Sie müssen also nicht für einen horizontalen Abstand zwischen aufeinanderfolgenden Zeichen sorgen.

**Aufgabe 6.6** Ausgabe der Namen der Gruppenmitglieder

Erstellen Sie zur Demonstration Ihrer bisherigen Arbeit eine Funktion mit dem Namen `StudentIdDemo()`, die abwechselnd die Kombination aus Vorname/Nachname und der Matrikelnummer für jeden Teilnehmer Ihrer Gruppe letztendlich auf dem Display darstellt. Der Wechsel zwischen den Datensätzen soll timergesteuert alle 5 Sekunden geschehen. Positionieren Sie die Matrikelnummer in einer neuen Zeile. Sollte Ihr Name zu lang für eine Displayzeile sein, können Sie beliebig abkürzen oder eine neue Zeile verwenden. Zentrieren Sie den Text sowohl in horizontaler, als auch in vertikaler Richtung. Lassen Sie zwischen den Zeilen fünf Pixel Abstand.