Silizium und Kupfer: Einführung in Programmierung von Mikrocontrollern

1 [120min] Platine löten

Platinen selbst sind häufig lediglich stabile Platten, auf denen Kupferbahnen gezogen worden sind. Sie können von nur einer Ebene auf der Vorderseite bis zu 14 Ebenen mit dazwischenliegenden Schichten haben. Verbunden werden die Schichten dann mit einem isolierendem Material unter höchst sauberen Bedingungen. Um einzelne Leiterbahnen auf verschiedenen Ebenen elektrisch zu verbinden, kann man Bohrungsränder mit Kupfer füllen, diese heißen dann Vias. Um die Leiterbahnen vor Korrosion zu schützen, wird häufig ein nichtleitender Lötstopplack auf alle nicht zu verlötenden Flächen aufgetragen. Seine namensgebende Funktion ist es den Fluss des Lots auf dem Lack zu verhindern.

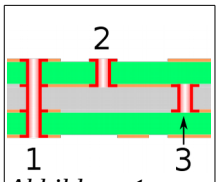


Abbildung 1: Vias bei 4 Ebenen

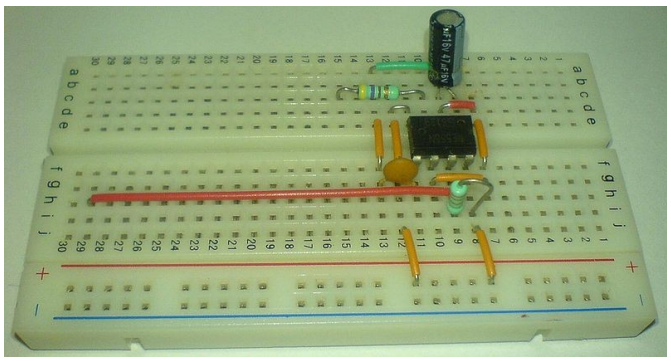


Abbildung 2: Breadboard zum Stecken

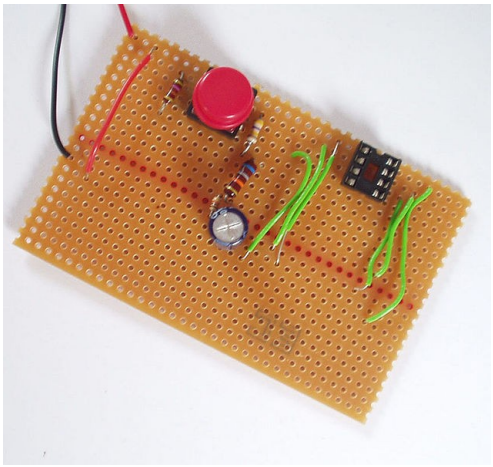


Abbildung 3: Lochrasterplatine zum Löten

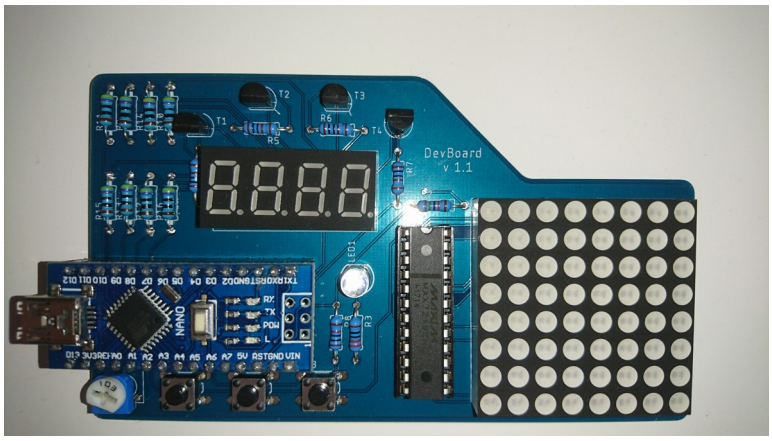
Zuerst müssen die Bauteile auf die Platine gelötet werden. Dazu wird die Platine mit einem Bauteil auf der Vorderseite (bei der Bedruckung) bestückt und auf der Rückseite auf den Pads fest gelötet. Anschließend wiederholst du das nun mit jedem Bauteil bis du fertig bist. Bei manchen Bauteilen (LEDs, Transistoren etc) ist auf die richtige Polung zu achten! Zu dem Löten selbst findest du viele ausführliche oder schnelle Anleitungen im Internet. Beachte die richtige Reihenfolge der Pins bei den Displays. Der jeweils erste Pin ist mit einem Kreis auf der Platine markiert.

Hier noch ein paar Tipps:

- Bevor du anfängst, prüfe die Vollständigkeit des Bausatzes mit Hilfe des Datenblatts und sortiere die Widerstände nach Wert.
- Fange mit den flachsten Bauelementen an und arbeite dich mit steigender Dicke voran. So kannst du das Board ohne Tischklemme löten.
- Befestige die Elemente auf der Oberseite mit Tesa-Film, während du sie festlötest, damit sie möglichst nah an der Platine befestigt werden.
- Bevor du die Stiftleisten an den Arduino lötest, löte die weiblichen Buchsenleisten auf der Platine fest und stecke die Stiftleisten in die Buchsenleisten. So verhinderst du, dass du die Stiftleisten schräg festlötest.

1) Auf welche zwei gängigen Verfahren kann eine Platine gefertigt werden und was sind die jeweiligen Vor- und Nachteile?
2) Versuche dich zuerst an einer kleinen Platine und übe dort das Löten.
3) Löte nun die Platine für das Dev-Board und teste es.

4) Schau dir den Schaltplan des Dev-Board an. Beschreibe die jeweilige Funktion aller Widerstände außer R1 und R2. (Schlüsselwörter zum Suchen: Transistor, Pull Up/Down)



Wenn du alles richtig gemacht hast, siehst du nach dem Anschließen der Platine über USB an den Computer, dass auf der 7 Segment-Anzeige die Segmente der Reihenfolge nach auf allen vier Stellen gleichzeitig angezeigt werden, auf der LED-Matrix ein X dargestellt wird und die Helligkeit der LED abhängig von der Trimmerstellung ist. Die Taster S1 bis S3 schalten im gedrückten Zustand die LED aus. Über den seriellen Monitor empfängst du im 2Hz Takt die aktuelle Trimmerstellung im Intervall [0; 1023].

2 [120min] Einführung Programmierung

Ein Mikrocontroller besteht aus einem Mikroprozessor für die Befehlsausführung und der verbauten Peripherie wie Pulsbreitenmodulations-Ausgängen (PWM), Analog/Digital-Wandler (AD), sowie serielle- und USB-Schnittstellen. Der auf dem Arduino verbaute Mikrocontroller ist der Atmel328. Mit der genaueren Funktion und Verwendung der PWM- und AD-Schnittstellen werden wir uns erst in späteren Kapiteln beschäftigen. Der AD-Wandler verfügt über eine 10-Bit Auflösung, die PWM-Ausgänge nur über eine 8-Bit Auflösung.

Wenn man von einer Auflösung spricht (z.B. 10 Bit), ist damit gemeint, was der maximale zulässige Wert ist. Eine 10-Bit-Zahl (eine Binärzahl mit 10 Stellen) kann alle Zahlen zwischen 0 und 1023 darstellen. Also wenn ein 0..5V Analog/Digital-Wandler eine 10-Bit Auflösung hat, dann bedeutet im Quelltext eine Ausgabe von 1023, dass 5V anliegen. Bei dem Wert 512 würden 2,5V anliegen.

Der Atmel328 arbeitet noch immer nach den Grundprinzipien der Von-Neumann-Architektur aus dem Jahr 1945.

1) Was sind die vier Hauptkomponenten der Von-Neumann-Architektur?
2) Was ist der sogenannte Von-Neumann-Flaschenhals?
3) Wie viele verschiedene Werte hat eine 10-Bit Auflösung?
4) Stelle die Zahlen 127, -127, 313, -755, 1000 und -1000 als Binärzahl im 10-Bit Zweierkomplementär dar.

Um das Board programmieren zu können, benötigst du die Arduino-IDE (**I**ntegrated **D**evelopment **E**nvironment). Arduino ist eine Open-Source Plattform, die in verschiedensten Formen und Ausstattungen angeboten wird. Die IDE übersetzt den Quelltext (mit C++ als Programmiersprache mit einigen zusätzlichen Befehlen) in einen für den Prozessor interpretierbaren Bitcode und übermittelt ihn mittels eines speziellen Protokolls an den Controller, der ihn dann speichert.

Als Quellen zum Lernen der Programmierung eignen sich www.arduino-tutorial.de und die in der IDE mitgelieferten Beispiele. Eine Übersicht über alle Arduino-spezifischen Befehle findest du unter www.arduino.cc/reference/de .

Für den Anfang beschränken wir uns darauf, dass es für die Programmierung nur zwei Funktionen gibt: `setup()` und `loop()`. Während `setup()` nur einmalig ganz am Anfang eines Programms aufgerufen wird, wird `loop()` unendlich oft wiederholt ausgeführt.

```
/* i: Integer für die Pinnummer
 * m: Modus aus {INPUT, OUTPUT, INPUT_PULLUP}
 * s: Signal aus {HIGH, LOW}
 * ms: Zeit in Millisekunden, als Integer gespeichert*/

// Diese Funktionen sind nützlich:
pinMode(i, m); // Definiert den zu verwendenden Modus des Pins
digitalWrite(i, s); // Definiert das Signal an einem Ausgang
s = digitalRead(i); // Erkennt das Signal an einem Eingang
delay(ms); // Verzögert die weitere Ausführung des Programms
```

5) Schau in den Schaltplan. An welche auswertbaren Arduino-Pins (also Input/Output Pins) sind die Taster S1, S2, S3, sowie die LED angeschlossen?
6) Schreibe ein Programm, mit welchem die LED mit 2Hz blinkt. Der Pin für die LED soll in einer Variable gespeichert sein, die bei Funktionsaufrufen verwendet wird. (Siehe Beispiele)

Um Bedingungen zu prüfen, verwendet man `if/else` Konstruktionen. Sie prüfen, ob eine definierte Bedingung wahr (`true`) oder falsch (`false`) ist. Sie beginnen mit `if (b) {c}`, wobei `b` für die Bedingung und `c` für den Code steht, der ausgeführt wird, wenn die Bedingung wahr ist. Es eignet sich den Code innerhalb von Konstruktionen um eine Position mit der Tab-Taste einzurücken, da dies die Lesbarkeit des Quelltextes erheblich steigert.

Mehr zu diesem Thema findest auf www.arduino.cc/reference/en/#structure unter „`if...else`“, „`else`“, „`Comparison Operators`“ und „`Boolean Operators`“.

7) Welche booleschen (logischen) und vergleichenden Operatoren gibt es in C gemäß der Arduino Referenz?
8) Schreibe ein Programm, mit welchem die LED mit 2Hz blinkt. Wenn Taster S1 gedrückt gehalten wird, soll sie mit 5Hz blinken. (Beachte, dass die Taster S1 und S2 über `INPUT_PULLUP` initialisiert werden müssen)

Mit Pulsbreitenmodulation (**p**ulse **w**idth **m**odulation = pwm) kann man die umgesetzte Leistung in elektrischen Bauelementen, die sich sonst sehr unpraktikabel oder nur teuer „dimmen“ lassen würden, einstellen. Somit kann man etwa eine LED je nach PWM-Auflösung stufenlos dimmen, ohne, dass die Spannung an der oder der Strom durch die LED variiert werden muss. Zur Verwendung dieses Features hat unser Arduino einige 8-bit (0..255) PWM-kompatible Pins, die im Schaltplan mit einer Tilde („~“) gekennzeichnet werden.

Dazu muss der Pin als `OUTPUT` definiert worden sein und das Feature mit `analogWrite(i, v);` (mit `i` für Pinbezeichner und `v` für den Wert) abgefragt worden sein.

9) Erkundige dich zu Pulsbreitenmodulation. Zeichne ein Strom/Zeit Diagramm durch eine „PWM-gedimmte“ Glühbirne, die mit einem Drittel ihrer maximalen Leuchtkraft leuchtet.
10) Schreibe ein Programm, mit welchem die LED ganz dunkel anfängt und innerhalb von zwei Sekunden stufenlos maximale Helligkeit erreicht und anschließend wieder innerhalb von zwei Sekunden auf den ausgeschalteten Zustand herunter dimmt. Der Vorgang wiederholt sich von vorn unendlich oft.

3 [!!!min] Paralleles arbeiten

3.1 [!!!min] Taster

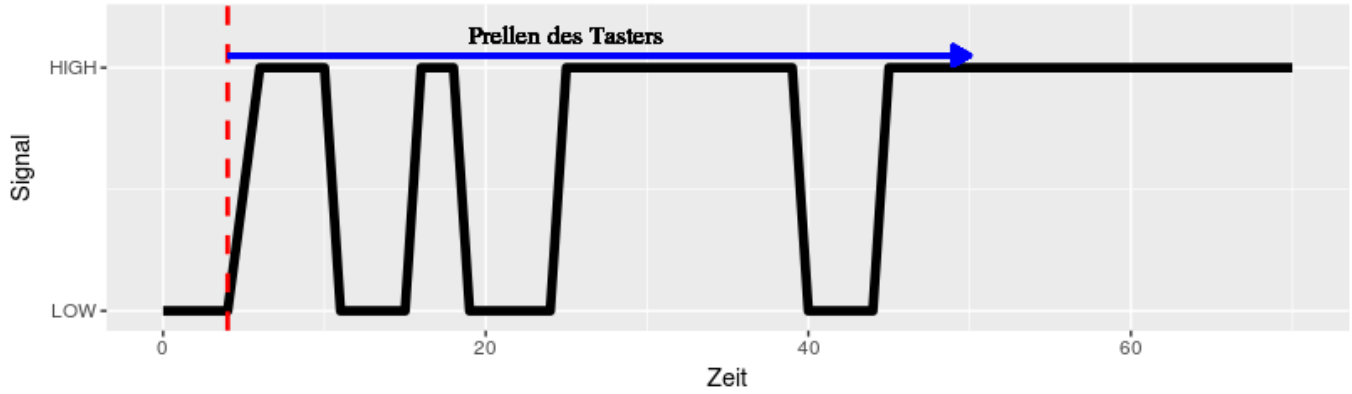


Abbildung 4: Prellen des Tasters

Wenn man versucht einen Taster mit Rastfunktion zu programmieren, so dass er lediglich auf eine „`rising edge`“ (steigende Flanke) wartet und dann einen Boolean im Speicher negiert, wird man schnell feststellen, dass es in der Praxis nicht zu dem gewünschten Resultat führt. Grund dafür ist das sogenannte Prellen (engl. bouncing) der gefederten Schaltkontakte, die noch kurz nach Betätigung ihren Zustand wechseln.

1) Schreibe ein Programm, bei dem die LED ihren Zustand wechselt, wenn der Taster kurz gedrückt wurde. (Die verwendeten Taster haben eine gemessene maximale Prellzeit von etwa 50ms, du benötigst für diese Aufgabe zwei Booleans als Variablen)

Wir haben uns bisher nur C und Arduino interne Funktionen angeguckt. Der Programmierer hat natürlich auch die Möglichkeit Funktionen zu erstellen, die er an späteren Stellen im Quelltext wieder aufrufen kann. Um mehr dazu zu erfahren, gehe auf www.arduino.cc/en/Reference/FunctionDeclaration . Solche Funktionen wirst du für Interrupts benötigen.

2) Was sind Interrupts und wie verwendet man sie bei Arduinos? Welche zwei verschiedenen (in die Hardware implementierte) Interrupts gibt es?
4) Welche Pins des verwendeten Arduinos Nanos unterstützen Interrupts? Welche/-s Bauelement/-e sind damit verbunden?
5) Überarbeite das Programm aus der ersten Teilaufgabe mit Interrupts.

3.2 [!!!min] Multitasking

Als nächstes soll ein Programm entwickelt werden, welches wieder eine LED mit einer Frequenz von 2Hz blinken lässt. Außerdem soll er mit 5Hz über die serielle Schnittstelle den Schaltzustand des Tasters S1 und S2 übermitteln. Auslesen lässt sich das in der Arduino IDE über den seriellen Monitor.

Um zwei oder mehr verschiedene Arbeiten mit unterschiedlichen Frequenzen abzuarbeiten muss man bei den meisten Mikrocontrollern mit Scheinparallelität arbeiten. Es gibt aber inzwischen auch einige, die echte Nebenläufigkeit mit Hilfe eines Schedulers unterstützen, was bei uns aber nicht der Fall ist. Bei uns muss die Hauptschleife ohne Verzögerung ausgeführt werden. Bedingungen abhängig von der Zeitdifferenz seit der letzten Ausführung des Prozesses steuern, welche Funktionen aufgerufen werden sollen.

```
// Deklariere Variablen hier, damit sie beim Neustart von loop() ihren
// Wert behaelt (und in allen Funktionen verfuegbar ist)
int pinLed = 5, pinS1 = 3, pinS2 = 2;
unsigned long lastBlink = 0;
unsigned long lastSerial = 0;

void setup () {
  Serial.begin(9600); // Verbindung hat BaudRate von 9600
  pinMode(pinLed, OUTPUT);
  pinMode(pinS1, INPUT_PULLUP);
  pinMode(pinS2, INPUT_PULLUP);
}

void loop () {
  if (millis() - lastBlink >= 250) {
   // Umschalten mit einem Ternary Operator
   digitalWrite(pinLed, digitalRead(pinLed) == 1 ? 0 : 1);
   lastBlink = millis();
  }
  if (millis() - lastSerial >= 200) {
   Serial.print("S1: " + (digitalRead(pinS1)==1 ? "Open" : "Closed"));
   Serial.println(" S2: " + (digitalRead(pinS2)==2 ? "Open" : "Closed"));
   lastSerial = millis();
  }
}
```

1) Was ist eine Baudrate?
2) Wie viele Bits hat eine Variable vom Typ `unsigned long` zur Verfügung? Was ist damit größte darstellbare Zahl?
3) Warum ist es kein Problem, wenn die Variablen zur Speicherung der ms seit Programmbeginn nach dem Erreichen des Maximalwerts überlaufen?
4) Erkundige dich zu Arrays für Arduino. Wie erstellt man eins? Wie speichert man einzelne Werte in einem? Wie ruft man diese Werte ab?

Nun ist der Schritt zu deinem eigenen Scheduler gar nicht mehr so weit. Alles was du brauchst sind drei gleich lange Arrays. Das eine Array speichert die aufzurufende Funktionen für die Prozesse. Das zweite Array speichert die Ausführungsintervalle. Und das letzte Array speichert den Wert von millis(); ,seitdem die Funktion zuletzt aufgerufen wurde.

```
// Ein Array von Funktionen (genauer Pointer auf Funktionen) wird so deklariert:
void (*arrayName[])() = {funktionsName1, funktionsName2};
```

Nun kannst du mit Hilfe einer `for`-Schleife (siehe Beispiel) prüfen, ob das Programm aufgerufen werden soll und wenn ja direkt aufrufen. Dies erfordert, dass jeder Prozess seine eigene Funktion hat.

```
int someNumbers[] = {2, 13, 11, 9, 112}
/* Die Division ist notwendig, weil sizeof() die Anzahl an Bytes in einer
 * Variablen, Konstanten oder eines Datentyps zurück gibt. Die Laenge eines
 * Integers ist abhaengig des verwendeten Mikrocontrollers. Bei ATmega Chips
 * ist ein Integer 2 Byte gross.*/
for (int i=0; i < sizeof(someNumbers)/sizeof(int); i++) {
  Serial.println(someNumbers[i]);
}
```

5) Setze deinen eigenen Scheduler um und teste ihn mit dem oben beschriebenen Programmablauf (LED 2Hz und serielle Kommunikation 5Hz)

3.3 [!!!min] Multiplexing

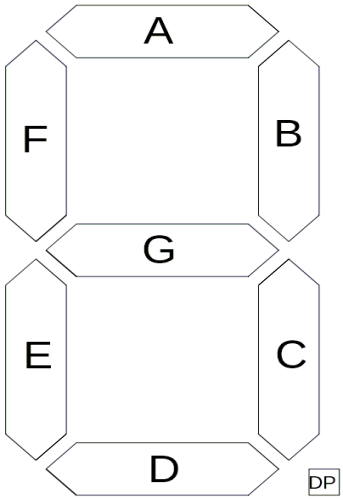


Abbildung 5: Bezeichnung der Segmente eines Elements

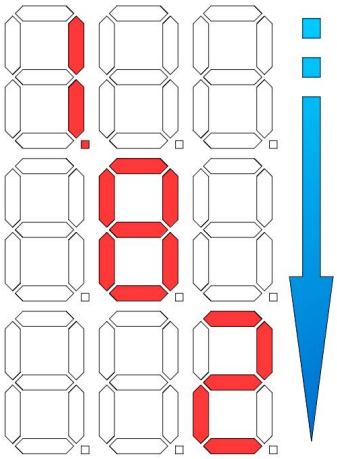


Abbildung 6: Bei hoher Frequenz erkennt ein Mensch die Zahl 1,82

Eine 4-stellige 7-Segment-Anzeige besteht aus 32 einzelnen LEDs, die alle separat angesteuert werden können. Ein kurzer Blick in den Schaltplan verrät aber schon, dass nicht jede LED seine eigene Leitung zum Mikrocontroller hat, sondern lediglich acht Leitungen für die sieben Segmente und den Dezimalpunkt, sowie vier Leitungen für die einzelnen Stellen eine Anbindung an den Controller haben.

Um nun eine Zahl darzustellen muss mit hoher Wiederholungsfrequenz jedes Element kurz seine designierte Ziffer darstellen. Durch die Trägheit der menschlichen Bildwahrnehmung wird die vollständige Zahl ohne erkennbare Flackereffekte, optimalerweise sogar mit nahezu maximaler Helligkeit der LEDs, wahrgenommen.

1) Schreibe eine Funktion, welche als Argumente
   1. das ausgebende Element (1 bis 4),
   2. die darzustellende Ziffer (0 bis 9) und
   3. einen Wahrheitswert (=Boolean) für den Dezimalpunkt

   entgegen nimmt und die Ziffer auf dem Display darstellt.
2) Schreibe ein Programm, mit dem du nun das Multiplexing umsetzt und beliebige 4 stellige Zahlen darstellen kann. Verwende dazu die in der vorherigen Teilaufgabe umgesetzte Funktion.

4 [!!!min] Binärdaten

4.1 [!!!min] Bitweise Operationen

Wir haben uns bereits einmal kurz mit Binärzahlen beschäftigt, doch die Relevanz dieses Themas für die Informatik ist so groß, dass wir uns dieses Thema noch einmal etwas genauer anschauen Kenntnisse in diesem Thema ermöglichen für manche Probleme einfache und sehr elegante Lösungen. Berechnungen mit Binärzahlen lassen sich mit Binärzahlen genau so durchführen, wie sie auch im Dezimalsystem durchgeführt werden würden.

```
  10011100 = 156     |       10011100 = 156    |   10011100 x 00010010 = 156*18
+ 00010010 = 18      |     - 00010010 = 18     |             10011100
____1_____           |       _______1__        |   +      10011100___
  10101110 = 174     |       10001010 = 138    |          10101111100 = 2808
```

Während Zahlen in C immer als Binärzahlen gespeichert und abgerufen werden, ist das für den Programmierer meist nicht relevant, da die Programmiersprache zwischen verschiedenen Basen bei der Eingabe unterscheidet:

- **Dezimalzahl** (Basis 10): Das ist der Standard für eingegebene Zahlen in C.
  Beispiel: `int dezimalzahl = 30;`
- **Binärzahl** (Basis 2): Um Zahlen als Binärzahl einzutragen, muss man ein „`B`“ vorne anfügen (maximal 8 Bit).
  Beispiel: `int binaerzahl = B11110;`
- **Oktalzahl** (Basis 8): Weniger Relevant. Zum Speichern muss eine „`0`“ vorne angefügt werden.
  Man kann eine Oktalzahl aus einer Binärzahl ermitteln, indem man von rechts beginnend jeweils 3 Stellen zusammenfasst und die Wertigkeit pro Block ermittelt. Ist der vorderste Block nicht drei Stellen lang, wird er einfach vorne mit Nullen aufgefüllt. Jeder Block repräsentiert immer eine Stelle in der Oktalzahl.
  Beispiel: 011 110 → `int oktalzahl = 036;`
- **Hexadezimalzahl** (Basis 16): Mit Hexadezimaldarstellungen kann man 8-Bit Binärzahlen mit nur zwei Stellen darstellen. Eine Stelle einer Hexadezimalzahl kann die Dezimalzahlen 0 bis 15 darstellen. Damit der Bereich 10-15 nicht zwei Stellen in Anspruch nimmt, werden diese mit den Großbuchstaben A-F ersetzt.
  Das Verfahren zur Umwandlung ist gleich mit dem der Oktalzahl, außer, dass von rechts beginnend die Binärzahl in vierer-Blöcke unterteilt wird. Zum Speichern muss ein „`0x`“ vorne angefügt werden.
  Beispiel: 0001 1110 → `int hexadezimalzahl = 0x1E;`

1) Ermittle die Binär- und Hexadezimalzahlen von 56, 61, 22 und 12.
2) Berechne im Binären: 56 + 61, 61 – 22 und 22 * 12 und gebe sie anschließend als Hexadezimalzahlen an.
3) Was sind `0xF5`, `0x5A` und `0x13` als Dezimalzahlen?

Mit C kann man Binärzahlen (dabei ist es egal, mit welcher Basis sie angegeben wurde) auch bitweise verknüpfen oder modifizieren. Diese Funktionen heißen „Bitwise Operations“:

- **Bitweises NICHT** (~): Negiert jede Stelle einer Binärzahl
  Beispiel: `~0x5F` ergibt `B10100000` bei einer 8-Bit Zahl
- **Bitverschiebung Links** (<<): Verschiebt die Stellen einer Zahl um eine angegebene Anzahl an Stellen nach links. Die zu modifizierende Zahl wird links vom Operator, die Anzahl der Stellen rechts vom Operator angegeben. Stellen, die links über die vom Datentyp angegebene Maximallänge hinausgehen würden, verfallen.
  Beispiel: `0x0F << 4` ergibt `0xF0`
- **Bitverschiebung Rechts** (>>): Verschiebt die Stellen einer Zahl um eine angegebene Anzahl an Stellen nach rechts. Die zu modifizierende Zahl wird links vom Operator, die Anzahl der Stellen rechts vom Operator angegeben. Stellen, die rechts über die vom Datentyp angegebene Maximallänge hinausgehen würden, verfallen.
  Beispiel: `B01100110 >> 3` ergibt `B00001100`
- **Bitweises UND** (&): Verknüpft zwei Binärzahlen miteinander, indem die jeweils gleichen Stellen mit dem UND-Operator verbunden werden.
  Beispiel: `B0011 & B0101` (oder anders herum) ergibt `B0001`
- **Bitweises ODER** (|): Verknüpft zwei Binärzahlen miteinander, indem die jeweils gleichen Stellen mit dem ODER-Operator verbunden werden.
  Beispiel: `B0011 | B0101` (oder anders herum) ergibt `B0111`

- **Bitweises Exklusives Oder** (^): Verknüpft zwei Binärzahlen miteinander, indem die jeweils gleichen Stellen mit dem XOR-Operator verbunden werden.
  Beispiel: `B0011 ^ B0101` (oder anders herum) ergibt `B0110`

Verwende für diese Aufgaben die bitweisen Operatoren!

4) Schreibe eine Funktion, die prüft, ob eine Komplementärzahl vom Datentyp Integer negativ oder positiv ist.
5) Schreibe eine Funktion, die eine ungerade Zahl auf die nächsttiefere gerade Zahl abrundet (für alle positiven Zahlen). Gerade Zahlen sollen erhalten bleiben.
6) Schreibe eine Funktion, dass die n-te Stelle einer Binärzahl auf 1 setzt.
7) Schreibe eine Funktion, dass die n-te Stelle einer Binärzahl auf 0 setzt.

4.2 *[!!!min]* Ansteuern integrierter Schaltkreise

Das im Kapitel 3.3 betrachtete Multiplexing von Ausgängen kann auch auf Eingänge angewendet werden, um zum Beispiel bei vielen Tastern weitere wertvolle Pins am Mikrocontroller zu sparen. Doch diese direkte Form des Multiplexings hat immer noch entscheidende Nachteile. Es werden zu viele „überqualifizierte“ Ein-/Ausgänge für sehr einfache Aufgaben verschwendet und die Zeitverzögerung der Darstellung oder Auswertung nimmt bei Mikrocontrollern ohne integrierte echte Nebenläufigkeit linear zu.

Um dem entgegen zu wirken kommen häufig integrierte Schaltkreise (**I**ntegrated **C**ircuits = IC) zum Einsatz. Es handelt sich dabei um häufig komplexe Schaltungen, die nur über eine minimale Anzahl an Pins zur Steuerung der Ein- und Ausgabe angeschlossen werden müssen. Ein solcher ist der Maxim MAX7219, den wir für unser Board verwenden. Es handelt sich bei ihm um einen Multiplexer IC (= Muxer), der 8x8 LEDs betreiben kann. Es besteht auch die Möglichkeit diesen IC um nahezu beliebig viele weitere Muxer des selben Typs zu erweitern, ohne dass der Mikrocontroller weitere Pins aufgeben müsste.

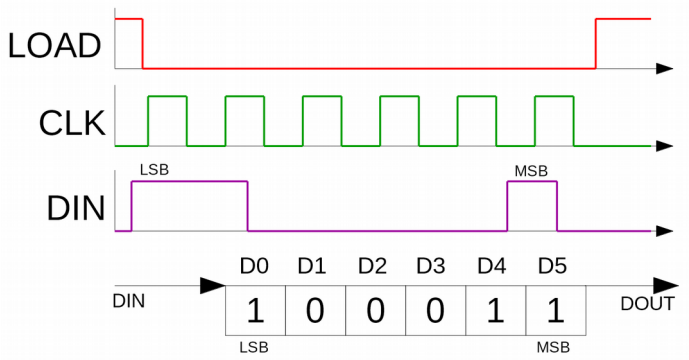


Abbildung 7: Beispielhafter 6Bit MSB-First Schieberegister empfängt den Befehl "110001".

In unserem Beispiel soll er die 8x8 LED Dot Matrix betreiben, für die wir in einem klassischem Ansatz zum Multiplexing 16 Pins benötigen würden. Er ermöglicht uns diese Anzahl auf nur 3 Pins zu reduzieren:

1. **D**ata **In** (DIN): Über diesen Eingang empfängt der IC Binärdaten (Ein- und Ausschalten) für die Instruktionen, welche Ausgänge er schalten soll. Es kann entweder mit der höchsten Stelle (**M**ost **S**ignificant **B**it = MSB) oder der niedrigsten Stelle (**L**east **S**ignificant **B**it = LSB) eine Übertragung begonnen werden.
2. **Cl**oc**k** (CLK): Weil DIN keiner definierten Geschwindigkeit unterliegt muss die Geschwindigkeitsvorgabe zur Interpretierung der DIN-Werte vom Mikrocontroller kommen. Das CLK-Signal vom Arduino kommend muss ständig zwischen `HIGH` und `LOW` (1 und 0) mit einer maximalen Geschwindigkeit von 10MHz alternieren. Mit jeder Rising Edge des CLK-Signals wird der aktuelle Wert des DIN-Signals in den Schieberegister geschoben. Der hinterste Wert des Registers wird an dem DOUT (**D**ata **Out**)-Pin angelegt.
3. **Load**: Jeder in den Register geschobene Befehl muss nachdem er vollständig eingetragen wurde auch aktiviert werden. Das passiert, wenn der Pin Load eine Rising Edge erzeugt. Es ist aus Kompatibilitätsgründen ratsam den Load Pin vor dem Beginn einer Übertragung auf `LOW` zu und am Ende wieder auf `HIGH` zu setzen.

Mit Hilfe der `shiftOut` Funktion kann man bei einem Arduino sehr einfach Schieberegister bedienen. So kann man den in der Abbildung 7 abgesendeten Befehl „`100011`“ wie folgt erzeugen:

```
digitalWrite(loadPin, LOW);
shiftOut(dinPin, clkPin, MSBFIRST, 0x31); // 0x31 = B00110001
digitalWrite(loadPin, HIGH);
```

Mehr zu der Funktion `shiftOut` findest du in der Arduino Referenz online. Informationen zu dem Schieberegister des MAX7219 sind angehängt. Nutze den Schaltplan des DevBoards, um die Pinbelegung herauszufinden.

1) Wie viele Bits hat der Schieberegister eines MAX7219? Mit welchem Bit muss bei dem Befüllen begonnen werden? Wie ist ein Befehl Befehl des MAX7219 aufgebaut?
2) Der MAX7219 startet im „`shutdown mode`“ und muss zuerst initialisiert werden. Welche Befehle muss man ihm zur Initialisierung schicken?
3) Gebe auf der LED Matrix ein großes X aus.
4) Schreibe eine Funktion, mit der du beliebige Muster auf der LED-Matrix darstellen kannst.

5 [!!!min] Analogdaten

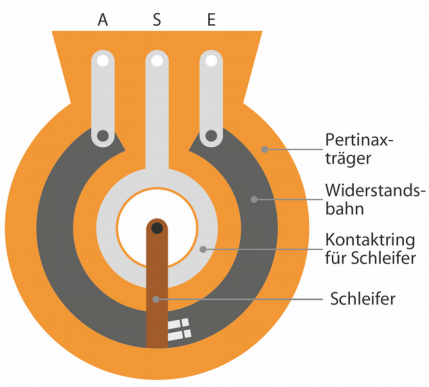


Abbildung 8: Funktionsprinzip eines Potentiometers. Quelle: heise.de

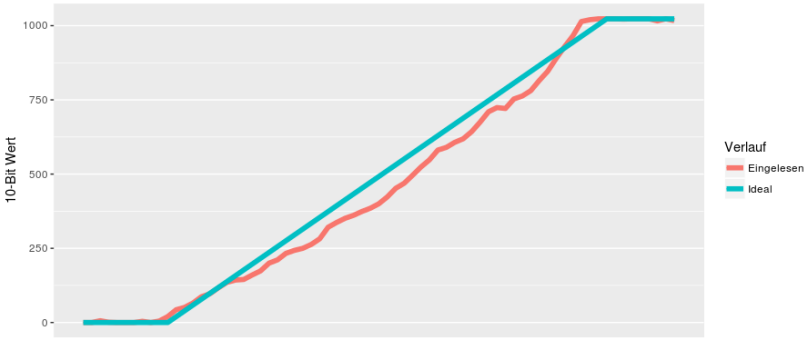


Abbildung 9: Durch manuelles Drehen erfasste Analogwerte als roter Graph "Eingelesen"

In diesem Kapitel betrachten wir den Potentiometer, oder aufgrund seiner Bauform den Trimmer, genauer. Bei einem Potentiometer handelt es sich um einen variablen Widerstand mit drei Kontakten. Zwischen zwei von ihnen ist ein fester Widerstand (bei uns sind es 10kΩ) in Form einer länglichen Kohleschicht verbaut. Der dritte Kontakt ist mit einem Schleifkontakt verbunden, der mit der Drehung des Potentiometers seine Position auf der Kohleschicht ändert und so an verschiedenen Stellen verschiedene Widerstände zu den beiden außenliegenden Pins erzeugt.

Unser Ziel ist es diesen Potentiometer mit einer 4-Bit Auflösung (8 verschiedene Werte) auszuwerten. Mit Hilfe der Funktion `analogRead` unter Angabe des auszulesenden Pins kann die Spannung zwischen Pin und Gnd gemessen und mittels eines auf dem Arduino befindlichen Analog/Digital-Wandlers in einen 10-Bit Wert gewandelt und zurückgegeben werden. Dabei werden als Grenzen standardmäßig die boardinternen 5V für den Wert 1023 (maximal) und Gnd für den Wert 0 (minimal) festgelegt.

Liegt am auszulesenden Analogpin A7 also beispielsweise 2V an (Potentiometer etwas über die Hälfte gedreht), gibt der Aufruf der Funktion `analogRead(A7)` also „409“ ((1023/5)*2) zurück.

1) Das Drehen des Trimmers gegen den Uhrzeigersinn erhöht den Wert, den analogRead ausgibt. Schreibe eine Funktion, die das Verhalten des Trimmers invertiert, also im Uhrzeigersinn den Wert erhöht.
2) Erweitere die Funktion so, dass sie statt den 10-Bit Wert, einen entsprechenden 4-Bit Wert ausgibt.

Doch in der Welt der Elektrotechnik ist wohl nichts perfekt und so müssen auch an dieser Stelle Probleme angegangen werden. Wie in der Abbildung 9 zu sehen ist, liefert der Potentiometer keine perfekten Ergebnisse, die zweifelsfrei und unformatiert auf die Stellung des Trimmers zurückführbar sind. Selbst wenn der Trimmer nicht bewegt wird zittern die Werte. Dies liegt an elektromagnetischen Störungen aus dem Umfeld, die die Ergebnisse beeinflussen.

Eine Möglichkeit dieses Problem zu beheben wird in der Arduino Beispielsammlung unter dem Stichwort „Smoothing“ gegeben. Dies wollen wir hierbei nicht weiter verfolgen, da in unserem Fall eine Hysterese eine bessere Lösung verspricht.

3) Erweitere die Funktion erneut so, dass sie an den „Schwellwerten“ zwischen zwei Ausgabewerten nicht mehr „zittert“. Erreiche dies durch eine Art Hysterese zwischen zwei Werten. (Hinweis: Du musst prüfen, ob sich der gemessene Wert außerhalb der Übergangsschwelle und einem dazu festgelegtem Toleranzbereich befindet; Du kannst die Funktion `abs` verwenden)

6 [!!!min] Zusammenfügen des Puzzles

Wenn du bis hier hin alle Aufgaben lösen konntest, dann Herzlichen Glückwunsch! Denn du kennst nun alle benötigten Techniken um das devBoard vollumfänglich programmieren zu können. Um dir die Arbeit zu erleichtern und das Programm effizienter zu machen möchte ich dir in diesem Kapitel noch einige Hilfestellungen bei der finalen Entwicklung des Spieleklassikers „Snake“ geben.

Um die folgenden Aufgaben bearbeiten zu können musst du die „Programmteile fertiges Spiel“ (siehe Kapitel „7: Anhänge“) herunterladen.

6.1 Spielregeln

1. Beim Start des Programms ist die „Schlange“ auf der 8x8 LED Matrix 2 Elemente groß und befindet sich oben links. Sie ist in Bewegung nach rechts.
2. Die 7 Segment Anzeige zeigt den im EEPROM gespeicherten Highscore auf den linken beiden Stellen. Wenn kein Highscore gesetzt ist, zeigt es „--“ an. Die aktuelle Länge der Schlange wird auf den beiden rechten Stellen angezeigt. Gleichzeitig zeigt die LED an, wie weit man sich dem Highscore genähert hat. Je heller sie ist, desto näher ist man dran. Wenn man den Highscore erreicht hat oder keiner definiert ist, ist die LED aus.
3. Ein drei Sekunden langer Druck auf den Taster S3 löscht den Highscore.
4. Ein Druck auf den Taster S1 lässt die Schlange „links abbiegen“. Ein Druck auf den Taster S2 lässt die Schlange „rechts abbiegen“.
5. Mit dem Poti lässt sich die Fortbewegungsgeschwindigkeit der Schlange in acht Stufen zwischen 100ms und 2s pro Schritt einstellen. Die aktuelle Geschwindigkeitseinstellung wird mit Hilfe der Punkte auf der 7 Segment-Anzeige dargestellt.
6. Das Überschreiten der Ränder mit der Schlange führt dazu, dass die Schlange am gegenüberliegendem Ende weiter fortgesetzt wird.
7. Es werden zufällig Punkte generiert, die mit einer Frequenz von 10Hz auf der 8x8 Led Matrix blinken. Sie dürfen nicht innerhalb der Schlange platziert werden. Das Erreichen dieser Punkte mit dem Kopf der Schlange führt dazu, dass sie ein Element länger wird und ein neuer Punkt generiert wird.
8. Sobald der Schlangenkopf auf ein Feld geht, welches bereits von einem anderen Schlangenteil besetzt ist, wird das Spiel beendet. Dabei wird gegebenenfalls der Highscore aktualisiert (sofern die erreichte Länge größer als der Highscore ist) und die Schlange bewegt sich nicht mehr. Außerdem wird auf dem 7-Segment Display im Sekundentakt abwechselnd der Schriftzug „noob“ und der normale Wert (Highscore und aktuelle Länge) ausgegeben. Das Neustarten des Controllers mittels des Reset-Tasters ist erforderlich, um ein neues Spiel zu starten.

6.2 Datentypen

Die wichtigsten Datentypen für unser Projekt sind Datentypen, die ganze Zahlen repräsentieren können. Bisher haben wir diesen kaum Aufmerksamkeit gewidmet und dafür stets den Datentypen `int` verwendet. Der Datentyp `int` ist jedoch in Wirklichkeit nur eine alternative Schreibform für `int16_t`, da er aus zwei Bytes (16 Bits) besteht und vorzeichenbehaftet ist. Somit kann er Zahlen zwischen -32768 und 32767 darstellen. Wenn man möchte, dass der Datentyp nur Zahlen im positiven Bereich darstellen kann, dann fügt man vor dem „int“ ein „u“ an. Ein `uint16_t` kann entsprechend Zahlen zwischen 0 und 65535 speichern.

Dieser Zahlenbereich ist jedoch in unserem Beispiel häufig überdimensioniert. Überdimensionierte Datentypen führen zu einer langsameren Verarbeitung des Programmablaufs und höherem Bedarf an Arbeitsspeicher zur Ausführungszeit. Statt nur eines zwei Bytes großen Datentyps, kann man bei Integers auch auswählen, wie groß er sein soll. Er kann 1, 2, 4 oder 8 Bytes groß sein. In den meisten Fällen benötigt man jedoch keine 8 Byte großen Zahlenbereiche. Wie groß er dann sein soll, wird durch die numerische Angabe hinter „int“ als Anzahl von Bits festgelegt.

Datentyp	Mit Vorzeichen		Ohne Vorzeichen	
1 Byte Integer	`int8_t`, `char`	-128 bis 127	`uint8_t`, `byte`, `unsigned char`	0 bis 255
2 Byte Integer	`int16_t`, `int`	-32.768 bis 32.767	`unit16_t`, `word`, `unsigned int`	0 bis 65.535
4 Byte Integer	`int32_t`, `long`	-2.147.483.648 bis 2.147.483.647	`uint32_t`, `unsigned long`	0 bis 4.294.967.295
8 Byte Integer	`int64_t`, `long long`	-2^{63} bis $(2^{63} - 1)$	`uint64_t`, `unsigned long long`	0 bis $(2^{64} - 1)$

Zuletzt bleibt noch zu erwähnen, dass man noch mehr Arbeitsspeicher sparen kann, indem man manche Werte als „Konstanten“ mit der Bezeichnung „const“ vor dem Datentyp definiert. Konstanten sind alle die Werte, bei der man weiß, dass sie sich während der Programmausführung nicht ändern werden. Beispielsweise sehen Pinbezeichnung häufig so aus: `const uint8_t pinLed = 5;` .

1) Füge die richtigen Datentypen im Abschnitt 1 in den zur Verfügung gestellten Programmteilen ein, indem du die „???“ ersetzt.
2) Vervollständige den Abschnitt 2 mit dem Programmcode, der die Geschwindigkeit der Schlange anhand der Stellung des Potentiometers regelt.

6.3 Strukturierungshilfen

Wie dir bereits vielleicht aufgefallen ist, speichert die Arduino IDE die Sketche in Textdateien mit dem Namen deines Projektes und der Dateiendung „.ino“ in einem Ordner, der ebenfalls nach deinem Projekt benannt ist. Wenn du versuchst, nur einen der beiden (Ordner oder Textdatei) umzubenennen und dann das Sketch in der Arduino IDE zu öffnen, wirst du feststellen, dass die IDE dir sagt, dass die Datei nicht in einem zugehörigem Ordner ist und deshalb nicht geöffnet werden kann.

Das liegt daran, dass es in der Arduino Welt die Möglichkeit gibt, den Quelltext auf mehrere Dateien aufzuteilen und damit die Lesbarkeit zu erhöhen. Dabei gibt es immer eine „Hauptdatei“, die zuerst geladen wird, und das ist die Datei, die den gleichen Namen wie der Ordner, in dem sie enthalten ist, hat. Am einfachsten kannst du eine neue Datei erstellen, indem du in der IDE oben rechts auf den kleinen nach unten gerichteten Pfeil und dann auf „Neuer Tab“ klickst.

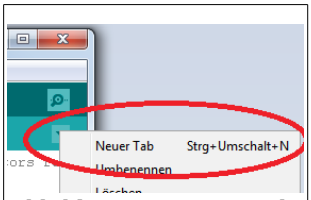


Abbildung 10: Neuer Tab in der Arduino IDE

Die Auslagerung des Quelltextes auf mehrere Dateien ist zwar schon ein guter Schritt, aber noch nicht optimal. Denn nun kann es vorkommen, dass versehentlich verschieden gemeinte Ausdrücke (Variablennamen, Funktionsnamen) mit gemeinsamer Bezeichnung in mehreren Dateien vorkommen. Das kann im besten Fall zu einer Fehlermeldung und im schlimmsten Fall zu einem schwer Auffindbarem Bug führen. Um diese potentielle Fehlerquelle abzuschalten gibt es in C++ „Namespaces“:

```
namespace MeinSpace1 {
  uint8_t meineVariable = 1;
  void meineFunktion () {Serial.println("Apfelkuchen");}
  // Innerhalb von Namespaces kann man "MeinSpace1::" weglassen
  void printMeineVar () {Serial.println(meineVariable);}
}
namespace MeinSpace2 {
  uint8_t meineVariable = 2;
  void meineFunktion () {Serial.println("Pflaumenkuchen");}
}
meineFunktion(); // Gibt Fehler aus, es fehlt die Angabe des namespaces
MeinSpace2::meineFunktion(); // Gibt "Pflaumenkuchen" aus
Serial.println(MeinSpace1::meineVariable); // Gibt "1" aus
```

Die Arduino IDE schränkt uns aber leider ein, indem bei der Kompilierung die verschiedenen Projektdateien der anderen Tabs erst am Ende der Hauptdatei angefügt werden. Dadurch kann es vorkommen, dass man z.B. in der `loop`-Funktion nicht auf Funktionen in Namespaces zurückgreifen kann, die in einem anderen Tab definiert hat. Dieses Problem beseitigt man mit einem Verfahren, dass sich Prototyping nennt. Man definiert vor den Funktionsaufrufen die Signaturen der Funktionen und definiert den Rest im Nachhinein nochmal.

```
// Datei: Hauptdatei
namespace MeinSpace {
  uint8_t meineVariable = 1;
  bool isGreater(uin8t_t, uint8_t);
}
// Ab hier koennen die Elemente aus dem Namespace verwendet werden

// Datei: MeinSpace.ino
bool MeinSpace::isGreater (uin8t_t a, uint8_t b) {
  return (a > b);
}
```

1) Erstelle die Prototypen für alle Funktionen im Namespace „DotMatrix“.
2) Vervollständige die Funktionen `Game::turnLeft()`, `Game::turnRight()` und `Game::iterate()`.
3) Vervollständige die Funktion `SevSeg::iterate()`.

Zuletzt sehen wir uns noch einmal an, wie wir uns mit Hilfe von anonymen Funktionen etwas Schreibarbeit sparen können. Anonyme Funktionen sind Funktionen, die über keinen Bezeichner, also Namen, verfügen. In vielen Programmiersprachen, C++ eingeschlossen, werden sie gelegentlich auch als Lambda-Funktionen bezeichnet.

Man verwendet sie immer dann, wenn eine Funktion etwa als Argument an eine andere Funktion übergeben werden soll, wie wir es beispielsweise bereits einmal im Kapitel 3.1 gemacht haben, als wir eine Funktion zum Anlegen eines Interrupts erstellen sollten.

Eine anonyme Funktion wird in C++ so angelegt: `[capture](Parameter)->Ausgabetyp {Funktionskörper}`

- **„Capture“**: Gibt an, welche Variablen der Umgebung (Scope) im Funktionskörper verwendet werden können. Mit beispielsweise `[a, b]` werden die Variablen `a` und `b` kopiert dem Funktionskörper zur Verfügung gestellt.
- **„Parameter“**: Auch Argumente genannt. Diese Parameterliste hat den selben Aufbau wie in einer normalen Funktionsdefinition.
- **„Ausgabetyp“**: Der Datentyp der Ausgabe. In einer normalen Funktionsdefinition wird dieser immer vor den Bezeichner geschrieben.
- **„Funktionskörper“**: Enthält die Befehle, die in der Funktion ausgeführt werden sollen.

Dieses Beispiel soll zeigen, wie man eine anonyme Funktion verwenden kann, um mit einem Interrupt einen sekündlich fortschreitenden Timer zurück zu setzen:

```
uint8_t wert = 0; // Anzahl der Sekunden seit dem letzten Reset
uint32_t lastCounted = 0; // ms, seitdem der Wert das letzte mal erhoeht wurde

void setup () {
  pinMode(3, INPUT_PULLUP);
  attachInterrupt(DigitalPinToInterrupt(3), [&wert]()->void {
    wert = 0;  // Bei Capture & verwendet, damit der Wert auch außerhalb der
               // Funktion auf 0 gesetzt wird. Theorie dahinter irrelevant.
  }, FALLING);
}

void loop () {
  if (millis() - lastCounted >= 1000) {
    lastCounted = millis();
    wert++; // Entspricht: wert = wert + 1;
  }
}
```

4) Schreibe eine anonyme Funktion in den Array `schedulerFunctions` mit dem Index 2.

6.4 EEPROM

Der verwendete Mikrocontroller besitzt eine EEPROM-Einheit. EEPROM ist ein wiederbeschreibbarer, nicht flüchtiger Speicher und steht für „**E**lectrically **E**rasable **P**rogrammable **R**ead **O**nly **M**emory“. Man kann ihn sich wie eine günstige Alternative zu Flash-Speicher (etwa in USB-Sticks) oder Magnetplattenspeicher (in Hard-Drive-Disks, Festplatten) vorstellen, die ihre gespeicherten Werte auch nach einer Trennung der Spannungsquelle noch behalten. Wir benötigen ihn, um den erreichten Highscore zu speichern.

1) Informiere dich, wie du den EEPROM eines Arduinos auslesen und überschreiben kannst. Vervollständige die Funktionen `Saver::write` und `Saver::read`.
2) Beim Neukauf eines Arduinos sind die Bytes eines EEPROMs in einem nicht festgelegtem Zustand. Wie kann man verhindern, dass der zufällige Startwert wie ein echter Highscore gewertet wird? Setze deine Idee in `Saver::write` und `Saver::read` um.

Der Aufbau von EEPROMs haben aber den Nachteil, dass sie vergleichsweise schnell verschleißen, wenn sie zu oft überschrieben werden. Der Verschleiß äußert sich durch Symptome wie nicht mehr überschreibbare Bits, die ihren vorherigen Wert beim Überschreiben behalten oder Bits, die ihren Wert von benachbarten Bits übernehmen. Der Verschleiß ist dabei maßgeblich von diesen Faktoren abhängig:

- **Lösch-/Schreibzyklen:** Jeder Zyklus verringert die Lebenszeit, wodurch die Anzahl der Zyklen auf ein Minimum reduziert werden sollte.
- **Speichermodus:** Es gibt zwei wesentlich unterschiedliche Speichermodi. Bei Byte-Write wird immer nur ein Byte überschrieben, bei Page-Mode werden immer 4 Bytes gemeinsam überschrieben. Der Page-Mode sorgt für ausgeglichene Potentiale zwischen den Bytes und reduziert somit den Verschleiß.
- **Temperatur und Spannung:** Je geringer die Temperatur und die Spannung bei den Lösch- und Schreibvorgängen ist, desto besser ist es.

Die meisten Anbieter von EEPROMs geben ihre Lebenszeit mit etwa 1 Million Lösch-/Schreibzyklen an. Viele Messungen von Hobbybastlern ergeben jedoch deutlich längere Lebenszeiten von 11 – 13 Millionen Zyklen. Um Fehler zu erkennen und automatisch korrigieren zu können, werden Algorithmen zur Fehlerkorrektur eingesetzt. Ein sehr häufig verwendeter Algorithmus dafür ist der „Solomon-Reed-Algorithmus“, den wir hier allerdings nicht weiter betrachten werden.

3) Überlege dir eine einfache Methode, um den Highscore (Maximalwert $64=2^6$) mit den verbleibenden 2 Bits so abzuspeichern, dass ein einfacher Speicher- oder Lese-Fehler erkannt werden kann.
4) Setze deine Methode in den Funktionen `Saver::write` und `Saver::read` um. Sollte ein Fehler erkannt werden, soll der Highscore auf „0“ zurück gesetzt werden.

6.5 Spiellogik

!!! Abstrakter Datentyp

7 Anhänge

Alle Anhänge findest du auf ***https://jlus.de/index.php/devboard***

1. Datenblatt Dev-Board
2. Datenblatt Arduino Nano, komplett
3. Datenblatt Max7219
4. Datenblatt 8x8 LED Matrix
5. Datenblatt 4 Stellen 7-Segment-Anzeige
6. Testprogramm
7. Programmteile fertiges Spiel