Neopixel mit Nano-Board ansteuern

Schon seit einiger Zeit gibt es preiswerte Neopixel-Bänder (vgl. Abb. 1). Auf diesen sitzen Bausteine mit jeweils 3 LED für die Grundfarben R(ot), G(rün) und B(lau) sowie einer Steuereinheit. Bei dieser Steuereinheit handelt es sich meist um den WS2812. Das Interessante dabei ist, dass hierbei über eine einzige Datenleitung sämtliche Steuereinheiten mit ihren LEDs einzeln gesteuert werden können.

Abbildung 1

In letzter Zeit sieht man statt der Neopixel-Bänder auch Neopixel-Streifen oder Ringe (vgl. Abb. 2). Diese sind m. E. gut für erste Experimente geeignet. Die Preise (Juli 2020): 1 m Neopixelband mit 30 RGB-LEDs bei Ebay für unter 4 Euro aus Hongkong oder Taiwan und 3 Ringe mit jeweils 12 RGB-LEDs für unter 10 Euro bei Amazon Prime, jeweils inkl. Porto.

Abbildung 2

1 Funktionsweise

Eine Neopixel-Einheit besteht aus einem WS2812-Chip und einer RGB-LED, die durch diesen Chip gesteuert wird. Chip und RGB-LED werden über Vcc (3 - 5 V) und GND mit Strom versorgt (vgl. Abb. 3.). Dabei wird die Helligkeit der drei LEDs jeweils durch eine 8-Bit-Wert eingestellt: Der Wert 0 schaltet die LED aus, maximale Helligkeit wird mit dem Wert 255 erreicht. **Um eine RGB-LED zu steuern, sind also 24 Bit erforderlich**. **Diese werden seriell über die DI-Leitung an den WS2812 übertragen**.

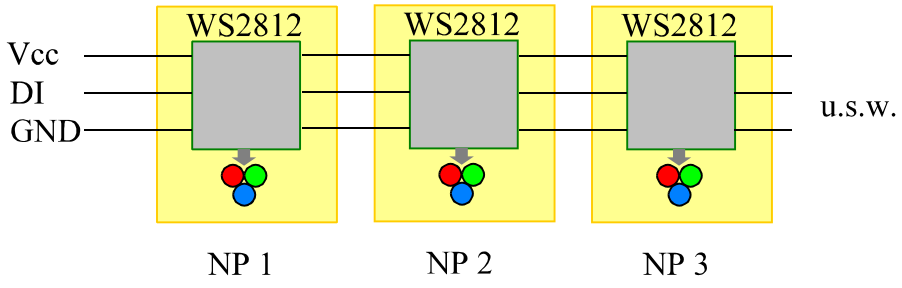


Abbildung 3

Dabei ist die Reihenfolge der einzelnen Bits: G7, G6, ..., G0, R7, R6, ..., R0, B7, B6, ..., B0. Wie sind nun die einzelnen Bits auf der DI-Leitung zu kodieren. Das Datenblatt gibt folgende Signale für die 0- und 1-Bits vor:

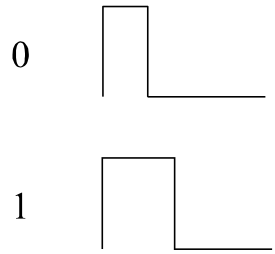


Abbildung 4

Dabei werden für die **High- und Low-Phasen** jeweils folgende Angaben gemacht:

Signal	High	Low	Low (nach J. Levine, s. u.)
0	200 - 500 ns	650 - 950 ns	bis ca. 5000 ns
1	550 - 850 ns	450 - 750 ns	bis ca. 5000 ns

Nun wird der Atmega328 des Nano-Boards üblicherweise mit 16 MHz getaktet. Das bedeutet: Ein einziger Takt dauert 62,5 ns. Die meisten AVR-Maschinen-Befehle benötigen 2 Takte, also schon mehr als 100 ns. Befehle einer Hochsprache wie z. B. BASCOM erfordern häufig deutlich mehr Zeit; in der Regel sind deren Ausführungszeiten auch nicht bekannt. Auf den ersten Blick scheint es demnach nicht möglich zu sein, Neopixel mit BASCOM anzusteuern.

Never say never - heißt es schon bei James Bond. Und in der Tat gelingt das sogar verblüffend einfach, ohne den Einsatz von Timern oder Interrupts. Die Lösung findet man bei der Webseite

https://wp.josh.com/2014/05/13/ws2812-neopixels-are-not-so-finicky-once-you-get-to-know-them/

von Josh Levine. Er hat herausgefunden, dass der WS2812 sehr tolerant ist, was die Länge der Low-Phasen angeht; sie können 5000 ns und mehr bertragen (vgl. obigeTabelle). Auf diese Webseite bin ich gestoßen über den Beitrag

https://www.elektronik-labor.de/AVR/Neopixel.html

von Dirk Beilker. Dort stellt er auch ein vollständiges BASCOM-Programm vor. Mehr dazu in Abschnitt 2. Zuvor wollen wir uns noch klar machen, wie die Neopixel-Kette funktioniert.

Werfen wir dazu einen Blick auf die Abb. 3: Vom Mikrocontroller werden an den DI-Eingang von NP1 serielle Signale wie in Abb. 4 dargestellt gesendet. Der WS2812-Baustein von NP1 empfängt die Signale, reicht diese aber nicht an den folgenden Baustein NP2 weiter. Sobald NP1 die ersten 24 Bit empfangen hat, wertet sein WS2812-Chip sie aus und steuert “seine” 3 LEDs entsprechend den empfangenen Helligkeitswerten. Die darauf folgenden Bits wertet der WS2812-Chip von NP1 nicht mehr aus; dafür leitet er sie weiter an den nächsten Neopixel-Baustein, NP2. Dieser geht nun genauso vor: Er steuert mit den nächsten 24 Bit “seine” 3 LEDs

und leitet erst anschließend die nachfolgenden Signale an NP2 weiter. Auf diese Weise erhalten alle folgenden Neopixel-Bausteine nach und nach ihre Steuerwerte.

Erst wenn der Mikrocontroller ein **Reset-Signal** an NP1 sendet, werden dieser und die restlichen WS2812-Chips wieder in den Start-Zustand versetzt. Dieses Reset-Signal besteht aus einem **Low-Signal von mindestens 50 µs**.

2 Programm-Idee

Wir gehen davon aus, dass die Steuerwerte für die LEDs in den Byte-Arrays R, G und B bereits abgelegt worden sind. Der DI-Eingang von NP1 ist mit PortB.2 des Nano-Boards verbunden. Die gesamte Steuerung der Neopixel-Kette erfolgt durch folgende Schleife:

```
for N = 1 to Anzahl_neopixel
  if G(n).7 = 0 then
     PORTB.2 = 1
     NOP:NOP:NOP:NOP
     PORTB.2 = 0
  else
     PORTB.2 = 1
     NOP:NOP:NOP:NOP:NOP:NOP:NOP:NOP:NOP
     PORTB.2 = 0
  end if
  if G(n).6 = 0 then
    ' wie bei G(n).7 ...

  '... restliche Bits von G(n) senden

  '... genauso die Werte von R(n) und B(n) übertragen
next N
```

Manch einer fragt sich vielleicht an dieser Stelle schon: Warum werden hier die Befehlsfolgen für jedes Bit einzeln hingeschrieben. Wäre es nicht ratsam, dies mit Hilfe einer Schleife zu erledigen, etwa in der Art

```
for N = 1 to Anzahl_neopixel
  for Bitnummer = 7 to 0 step - 1
    if G(n).Bitnummer = 0 then
      PORTB.2 = 1
      NOP:NOP:NOP:NOP
      PORTB.2 = 0
    else
      ...
    End if
  Next Bitnummer
  ... weitere Farben
next N
```

Die Antwort lautet: Im Prinzip Ja! Aber die zusätzliche Schleife kostet weitere Zeit und noch schlimmer verhält es sich mit dem Zugriff auf ein einzelnen Bit, wenn die Bitnummer eine Variable ist: Ein solcher Zugriff ist zwar möglich, dauert aber erheblich länger als bei einer Konstanten; außerdem hängt die benötigte Zeit auch noch von der Höhe der Indexnummer ab! Diesen Umstand habe ich ausführlich auf meiner Webseite

http://www.forum.g-heinrichs.de/viewtopic.php?f=16&t=131&p=188#p188

erläutert. In Hinblick auf die oben schon angesprochene Zeitproblematik würde all dies zu immensen Schwierigkeiten führen. Bleiben wir also bei dem ursprünglichen Quellcode.

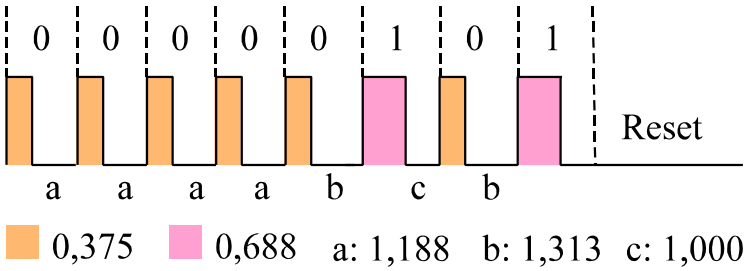
Beginnen wir mit dem Fall, dass G(n).7 den Wert 0 hat. In diesem Fall werden die Befehle

```
PORTB.2 = 1
NOP:NOP:NOP:NOP
PORTB.2 = 0
```

ausgeführt. Zunächst werden PortB.2 bzw. DI auf 1 gesetzt. Ab jetzt müssen wir die benötigte Zeit stoppen: Bei den folgenden vier NOP-Befehlen handelt es sich um Befehle, die nichts bewirken (No OPeration); sie benötigen jeweils 1 Takt. Durch den folgenden Befehl wird PortB.2 bzw. DI wieder auf 0 zurückgesetzt; dazu sind zwei Takte erforderlich. Insgesamt dauert die High-Phase also $4 \cdot 1 + 2 = 6$ Takte. Das entspricht 6/16 ms, also 375 ns. Dieser Wert liegt fast genau in der Mitte des Toleranzintervalls für den High-Zustand des 0-Signals.

Wie sieht es bei diesem 0-Signal nun mit der Dauer des Low-Zustands aus? Im Quelltext sieht man keine expliziten Befehle zum Warten. Tatsächlich ist es so, dass nach dem Zurücksetzen von PortB.2 von BASCOM eine ganze Reihe von Befehlen abgearbeitet werden müssen: Zunächst muss der else-Teil übersprungen werden. Dann muss der nächste Puls vorbereitet werden: Für die Überprüfung G(n).6 = 0? muss zunächst der Wert von n aus dem Speicher geholt und damit G(n) bestimmt werden. Von diesem Array-Wert wird jetzt das Bit 6 ermittelt. Nun wird überprüft, ob dieses Bit den Wert 0 hat. Anschließend wird die High-Phase des nächsten Signals mit PortB.2 = 1 eingeleitet. Bis dahin vergeht viel Zeit; sie entspricht der Low-Phase des 0-Signals von Bit 7.

Mit einem Logik-Analysator habe ich die High- und Low-Phasen für das Byte 5 = &B00000101 einmal aufgezeichnet. Das Ergebnis ist in Abb. 5 zu sehen.



Alle Angaben in µs

Abbildung 5

Offensichtlich braucht das BASCOM-Programm auf dem Atmeg328 ca. 1,188 µs für die oben angegebene Befehlsfolge, welche die Low-Phase von Bit 7 ausmachen. Das entspricht 19 Takten - kein Wunder bei der umfangreiche Aufgabe, die zu erledigen war.

Die festgestellt Länge der Low-Phase liegt außerhalb des offiziellen Toleranzintervalls, aber sie ist deutlich kleiner als die von J. Levine ermittelte obere Grenze. Deswegen wird der WS2812 diese Phase auch als Low-Phase des Bits 7 akzeptieren.

Nachdem PortB.2 nun wieder den Wert 1 angenommen hat, beginnt die High-Phase des Bits 6; und die nächsten Phasen verlaufen ähnlich den bisher geschilderten Phasen.

Schauen wir uns nun an, wie sich das Programm bei einem 1-Signal verhält. In Abb. 5 ist das bei dem Bit mit der Nummer 2 der Fall. Jetzt springt es in den else-Teil. In diesem tauchen nicht nur 4, sondern 9 NOP-Befehle auf. Demzufolge dauert die High-Phase nun 11 Takte; das entspricht einer Zeit von 0,688 µs. Das liegt ziemlich genau in der Mitte des Toleranzintervalls für die High-Phase des 1-Signals!

Das Timing-Diagramm bestätigt diesen Zeitwert. Auch die anderen gemessenen Zeitwerte für die Low-Phasen liegen in dem Toleranzbereich. Wie auch schon beim 0-Signal können wir die Dauer der Low-Phasen auch bei den 1-Signalen nicht aus den BASCOM-Befehlen herleiten. Dazu müsste man einen genaueren Blick auf den von BASCOM erzeugten Maschinencode werfen.

Allerdings soll hier noch auf einen Umstand hingewiesen werden: Die Dauer einer Low-Phase hängt nicht nur davon ab, ob es zu einem 0- oder einem 1-Signal gehört. Vielmehr ist auch von Bedeutung, welcher Signal-Typ folgt. Die Signale für die Bits 4 und 3 in Abb. 5 machen dies deutlich: Bei beiden handelt es sich um 0-Signale; aber während die Low-Phase bei Bit 4 eine Länge von 1,188 µs hat, dauert die Low-Phase bei Bit 3 etwas länger, nämlich 1,313 µs. Wie kann man sich diese Abweichung erklären? Nun, im Gegensatz zu den Bits 7, 6, 5 und 4 folgt auf das Bit 3 kein 0-Signal, sondern ein 1-Signal. Um die High-Phase für dieses 1-Signal zu erzeugen, muss das Programm über den Then-Teil in den Else-Teil springen; offensichtlich erfordert dies zusätzliche Zeit (2 Takte), und das führt zu einer entsprechenden Verlängerung der Low-Phase von Bit 3.

3 Praktische Überlegungen

Wie lange dauert es bei einem 12-Ring ungefähr, bis unser Programm alle RGB-LEDs entsprechend den vorgegebenen Farb-Arrays eingestellt worden sind? Machen wir eine grobe Abschätzung: Die einzelnen Bits benötigen im Durchschnitt etwa 1,7 µs. Die Steuerung eines einzelnen Neopixels erfordert also $24 \cdot 1{,}7\ \mu s \approx 40\ \mu s$, und für einen 12-Ring brauchen wir dann ungefähr 480 µs. Diese Signalfolge müssen wir allerdings noch mit einem Reset-Signal von mindestens 50 µs abschließen.

In rund einer halbe Millisekunde kann man also einen 12-Ring vollständig steuern. Das macht deutlich: Rasche Farbwechsel sollten kein Problem darstellen: Ob Dimmen oder hektische Stroboskop-Effekte, hier gibt es praktisch keine Grenzen, zumindest was die eigentliche

Ansteuerung des Rings angeht. Wenn man eine Choreographie plant, muss man allerdings immer bedenken, dass auch die Festlegung der Werte für die Farb-Arrays Zeit in Anspruch nimmt.

Als einfaches Beispiel soll das folgende Programm dienen. Hier wird lediglich eine Dimmer-Funktion für die blauen LEDs erzeugt. Dazu wird in einer Endlos-Schleife der Farbwert mit Hilfe der Byte-Variablen *D* Schritt für Schritt um 5 erhöht; die blauen LEDs leuchten dann immer heller, bis es zu einem Overflow bei D kommt. Der Dimmvorgang beginnt von Neuem.

```
D = 0

Do
  Waitms 20             'Dimm-Geschwindigkeit
' Farb-Arrays erstellen:
  For N = 1 To Anzahl_neopixel
    G(n) = 0
    R(n) = 0
    B(n) = D
  Next N
  D = D + 5             'Farbwert inkrementieren
' Neopixel steuern:
  Portb.2 = 0           'Reset
  Waitus 60
  For N = 1 To Anzahl_neopixel
  '... (s. o.)
  Next N
Loop
```

4 Geht es auch kürzer?

Wir haben bereits darauf hingewiesen, dass das bislang benutzte Programm recht lang (und auch wenig elegant) ist, weil hier die Befehlsfolgen zur Erzeugung der Signale für jedes Bit einzeln hingeschrieben werden. Allerdings haben wir auch festgestellt, dass bei diesem Programm die Low-Phasen noch deutlich unter der Grenze von ca. 5 µs liegen. Vielleicht gelingt es ja doch, die vielen Befehlsfolgen für die einzelnen Bitnummern durch eine Zählschleife mit einer einzigen Befehlsfolge zu ersetzen. Da variable Bitnummern - wie bereits erwähnt - aber recht zeitaufwändig sind, greifen wir auf die einzelnen Bits eine Farbwerts mit einem Trick zu:

```
  Merke = G(n)
  For I = 7 To 0 Step -1
    If Merke.7 = 0 Then
      Portb.2 = 1 : Nop : Nop : Nop : Nop : Portb.2 = 0
    Else
      Portb.2 = 1 : Nop : Nop : Nop : Nop : Nop : Nop : Nop : Nop
      Portb.2 = 0
    End If
    Shift Merke, left
  Next I
```

Der Programmabschnitt funktioniert folgendermaßen: Zunächst speichern wir den Farbwert G(n) in einer Variablen *Merke*. In der Zählschleife greifen wir für den Vergleich nun immer auf das höchstwertige Bit (also das Bit mit der Bitnummer 7) von der Variablen *Merke* zu, führen am Ende jeden Schleifendurchlaufs aber bei der Variablen *Merke* einen Links-Shift durch. Am Anfang des ersten Schleifendurchlaufs steht in *Merke*.7 das Bit 7 von G(n), am Anfang des nächsten Schleifendurchlaufs steht in *Merke*.7 das Bit 6 von G(n), usw. So werden nacheinander alle Bits von G(n) der Reihe nach isoliert und entsprechend in Signale umgesetzt.

Natürlich müssen wir jetzt davon ausgehen, dass die Low-Phasen länger sein werden als bei dem Programm aus Abschnitt 2; es muss nämlich jetzt zusätzlich bei jedem Schleifendurchlauf der Schleifenindex dekrementiert, die Schleifenbedingung überprüft und ein Sprung durchgeführt werden. Immerhin haben wir es aber vermieden, mit einer variablen Bitnummer auf die einzelnen Bits von G(n) zuzugreifen.

Mit einem Logik-Analysator habe ich die neue Programmversion getestet (Abb. 6). Hier können wir erkennen, dass die Low-Phasen wie erwartet deutlich länger sind als in der Version aus Abschnitt 2, aber sie bleiben alle unter der kritischen Grenze von 5,0 µs.

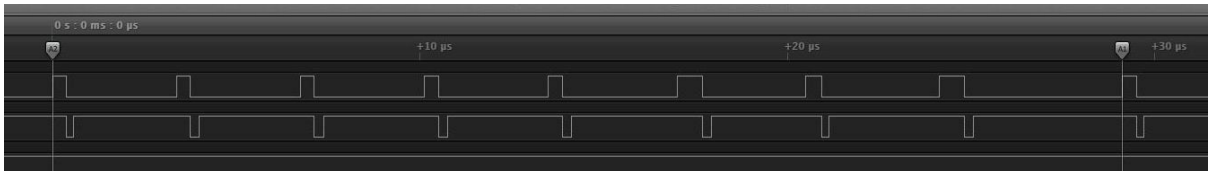


Abbildung 6

Nach diesen Erfahrungen habe ich es dann doch einmal mit einer Schleife mit variablen Bitnummern versucht. Der Logikanalysator zeigt, dass nun tatsächlich auch Low-Phasen auftauchen, die länger als 5,0 µs dauern. Überraschenderweise ließ sich mein Neopixel-Ring mit 12 RGB-LEDs auch mit diesem Programm steuern!

Ich denke, dass die Version 2 einen guten Kompromiss zwischen der Programmlänge auf der einen Seite sowie einer raschen und sicheren Übertragung auf der anderen Seite darstellt.