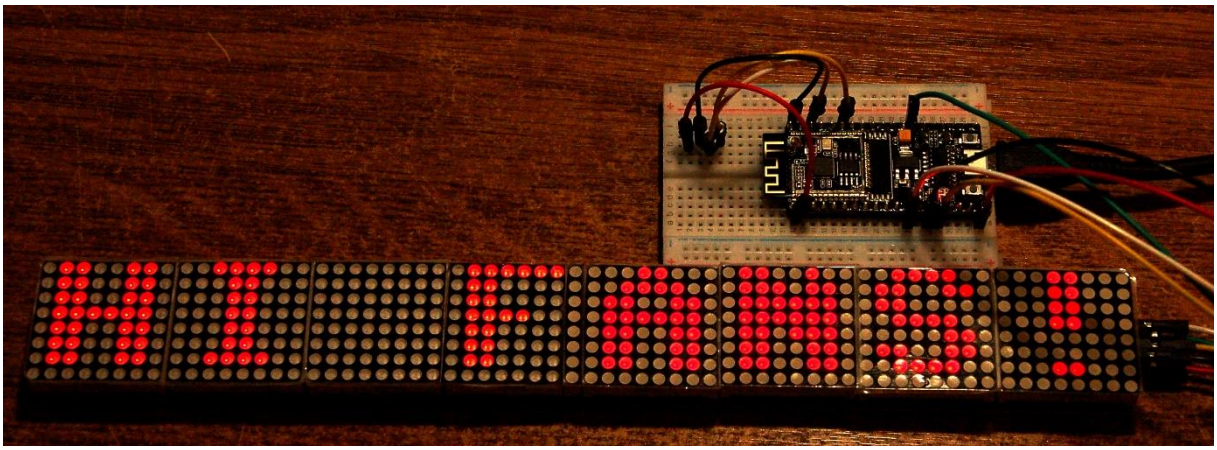
Achter-Display in Aktion

Dieser Beitrag ist auch als PDF-Dokument verfügbar.

Eine Großleinwand ist es sicher nicht, aber das Display, aufgebaut aus mehreren Matrixelementen mit 8x8 LEDs ist auch aus fünf Metern Entfernung noch gut ablesbar. Ich habe Module mit je vier Einzeldisplays kaskadiert und mit einem MicroPython-Modul zur Ansteuerung versehen. Die Klasse MATRIX aus dem Modul matrix8x8.py hat elementare Methoden aus der Klasse Framebuffer geerbt. Dazu zählen Pixel, Linien, Rechtecke und Textausgabe. Darüber hinaus stellt MATRIX aber auch komplexe Methoden wie Verschieben, Rollen und Blinken zur Verfügung. Natürlich ist das Display beliebig verlängerbar von einem Einzelelement bis zu Kaskaden aus mehreren Vierergruppen. Angesteuert wird das Display über einen reduzierten SPI-Bus. Die MISO-Leitung hat keine Funktion und wird daher eingespart. Interesse geweckt? Dann willkommen zu einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Das Mammut-Matrix-Display unter MicroPython

Bevor wir mit dem Erstellen der Klasse MATRIX beginnen, ein paar Informationen zum verwendeten LED-Modul und dessen Treiberchip MAX7219, das erleichtert das Verständnis bei der Programmierung.

Die Mäxe und der Framebuffer

Der MAX7219 ist nativ ein Hardwaretreiber für 7-Segment-Displays mit 8 Digits. Das heißt, er kann 8 Segmente, inclusive Dezimalpunkt, für 8 Anzeigeeinheiten (Ziffern) ansteuern. Das Ganze ist gemultiplext. Daher reichen 8 + 8 = 16 Leitungen für 64 LEDs. Unter dem Strich ist es aber nun egal, ob 8 Segmente oder 8 einzelne LEDs in einer Zeile in dem Moment an +Vcc gelegt werden, wenn gerade die von allen 8 Segmenten zusammengeführten Kathoden auf GND gehen. Wenn man es genau sieht, wurde der MAX7219, der unter jedem einzelnen Matrixdisplay sitzt, also zweckentfremdet. Statt 8 Stück 7-Segmentanzeigen erhalten wir also nur ein Digit mit 8 Zeilen und 8 Spalten. Dafür kann unser Digit aber außer Ziffern und einer Handvoll

Zeichen, wie bei einer 7-Segmentanzeige, auch alle Groß- und Kleinbuchstaben des englischen Alphabets und sogar Grafikelemente, wie Linien, Pixel und Rechtecke sowie beliebige Muster anzeigen.

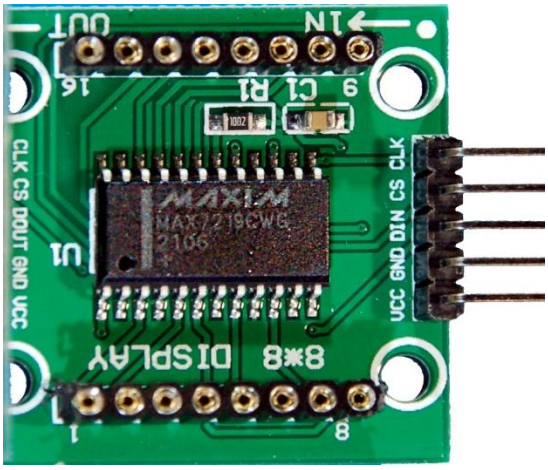

Abbildung 1: Der Displaytreiber MAX7219

Die besagten Eigenschaften werden durch das MicroPython-interne Modul **framebuf** mit der Klasse **Framebuffer** zur Verfügung gestellt. Der Name dieser Klasse ist Programm, denn damit das alles funktionieren kann, brauchen wir einen Auffangspeicher für die Pixelmuster des Displays. Dieser Speicher wird in unserer Klasse **MATRIX** in Form eines **Bytearrays** mit dem Namen **buffer** zur Verfügung gestellt. Die einzelnen Befehle füllen den Buffer Stück für Stück, bis wir ihn schließlich als Ganzes an die Mäxe des Displays schicken. Die Abbildung 2 zeigt den Zusammenhang für vier kaskadierte einzelne Matrixblöcke.

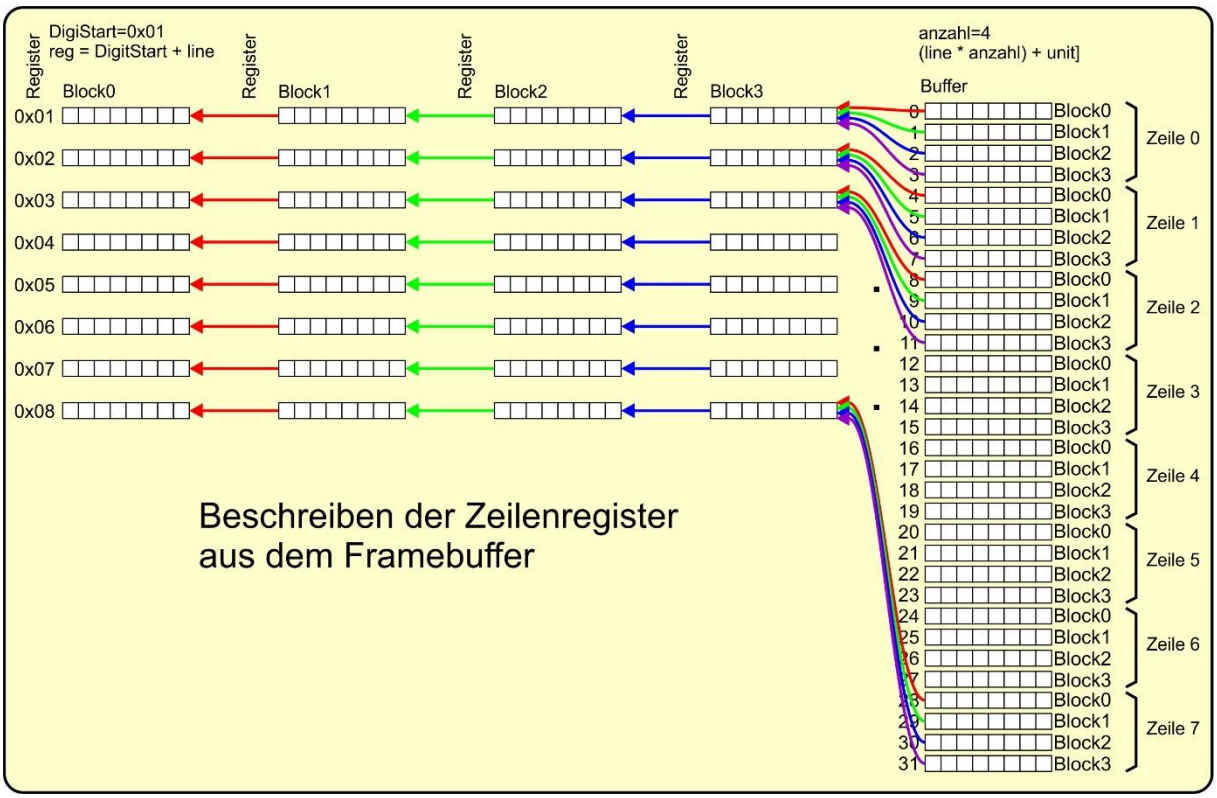

Abbildung 2: Matrix-Display und Framebuffer

Jeder MAX7219 besitzt, neben einigen anderen Registern, 8 Zeilenregister mit den Adressen von 0x01 bis 0x08. Eine 1 in einem Bit lässt die entsprechende LED im Display aufleuchten.

Im ESP32 oder ESP8266 habe ich den Framebuffer als Array, also als Folge von einzeln adressierbaren Bytes eingerichtet. Jedes Zeilen-Register in einem der Chips entspricht einem Byte im Framebuffer. Die Zuordnung ist so organisiert, dass die ersten 4 Bytes aus dem Buffer der ersten Zeile in den 4 Blöcken entsprechen. Die vier Bytes werden unter Angabe der Registeradresse 0x01 über den SPI-Bus nacheinander zum MAX7219 von Block 3 gesendet. Nachdem dieser Chip das Byte, das eigentlich für Block 0 gedacht ist, erhalten hat, senden wir ihm ein zweites Byte, das ist für Block 1 gedacht. Während Block 3 das empfängt, sendet er das zuvor erhaltene an den Block 2 weiter… Wenn Block 3 dann endlich das für ihn selbst gedachte Byte empfangen hat, ist das zuerst versandte Byte für Block 0 auch in Block 0 angekommen, denn jeder Treiberchip gibt ein Byte weiter, wenn er eines empfängt. Alle Chips liegen ja an derselben Taktleitung und an der derselben Freigabeleitung. Jede der 8 Zeilen des Displays wird nach demselben Schema übertragen. Hätten wir 8 Blöcke, bräuchten wir 8 Bytes für den Inhalt der ersten kompletten Zeile des Displays im Framebuffer und 8 Sendebefehle zum Durchschieben.

Die Taktgeschwindigkeit auf dem SPI-Bus beträgt nominell 10MHz. Wie schnell die Taktung wirklich ist habe ich nicht gemessen, sie ist in jedem Fall schnell genug, um einen fließenden Betrieb zu gewährleisten.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F oder ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board
nach Bedarf	MAX7219 8x8 1 Dot Matrix MCU LED Anzeigemodul oder MAX7219 8x32 4 in 1 Dot Matrix LED Anzeigemodul
1	KY-004 Taster Modul
1	LM2596S DC-DC Netzteil Adapter Step down Modul
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins

Die Hardware ist überschaubar. Zum Controller, ESP32 oder ESP8266, werden mehrere Blöcke von Matrixanzeigen im Format 8x8 Pixel benötigt. Die Breite der gesamten Anzeige, also die Anzahl an Elementen, richtet sich nach Ihren Bedürfnissen und ist frei skalierbar.

Zum Ankoppeln habe ich die Vierergruppen mit kurzen Litzen und einer 5-poligen Buchsenleiste versehen. Die passt genau zwischen Display und MAX7219. Die gekoppelten Blöcke schließen dadurch nahtlos aneinander an.

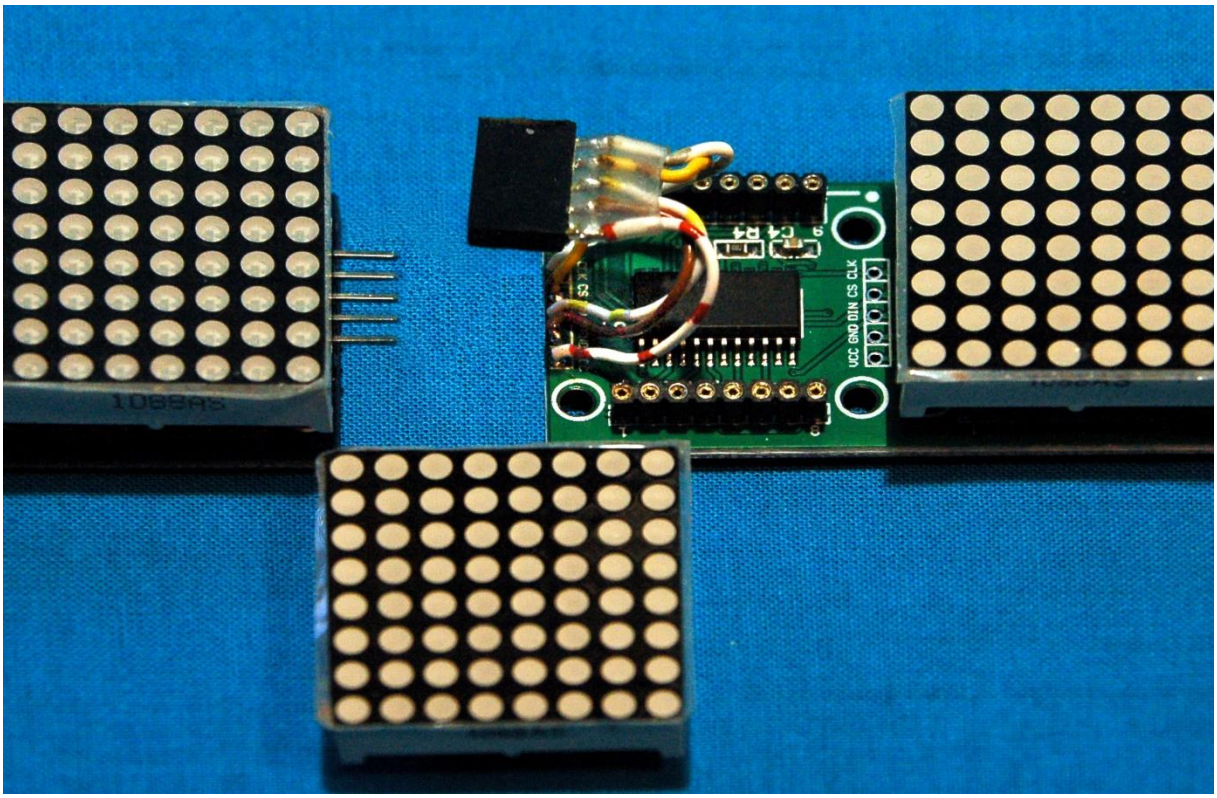

Abbildung 3: Kaskadierung mit Buchsenleiste

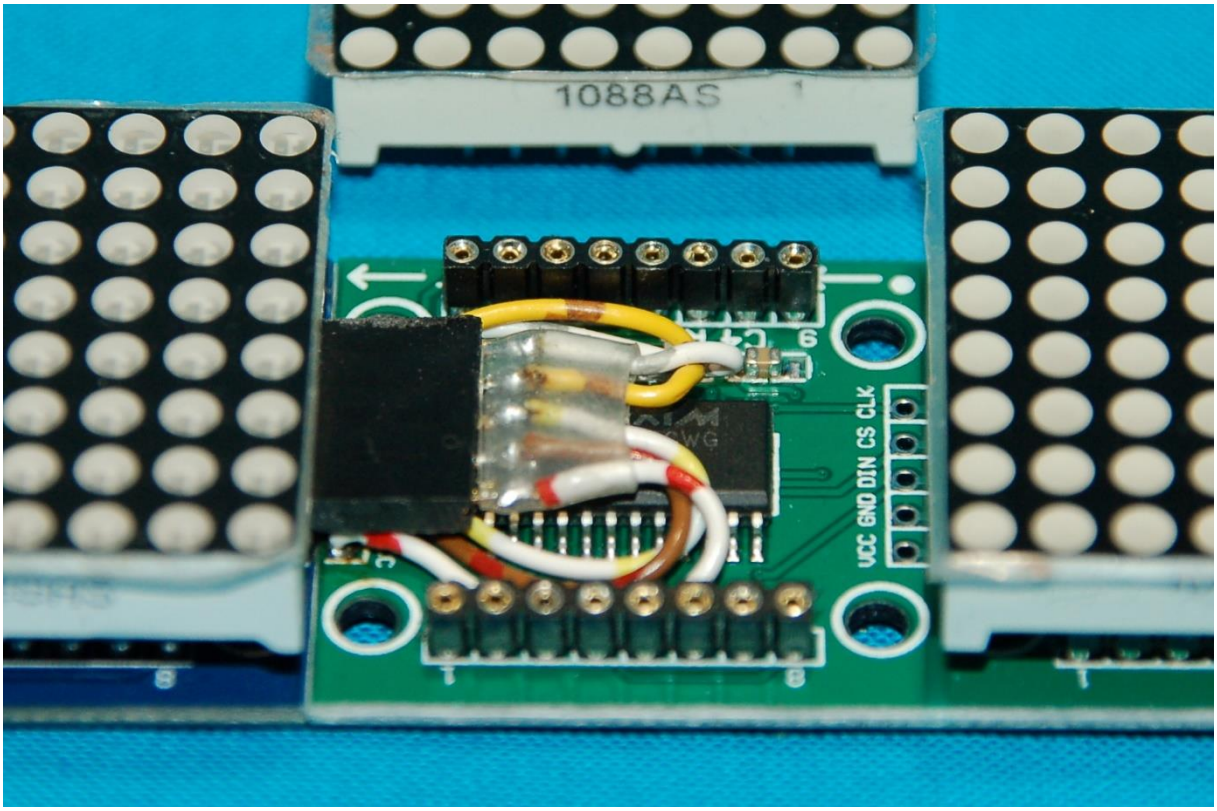

Abbildung 4: Angekoppelt

Die Schaltung ist sehr übersichtlich. Von der Anzeige ist nur ein Element eingetragen. Es kann ohne weiteres um eine oder mehrere 4-er-Gruppen erweitert werden, wie in Abbildungen 3 und 4 gezeigt.

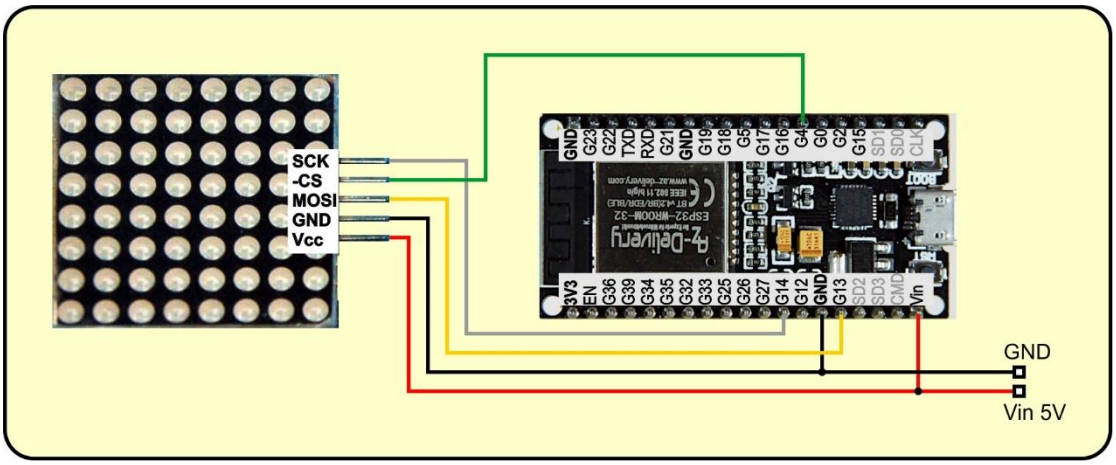

Abbildung 5: Mammut-Matrix-Schaltung

Als 5V-Versorgung für mehrere 8x8-Elemente sollte eine Spannungsversorgung mit ausreichend Stromstärkereserven verwendet werden. Ich verwende dafür ein Modul mit Step-Down-Converter, das bis zu 2A (3A kurzzeitig) liefern kann. Bei variabler Eingangsspannung von 5 bis ca. 20V wird der Ausgang auf 5V mit dem Trimm-Poti eingestellt.

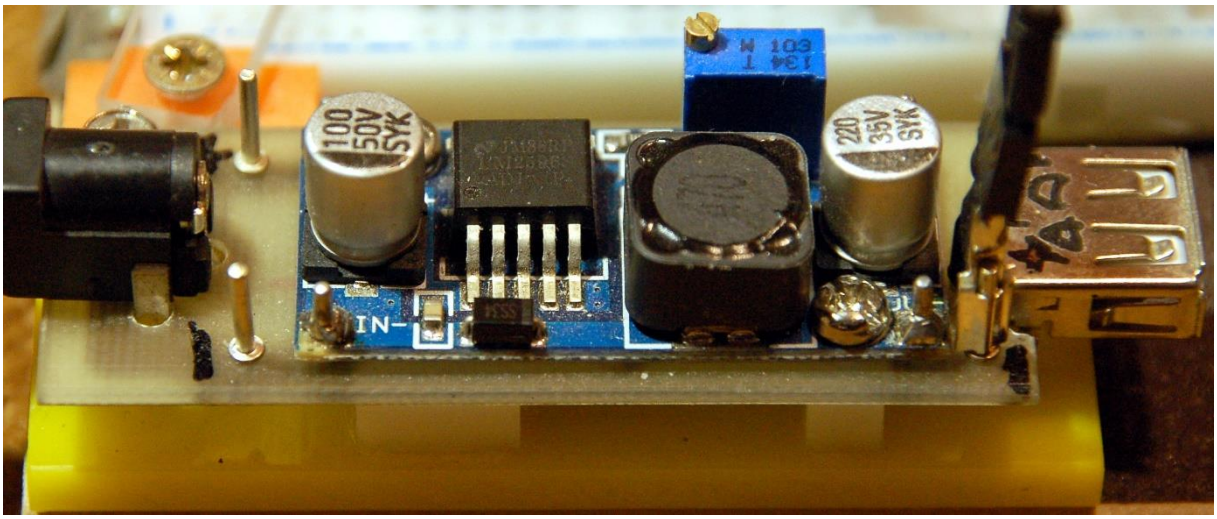

Abbildung 6: Spannungsversorgung mit Buck-Konvertrer auf meiner Trägerplatine

Ich habe mir dazu eine Basisplatine entworfen, mit Rohrbuchse am Eingang und USB-A-Buchse am Ausgang, das Ganze ist auf einer Plexiglas-Grundplatte montiert.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware

Bitte eine Stable-Version aussuchen
ESP8266 mit 1MB Version 1.18 Stand: 25.03.2022 oder
ESP32 mit 4MB Version 1.18 Stand 25.03.2022

Die MicroPython-Programme zum Projekt:

matrix8x8.py Treibermodul für den MAX7219
matrixtest.py Demoprogramm für ein x mal 8 -Matrixdisplay

Sonstige Software:

Packet Sender Downloadseite
Packet Sender Windows Install Version
Packet Sender Windows Portable
Packet Sender Linux

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 05.02.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Das Mammut-Matrix-Modul

Die Funktionsweise des MAX7219 habe ich eingangs ja schon erläutert. Somit können wir sofort mit der Programmierung des Treibermoduls beginnen.

Das Programm matrix8x8.py wird unter der MIT License kostenlos zur Verfügung gestellt und bietet neben dem Interface zur Hardware einige Features, welche die Anwendung wesentlich erleichtern.

So überschaubar wie die Hardware ist auch der Import von Modulen. Essentiell ist das Modul **framebuf**, von dem uns speziell die Klasse **Framebuffer** interessiert. Für die Zeitsteuerung importieren wir vom Modul **time** die Methoden **sleep** und **sleep_ms**. Wir schauen uns einmal an, was uns framebuf alles zu bieten hat.

```
>>> import framebuf
>>> framebuf
<module 'framebuf'>
>>> dir(framebuf)
['__class__', '__name__', 'FrameBuffer', 'FrameBuffer1',
'GS2_HMSB', 'GS4_HMSB', 'GS8', 'MONO_HLSB', 'MONO_HMSB',
'MONO_VLSB', 'MVLSB', 'RGB565']
>>>
```

Neben der Klasse Framebuffer ist die Konstante **MONO_HLSB** wichtig. Sie sorgt dafür, dass in unserem **horizontal** orientierten Display das LSB, also das niederwertigste Bit, im Framebuffer auch im LSB des Zeilen-Registers des MAX7219 landet. MONO_HMSB würde die Zeichen an der korrekten Position im Display seitenverkehrt anzeigen. Es ist ein interessantes Unterfangen, herauszufinden, was man damit alles anstellen kann.

Es folgen die Deklarationen für die Register des MAX7219. Die Information dazu findet sich im Datenblatt von Maxim Integrated.

Damit sind wir auch schon bei der Definition der Klasse MARTIX angekommen. Ich habe beschlossen, die Klasse Framebuffer zu beerben. Deshalb steht ihr Name in den runden Klammern hinter MATRIX. Das ist praktisch und in diesem Fall unbedenklich, weil keine der von mir definierten Methoden denselben Namen haben wird, wie eine Methode von Framebuffer. Damit wird auch keine dieser Methoden überschrieben. Und praktisch ist es deshalb, weil ich jede Methode aus Framebuffer so aufrufen kann, als wäre sie in MATRIX definiert. Beide Scopes sind also verschmolzen.

```
class MATRIX(framebuf.FrameBuffer):

    def __init__(self, spi, csPin, anzahl=4):
        self.spi = spi
        self.cs = csPin
        self.buffer=bytearray(anzahl*8)
        self.buf=bytearray(2)
        self.anzahl=anzahl
        super().__init__(self.buffer, self.anzahl*8, 8,\
                         framebuf.MONO_VLSB)
        self.initDisplay()
        self.blank=(0,0,0,0,0,0,0,0)
```

Die Funktion – Verzeihung, in einer Klasse heißen Funktionen ja Methoden, also die Methode **__init__()** ist der Konstruktor der Klasse. Sie wird dazu benutzt, um nach dem Bauplan der Klasse ein Objekt davon zu erstellen. Aufgerufen wird der Konstruktor später über den Namen der Klasse.

Der Parameter **self** ist eine Referenz auf ein erstelltes Objekt. Wir übergeben an die Methode das SPI-Objekt, das im aufrufenden Programm erstellt werden muss, das GPIO-Pin-Objekt für die Chipauswahl und die Anzahl von Matrixelementen.

Danach erzeugt der Konstruktor die objektspezifischen Objekte und Variablen, unter anderem den Framebuffer und ein bytearray der Länge 2, welches zur Kommunikation mit dem MAX7219 methodenübergreifend benötigt wird. Auch die Anzahl von Anzeigeelementen wird für das entsprechende Objekt festgelegt.

Jetzt folgt die Initialisierung der Klasse Framebuffer, von der wir geerbt haben. Ich gebe eine Referenz auf das Array **buffer**, die Anzahl der Matrixelemente, deren Höhe in Pixel und die Information zur Organisation von Framebuffer-Byte zu MAX7219-Register weiter.

Der Aufruf der Methode **initDisplay**() stellt alle beteiligten Mäxe auf Startposition. Zum Schluss erzeuge ich ein Tuple **blank**, das dazu dient, alle LEDs in einem Block zum Erlöschen zu bringen, wenn die Methode **shape**() darauf zugreift.

Als Nächstes brauchen wir eine Routine **initDisplay**(), die das Display initialisiert. Die Konfigurations-Register aller Mäxe müssen mit den Startwerten versorgt werden. Den Datensatz dazu bereitet die Methode **initDisplay**() vor, und die Methode **writeCommand**() sendet die Nachricht an alle eingebundenen Mäxe. Der Anzeigebereich wird gelöscht und die Löschung mittels **show**() sichtbar bemacht.

```
    def initDisplay(self):
        for cmd,val in (
            (Shutdown,0x00), # Display aus
            (DecodeMode, NoDecode),
            (Intensity, 0x08),
            (DisplayTest, 0x00),
            (ScanLimit, 0x07),
            (Shutdown,0x01), # Display an
            ):
            self.writeCommand(cmd,val)
            self.fill(0)
            self.show()
```

In dieser Methodendefinition taucht der Befehl **writeCommand** auf. Diese Methode ist eine der drei grundlegenden Funktionen, welche die unmittelbare Kommunikation mit den Mäxen erledigt. Dazu kommen wir gleich. Die Initialisierung wird abgeschlossen durch das Löschen des Displays. Die Methode **fill** aus der Klasse Framebuffer setzt einfach alle Bytes im "Bildschirmspeicher" auf 0. Damit das am Display sichtbar wird, muss der Buffer an die Mäxe geschickt werden, das macht die Methode **show**().

Die Methode **writeCommand**() nimmt die beiden Parameter **cmd** und **val**. **cmd** ist eine Registeradresse im MAX7219 und **val** der Wert, der geschrieben werden soll und zwar an alle Mäxe. Die Registeradresse kann man auch als Kommando-Code auffassen. So dient das Register Shutdown = 0x0C dazu, alle LEDs eines Elements aus- oder einzuschalten.

```
    def writeCommand(self,cmd,val):
        self.buf[0]=cmd
        self.buf[1]=val
        self.cs(0)
        for unit in range(self.anzahl):
            self.spi.write(self.buf)
        self.cs(1)
```

Die Parameter werden als Integerwerte übergeben. spi.write() will aber ein Objekt, das auf dem Buffer-Protokoll basiert. Wir haben im Konstruktor bereits ein Buffer-Objekt mit zwei Elementen erzeugt. Als Instanz-Objekt können wir es in jeder Methode verwenden, Stichwort: Recycling. Den Elementen weisen wir jetzt einfach die Referenzen auf die Parameter zu und haben damit das benötigte Bytes-Objekt. Wir legen die Chipselect-Leitung auf 0, und die for-Schleife schiebt nun die 16 Bits mit dem MSB voraus durch alle Mäxe bis zum Linksaußen. Jeder Chip hat damit denselben Befehl erhalten, der mit **self.cs(1)** übernommen wird.

Die Methode **write**() sendet den Inhalt des Bytearrays **buffer**. **buffer** sollte natürlich die Länge haben, die gleich zweimal der Anzahl der Matrixelemente ist, damit auch wirklich jedes anvisierte Ziel getroffen wird. Für jeden Befehl sind ja zwei Bytes zu versenden, das Kommando und der Registerinhalt.

```
    def write(self, buffer):
        self.cs(0)
        self.spi.write(buffer)
        self.cs(1)
```

Anders als mit **writeCommand**() können mit **write**() unterschiedliche Befehle an mehrere Mäxe gesendet werden. Das folgende Array würde zum Beispiel das erste und das letzte Element einer 8-er Reihe auf die Helligkeitsstufe 3 setzen und die Elemente 3 und 4 ausblenden.

```
>>> d.text("12345678",0,0)
>>> d.show()
>>>
b=bytearray(b'\n\x03\x00\x00\x00\x00\x0C\x00\x0C\x00\x00\x00\x00\x00\n\x03')
>>> d.write(b)
>>>
```

Die Nullbytefolge \x00\x00 stellt den NoOp-Befehl dar, der an den Elementen 1, 2, 5 und 6 keine Änderung bewirkt. Die NoOps sind als Füller nötig, weil es beim MAX7219 keine Möglichkeit gibt, gezielt Elemente anzusteuern, wie es bei Neopixel-Displays passiert.

writeAt() ist die dritte Methode mit Direktwirkung. Sie nimmt die Nummer der Einheit, ein Kommando und einen Wert als Ganzzahl. Nach der Erzeugung des Null-Bytearrays der Länge 2 mal Anzahl Matrixelemente wird die durch **unit** adressierte Stelle mit dem Kommandocode und dem Wert belegt. Dann gehen die Bytes des Arrays auf die Reise.

```
    def writeAt(self, unit, cmd, val):
        buf_= bytearray(self.anzahl*2)
        buf_[unit*2]=cmd
        buf_[unit*2+1]=val
        self.cs(0)
        self.spi.write(buf_)
        self.cs(1)
```

Die Methode **_writeLine**() steht von ihrer Bedeutung her zwischen den eben behandelten LOW-Level-Methoden und den Premiummethoden. Sie dient dazu, die acht Bytes eines Musters (shape) in die richtige Position des Framebuffers zu schreiben. Die Arbeitsweise ist nicht pixelorientiert, sondern an den Elementen ausgerichtet. Benutzt wird diese eigentlich private Methode von der Methode **shape**(), welche grafische Muster in einem Matrixelement darstellt.

```
    def _writeLine(self,unit,line,val):
```

```
        self.buffer[(line * self.anzahl) + unit]=val
```

Bei der Entwicklung eigener Methoden zur Ansteuerung der LED-Matrix hat sich die Methode **printBuffer**() als sehr nützlich erwiesen. Nach der Verwendung von Befehlen, die in den Framebuffer schreiben, kann man damit dessen Inhalt ausgeben lassen. Die Bytes werden als 0-1-Folgen mit vollen 8 Zeichen Länge ausgegeben.

```
    def printBuffer(self):
        for i in range (len(self.buffer)):
            print("{:08b}".format(self.buffer[i]))
```

Alle nun folgenden Methoden zähle ich bereits zur High Society, weil sie sich der LOW-Level-Befehle bedienen, um mit den Mäxen zu kommunizieren. Sie erledigen jeweils auch eine ganz spezielle Aufgabe.

setIntensity() stellt die Helligkeit aller Matrixelemente auf einen Helligkeitswert von 0 bis 15 ein, indem sie **writeCommand**() die Registernummer und den Wert übergibt.

```
    def setIntensity(self,val):
        intensity=val if 0<=val<=15 else 15
        self.writeCommand(Intensity, intensity)

    def shutdown(self,state=True):
        val=ShutdownMode if state else NormalMode
        self.writeCommand(Shutdown,val)
```

shutdown() geht ähnlich vor, um alle Elemente dunkel zu tasten (**True** = default) oder einzublenden (**False**). Wenn es, wie hier, nur um die entsprechende Zuweisung von Werten geht, verwende ich gerne Conditional Expressions (bedingte Ausdrücke). So geht in einer Zeile, was sonst vier Zeilen beansprucht.

val=ShutdownMode if state else NormalMode

statt

```
if state:
   val = ShutdownMode
else:
   val = NormalMode
```

Wie bekommen wir denn nun Zeichen und grafische Elemente in die Anzeige? Wenn Sie sich die Datei **matrix8x8**.py bereits heruntergeladen und durchgeschaut haben, werden Sie keinerlei Befehle dazu gefunden haben. Die sind aber implizit bereits ab der ersten Zeile im Namensraum der Klasse **MATRIX** vorhanden. Sie stammen nämlich alle aus dem Erbe der Klasse **FrameBuffer**. Ich schiebe einfach einmal eine Aufstellung dazwischen.

fill(c)	gesamtes Display	Display füllen oder löschen
fill_rect(x,y,b,h,c)	pixelorientiert	gefülltes Rechteck

		x,y linkeobere Ecke b,h Breite und Höhe
hline(x,y,w,c)	pixelorientiert	Horizontale Linie x,y Startpunkt (links) w Breite/Länge
line(x,y,xb,yb,c)	pixelorientiert	x,y Startpunkt, xb,yb Endpunkt
pixel(x,y,c)	pixelorientiert	Punktkoordinaten
rect(x,y,b,h,c)	pixelorientiert	Umrandung x,y linkeobere Ecke b,h Breite und Höhe
scroll(x,y)	pixelorientiert	Displayinhalt verschieben mit Vektor x,y
text(t,x,y,c)	pixelorientiert	Textausgabe x,y linkeobere Ecke
vline(x,y,h,c)	pixelorientiert	vertikale Linie x,y startpunkt (oben) h Höhe

c=0 | 1: dunkel | hell
x=0..(Anzahl Elemente * 8) - 1
y=0..7
b=1..(Anzahl Elemente * 8) - x
h=1..8-y
t Textkonstante oder String-Variable

Damit die Objekte sichtbar werden, muss der Framebuffer zum Display geschickt werden. Das macht die Methode **show**(). Sie schiebt Zeile für Zeile die entsprechenden Bytes aus dem Framebuffer in die Register der Anzeigeblöcke. Vergleichen Sie dazu die Abbildung 2.

```
    def show(self):
        for line in range (8):
            self.cs(0)
            for unit in range(self.anzahl):
                self.buf[0]=DigitStart+line
                self.buf[1]=self.buffer[(line * self.anzahl)\
                                        + unit]
                self.spi.write(self.buf)
            self.cs(1)
```

Nachdem Sie jetzt wissen, wie die Anzeige mit Inhalten zu füllen ist, schauen wir uns die restlichen Methoden von MATRIX an.

blink() führt das aus, was der Name verspricht, und lässt die Anzeige blinken. Sie geht zuerst für **off** Millisekunden aus und dann für **on** Millisekunden an. Letzteres fällt natürlich nicht auf, wenn danach die Anzeige an bleibt. Mit **cnt** geben wir an, wie oft der Vorgang wiederholt wird. Erst ab cnt=2 fällt das Blinken auf.

```
    def blink(self,on,off,cnt=1):
        for i in range(cnt):
```

```
            self.shutdown()
            sleep_ms(off)
            self.shutdown(False)
            sleep_ms(on)
```

```
    def clear(self,show=True):
        self.fill(0)
        if show:
            self.show()
```

clear() nutzt die Methode **fill**(), um das Display mit Hintergrundfarbe zu füllen. Sichtbar wird die Löschung sofort, wenn **show=True** übergeben wird. Mit **False** warten wir, bis zum nächsten **show**(), explizit im Programmtext oder implizit in einer Methode.

shape() verlangt die Nummer eines Anzeigeelements und ein Tuple oder eine Liste mit den 8 Bitmustern einer 8x8-Grafik, um diese mit Hilfe der Methode **writeLine**() in den Framebuffer zu schreiben. **show**=**True** bringt das Muster sofort zur Anzeige.

```
    def shape(self,unit,buf,show=False):
        for line in range(8):
            self._writeLine(unit,line,buf[line])
        if show:
            self.show()
```

Das folgende Beispiel stellt ein Herzchen in Anzeigeelement 0 dar. **True** führt dazu, dass das Muster auch sofort angezeigt wird.

```
>>> heart=( 0b00000000,
      0b01101100,
      0b11111110,
      0b11111110,
      0b01111100,
      0b00111000,
      0b00010000,
      0b00000000,
    )

>>> shape(0,heart, True)
```

clearUnit() nutzt die Methode shape, um ein Null-Muster, das durch **self.blank**=(0,0,0,0,0,0,0,0) im Konstruktor definiert ist, über den Inhalt im Framebuffer zu schreiben.

```
    def clearUnit(self,unit,show=False):
        self.shape(unit,self.blank)
        if show:
            self.show()
```

```
    def clearFT(self,von,bis=None,show=False):
        bis_=self.anzahl
        if bis is not None:
            bis_=self.anzahl if bis>=self.anzahl else bis
        for unit in range(von, bis_):
            self.shape(unit,self.blank)
        if show:
            self.show()
```

Auch **clearFT**() nutzt die Methode **shape**(). Sie löscht die Matrixelemente von **von** bis **bis**, falls dafür ein Wert angegeben wurde. Fehlt die Zuweisung an **bis**, wird bis zum Anzeigeende gelöscht.

pixelShift() verschiebt den Anzeigeinhalt pixelweise um die Entfernung **distance**. Die Richtung wird durch den Parameter **direction** festgelegt. mit **delay**=50 ms erfolgt der Positionswechsel fließend.

```
    def pixelShift(self,distance, direction="left",delay=50):
        richtung=direction.upper()
        dir_={
            "LEFT":(-1,0),
            "RIGHT":(1,0),
            "UP":(0,-1),
            "DOWN":(0,-1),
            }
        x,y=dir_[richtung]
        for i in range(distance):
             self.scroll(x,y)
             sleep_ms(delay)
             self.show()
```

Das Dictionary **dir** übersetzt die Klartextbegriffe in die entsprechenden Vektoren, welche die Methode **scroll**() aus der Klasse FrameBuffer verlangt. Weitere Richtungsabgaben können natürlich ergänzt werden. Damit die Verschiebung effektvoll ist, sollte der Wert für **delay** nicht zu klein gewählt werden. Zu große Werte führen zu stop and go und sind nützlich, wenn man genau verfolgen möchte, was passiert, zum Beispiel bei der vorletzten Methode **roll**().

```
    def roll(self,text, cnt=1, multiply=False, delay=50):
        l=len(text)
        text_= text if text[l-1]==" " else text+" "
        #print(text_)
        t=text_
        if multiply:
            while len(t) <= (self.anzahl//4)*3:
                t+=text_
        t = t + " "*(self.anzahl)
        for w in range(cnt):
            for s in range(len(t)):
                self.clearUnit(self.anzahl-1)
                self.text(t[s],(self.anzahl-1) * 8,0,1)
```

```
                self.pixelShift(8,"left",delay)
```

Die Methode rollt Texte über das Display. Sie werden in **text** übergeben. Mit **cnt** geben wir an, wie oft der Prozess wiederholt werden soll. Texte, die kürzer als drei Viertel der Anzeigenlänge sind, können durch **multiply**=True vervielfacht werden. Den Parameter **delay** kennen wir bereits von **PixelShift**(), wo er hier auch zur Anwendung kommt.

Damit der Text sauber rollt und keinen Kometenschweif hinter sich herzieht, muss zumindest die letzte Pixelspalte rechts aus sein. Einige Zeichen sind aber 8 Pixel breit. Genau die würde Probleme bereiten. Deshalb untersuchen wir, ob der Text mit einem Leerzeichen endet. Tut er das nicht, fügen wir eines ein. Das erledigen wir wieder mit einer Conditional Expression. Im nächsten Schritt wird der Text, falls nötig, so lange mit sich selbst addiert, bis er mindestens drei Viertel solang ist, wie die Anzeige Elemente hat.

Jetzt lassen wir die Geschichte rollen, so viele Runden, wie **cnt** vorgibt. Die Rechtsaußen-Position wird gelöscht, dann fügen wir genau dort das s-te Zeichen aus dem Text ein und schieben das gesamte Display um 8 Positionen nach links.

Ein Gadget habe ich noch, die Methode **center**(), die genau das tut, was der Name verspricht – falls das möglich ist. Texte, die kürzer als die Anzeigenbreite sind, werden zentriert, längere Texte werden gerollt und zwar einmal aus der Anzeige hinaus.

```
    def center(self,text):
        pixBreite=self.anzahl*8
        mitte=pixBreite//2
        l=len(text)
        if l > self.anzahl:
            self.roll(text, velocity=100)
        else:
            startPos=mitte-(l*8)//2
            self.text(text,startPos,0,1)
            self.show()
```

Das Testprogramm

Natürlich brennen Sie schon darauf, die neue Klasse auszuprobieren. Das geht in Handarbeit über das Terminal von Thonny, aber viel schöner kann das das Programm matrixtest.py. Laden Sie es doch gleich einmal herunter, öffnen Sie es in einem Editorfenster und starten Sie es dort mit F5. Dann lehnen Sie sich zurück, klopfen sich auf die Schulter und genießen die Demo.

Ausblick

Sollte es jetzt an Projekten für die Anwendung der Klasse MATRIX fehlen, dann kann ich Ihnen mit dem nächsten Blogpost aus dieser Reihe ein Thermo- Hygrometer mit

dem SHT21 ankündigen. Dieses Modul wird über den I2C-Bus angesteuert. Zusammen mit der Mammut-Matrix ergibt das ein Überwachungsgerät, das auch aus größerer Entfernung gut ablesbar ist. Es ist Frühjahr, vielleicht haben Sie ja ein Treibhaus, dessen Klima Sie vom Fenster aus überwachen möchten? Dann ist das genau das Richtige für diesen Zweck.

Bis dann!