Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

Praktikum Systemprogrammierung

Versuch 6

LED-Matrix

Lehrstuhl Informatik 11 - RWTH Aachen

8. Dezember 2023

Commit: 48145a2f

Inhaltsverzeichnis

Dieses Dokument ist Teil der begleitenden Unterlagen zum *Praktikum Systemprogrammierung*. Alle zu diesem Praktikum benötigten Unterlagen stehen im Moodle-Lernraum unter https://moodle.rwth-aachen.de zum Download bereit. Folgende E-Mail-Adresse ist für Kritik, Anregungen oder Verbesserungsvorschläge verfügbar:

support.psp@embedded.rwth-aachen.de

6 LED-Matrix

In diesem Praktikumsversuch sollen Hardware-Treiber für eine farbige LED-Matrix und ein neues Eingabegerät implementiert werden. Diese werden dann für das klassische Spiel *Snake* verwendet.

6.1 Versuchsinhalte

- Konfiguration des internen AD-Wandlers
- LED-Matrixtreiber
- Joysticktreiber
- Das Spiel *Snake*

6.2 Lernziele

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Punkte:

- Ansteuerungskonzept der LED-Matrix
- Ressourceneffiziente Speicherverwendung
- Funktionsweise des Joysticks

6.3 Grundlagen

In diesem Abschnitt wird der grundlegende Aufbau der neuen Hardware erläutert.

6.3.1 LED-Matrix

Die Matrix ist aus 32×32 Pixel in einer Gitteranordnung aufgebaut. Jedes Pixel besteht aus drei LEDs, je eine für die Farben Rot, Grün und Blau. Jede LED hat ihre eigene Steuerleitung und kann entweder vollständig ein- oder ausgeschaltet werden.
Um diese insgesamt $3 \cdot 32 \cdot 32 = 3072$ Leitungen individuell ansteuern zu können, befinden sich auf der Platine der Matrix mehrere Latch-Bausteine, Schieberegister und ein Decoder. Es wird nun zunächst das Konzept dieser elektrischen Bauteile erläutert und anschließend deren Verschaltung an der Matrix.

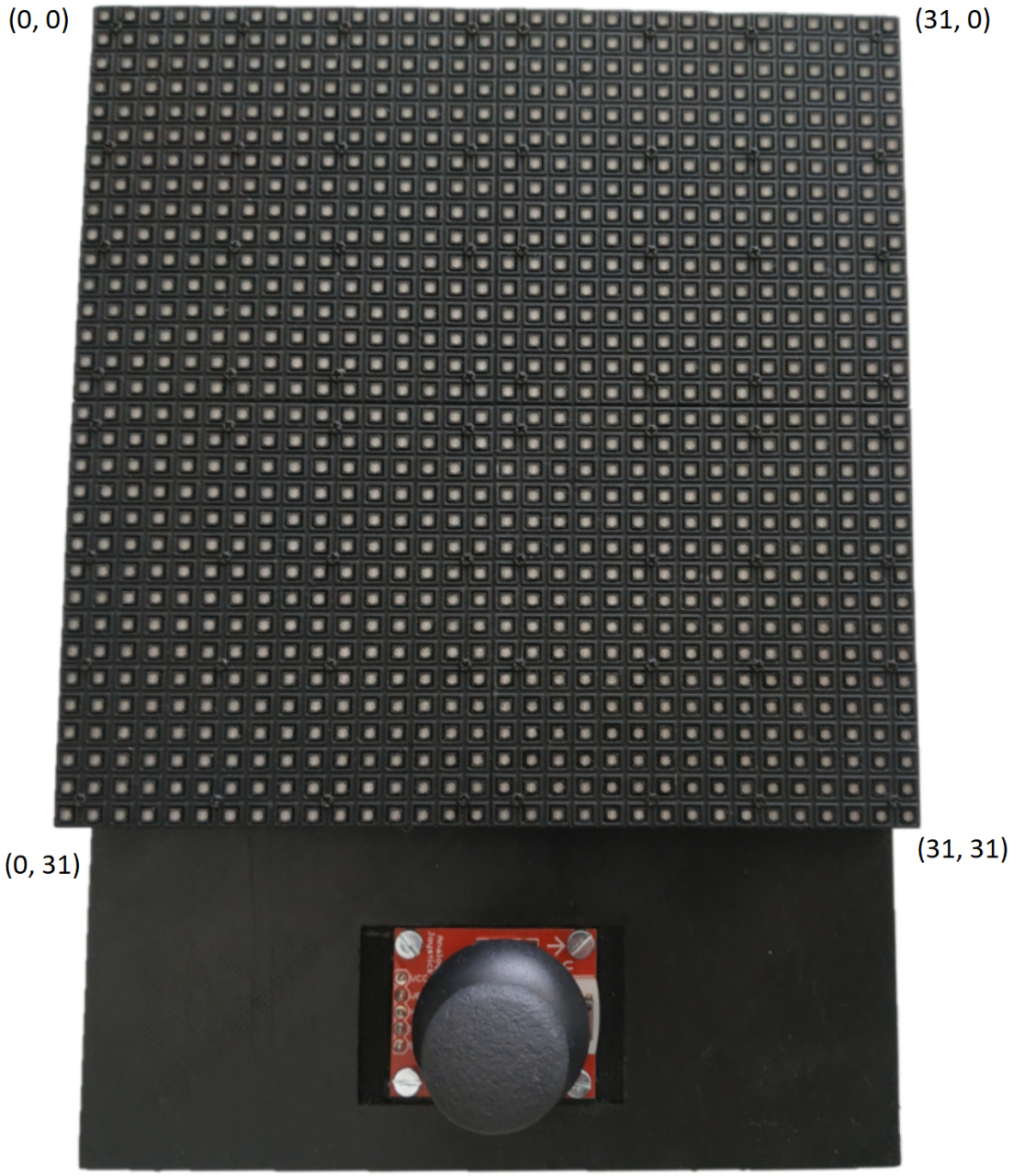

Abbildung 6.1: Die LED-Matrix und das in diesem Versuch verwendete Koordinatensystem

Latch-Baustein Ein Latch ist ein Bauteil zum Zwischenspeichern mehrerer Bits. Er besitzt gleich viele Dateneingänge wie Datenausgänge sowie die Steuerleitung *Latch Enable*

(LE). Ist LE = 1, werden die Eingänge auf die Ausgänge durchgeschaltet. Der Wert am ersten Ausgangspin entspricht also dem Wert am ersten Eingagspin usw. Wird LE = 0 gesetzt, bleiben die Ausgangspins konstant und behalten ihre Werte, unabhängig der Eingangspins.

Schieberegister Ein n-Bit Schieberegister speichert n Bits. Es hat für jedes Bit einen Ausgangpin, aber nur einen Eingangspin sowie eine Steuerleitung *Clock* (CLK). Jedes Mal, wenn die Clock von 0 auf 1 wechselt (Steigende Flanke), wird das am Eingangspin anliegende Signal in die erste Stelle des Schieberegisters eingelesen, jedes bereits gespeicherte Bit wird um eine Stelle weitergeschoben und das letzte Bit wird verworfen.

Decoder Ein n-Bit Decoder besitzt n Eingangspins und 2^n Ausgangspins. Liegt an den Eingangspins eine mit n Bits kodierte Binärzahl x an, liegt am $x+1$-ten Ausgangspin eine logische 1 an, an allen anderen eine logische 0.

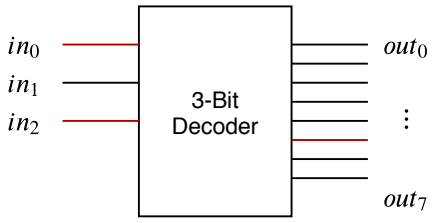

Abbildung 6.2: Ein 3-Bit Decoder, an dem eine binäre 5 anliegt

Gesamtverschaltung Die im Praktikum verwendete LED-Matrix wird durch eine Kombination der zuvor erläuterten Bauteile angesteuert. Diese sind so verschaltet, dass die Anzahl der durch den Mikrokontroller zu beschaltenden Steuerleitungen stark reduziert wird. Dadurch können allerdings immer nur zwei Zeilen gleichzeitig leuchten und alle anderen Pixel bleiben ausgeschaltet. Ein vollständiges Bild wird erzeugt, indem die gemeinsam leuchtenden Doppelzeilen so schnell hintereinander angesteuert und zum Leuchten gebracht werden, dass es aufgrund der Trägheit des menschlichen Auges so aussieht, als leuchteten alle Pixel gleichzeitig. Steuert man die Matrix zu langsam an, nimmt man jedoch ein Flackern wahr. Um diesen Effekt zu reduzieren, sind zwei gleichzeitig leuchtende Zeilen nicht zwei benachbarte Zeilen der Matrix, sondern um je 16 Zeilen verschoben. Es leuchtet daher immer eine Zeile der oberen Hälfte der Matrix und eine Zeile der unteren Hälfte der Matrix. Im Folgenden wird so ein gemeinsam leuchtendes Zeilenpaar *Doppelzeile* genannt.
Da es insgesamt 16 ($= 2^4$) Doppelzeilen gibt, besitzt die LED-Matrix vier Eingänge, um eine Doppelzeile auszuwählen. Ein 4-Bit Decoder verbindet die Kathoden der LEDs der gewählten Doppelzeile mit *Ground*, sodass nur diese bei entsprechender Beschaltung der Anode leuchten können.
Hat man eine Doppelzeile ausgewählt, müssen 192 Leitungen beschaltet werden (2 Zeilen$\cdot$ 32 Pixel $\cdot$ 3 LEDs). Dies geschieht mithilfe von insgesamt 12 MBI5024-Chips, die jeweils

16 Leitungen beschalten. In Abbildung 6.3 ist der innere Aufbau eines MBI5024-Chips schematisch dargestellt.

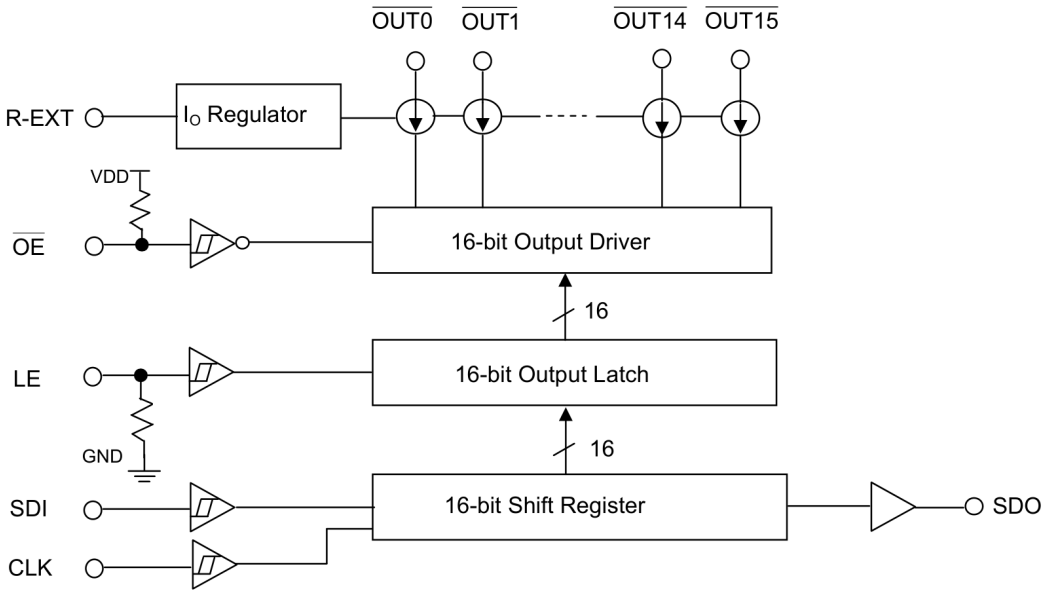

Abbildung 6.3: Blockdiagramm des MBI5024

Der MBI5024 beinhaltet ein 16-Bit Schieberegister und hat somit einen Dateneingang (`SDI`), einen Clockeingang (`CLK`) und 16 Ausgänge (`OUT0` bis `OUT15`). Er besitzt einen Ausgang `SDO`, auf dem das beim Schiebeprozess herausfallende Bit ausgegeben wird. Außerdem bietet er die Möglichkeit, an `R-EXT` einen externen Widerstand anzuschließen, um die an allen LEDs abfallende Spannung zu verändern.
Die Inhalte des Schieberegisters werden nicht direkt auf den Ausgängen ausgegeben, sondern sind zunächst mit einem 16-Bit Latch verbunden, dessen Steuerleitung `LE` am MBI5024 als Eingangspin ausgeführt ist. Die Ausgänge des Latches sind mit einem *Output Driver* verbunden, der die logischen Signale des Latches auf für die LEDs passende Spannungen umsetzt. Der Output Driver wird durch die invertierte Leitung $\overline{\texttt{OE}}$ (*Output Enable*) ein- und ausgeschaltet.

6.3.2 Aktualisierung einer Doppelzeile

Die Matrix besitzt sechs Dateneingänge (`R1`, `G1`, `B1`, `R2`, `G2`, `B2`). Jeder Dateneingang ist dazu da, die LEDs einer bestimmten Farbe der ausgewählten Doppelzeile zu beschalten. `R1`, `G1` und `B1` sind dazu da, die LEDs der Zeilen 0 bis 15 zu beschalten (obere Zeile einer Doppelzeile), `R2`, `G2` und `B2` sind analog dazu da, die Zeilen 16 bis 31 zu beschalten (untere Zeile einer Doppelzeile).
Dazu ist jeder Dateneingang mit `SDI` eines MBI5024-Chips verbunden, der mit einem weiteren MBI5024-Chip verkettet ist. Die Verkettung wird mit dem `SDO`-Ausgang des

Chips realisiert und bewirkt, dass sich die 16-Bit Schieberegister zweier Chips wie ein 32-Bit Schieberegister verhalten. Dadurch können nacheinander 32 Bits an einen Dateneingang angelegt und in die Schieberegister der zwei mit ihm verbundenen MBI5024-Chips eingelesen werden. Die Ausgänge der MBI5024-Chips sind jeweils mit den LEDs der passenden Hälfte und Farbe verbunden, abhängig davon, mit welchem Eingangspin der LED-Matrix der Chip verbunden ist. Beispielsweise ist der erste Ausgang des ersten mit `R1` verbundenen MBI5024-Chips mit allen roten LEDs der oberen Hälfte der ersten Spalte der LED-Matrix verbunden (siehe Abbildung 6.4). Es können nur die LEDs der ausgewählten Doppelzeile leuchten, da der Decoder nur diese mit *Ground* verbunden hat.

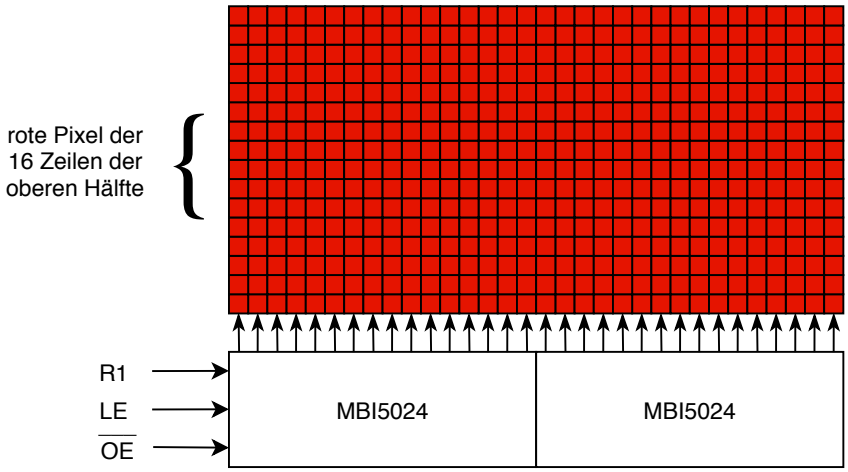

Abbildung 6.4: Die Ausgänge der Chips, die mit `R1` verbunden sind, sind mit allen roten LEDs der oberen Hälfte der Matrix verbunden.

Die Aktualisierung einer Doppelzeile läuft nun wie folgt ab: Zuerst gilt es, eine Doppelzeile auszuwählen. Dann kann man gleichzeitig 6 Bits an die Eingänge der Matrix anlegen, die Clock der Schieberegister der Chips auf 1 und wieder auf 0 ziehen, und damit die Bits einlesen. Dieser Prozess wird 32 mal wiederholt, um alle 192 Bits in den Schieberegistern zu füllen. Anschließend sollen die Inhalte des Schieberegisters im Latchbaustein gespeichert werden. Dazu muss zunächst `LE` auf 1 und dann wieder auf 0 gezogen werden. Nun kann durch Aktivieren von $\overline{\texttt{OE}}$ der gespeicherte Inhalt auf die LEDs ausgegeben werden. Die Eingänge `CLK`, `LE` und $\overline{\texttt{OE}}$ aller MBI5024-Chips sind zusammengeführt und Eingangspins der Matrix. Man führt die durch diese Steuerleitungen veranlassten Aktionen daher zeitgleich auf allen Chips durch. Abbildung 6.5 stellt den zeitlichen Signalablauf bei der Aktualisierung eines MBI5024-Chips in einem Diagramm dar.

Hinweis

- Der Output Driver darf deutlich länger angeschaltet sein als im Timing-Diagramm dargestellt. Lassen sie den Driver daher so lange es sinnvoll ist aktiviert, um Flackern zu reduzieren.
- Der Output Driver muss ausgeschaltet sein bevor die LED-Zeile gewechselt wird, da ansonsten der Inhalt der vorherigen Zeile noch in der Neuen angezeigt wird. Dies führt sonst zu Bildfehlern.

6.3.3 Joystick

Ein Joystick ist ein in zwei Dimensionen bewegbarer Steuerhebel zur Richtungseingabe. Er wird durch zwei Potentiometer realisiert, die als Spannungsteiler zwischen V_{CC} (5 V) und *Ground* (0 V) fungieren. Jedes der Potentiometer liefert somit ein analoges Signal, das die Neigung des Hebels in eine der Achsen (horizontal oder vertikal) wiederspiegelt. In Ruheposition befinden sich beide Potentiometer in mittiger Position, wodurch eine Spannung von ca. 2,5 V am Ausgang anliegt. Beim Auslenken nach oben bzw. unten liefert das Potentiometer, das zur vertikalen Achse gehört, dann eine entsprechend höhere bzw. niedrigere Spannung bis hin zu annähernd 0 V bzw. 5 V.
Als weitere Eingabemöglichkeit kann auf den Joystick gedrückt werden, wodurch ein Schalter geschlossen wird, der Pin A7 mit *Ground* verbindet (siehe Pinbelegung 6.7).

Analog-Digital-Wandlung

Aus Versuch 1 sind Sie bereits mit den Grundlagen der Analog-Digital-Wandlung vertraut. Daher wird an diesem Punkt nur noch auf die Nutzung des internen AD-Wandlers (ADC) des Mikrocontroller eingegangen. Da eine Analog-Digital-Wandlung verhältnismäßig kompliziert ist und viel interne Hardware benötigt, verfügt der ATmega644 nur über einen ADC. Um dennoch das Messen mehrerer analoger Signale zu ermöglichen, ist dieser über einen Multiplexer mit mehreren Pins (Kanälen) verbunden. Der Multiplexer wird über das *ADC Multiplexer Selection Register* (`ADMUX`) des Mikrocontrollers konfiguriert.

7	6	5	4	3	2	1	0
`REFS1`	`REFS0`	`ADLAR`	`MUX4`	`MUX3`	`MUX2`	`MUX1`	`MUX0`

Tabelle 6.1: ADC Multiplexer Selection Register (`ADMUX`)

Tabelle 6.1 veranschaulicht die Aufteilung des `ADMUX` Registers. Die einzelnen Bits werden im Folgenden erläutert. Eine genaue Beschreibung können Sie dem Datenblatt des ATmega644 in *Kapitel 21.9.1* entnehmen.

- Bit 7 und 6 dienen der Auswahl der Referenzspannung (*Reference Selection*, `REFS`).

`REFS1`	`REFS0`	Referenzspannung
0	0	Externe Referenzspannung (Pin `AREF`)
0	1	V_{CC}
1	0	Interne Referenzspannung mit 1,1 V
1	1	Interne Referenzspannung mit 2,56 V

Tabelle 6.2: Auswahl der Referenzspannung

- Bit 5 ermöglicht das Verändern der Reihenfolge, in der die 10 Bit des Ergebnisses abgelegt werden (*ADC Left Adjust Result*, `ADLAR`). Ist das Bit gesetzt, werden die

acht höheren Bits im `ADCH` abgelegt, die beiden niedrigeren im `ADCL`. Ist das Bit nicht gesetzt, werden die beiden höheren Bits im `ADCH` abgelegt, die acht niedrigeren im `ADCL`.

- Mit Bit 4 bis 0 lässt sich einstellen, an welchem Pin die analoge Spannung, die als nächstes mit dem internen ADC konvertiert werden soll, angelegt ist (*Analog Channel and Gain Selection Bits*, `MUX`).

Darüber hinaus können diverse Einstellungen im *ADC Control and Status Register A* (`ADCSRA`) vorgenommen werden. Details können Sie *Kapitel 21.9.2* im Datenblatt des ATmega644 entnehmen.

Lernerfolgsfragen

- Wofür werden die Schieberegister in der Verschaltung mit den LEDs benötigt?
- Wofür werden die Latches in der Verschaltung mit den LEDs benötigt?
- Über wie viele Pins wird die Matrix gesteuert?

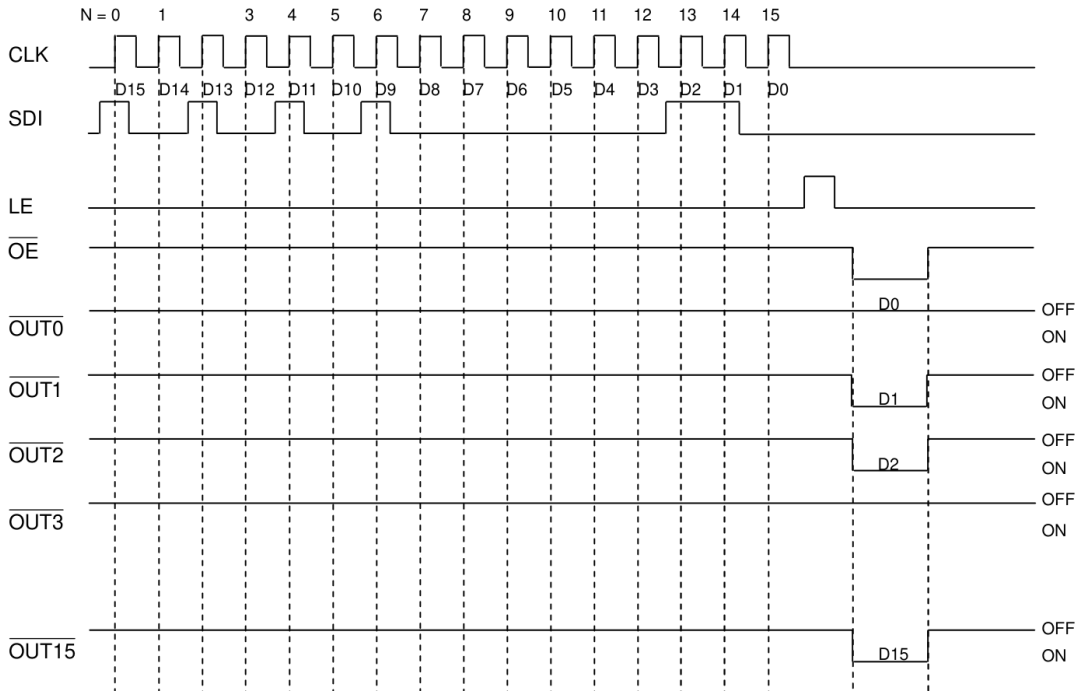

Abbildung 6.5: Signale beim Aktualisieren eines MBI5024-Chips

6.4 Hausaufgaben

Achtung

Passen Sie das `define VERSUCH` auf die aktuelle Versuchsnummer an.

Implementieren Sie die in den nächsten Abschnitten beschriebenen Funktionalitäten. Halten Sie sich an die hier verwendeten Namen und Bezeichnungen für Variablen, Funktionen und Definitionen.
Lösen Sie alle hier vorgestellten Aufgaben zu Hause mithilfe von Microchip Studio 7 und schicken Sie die dabei erstellte und funktionsfähige Implementierung über Moodle ein.

Achtung

Alle im Versuch geforderten Funktionalitäten müssen implementiert sein!
Ihre Abgabe soll dabei die `.atsln`-Datei, das Makefile sowie den Unterordner mit den `.c/.h`-Dateien inklusive der `.xml/.cproj`-Dateien enthalten. Es sollen keine Testtasks, PDFs oder Doxygen-Dateien abgegeben werden.

Beachten Sie bei der Bearbeitung der Aufgaben die angegebenen Hinweise zur Implementierung! Ihr Code muss ohne Fehler und ohne Warnungen kompilieren sowie die Testtasks mit aktivierten Compileroptimierungen bestehen. Wie Sie die Optimierungen einschalten ist dem begleitenden Dokument in Abschnitt *6.3.2 Probleme bei der Speicherüberwachung* zu entnehmen.

Achtung

Verwenden Sie zur Prüfung auf Warnungen den Befehl „Rebuild Solution“ im „Build“-Menü des Microchip Studio 7. Die übrigen in der grafischen Oberfläche angezeigten Buttons führen nur ein inkrementelles Kompilieren aus, d.h. es werden nur geänderte Dateien neu kompiliert. Warnungen und Fehlermeldungen in unveränderten Dateien werden dabei nicht ausgegeben.

Hinweis

Da die Ports B und D für andere Zwecke benötigt werden, muss in diesem Versuch von der Benutzung des externen SRAMs abgesehen werden. Somit ist es ratsam, den externen SRAM wie auch den dazugehörigen Heap aus der Liste Ihrer Speichermedien bzw. Heaps zu entfernen.

6.4.1 Ändern des LCD-Ports

Durch die Erweiterungsplatine, die den ATmega644 mit der LED-Matrix verbindet, ist das LCD in diesem Versuch nicht mit `Port A`, sondern mit `Port B` verbunden. Deswegen muss die Datei `lcd.h` angepasst werden. Dies betrifft die Zeilen 48, 51 und 54.

6.4.2 Matrixtreiber

Im Folgenden wird die Implementierung des *Matrixtreibers* erläutert, der die Beschaltung der LEDs der Matrix realisiert.

Framebuffer Um das Zeichnen auf der Matrix für das Anwendungsprogramm komfortabel zu gestalten, werden die anzuzeigenden Inhalte zunächst in einem internen Speicher, dem sogenannten *Framebuffer*, abgelegt und von dort über den Matrixtreiber auf den LEDs angezeigt. Ein Framebuffer kann eine beliebige Datenstruktur sein, in der das aktuelle auf der Matrix anzuzeigende Bild vorgehalten wird. Um den Framebuffer effizient auf die Matrix übertragen zu können, sollten die Daten jedoch so im Framebuffer abgespeichert werden, dass sie bei der Aktualisierung einer Doppelzeile direkt an Port D angelegt werden können.
Im einfachsten Fall ist der Framebuffer ein aus 32 Spalten und 16 Doppelzeilen bestehendes, globales, zweidimensionales Array. Ein Byte des Arrays speichert gleichzeitig die Farbe des Pixels an Positionen (x, y) und $(x, y + 16)$ und hat die in Tabelle 6.3 dargestellte Form. Man beachte, dass dies genau der Pinbelegung von *Port D* entspricht. Somit können wir die Farbwerte für 2 Pixel in einer Variable mit Typ `uint8_t` speichern, mit einem ungenutzten Speicherplatz von 2 Bit pro Byte.

7	6	5	4	3	2	1	0
-	-	B2	G2	R2	B1	G1	R1

Tabelle 6.3: Datenlayout einer Zelle des Framebuffers

Zur regelmäßigen Übertragung des Framebuffers zur Matrix wird die Beschaltung der Steuerleitungen unabhängig des Schedulers von einer eigenen *Interrupt Service Routine* (ISR) übernommen. Diese wird mit hoher Frequenz aufgerufen und soll in jedem Durchlauf den Inhalt einer Doppelzeile aktualisieren.

Lernerfolgsfragen

- Wie viele Bytes benötigt ein komplettes Bild der Matrix im hier beschriebenen Layout?
- Warum ist es wichtig, Farbkodierungen schnell auslesen zu können?
- Warum wird die Beschaltung von einer eigenen ISR übernommen, und nicht etwa von einem Prozess?

Farbdarstellung Da die 3 LEDs eines Pixels entweder vollständig ein- oder ausgeschaltet sind, sind nativ nur 8 verschiedene Farben pro Pixel möglich. Man kann jedoch mehr Farben darstellen, indem man die LEDs schnell pulsieren lässt. Der Rotanteil der Farbe eines Pixels ließe sich in etwa halbieren, indem die rote LED nur in jedem 2. Aktualisierungszyklus eingeschaltet wird. Allgemein beeinflusst das An-/Aus-Verhältnis einer LED die wahrgenommene Helligkeit einer Farbe.
Dieses Verhältnis ließe sich ändern, indem man den Framebuffer dupliziert und die ISR die Daten abwechselnd entweder aus dem ersten oder dem zweiten Segment auslesen lässt. Folgend nennen wir diese unterschiedlichen Segmente *Ebenen*. Das Überlagern mehrerer Ebenen erlaubt nun eine anteilige Farbmischung (siehe Abbildung 6.6).
Die Ebenen können als einzelne Arrays angelegt werden. Eleganter ist es jedoch, sie mithilfe einer zusätzlichen Dimension des oben beschriebenen Framebuffers zusammenzufassen. Der Framebuffer ist dann ein dreidimensionales Array: der erste Index selektiert die Ebene, der Zweite die Doppelzeile, der Dritte die Spalte.

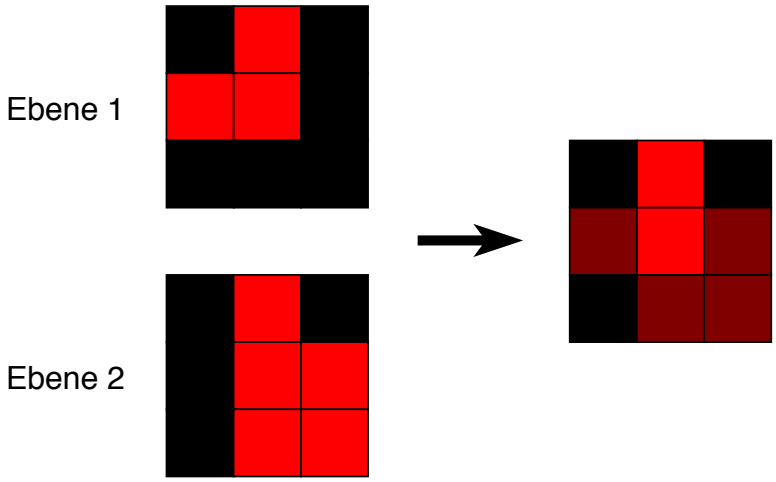

Abbildung 6.6: Farbebenen des Framebuffers

Gewichtete Ebenen Gleichmäßig gewichtete Ebenen zur Farbdarstellung sind nicht sehr effizient, da bei Vertauschung der Ebenen die gleichen Farben erzeugt werden. Somit wird der Speicher nicht optimal genutzt. Besser wäre es, dass die erste Ebene doppelt so lange bzw. doppelt so oft angezeigt wird wie die zweite Ebene. Verallgemeinert lässt sich sagen, dass die n-te Ebene doppelt so oft angezeigt wird, wie die $n+1$-te.

Im Codegerüst ist Ihnen das `struct Color` vorgegeben, das drei `uint8_t r`, `g` und `b` enthält und als Datentyp zur Farbdarstellung genutzt werden soll. Mit diesem Datentyp können in gewohnter Weise RGB-Farben dargestellt werden.
Beispielsweise wird Violett (50% Rot, 0% Grün, 100% Blau) als
`(Color){ .r = 0x80, .g = 0x00, .b = 0xFF }` dargestellt.
Die numerische RGB-Darstellung einer Farbe ergibt sich direkt aus den in den Ebenen gesetzten Bits. Ist das Bit von R, G oder B beispielsweise in der höchsten Ebene gesetzt, leuchtet diese LED mit mindestens halber Intensität. Das heißt, dass das MSB in der RGB-Farbdarstellung auch gesetzt ist.
Verallgemeinert lässt sich sagen, dass aus einem gesetzten Bit von R, G oder B in der n-ten Ebene folgt, dass Bit $8-n$ in der numerischen RGB-Farbdarstellung gesetzt ist. Die Anzahl an Ebenen bestimmt also, mit wie vielen Bits eine Farbe kodiert werden kann, d.h. die *Farbtiefe*.
Experimentieren Sie in Ihrer Implementierung mit der Anzahl an Ebenen. Es müssen mindestens zwei Ebenen benutzt werden, um die Testtasks zu bestehen.

Hinweis

Angesichts des großen Speicherbedarfs des Framebuffers ist es Ihnen gestattet, den Heapoffset zu erhöhen oder die Heapstruktur vollständig zu verwerfen.

Hinweis

Die menschliche Helligkeitswahrnehmung ist nicht linear und hängt stark von der Lichtfarbe ab. Es wird eine Lichtquelle deswegen nicht als exakt halb so hell wahrgenommen, wenn Sie mit halber Intensität leuchtet bzw. nur halb so oft leuchtet, wie eine andere. In diesem Versuch reicht es, dass zu sehen ist, dass mit Ihrer Implementierung auch dunklere Farben darstellbar sind, die nicht zu den 8 nativen Farben gehören.

ISR Die folgend beschriebene Funktionalität ist in der Datei `led_paneldriver.c` zu implementieren. Implemenentieren Sie zunächst die Funktion `panel_init()`, die die zur

Beschaltung der Matrix erforderlichen Pins konfiguriert. Füllen Sie danach die im Codegerüst vorhandene ISR zur Aktualisierung der Matrix aus. Um die ISR übersichtlich zu halten, lagern Sie Teilabschnitte in die folgenden Unterfunktionen aus:

- `panel_latchEnable/Disable`
- `panel_outputEnable/Disable`
- `panel_setAddress`
- `panel_setOutput`
- `panel_CLK`

Überlegen Sie sich, welche Parameter und Rückgabewerte Sie für die Funktionen vorsehen und wählen Sie entsprechende Datentypen. Die von uns bereits vorgegebene Funktion `panel_initTimer()` konfiguriert den für die ISR relevanten Timer, sodass die ISR mit einer hohen Frequenz aufgerufen wird. Rufen Sie diese an einer geeigneten Stelle im Betriebssystem auf.
In der ISR soll der in Abschnitt 6.3.2 beschriebene Vorgang mit den Farben der aktuellen Ebene für eine Doppelzeile durchgeführt werden. Beachten Sie die invertierte Semantik von $\overline{\texttt{OE}}$ und dass beim Setzen der Clock keine Wartezeit notwendig ist. Überlegen Sie sich, wie man es bewerkstelligen kann, im nächsten Aufruf der ISR die nächste Doppelzeile zu beschalten. Beachten Sie, dass nach 16 Durchläufen der ISR, d.h. der Aktualisierung aller Doppelzeilen, entschieden werden muss, in welche Ebene gewechselt werden muss.

Hinweis

Innerhalb von atomaren Sektionen wird die ISR nicht aufgerufen, weshalb die Aktualisierung der Matrix in solchen Sektionen nicht stattfindet. Eine unregelmäßige Aktualisierung der Matrix führt zu Flackern, was vermieden werden soll. Achten sie daher darauf, dass atomare Sektionen möglichst selten und so kurz wie möglich sind. Die folgenden Funktionen in SPOS stellen atomare Sektionen dar:

- Die gesamte Scheduler-ISR. Hier kann die Wahl einer effizienten Schedulingstrategie helfen, die atomare Sektion kurz zu halten.
- Alle LCD-Ausgabefunktionen. Verwenden sie das LCD in diesem Versuch insgesamt sparsam und nutzen, wenn möglich, die LED-Matrix zur Textausgabe. Wenn das LCD zur Anzeige von variablen Daten (z.B. einer Punktzahl) verwendet wird, sollte das LCD nur bei einer Änderung aktualisiert werden.
- Die Funktionen `enterCriticalSection` und `leaveCriticalSection`. Die atomare Sektion ist hier sehr kurz, kann im Extremfall aber auch zu Problemen führen (wenn z.B. in einer Schleife sehr oft hintereinander eine kurze Funktion aufgerufen wird, die als kritische Sektion implementiert ist)

Flackerreduktion In folgendem Beispiel gibt es drei Ebenen (`0`, `1`, `2` in absteigender Gewichtung) und man möchte die Farbe Rot mit halber Intensität anzeigen, d.h. $r =$ `0b10000000` und es werden nur die drei höchsten Bits berücksichtigt. Die höchste Ebene wird für vier vollständige Aktualisierungsvorgänge angezeigt, die mittlere für zwei, und die niedrigste für einen. Es gibt also einen Zyklus über sieben Bildaktualisierungen. Naiv könnte man diese Anzahlen hintereinander auf den Zyklus verteilen, wie in Tabelle 6.4 dargestellt. Dies wird jedoch ein sichtbares Flackern verursachen, da die roten LEDs zuerst einige Zeit eingeschaltet, und dann wieder für einige Zeit ausgeschaltet ist.

Nr. der Bildaktualisierung	1.	2.	3.	4.	5.	6.	7.
Zur Aktualisierung genutzte Ebene	0	0	0	0	1	1	2
Rote LED des Pixels	1	1	1	1	0	0	0

Tabelle 6.4: Zustand der roten LED im Beispiel in den ersten sieben Aktualisierungen des vollständigen auf der LED-Matrix angezeigten Bildes

In Tabelle 6.5 ist eine bessere Reihenfolge dargestellt: Ebene `0` wird nicht mehr vier mal hintereinander dargestellt. Beachten Sie, dass die dargestellte Farbe immer noch die selbe ist, da über den Zyklus betrachtet die Zeitspannen, die die einzelnen Ebenen angezeigt werden, gleich geblieben sind. Überlegen Sie sich für ihre Ebenenanzahl eine

gute Reihenfolge, in der die anzuzeigenden Ebenen ausgewählt werden.

Nr. der Bildaktualisierung	1.	2.	3.	4.	5.	6.	7.
Zur Aktualisierung genutzte Ebene	0	0	1	1	0	0	2
Rote LED des Pixels	1	1	0	0	1	1	0

Tabelle 6.5: Zustand der roten LED im Beispiel bei verbesserter Reihenfolge der selektierten Ebenen

6.4.3 Draw-Methoden

Durch den Display-Treiber wird die Möglichkeit gegeben, die im Speicher hinterlegten Ebenen auf der Matrix zu visualisieren. Um das Display komfortabel benutzen zu können, sind nun diverse Hilfsfunktionen notwendig, um die Ebenen selbst zu beschreiben. Dies beinhaltet das Setzen von einzelnen Pixeln sowie komplexere Zeichenoperationen. Implementieren Sie die folgenden *Draw-Funktionen*.
Die Methoden `draw_letter`, `draw_decimal` und `draw_number` sind bereits vorgegeben und können in der weiteren Implementierung von Ihnen genutzt werden. Das Verhalten dieser Funktionen kann aus der Dokumentation im Source-Code abgeleitet werden.

draw_setPixel Die Funktion `void draw_setPixel(uint8_t x, uint8_t y, Color color)` setzt das Pixel in Spalte x und Zeile y auf die übergebene Farbe. Hierbei ist zu beachten, dass der Punkt $(0, 0)$ das Pixel in der oberen linken Ecke der LED-Matrix bezeichnet, der Punkt $(31, 0)$ das Pixel in der oberen rechten Ecke. Teilen Sie die Bits der übergebenen Farbe wie in Abschnitt 6.4.2 beschrieben auf die Ebenen auf.

draw_getPixel Die Funktion `Color draw_getPixel(uint8_t x, uint8_t y)` liest den Framebuffer aus und rekonstruiert die Farbe des an der Stelle der übergebenen Koordinaten stehenden Pixels.

draw_filledRectangle Diese Funktion zeichnet ein Rechteck, das durch die Koordinaten des oberen linken Eckpunktes (x_1, y_1) und des unteren rechten Eckpunktes (x_2, y_2) sowie der Füllfarbe definiert wird.

draw_fillPanel Diese Funktion füllt die ganze LED-Matrix mit der übergebenen Farbe.

draw_clearDisplay Diese Funktion löscht das Bild der LED-Matrix, d.h. es leuchtet keine LED mehr.

6.4.4 Joysticktreiber

Die Funktionalität des Joystick-Treibers soll in den Dateien `joystick.c/.h` implementiert werden. Die Dateien sind von Ihnen anzulegen. Implementieren Sie folgende Datenstrukturen und Funktionen.

enum Direction Legen Sie in `joystick.h` ein `enum Direction` mit passendem `typedef` an, durch das fünf mögliche Werte zur Richtungseingabe durch den Joystick repräsentiert werden können. Halten Sie sich dabei an die Bezeichner `JS_LEFT, JS_RIGHT, JS_UP, JS_DOWN, JS_NEUTRAL`.

void js_init() Diese Funktion konfiguriert die Input-Pins sowie den internen AD-Wandler mithilfe der Register `ADMUX` und `ADCSRA`. Es soll eine interne Referenzspannung von V_{cc} benutzt werden.

uint16_t js_getHorizontal() und js_getVertical() Diese Funktionen sollen mithilfe des internen AD-Wandlers den Analogpegel je eines Potentiometers auslesen und den gemessenen Wert zurückgeben.

Direction js_getDirection() Diese Funktion soll mithilfe von `js_getHorizontal` und `js_getVertical` die Richtung des Joysticks ableiten können. Für die Neutralposition soll ein Toleranzbereich von etwa 1,0 V um die Ruheposition verwendet werden.

bool js_getButton() Diese Funktion gibt `true` zurück, wenn gerade auf den Joystick gedrückt wird, d.h. der in ihm verbaute Schalter geschlossen ist. Andernfalls gibt diese Funktion `false` zurück.

6.4.5 Anwendungsprogramm: Snake

Ihre Implementierung sollten sich an den gängigen Spielregeln orientieren. Der Spieler steuert per Joystickeingabe eine Schlange über das Spielfeld. An zufälligen Positionen auf dem Spielfeld erscheint Nahrung von der Größe eines Pixels. Ziel des Spiels ist es, möglichst viel Nahrung zu fressen. Mit jeder aufgenommenen Nahrungseinheit wächst die Länge der Schlange um ein Pixel. Wenn die Schlange in sich selbst oder eine Wand navigiert wird, ist das Spiel verloren. Bei verlorener Partie soll ohne Neustart des Mikrocontrollers erneut gespielt werden können. Zusätzlich zum eigentlichen Spiel soll der Punktestand – die aktuelle Menge gefressener Nahrung – angezeigt werden. Auch der Highscore aus vorherigen Partien soll zu jeder Zeit sichtbar sein.

Spielmenü Durch Drücken des Joystick-Buttons muss das laufende Spiel unterbrochen werden können. Das angezeigte Bild soll gelöscht und ein Pausenmenü geöffnet werden. Das Menü muss wieder geschlossen werden können – bspw. durch erneutes Drücken auf den Joystick – und das Spiel im Zustand vor der Pausierung fortgeführt werden. Das Menü braucht keine Funktionalität, der Kreativität sind aber keine Grenzen gesetzt.

6.4.6 Tipps zur Snakeimplementierung

Kodierung der Schlange Um Richtung und Position eines Teils des Schlangenkörpers (im Folgenden ein *Snakebit* genannt) effizient abzuspeichern, kann eine der Heapmap ähnliche Kodierung verwendet werden. Es genügt, die Richtung, in die sich ein Snakebit

bei der nächsten Bewegung der Snake bewegen wird, zu speichern und diese Informationen aufeinanderfolgend in einem Array abzuspeichern. Dabei gehört die erste gespeicherte Richtung zu dem Snakebit, das der Nachfolger des Schlangenkopfes ist. Die zweite Richtung dann zum zweiten Snakebit usw.
Wenn die absolute Position des Schlangenkopfes bekannt ist, ergibt sich dadurch eine eindeutige Darstellung der Schlange, da sich jedes Snakebit, bis auf den Kopf, pro Schritt immer auf die Position seines Vorgänger-Snakebits bewegt, und dessen vorherige Richtung übernimmt.
Da es 4 Bewegungsrichtungen gibt, reichen 2 Bit zum Speichern der Richtungsinformation.
Nach dem beschriebenen Vorgehen lässt sich eine Schlange maximaler Länge also in $2S$ Bit kodieren, wobei S die Anzahl der Felder der Spielfläche ist. Maximal kann S also 1024 annehmen, sodass ein Array der Größe 2048 Bit = 256 Byte immer ausreichend ist.

Lernerfolgsfragen

- Welche alternative Datenstruktur bietet sich zum Abspeichern der Schlange an?
- Welchen Overhead erwarten Sie, falls Sie sich bei der Implementierung für Ihre eigene Alternative entscheiden?

Aktualisierung des Spielzustands Bei jedem Schritt der Schlange muss der Spielzustand aktualisiert werden. Hierzu ist es nötig, erst den Joystick auszulesen, und auf Basis des Rückgabewertes zu entscheiden, in welche Richtung der Schlangenkopf bewegt werden soll.
Führen Sie diese Bewegung hypothetisch durch, und berechnen Sie, welche Folgen dies für den Spielzustand hat. Mögliche Folgen sind eine Kollision mit Spielfeldrand, der Schlange selbst oder das Aufsammeln von Nahrung, mit oder ohne Sieg als Folge.

Bewegung der Snake Wenn die Snake sich ein Pixel weiterbewegt, ändern sich optisch nur die Kopfposition und die Position des letzten Snakebits. Aus diesem Grund werden die Snakebits in einem Ringbuffer gespeichert. Hierbei zeigt die Variable `head` auf das erste Element im Buffer und die Variable `tail` auf die erste freie Stelle nach dem letzten Element. Beachten Sie, dass die Elemente des Ringbuffers in unserem Fall mit zwei Bit kodierte Richtungen der Schlange sind. Aus diesem Grund muss über Bitoperationen auf die Einträge zugegriffen werden.
In dem Ringbuffer zeigt `head` auf die Richtung, die zum ersten Snakebit gehört, der nächste Eintrag beinhaltet die Richtung des zweiten Snakebits und so weiter. Eine Bewegung aller Snakebits um eins entspricht nun nur noch dem gleichzeitigen Verschieben

von `head` und `tail` um eine Position. Die entstehende Lücke an der Position von `head` im Buffer muss mit der Richtung beschrieben werden, in der vom alten Schlangenkopf aus nach Interpretation des Joysticks die neue Kopfposition erreicht wird. Anschließend muss die Position des Kopfes aktualisiert werden. Der Kopf ist nicht Teil des Ringbuffers, da er richtungslos ist, bis im nächsten Schritt der Spiellogik eine Richtung ermittelt wird.
Wenn mit der Bewegung Nahrung gefunden wurde, wird das letzte Element der Schlange im Buffer nicht gelöscht, sondern unverändert beibehalten. Effektiv verlängert sich die Schlange um ein Snakebit.

Verwaltung des Spielzustands Die Verwaltung des Spielgesamtzustandes kann beispielsweise durch Abspeichern folgender Informationen realisiert werden:

- Position Schlangenkopf
- Highscore
- Momentane Punktzahl
- Richtung und Position der Teile des Schlangenkörpers
- Position Nahrung

Gegebenenfalls lohnt es sich, weitere Informationen, wie die vorherige Position des Schlangenkopfes oder des letzten Snakebits zwischenzuspeichern, um den Spielzustand einfacher und schneller aktualisieren zu können.

Grafische Ausgabe des Spielzustands Um die Änderungen des Spielzustandes optisch umzusetzen, müssen die entsprechenden Pixel im Framebuffer gesetzt oder gelöscht werden. Um Punktestände darzustellen, können die gegebenen Draw-Methoden verwendet werden.

Vergrößerung des Prozessstacks Sollten durch Ihre Snakeimplementierung Stackinkonsistenzen auftreten, könnte es sein, dass Sie durch eine große Anzahl lokaler Variablen oder Funkionsaufrufen den Stackspeicher überschreiten. In diesem Fall ist es Ihnen gestattet, das Define `MAX_NUMBER_OF_PROCESSES` zu verringern. Dadurch hat jeder einzelne Prozess einen größeren Stack, da immer genau die Hälfte des internen SRAMs für Prozessstacks verwendet wird.

6.5 Zusammenfassung

Folgende Übersicht listet alle Typen, Funktionen und Aufgaben auf. Alle aufgelisteten Punkte müssen zur Teilnahme am Versuch bis zur Abgabefrist bearbeitet und hochgeladen werden. Diese Übersicht kann als Checkliste verwendet werden und ist daher mit Checkboxen versehen.

- ☐ Aufruf von **`void panel_initTimer()`** an einer geeigneten Stelle in SPOS
- ☐ `led_paneldriver.c/.h`
  - ☐ Globale Variable(n) frei wählbaren Arraytyps als `framebuffer`
  - ☐ Hilfsfunktionen zum Ansteuern der Eingangspins der LED-Matrix
    - ☐ `panel_init`
    - ☐ `panel_latchEnable/Disable`
    - ☐ `panel_outputEnable/Disable`
    - ☐ `panel_setAddress`
    - ☐ `panel_setOutput`
    - ☐ `panel_CLK`
  - ☐ Ausfüllen der ISR zur Erneuerung einer Doppelzeile
- ☐ `led_draw.c/.h`
  - ☐ `void draw_setPixel(uint8_t x, uint8_t y, Color color)`
  - ☐ `Color draw_getPixel(uint8_t x, uint8_t y)`
  - ☐ `void draw_filledRectangle(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, Color color)`
  - ☐ `void draw_fillPanel(Color color)`
  - ☐ `void draw_clearDisplay()`
- ☐ `joystick.c/.h`
  - ☐ `enum Direction` mit `typedef`
  - ☐ `void js_init()`
  - ☐ `uint16_t js_getHorizontal()`
  - ☐ `uint16_t js_getVertical()`
  - ☐ `Direction js_getDirection()`
  - ☐ `bool js_getButton()`
- ☐ `led_snake.c/.h`
  - ☐ Implementierung des Spiels Snake

- ☐ Das Spiel Snake
- ☐ Menü, bei dessen Aufruf das Spiel unterbrochen und ein Menübildschirm angezeigt wird. Der Spielstand muss anschließend wiederhergestellt werden können.

6.6 Testtasks

Eine genauere Beschreibung der Testtasks können Sie der im Moodle vorhandenen Datei `Testtasksbeschreibung.pdf` entnehmen. Beachten Sie, dass auch ihre Snakeimplementierung im Praktikumsversuch getestet wird.

Color Rendering Dieser Testtask stellt verschiedene Farbmuster dar, anhand derer überprüft werden kann, dass alle Farben korrekt dargestellt werden können. In der Testtaskbeschreibung sind die Muster genauer beschrieben und es werden Vergleichsbilder gegeben, um die eigene Implementierung zu prüfen. Dies kann als erster Test zur Funktion der ISR und der Funktion `draw_setPixel` verwendet werden. Beachten Sie, dass Sie mindestens 2 Ebenen implementieren müssen, um diesen Testtask zu bestehen. Das heißt, es müssen auch Farben mit abgeschwächter Intensität sichtbar sein.

Color Retrieval Dieser Testtask gibt ein Farbmuster auf der Matrix aus. Anschließend werden die Farbwerte mittels `draw_getPixel` pixelweise eingelesen und mit dem dargestellten Wert verglichen. Wenn Abweichungen auftreten, wird eine Fehlermeldung ausgegeben.

Joystick Dieser Testtask prüft, ob die beiden Achsen des Joysticks korrekt ausgelesen werden, und stellt Zahlenrepräsentationen der gelesenen Werte auf dem Panel dar. Außerdem wird mithilfe der Funktion `Direction js_getDirection()` ausgelesen, als welche Richtung die aktuelle Joystickposition interpretiert wird. Dies wird in Form eines Pfeils in der unteren rechten Ecke angezeigt.

6.7 Pinbelegungen

Port	Pin	Belegung
Port A	A0	Rowselect 0
	A1	Rowselect 1
	A2	Rowselect 2
	A3	Rowselect 3
	A4	frei
	A5	Joystick horizontal
	A6	Joystick vertikal
	A7	Joystick Button
Port B	B0	LCD Pin 1 (D4)
	B1	LCD Pin 2 (D5)
	B2	LCD Pin 3 (D6)
	B3	LCD Pin 4 (D7)
	B4	LCD Pin 5 (RS)
	B5	LCD Pin 6 (EN)
	B6	LCD Pin 7 (RW)
	B7	frei
Port C	C0	CLK
	C1	LE
	C2	Reserviert für JTAG
	C3	Reserviert für JTAG
	C4	Reserviert für JTAG
	C5	Reserviert für JTAG
	C6	$\overline{\text{OE}}$
	C7	Button
Port D	D0	R1
	D1	G1
	D2	B1
	D3	R2
	D4	G2
	D5	B2
	D6	frei
	D7	frei

Pinbelegung für Versuch 6 (*LED-Matrix*).