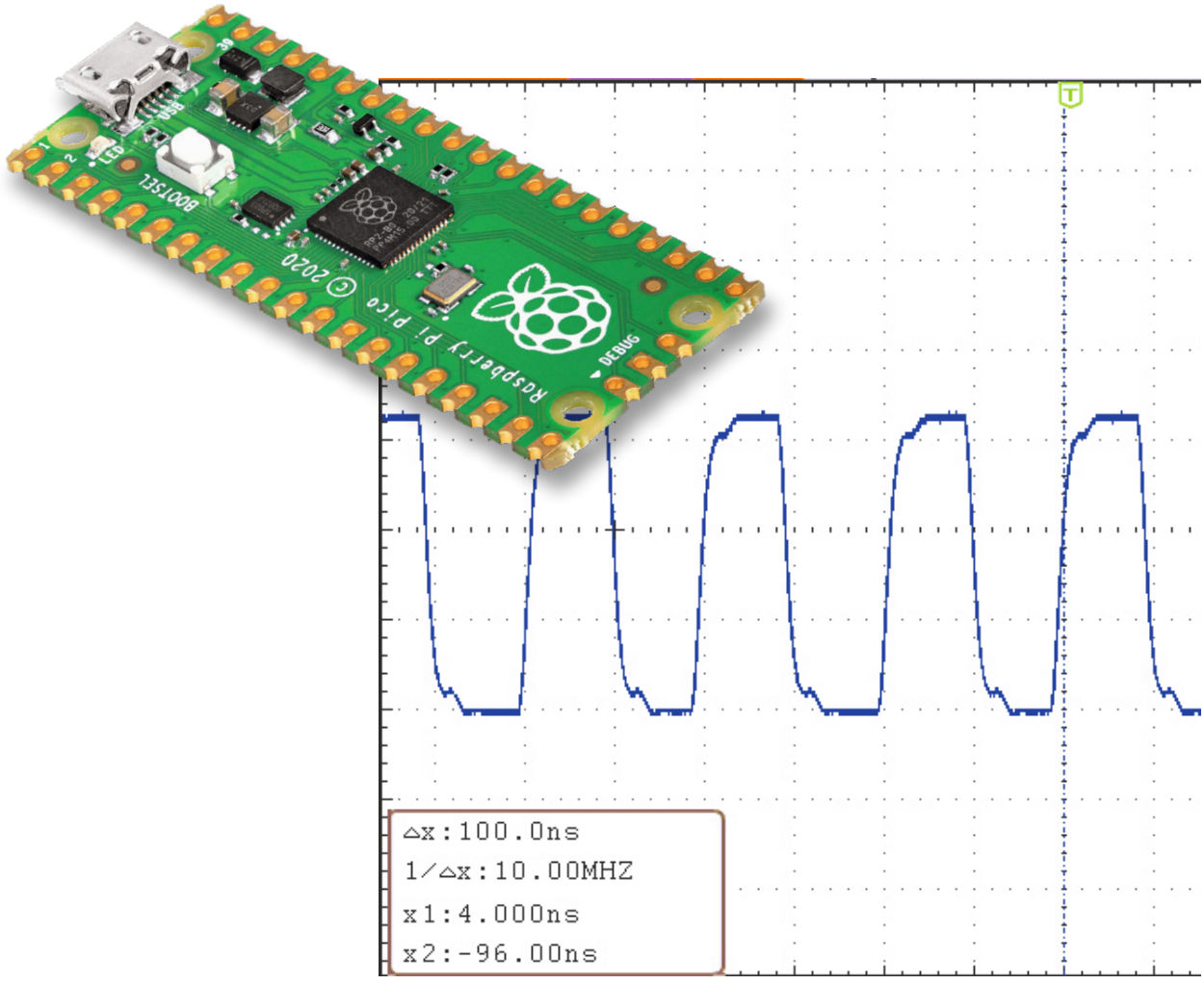
Raspberry Pi Pico

Pico State Machines

Teil 3

Eines der herausragenden Leistungsmerkmale des Raspberry Pi Pico sind die programmierbaren IO-Einheiten (PIOs). Diese ermöglichen es, fehlende Schnittstellen mit Echtzeitanforderungen zu implementieren. Normalerweise werden derartige Interfaces als controller-interne Hardware-Einheiten ausgeführt. Häufig werden hierfür auch CPLDs (Complex Programmable Logic Devices) oder FPGAs (Field Programmable Gate Arrays) eingesetzt. Für viele Anwendungen sind aber die neuen PIOs im RP2040 dafür ausreichend. Einige Projekte können damit deutlich kostengünstiger umgesetzt werden als mit den bekannten Bausteinen der großen Hersteller. Zudem sind die PIOs sogar über das Hauptprogramm des Pico rekonfigurierbar. Ein gewisser Nachteil ist allerdings, dass die PIOs im Vergleich zu ihren Konkurrenten in ihren Möglichkeiten etwas limitiert sind.



Universelle State Machines

Betrachtet man den Pico genauer, zeigt sich schnell, dass es sich nicht in jeder Hinsicht um einen vollkommen konventionellen Controller handelt. Bemerkenswert ist vor allem die Unterstützung der PIO-Funktionen. Dieses Leistungsmerkmal war bislang bei Controllern kaum zu finden. Die meisten Chips unterstützen verschiedene Kommunikationsprotokolle wie I2C und SPI. Auch der RP2040 ist mit internen UARTs, SPI- und I2C-Controllern ausgestattet. Dadurch kann der Pico bereits problemlos mit einer Vielzahl von gängigen Peripheriegeräten kommunizieren. Allerdings treten in der Praxis häufig Anwendungsfälle auf, die ältere oder proprietäre Protokolle erfordern. Zusätzliche Schnittstellen, wie etwa mehrere SPI-Busse an einem einzigen Controller können schnell zum Problem werden. Hier wird die PIO-Unterstützung des RP2040 schnell zum Retter in der Not.

Die PIOs werden über insgesamt acht State Machines programmiert. Diese können ihr Programm mit bis zu 133 MHz (bzw. 125 MHz auf dem Raspberry Pi Pico Board) abarbeiten. Es gibt nur neun Befehle, die dafür aber mit jeweils einem einzigen Systemtakt abgearbeitet werden. Logik- und Arithmetikbefehle fehlen praktisch komplett. Zudem hat jede PIO nur einen knapp bemessenen Programmspeicher von 32 Worten. Diese Einschränkungen werden eventuell zu weiteren Chip-Varianten führen.

Dennoch sind die programmierbaren IO-Einheiten gut geeignet, um fehlende Schnittstellen zu implementieren. Der Pico verfügt neben den gängigen seriellen Interfaces, von denen der RP2040 meist sogar jeweils zwei zur Verfügung stellt, noch über I²S (Audio), SDIO (SD-Card). Dazu werden auch verschiedene Beispielprojekte vorgestellt. Ein gewisser Nachteil ergibt sich jedoch aus der Tatsache, dass die Ports des Pico nicht 5-V-kompatibel sind.

Die PIOs werden über ihre State Machines programmiert: Es gibt zwei PIO-Blöcke mit je vier State Machines. Diese teilen sich den 32-Worte-Programmspeicher. Die State Machines können jederzeit gestartet und gestoppt werden. Da das zugehörige Programm über den Hauptprozessor neu geladen werden kann, ist es möglich, auf den gleichen Pins mehrere Protokolle anzubieten. Bei Bedarf können diese auch im laufenden Betrieb gewechselt werden.

Der Befehlssatz ist sehr kompakt ausgefallen. Da alle Befehle in nur einem Taktzyklus abgearbeitet werden, sind jedoch auch sehr schnelle Schnittstellen implementierbar. Die PIO-Programme sind meist sehr kurz, und bereits wenige Befehle können ein vollwertiges Programm darstellen.

Jede State Machine verfügt über zwei Universalregister (Bild 1). Dazu kommt noch je ein Shiftregister und ein FiFo (First-in First-out) mit vier Worten für IN und OUT bzw. ein einziges FiFo-Register mit acht Worten für die häufig auftretenden Anwendungen mit nur einer Datenrichtung.

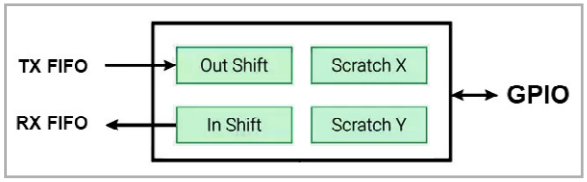


Bild 1: Vereinfachte Registerstruktur einer einzelnen PIO-State-Machine

Effiziente Programmierung

Dieser Beitrag kann keine vollständige Einführung in die Programmierung auf Maschinenebene bieten. Dennoch sollen im Folgenden die wichtigsten Grundlagen und Anweisungen vorgestellt werden. Danach folgen dann einige praktische Anwendungsbeispiele, die die Möglichkeiten der State Machines demonstrieren.

Prinzipiell ist eine PIO-Instanz mit einem eigenen kleinen Prozessor vergleichbar, der Codes völlig getrennt vom Hauptcontroller ausführen kann. Was also früher durch sogenanntes „Bitbanging" von Protokollen erreicht wurde und entsprechend CPU-Zyklen verbrauchte, kann nun über die PIO-Programmierung unabhängig vom Hauptrechnerkern erledigt werden.

Beim RP2040 umfasst jede PIO-Instanz vier Zustandsmaschinen (Bild 2), die jeweils eigene Befehle ausführen können, welche im gemeinsamen Befehlsspeicher gespeichert sind. Dieser Speicher kann 32 Befehle aufnehmen, und jeder Zustandsautomat kann jeden dieser Befehle verwenden. Ein Zustandsautomat kann alle GPIO-Pins auf dem Pico steuern. Dies wird über die IO-Mapping-Einheit erreicht.

Die State Machines eines PIO-Blocks können über Statusbits (IRQ Set, Clear, Status) untereinander kommunizieren und sich so synchronisieren oder Interrupts auslösen. Die Befehle können auf alle GPIO-Pins zugreifen, diese aber zumeist nicht explizit adressieren. Stattdessen werden die State Machines bei der Initialisierung auf GPIO-Bereiche gemappt. Dadurch kann z. B. das gleiche Programm von mehreren State Machines abgearbeitet werden, welchen jeweils nur andere Port-Pins zugewiesen sind. Pro State Machine gibt es vier solcher Mappings für die Befehle IN, OUT, SET (der Befehl MOV übernimmt je nach Quelle oder Ziel das IN- oder OUT-Mapping) und für „Side-Set" (s. u.).

Die Befehle haben implizite, feste Quellen, Ziele oder Datenbreiten. Die implizite Datenbreite wird beim Mapping eingestellt. Die insgesamt neun Befehle sind an klassische Assembler-Anweisungen angelehnt:

1. JMP springt zu einer festgelegten Programmstelle
2. WAIT wartet auf eine Bedingung
3. IN liest Bits von IO-Ports
4. OUT schreibt Bits in IO-Ports
5. PUSH schreibt das Input-Shift-Register ISR in die Empfangs-FiFo
6. PULL liest Daten aus der Sende-FiFo in das Output-Shift-Register
7. MOV kopiert Daten von einer Quelle zu einem Ziel
8. IRQ setzt/löscht ein PIO-IRQ-Flag (Statusbit)
9. SET kopiert Werte in X- oder Y-Register

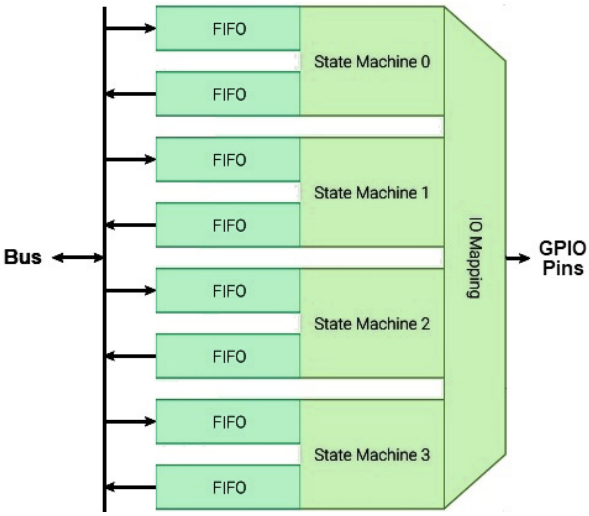


Bild 2: Zustandsmaschinen mit I/O-Mapping

Allerdings muss beachtet werden, dass die Befehle im Vergleich zu vielen Assembler-Anweisungen oft unerwartete Einschränkungen aufweisen oder zusätzliche Möglichkeiten bieten. Jeder der Befehle kann prinzipiell drei Aufgaben gleichzeitig erledigen:

1. Den eigentlichen Befehl ausführen
2. Über Side-Set bis zu 5 GPIOs ändern oder deren Richtung umschalten
3. Zusätzlich implizit bis zu 31 Wartetakte anhängen

Ein PIO-Assemblerbefehl sieht dann beispielsweise so aus:

```
mov pins, x side 0b01 [2]
```

„mov pins, x" ist der eigentliche Befehl (kopiere Register X zu den eingestellten GPIOs), „side 0b01" setzt zwei GPIOs laut Side-Set-Mapping, „[2]" definiert dann noch zwei Wartetakte.

Die Möglichkeit, Out-Pins mit Side-Set zu setzen, führt zu schnellen und effizienten Programmen. Deshalb lassen sich trotz des geringen Speicherumfangs auch komplexe Anwendungen umsetzen.

Die State Machines greifen auf die FiFos über ihr In- und Out-Shift-Register (ISR und OSR) zu. Die Steuerung des FiFo-Speichers erfolgt über die Befehle PUSH und PULL. Die Daten der Shift-Register werden mit IN und OUT auf die GPIOs ausgegeben bzw. eingelesen. Alternativ können FiFos auch über direkten Speicherzugriff (Direct Memory Access – DMA) beschrieben oder gelesen werden.

Die PIO-State-Machines sind so schnell, dass sogar Hochfrequenzanwendungen bis in den Megahertz-Bereich hinein möglich werden. So sind z. B. schnelle UARTs problemlos implementierbar. Aber auch Videoschnittstellen wie etwa VGA oder sogar DVI können so realisiert werden. So könnte der Pico sogar – ohne einen PC – direkt an einen VGA-Monitor angeschlossen werden. Selbst eigene Schnittstellenvarianten können mit PIO-Unterstützung von Grund auf neu erstellt werden. Prinzipiell wären sogar völlig neue Schnittstellen denkbar.

Jeder PIO-Block hat einen Befehlsspeicher mit 32 Befehlen. Vier Lese-Ports ermöglichen den gleichzeitigen Zugriff auf alle State-Machines.
Jeder State Machine verfügt über:

- Zwei 32-Bit-Schieberegister (OSR out, ISR in)
- Zwei Scratch-Register/temporäre Register (X, Y)
- Zwei FIFO-Datenpuffer (TX/RX)
- Einen Clock-Teiler
- GPIO-Zuordnung (Eingang, Ausgang, Set, Side-Set)
- Eine Schnittstelle für direkten Speicherzugriff (DMA)
- Einen Interrupt

PIO-Praxis

Um mit den State Machines in Python arbeiten zu können, müssen zunächst die entsprechenden Libraries importiert werden. Mann kann entweder die komplette Lib

```
import machine, rp2
```

oder nur die aktuell erforderlichen Module

```
from rp2 import PIO, StateMachine, asm_pi
```

importieren.

Um eine State Machine zu starten, benötigt man diese Parameter:

- Nummer der State Machine: 0 bis 7 zur Auswahl einer der acht State Machines
- Name des Programms (z. B. „blink")
- Arbeitsfrequenz der State Machine in Hz: Wertebereich 2000 bis 50000000
- GPIO-Basis-Pin

Beispiel:

```
sm = rp2.StateMachine(0, blink, freq=2000, ↩
set_base=machine.Pin(0))
```

Hier wird die erste State Machine (Nr. 0) angesprochen, und die Routine „blink" soll ausgeführt werden. Die State Machine soll dabei mit 2 kHz laufen und den Pin 0 ansprechen.

Im nächsten Schritt kann nun das Programm „blink_loop.py" erstellt werden (alle Programme sind im Download-Paket [1] zu diesem Artikel enthalten):

```
import machine, rp2

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink():
    label("loop_start")
    set(pins, 1)
    set(pins, 0)
    jmp("loop_start")

sm = rp2.StateMachine(0, blink, freq=2000, ↩
set_base=machine.Pin(0))
sm.active(1)
```

Das Ergebnis auf dem Oszilloskop zeigt die bekannten Tücken der Assembler-Programmierung (Bild 3). Anstelle eines sauberen, symmetrischen Signals erscheint eine unsymmetrische Rechteckfunktion. Die „Low"-Phasen sind doppelt so lange wie die „High"-Pulse. Dies liegt daran, dass der Jump-Befehl ebenfalls einen Maschinenzyklus beansprucht.

Hier bieten die beiden Anweisungen wrap_target() und wrap() eine elegante Lösung. Das wrap_target-Label ist eine spezielle Methode, bei der nicht explizit gesprungen wird. Die Anweisungen zwischen diesen beiden Befehlen werden so ausgeführt, als stünden sie direkt sequenziell hintereinander. Dies wird durch direktes Zurücksetzen des Programmzählers erreicht. Im Gegensatz zu einer expliziten Sprunganweisung erfordert diese Konstruktion keinen eigenen Taktzyklus.

Mit

```
@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink():
    wrap_target()
    set(pins, 1)
    set(pins, 0)
    wrap()
```

wird so eine exakte 50-Prozent-Rechteckschwingung erzeugt (blink_wrap.py). Die Funktion set() setzt in beiden Fällen den Pin-Status. Dabei wird mit „1" der Pin auf high gesetzt und dann mit „0" auf low. Der Code liefert nun das gewünschte Resultat an Pin 0 des Pico Boards (Bild 4).

Die Frequenz des Signals ergibt sich im zweiten Fall daraus, dass die Schleife genau zwei Anweisungen enthält. Jede Anweisung erfordert genau einen Maschinentakt. Dieser umfasst bei der gewählten Frequenz von 2 kHz exakt 500 µs. Somit beträgt die Ausgabefrequenz:

f = 1 / T = 1 / (2 * 500 µs) = 1 kHz

Falls kein Oszilloskop oder Frequenzzähler zur Verfügung steht, kann man die Ausgabefrequenz auch weiter reduzieren. Hierfür sind nop()-Anweisungen (no operation) in die Schleife einzufügen. Um die Frequenz auf 10 Hz zu reduzieren, muss die Periodendauer auf 100 ms gestreckt werden. Das entspricht 200 Maschinenzyklen. Der folgende Code (blink_10_Hz.py) erfüllt die Aufgabe:

```
import machine, rp2

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink():
    wrap_target()
    set(pins, 1)  [19]
    nop()         [19]
    nop()         [19]
    nop()         [19]
    nop()         [19]
    set(pins, 0)  [19]
    nop()         [19]
    nop()         [19]
    nop()         [19]
    nop()         [19]
    wrap()

sm = rp2.StateMachine(0, blink, freq=2000,↩
set_base=machine.Pin(25))
sm.active(1)
```

Dabei wurde von der Möglichkeit Gebrauch gemacht, den verwendeten Befehl durch Anhängen einer Zahl in eckigen Klammern mit zusätzlichen Leerlauf-Maschinenzyklen zu erweitern (side-set-mappin). Die Schleife enthält zehn Befehle, jeder Befehl führt einen Maschinenzyklus plus 19 weitere ohne Wirkung aus, somit also:

f = 1 / (10*(1+19)*500 µs) =
1 / (200*500 µs) = 1 / 100 ms = 10 Hz

Damit wurde die Port-Frequenz soweit heruntergeteilt, dass das Blinken einer angeschlossene LED für das menschliche Auge sichtbar wird. Durch Auswahl des Port 25 blinkt nun die Onboard-LED des Pico mit 10 Hz.

PWM durch Interferenzeffekt

Der große Vorteil von acht eigenständigen State Machines liegt natürlich darin, dass diese vollkommen unabhängig voneinander eingesetzt werden können. So lassen sich durch die Kombination mehrerer Einheiten interessante Effekte erzeugen. Eine Möglichkeit ist die Erzeugung eines pulsweitenmodulierten Signals (PWM) über einen Interferenzeffekt. Dabei steuern zwei Oszillatoren mit geringfügig unterschiedlicher Frequenz einen einzelnen Pin. Die Ausschaltfrequenz wird auf exakt 2000 Hz eingestellt, die Einschaltfrequenz dagegen auf 2001 Hz (PWM_interference.py):

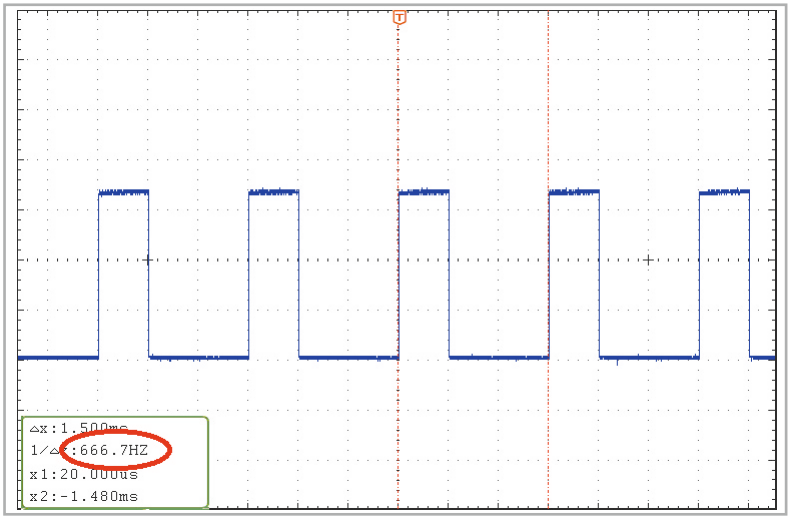


Bild 3: Unsymmetrisches Rechtecksignal an Port 0

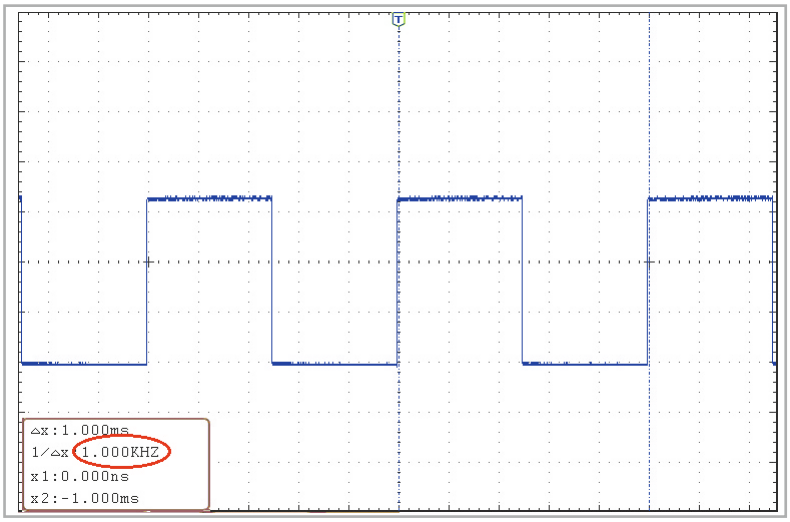


Bild 4: 1 kHz-Rechtecksignal an Port 0

```
from rp2 import PIO, StateMachine, asm_pio
from machine import Pin
import time

@asm_pio(set_init=PIO.OUT_LOW)
def led_off():
    set(pins, 0) [3]

@asm_pio(set_init=PIO.OUT_LOW)
def led_on():
    set(pins, 1) [3]

sm1 = StateMachine(1, led_off, freq=2000, set_base=Pin(25))
sm2 = StateMachine(2, led_on,  freq=2001, set_base=Pin(25))

sm1.active(1)
sm2.active(1)
```

Die Grundfrequenz der PWM ergibt sich aufgrund der zusätzlichen Wartezyklen zu

f_0 = 500 Hz

Da für eine volle PWM-Periode 2000 Zyklen durchlaufen werden, beträgt die Periodendauer der PWM:

T_PWM = 1/500 Hz * 2000 = 4 s

Die LED wird also innerhalb von 4 Sekunden zunehmend heller, bis sie wieder erlischt.

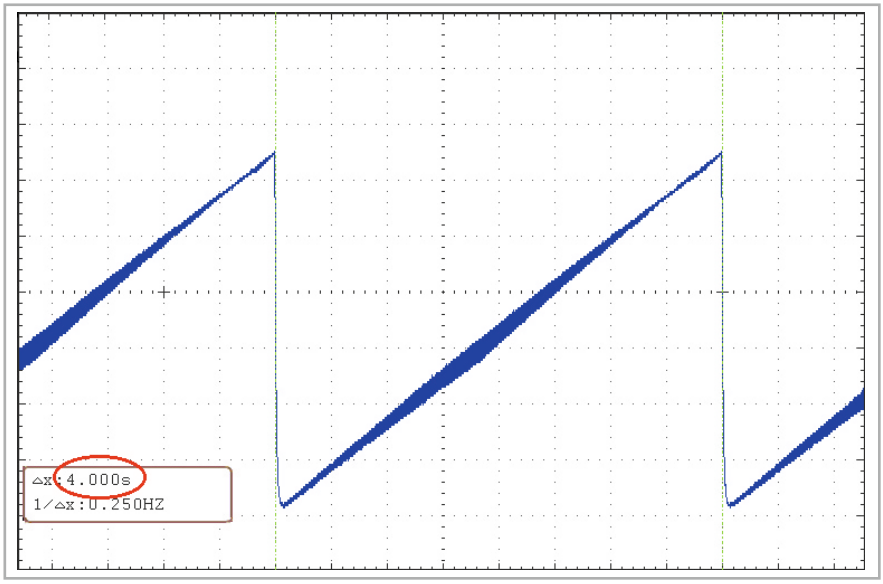


Bild 5: Quasi-analoges PWM-Signal

Bild 5 zeigt das tiefpass-gefilterte quasi-analoge PWM-Signal. Die Reste der Grundschwingung sind in der ansteigenden Flanke noch deutlich zu erkennen.

Schneller Signalgenerator bis 62,5 MHz

Das Blinken eine LED mit einigen Hertz oder auch eine PWM-Steuerung mit einigen Kilohertz kann nicht wirklich als „schnelles" Signal durchgehen. Nachdem das Funktionsprinzip der PIOs nun aber klar sein sollte, kann man sich auch an höhere Frequenzen wagen.

Die State Machines können mit Frequenzen von bis zu 125 MHz getaktet werden. Da zum Schalten mindestens zwei Takte (High on – Low on) erforderlich sind, können mit dem Pico Frequenzen von sehr beachtlichen 62,5 MHz erzeugt werden.

Prinzipiell muss dazu lediglich der Frequenzgenerator mit einer entsprechend hohen Frequenz betrieben werden (HF_generator_10 MHz.py):

```
import machine, rp2

@rp2.asm_pio(set_init=rp2.PIO.OUT_LOW)
def blink():
    wrap_target()
    set(pins, 1)
    set(pins, 0)
    wrap()

f=10_000_000

sm=rp2.StateMachine(0,blink,freq=2*f,set_base=machine.Pin(0))
sm.active(1)
```

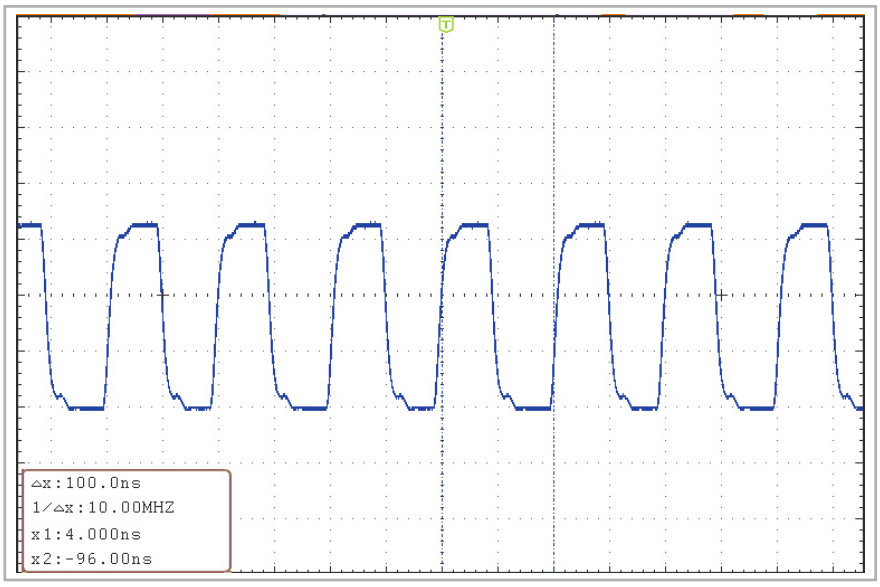


Bild 6: Pico als HF-Generator für 10 MHz

Die Taktfrequenz f wurde nun explizit definiert. Der Default-Wert ist auf 10 MHz gesetzt. Zu beachten ist, dass beim Start der State Machine die Taktfrequenz 2*f betragen muss, da je zwei Maschinenzyklen pro Periode erforderlich sind und dadurch die Frequenz praktisch halbiert wird. Bild 6 und Bild 7 zeigen das 10-MHz-Signal im Vergleich mit der maximal möglichen Frequenz von 62,5 MHz.

Während bei 10 MHz noch ein relativ klares Rechtecksignal sichtbar wird, zeigen sich bei über 60 MHz bereits die Grenzen der verwendeten Messtechnik (Oszilloskop-Bandbreite 100 MHz bei 1 GS/s). Inwieweit die Signalverzerrung (Anstiegszeit ca. 3 ns) auf die Messtechnik, den Aufbau oder den Pico selbst zurückzuführen ist, könnte nur mit erheblichem Mehraufwand und hochpreisigen Spezialmessgeräten sicher festgestellt werden.

Die Messungen zeigen jedoch, dass der Pico als sehr preisgünstiger HF-Generator in vielen Anwendungen nutzbringend eingesetzt werden kann. Der nächste Abschnitt liefert einige Beispiele dazu.

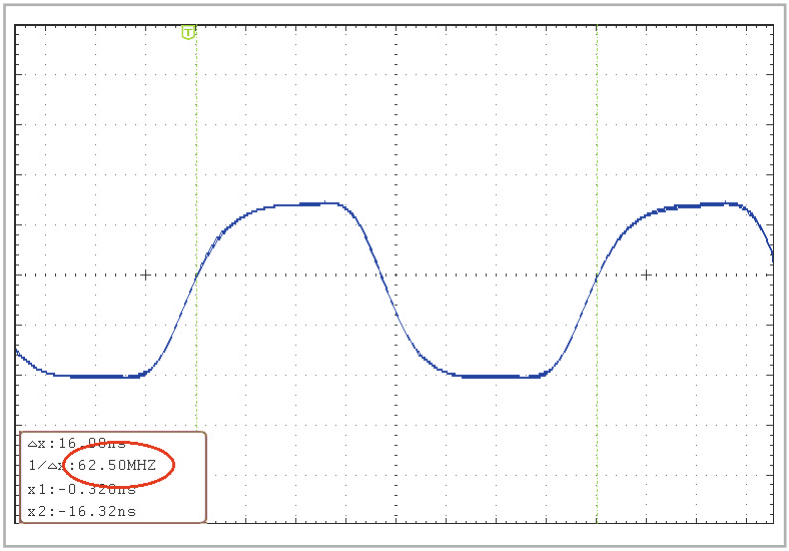


Bild 7: Bei 62,5 MHz zeigen sich bereits Bandbreitenprobleme.

Professionelle HF-Messtechnik für kleines Geld

Zunächst kann man das einfache HF-„Blinkprogramm“ mit einem simplen User-Interface versehen:

```
from rp2 import PIO, StateMachine, asm_pio
from machine import Pin
import time

@asm_pio(set_init=PIO.OUT_LOW)
def blink():
    set(pins, 0)
    set(pins, 1)

f=1_000 # initial frequency in kHz
pin=0

sm=StateMachine(0,blink,freq=2*f,set_base=Pin(pin))
sm.active(1)


while(True):
    a = input("Frequency in kHz (u: +10% - d: -10%): ")
    try:
        x=float(a)
        f=int(x*1000)
        sm.active(0)
        sm = StateMachine(2, blink, freq=f*2, set_base=Pin(pin))
        sm.active(1)
        print(str(f)+" Hz")
    except:
        print(a)
        if a=='u':
            f=int(f*1.1)
        if a=='d':
            f=int(f*0.9)
        sm.active(0)
        sm = StateMachine(2, blink, freq=f*2, set_base=Pin(pin))
        sm.active(1)
        print(str(f)+" Hz")
    time.sleep(0.1)
```

Damit lässt sich die gewünschte Frequenz sehr einfach über ein Serielles Terminal einstellen, ohne dass immer das Programm geändert werden muss. Zudem wurde eine Feineinstellung implementiert, die es erlaubt, die Frequenz über die Tasten u (für „up“) und d (für „down“) um jeweils 10 % zu justieren, ohne dass immer ein neuer Wert eingetippt werden muss. So entsteht ein Signalgenerator, der sich komfortabel und präzise auf Frequenzen zwischen 1 kHz und 62,5 MHz einstellen lässt.

Sweep-Generator

Man kann sogar noch einen Schritt weitergehen und einen Sweep-Generator realisieren. Damit lassen sich dann sogar beispielsweise die Resonanzkurven von Schwingkreisen bis in den HF-Bereich hinein vermessen.

Das Python-Programm dazu findet sich im Download-Paket [1] (s. HF_sweep_generator.py). Nach der Eingabe der Start- und Stoppfrequenz sowie der Schrittweite führt der Pico einen Frequenz-Scan im angegebenen Bereich aus. Wird diese Frequenz auf einen Schwingkreis gegeben, kann man mithilfe eines Oszilloskops die zugehörige Resonanzkurve aufnehmen. Mit einem Kondensator vom 15,5 nF und eine Spule mit 390 µH ergibt sich eine Resonanzfrequenz von 64,7 kHz. Das zugehörige Schaltbild für einen Parallel-Resonanzkreis zeigt Bild 8.

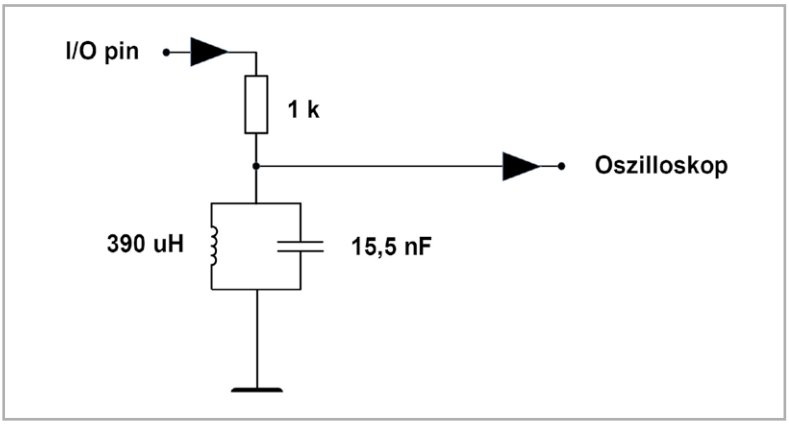


Bild 8: Vermessung eines Parallel-Resonanzkreises

Der Sweep-Generator wurde auf folgende Werte eingestellt:

Startfrequenz: 10 kHz
Stopfrequenz: 150 kHz
Step: 1 kHz

Die gemessene Resonanzkurve ist in Bild 9 dargestellt. Wie erwartet zeigt sich das Resonanzmaximum des Kreises bei etwas weniger als 65 kHz. Mit ein paar Programmzeilen wird der Pico damit zu einem wertvollen Hilfsmittel in jedem Elektronik-Labor.

Für das hier vorgestellte Anwendungsbeispiel wurde ein relativ niedriger Frequenzbereich gewählt. Der Pico ist aber durchaus in der Lage, auch Filter im MHz-Bereich zu vermessen. Allerdings müssen für aussagekräftige Resultate dann auch HF-taugliche Messaufbauten verwendet werden.

Neopixel-Steuerung: bunte LED-Ketten und -Ringe

Neben den Anwendungen im HF-Elektronik-Labor können die State Machines des Pico auch für alltäglichere Anwendungen eingesetzt werden. Mit den Neopixels-LEDs (Typbezeichnung WS2812B) stehen preisgünstige und weitverbreitete „Smart-LEDs“ zur Verfügung. Im Prototypenadaptersatz „digital“ (PAD4) sind beispielsweise acht dieser LEDs in der bedrahteten 8-mm-Bauform enthalten. Sie sind aber auch in vielen anderen Varianten als einzelne SMD-Bauteile, LED-Streifen oder Ringe etc. erhältlich. Das transparente Gehäuse dieser Bauelemente enthält rote, grüne und blaue LED-Chips zusammen mit einem Controller.

Ein spezielles Protokoll gestattet es, eine praktisch beliebig lange Reihe von LEDs mit nur einer Signalleitung anzusteuern. So wird auch für die Vollfarben-Ansteuerung von 100 oder mehr LEDs lediglich ein einziger Prozessor-Pin benötigt. Allerdings hat dieses Verfahren bei den Neopixel-LEDs auch zwei Nachteile:

- Das zur Ansteuerung benötigte Protokoll ist zeitintensiv
- Die LEDs sind nicht gamma-korrigiert

Auch hier können die State Machines des Pico zur Lösung beitragen. Durch das Implementieren des Steuerprotokolls auf einem Zustandsautomaten wird der Hauptprozessor entlastet. Werden sogar zwei State Machines eingesetzt, können durch sogenanntes „Dithering“ Pseudo-Helligkeitsstufen erzeugt werden. Das heißt, dass durch schnelles Umschalten zwischen verschiedenen Helligkeitsstufen sehr flüssige und vollkommen flimmerfreie Farbübergänge entstehen.

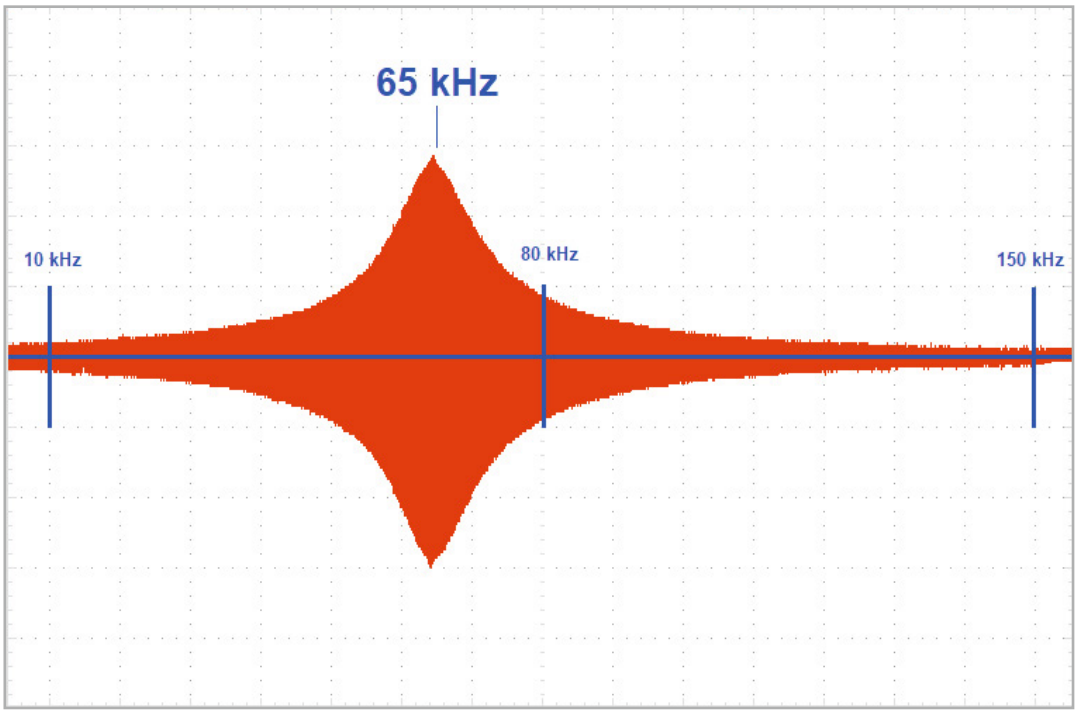


Bild 9: Resonanzkurve eines HF-Schwingkreises

Für die Ansteuerung der WS2812B-LEDs kann das folgende Beispiel-Programm verwendet werden:

```
import array, time, rp2
from machine import Pin

PIN_NUM=22
NUMBER_of_LEDS=3

@rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT, autopull=True, pull_thresh=24)
def ws2812():
    T1 = 2
    T2 = 5
    T3 = 3
    wrap_target()
    label("bitloop")
    out(x, 1)               .side(0)    [T3-1]
    jmp(not_x, "do_zero")   .side(1)    [T1-1]
    jmp("bitloop")          .side(1)    [T2-1]
    label("do_zero")
    nop()                   .side(0)    [T2-1]
    wrap()

sm=rp2.StateMachine(0, ws2812, freq=8_000_000, sideset_base=Pin(PIN_NUM))
sm.active(1)

ar = array.array("I", [0 for _ in range(NUMBER_of_LEDS)])

def pixels_show():
    dimmer_ar = array.array("I", [0 for _ in range(NUMBER_of_LEDS)])
    for i,c in enumerate(ar):
        r = int(((c >> 8) & 0xFF))
        g = int(((c >> 16) & 0xFF))
        b = int((c & 0xFF))
        dimmer_ar[i] = (g<<16) + (r<<8) + b
    sm.put(dimmer_ar, 8)
    time.sleep_ms(10)

def pixels_set(i, color):
    ar[i] = (color[1]<<16) + (color[0]<<8) + color[2]

pixels_set(0, (255, 0, 0))
pixels_set(1, (0, 250, 0))
pixels_set(2, (0, 0, 255))
pixels_show()
```

Die Funktion WS2812 sorgt dabei für die Programmierung der verwendeten State Machine. Sie erzeugt die im WS2812B-Protokoll verwendeten Impulse mit einer Frequenz von 800 kHz. Die Länge der Impulse bestimmt, ob eine 1 oder eine 0 erkannt wird. Der Code verwendet Sprünge für das Timing der einzelnen Farbbits. Die Variablen T1, T2 und T3 liefern die passenden Pulsbreiten, wobei wieder ein Taktzyklus für den Befehl selbst abgezogen werden muss.

Dann wird die State Machine mit 8 MHz am angegebenen Pin (22) gestartet. Die jeweils drei Farbwerte für jede LED werden in einem array (ar) abgespeichert. Die Funktion pixel_show() wandelt die einzelnen Farbwerte in Bitsequenzen mit jeweils 8 Bit Breite um. Mit pixel_set werden die Werte dann in das Array geschrieben. Über

```
pixels_set(0, (255, 0, 0))
```

wird also beispielsweise die erste LED mit der ersten Farbe (z. B. grün) in voller Helligkeit angesteuert. Über

```
pixels_set(1, (0, 250, 0))
```

erstrahlt die zweite LED dann z. B. in Rot usw. Neben den einzelnen Primärfarben kann so auch das gesamte Farbspektrum einschließlich der Farbe Weiß abgedeckt werden.

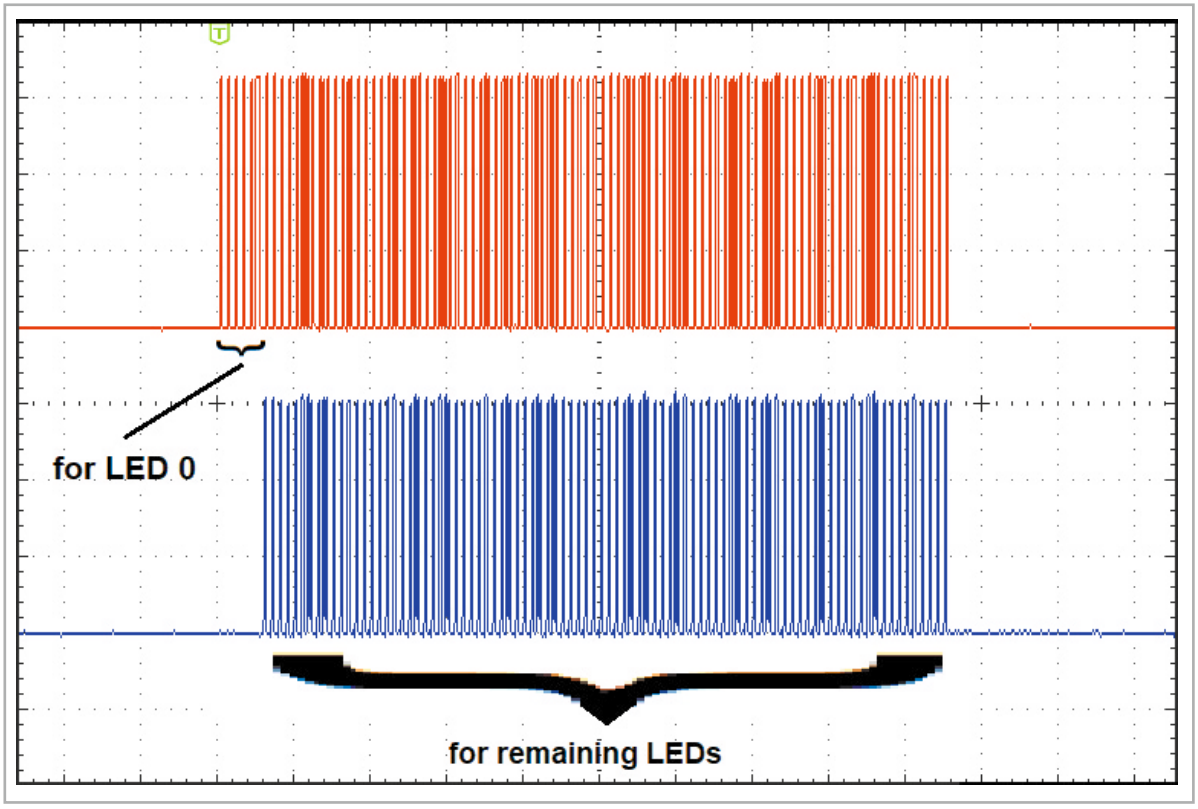


Bild 10: Jede Neopixel-LED kürzt das Datenpaket.

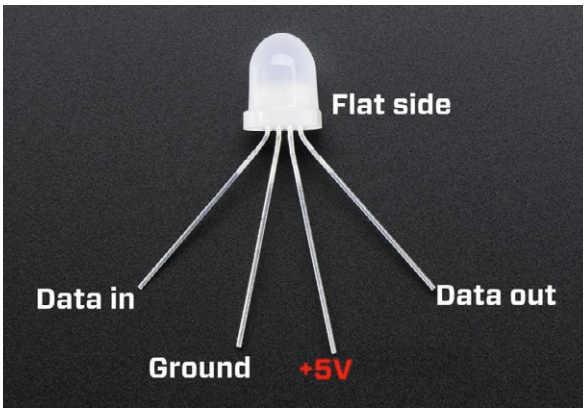


Bild 11: Pin-Belegung einer 8-mm-Neopixel-LED

Die LED-internen Controller werten das Bitmuster sequenziell aus und trennen das jeweils für sie bestimmte Bitmuster ab. Die folgenden LEDs erhalten also nur noch das entsprechend gekürzte Bitmuster (Bild 10).

Durch dieses Verfahren können die LEDs sehr einfach verschaltet werden. Über nur eine Datenleitung wird das Signal zu den LEDs übertragen. Diese verfügen über einen Eingangs-Pin (DI: Data in) und einen Ausgangs-Pin (DO: Data out) (Bild 11). Die problemlose Verkettung der LEDs zeigt Bild 12.

Die LEDs sind für Versorgungsspannungen zwischen 3 V und 5 V spezifiziert. Sie lassen sich also direkt mit der 3,3-V-Spannung des Pico betreiben.

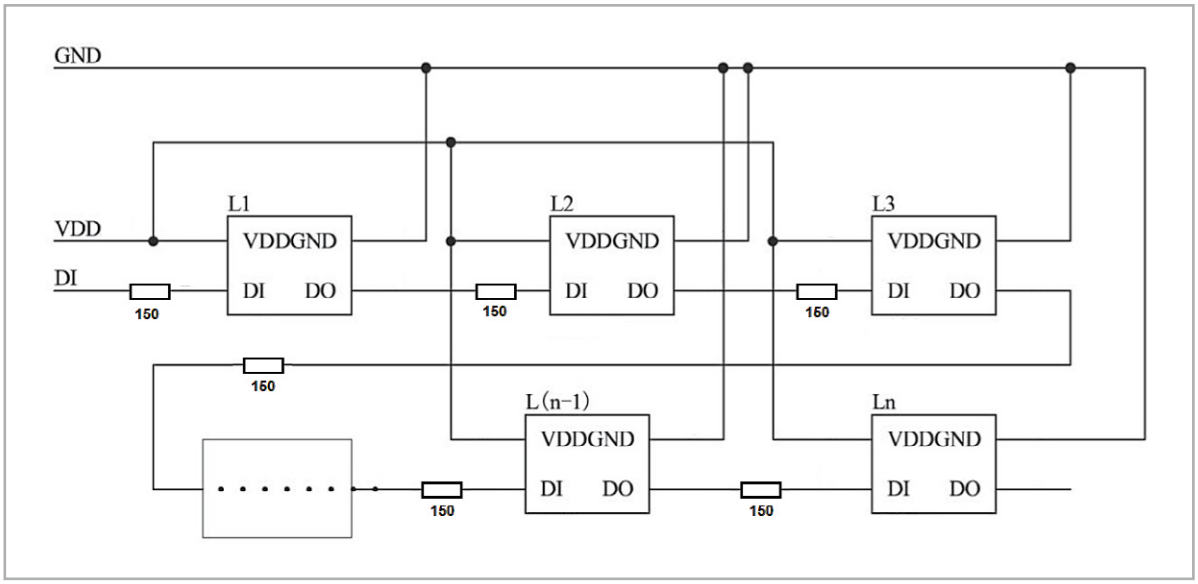
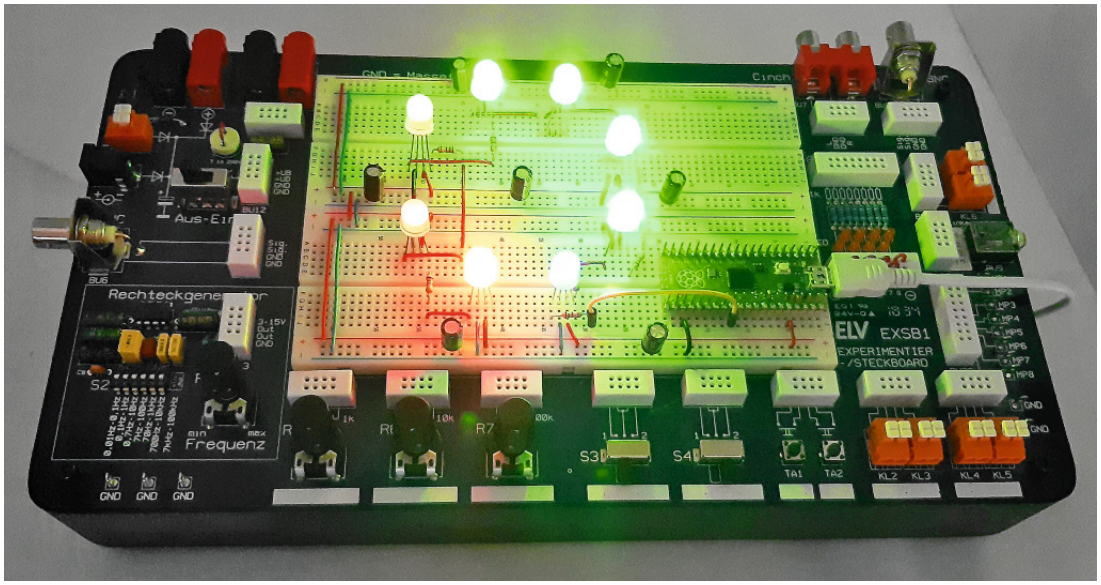


Bild 12: Schaltung zur Neopixel-Kette

Bild 13: Neopixel-Ring in Aktion



Allerdings ist zu beachten, dass die Stromaufnahme mit der Anzahl der LEDs stark ansteigt. Ab etwa neun bis zehn LEDs sollte daher eine externe Stromversorgung verwendet werden. Die 150-Ohm-Widerstände dienen zur Entkopplung der LEDs. Bei längeren Ketten sollte die Versorgungsspannung zudem durch Elkos (100 µF, 16 V) gepuffert werden. Bild 13 zeigt einen Aufbau mit acht Neopixel-LEDs auf dem EXSB-1-Aufbausystem.

Das dynamische Verhalten der Neopixel-Ansteuerung ist in einem Youtube-Video [2] zu sehen. Ein entsprechendes Beispielprogramm findet sich im Download-Paket [1].

Ausblick

In diesem Beitrag wurde gezeigt, wie die State Machines des Pico auf verschiedene Weisen eingesetzt werden können. Dabei bietet sich eine erstaunliche Breite von Einsatzmöglichkeiten. Ausgehend von einfachen LED-Blinkanwendungen über HF-Applikationen im MHz-Bereich bis hin zu einem eigenen Neopixel-Treiber reicht die Spanne. Die programmierbaren IO-Kanäle (PIO) des Pico machen den Chip so zu einem noch universelleren Werkzeug in der Hand des Anwenders.

Im nächsten Beitrag soll der Pico dann seine Leistungsfähigkeit im Bereich des Maschinellen Lernens und der Künstlichen Intelligenz (KI) demonstrieren. Natürlich kann der äußerst preisgünstige Chip nicht mit Multicore-Hochleistungsrechnern mithalten, aber einige einfache Anwendungen können doch recht beachtliche Resultate liefern. ELV

Material	Artikel-Nr.
Raspberry Pi Pico	251905
Experimentierboard EXSB1	153753
Bausatz Prototypenadapter für Steckboards PAD4, digital	155107

Weitere Infos

[1] Download-Paket: Artikel-Nr. 253236
[2] Youtube-Video zur Neopixel-Steuerung: https://www.youtube.com/watch?v=1EGP_3GKa_E

Alle Links finden Sie auch online unter: de.elv.com/elvjournal-links