Spielereien mit asm_pio + MicroPython auf dem PI PICO

Auf dem PI PICO kann man komfortabel mit MicroPython arbeiten und nebenbei kleine, taktgenau arbeitende Erweiterungen in ASM auf den PIO's hinzufügen, die unabhängig vom restlichen Programmcode ablaufen und weder Speicherplatz noch zusätzliche Rechenzeit beanspruchen.

"Just for fun" hier meine Spielereien mit asm_pio und MicroPython, in denen ich ausloten wollte, wie ich als Hobbyist im Bereich Homeautomation diese zusätzliche Hardware für mich nutzen könnte.

- Generatoren für symmetrische / asymmetrische Rechtecksignale
- Generatoren durch Ausgabe von 32-Bit-Patterns
- Timer/Counter für die Messung von Impulszeiten
- UART-Transmitter, optional mit Even/Odd Parity-Bit
- UART-Transmitter, optional mit 32-Byte Fifo-Sendebuffer
- UART-Receiver, optional mit 32-Byte Fifo-Empfangsbuffer
- Intervall-Summer
- PWM (bis zu 32 Bit) mit exponentieller Interpolation der Zwischenwerte
- IR-Remote Transmitter / Receiver (NEC-Protokoll, 38KHz)
- Taster zum Ein-/Ausschalten bzw. mit Ein-/Ausschaltverzögerung
- Inverter (Not-Gate)

Nachfolgend etwas ausführlichere Erläuterungen zu den Experimenten.

Generatoren ----------

pio_wave_sym.py
Dient dazu, ein symmetrisches Rechtecksignal durch Vorgabe der Frequenz zu erzeugen, hier sind die Dauer von Pulse und Pause gleich lang.
Die Anzahl der auszugebenden Pulse/Pause-Folgen kann festgelegt werden.

pio_wave_asym.py
Erzeugt ein asymmetrisches Rechtecksignal, bei dem die Dauer von Pulse und Pause getrrennt vorgegeben werden.
Die Anzahl der auszugebenden Pulse/Pause-Folgen muss festgelegt werden.

pio_wave_pattern_1.py
Beim Experimentieren mit "pull_thresh" und "autopull" ergab sich eine weitere Methode, um Impulsfolgen zu generieren:

Auf einem Output-Pin wird ein Bitmuster ausgegeben und mittels "autopull" immer wieder neu aus dem X-Register (in das es vorher manuell kopiert wurde) nachgeladen.
So lassen sich mit 3 asm_pio-Instruktionen Patterns als Rechteck-Signal mit Frequenzen ab 1Hz ausgeben (mit zusätzlichen Instruktionen kann man die Zeiten natürlich auch noch verlängern).
Somit für das Blinken von Leds etc. gut einsetzbar.
Die auszugegebende Impulsfolge kann durch Nachladen neuer Muster während des Programmablaufes jederzeit geändert werden.

pio_wave_pattern_led.py 1 (dauerhaft 1)
Als konkrete Anwendung hier die Ansteuerung einer LED mit unterschiedlicher Blinkfolge, die als Bitpatterns nacheinander an die Statemachine gesendet werden.

pio_wave_pattern_2.py
Die beiden vorausgegangenen Beispiele haben einen kleinen Schönheitsfehler:

Nachdem alle 32 Bits aus dem OSR auf dem Output-Pin ausgegeben sind, wird ein zusätzlicher Takt für "pull()" benötigt, um entweder das nächste Byte aus dem Fifo oder aus dem X-Register zu laden.
Das ist weniger geeignet in Situationen, in denen taktgenaue Signale gefordert sind.

Abhilfe schafft folgender Ansatz:
"pull_thresh" wird auf 31 begrenzt und im 32.Takt wird das "pull()" mit den erforderlichen, ausgleichenden "delays" ausgeführt.
Zu berücksichtigen ist dabei, dass nun der Pinstatus im 32.Bit gegenüber dem 31.Bit nicht mehr verändert werden kann.

Die Generatoren dienten z.B. dazu, zu Testzwecken die nachfolgenden Counter zu "füttern".

Timer/Counter ----------

pio_pulse_counter_32.py
Misst die Dauer zwischen der Steigenden und der Fallenden Flanke von Rechtecksignalen.
Das Ziel ist, Zeiten (= Takte) durch dekrementieren des X- bzw. Y-Registers aufzuaddieren. Denn für das direkte Addieren gibt es keine passende ASM-Instruktion.
Das jeweilige Zählregister wird z.B. mit 0xFFFF vorgeladen.
Während der Messung wird dekrementiert.
Am Ende beträgt der Messwert (0xFFFF - Registerstand).
Läuft der Zähler auf 0 über, dann wird die Messung abgebrochen und "0" ausgegeben.

Diese Variante zählt die Dauer des "High-levels" in einer Tiefe von bis zu 2**32, das Zählregister wird mit 0xFFFF_FFFF geladen.

pio_pulse_counter_16.py
Misst die Dauer von Pulse und Pause in einer Tiefe von 16 Bit und lädt die beiden Messergebnisse jeweils in ein gemeinsames Fifo-Register.
Das Hauptprogramm muss die Fifos schneller auslesen, als die Messergebnisse generiert werden - das funktioniert unter MicroPython bei einem Signal von bis zu ca. 20 KHz.

pio_pulse_counter_16_64,py
Wie der Vorgänger, aber hier arbeiten bis zu 8 Statemachines für die Aufzeichnung kollegial zusammen.
Jede Statemachine kann die Messergebnisse von bis zu 8 Takten in seinen Fifo-Registern aufnehmen, so dass insgesamt bis zu 64 Takte vermessen werden können, ohne dass die Fifos zwischenzeitlich ausgelesen werden müssen.
Ist also geeignet für schnelle, aber in der Anzahl auf 64 beschränkte Messungen.

Die Arbeitsweise:
Alle Statemachines lauschen am gleichen Input-Pin und zählen zunächst nur die Anzahl der eingehenden Impulse.
Die Statmachine 0 beginnt bereits beim ersten Takt mit der Aufzeichnung und

blockt nach 8 Takten beim "push()".
Die Statemachine 1 zählt bis 8, beginnt beim 9.Takt mit der Aufzeichnung und blockt nach de, 16. Takten.
Die Statemachine 2 zählt bis 16, beginnt beim 17.Takt mit der Aufzeichnung und blockt nach dem 24.Takt.
...
Die Statemachine 7 zählt bis 56, beginnt beim 57.Takt mit der Aufzeichnung und blockt nach dem 64.Takt.

In diesem Moment blocken alle Statemachines und weitere Aufzeichnungen sind nicht mehr möglich.
Die Statemachines müssen deaktiviert und ihre Register ausgelesen werden.
Damit liegen dann 64 Registereinträge je 32-Bit vor, die als Pulse- und Pause Längen interpretiert werden müssen.

Hier ein Beispiel:
Empfangen wurde per UART (0x55, 0x00, 0xFF, 0xAA) mit 9600 Baud

F_PIO: 30000000 Hz, resolution: 100.0 Nanosec.

```
#  00    1043 -   1043      Startbit   - 0 | Bit.0           - 1
#  01    1043 -   1043      Bit.1      - 0 | Bit.2           - 1
#  02    1044 -   1043      Bit.3      - 0 | Bit.4           - 1
#  03    1043 -   1043      Bit.5      - 0 | Bit.6           - 1
#  04    1043 -   1043      Bit.7      - 0 | Stopbit         - 1
#  05    9388 -   1043      Startbit + 8*0 | Stopbit         - 1
#  06    1044 -   9388      Startbit   - 0 | 8*0 + Stopbit   - 1
#  07    2086 -   1043      Startbit + 1*0 | Bit.1           - 1
#  08    1043 -   1043      Bit.2      - 0 | Bit.3           - 1
#  09    1043 -   1044      Bit,4      - 0 | Bit.5           - 1
#  10    1043 -      0      Bit.6      - 0 | Bit.7 + Stopbit - 1 (dauerhaft 1)
```

received: 11 words *eingefügte Erläuterungen sind kursiv gedruckt*

UART - Transmitter ----------

Mit Hilfe der PIO's lassen sich zusätzlich zu den beiden vorhandenen Seriellen Schnittstellen bis zu 8 weitere (als Sender oder Receiver) einrichten.

pio_uart_tx_osre.py

In der SDK gibt es bereits ein Beispiel für einen UART-Transmitter, der auf den PIOs arbeitet.
Dieses Beispiel kann man um eine Instruktion verkürzen:
In den Parametern der asm_pio-Funktion wird "pull_thresh = 8" gesetzt.
Dadurch wird ein interner Counter gesetzt, der bei jedem "out()" dekrementiert wird.
Mit "jmp(not_osre, 'next_bit')" werden nun solange die Bits ausgegeben, bis der Counter auf 0 steht.
Auf diesem Weg wird ein separater Bit-Counter über das X- oder Y-Register eingespart und ein Register freigehalten für andere Zwecke (die Instruktion "set(x, 7") entfällt).

pio_pull_thresh.py

Ein kleiner Exkurs um zu testen, ob "pull_thresh" ohne "autopull = True" funktioniert.

Dazu werden 32-Bit-Muster an die Statemachine geschickt und per "out(isr, 1)" direkt ins ISR weitergeleitet.
Der Vorgang wird solange wiederholt, wie "not_osre" nicht erfüllt ist.
Danach erfolgt ein "push()", so dass der Inhalt des ISR per sm.get() abgeholt werden kann.

Der Input ist immer gleich, der Output dagegen abhängig von push_thresh:

input : 0x12345678 (bei shift_out = right, shift_in = right)

output : 0x78000000 - bei push_thresh = 8
output : 0x56780000 - bei push_thresh = 16
output : 0x34567800 - bei push_thresh = 24
output : 0x12345678 - bei push_thresh = 32

Per "out()" wird nur die Anzahl an Bits rausgeshiftet, wie in "pull_thresh" festgelegt.
Danach ist "not_osre" nicht mehr erfüllt, "jmp(not_osre, 'label')" wird nicht mehr ausgeführt.

Zurück zum Thema UART.
Für den Fall, dass der Empfänger ein Paritätsbit erwartet, sind mehrere Varianten durchgespielt:

pio_uart_tx_parity_soft.py
In der ersten Variante berechnet der Sender das Paritätsbit selbst und schreibt 9 Bits ins Fifo der Statemachine (bei pull_thresh = 9) - das ist der einfachste Ansatz.

In den beiden anderen Variante berechnet die Statemachine selbst das Paritäts-bit, indem die ausgegebenen "high"-Bits mitgezählt werden.
Nachdem ein Bit ausgegeben ist, wird es auf "1" getestet.
Dazu muss es in das Y-Register manipuliert werden, damit ein "jmp(not_null, 'label')" ausgeführt werden kann.
Wenn "1", dann wird der Counter für die Paritätsbits (hier das X-Register) dekrementiert.
Das kostet einige zusätzliche Instruktionen bzw. einen zusätzlichen Pin als JMP-Pin (in der folgenden Variante).
Den Out-Pin gleichzeitig als In- und JMP-Pin zu nutzen, dass funktioniert nicht (eigentlich logisch, da ein Pin nur entweder "Input" oder "Output" sein kann).
Allerdings ist es möglich, den Status des Out-Pins per "in_(pin, 1)" ins ISR zu laden und anschlie0end mit "mov(y, isr)" ins Y-Register zu kopieren.

pio_uart_tx_parity_in.py
Diese Variante liest das Bit, das gerade per "out()" ausgegeben worden ist, per "in_(pins, 1)" wieder ins ISR ein, um es dann über den genannten Umweg im Y-Register zu testen.

pio_uart_tx_parity_jmp.py
Diese Variante ist schlanker, benötigt aber einen separaten Input-Pin, der direkt mit dem Out-Pin verbunden sein muss.
Auf diesem Weg kann der Bitstatus ohne Umwege ins Y-Register kopiert werden.

pio_uart_tx_32.py
Bei diesem Transmitter wird ohne Paritätsbit gesendet.
Es werden alle 8 Fifo-Register der 8 Statemachines als 64-Byte-Buffer für zu

sendende Daten genutzt.
Zu Beginn der Übertragung wird die Anzahl der zu sendenden Bytes ins Y-Register geschrieben. Danach folgen die Daten.
Diese Methode ist schnell für Datensätze bis zu 32 Byte.
Ist die Anzahl größer, gehen die Geschwindigkeitsvorteile allmählich verloren, da das Programm solange blockt, bis alle Words in die Fifos geschrieben sind.

Die Vorbereitung der Daten ist etwas aufwändiger, da jeweils 4 Bytes in der richtigen Reihenfolge in ein Word umkopiert werden müssen.

UART - Receiver ----------

pio_uart_rx_1_byte.py

In der SDK ist ebenfalls ein asm_pio-Beispiel für einen UART-Receiver enthalten.
Davon abweichend wird in den folgenden Variante das Startbit zusätzlich 2x getestet und es wird geprüft, ob nach dem letzten Bit an der Stelle des erwarteten Stopbits der Pegel "high" ist.
Der Programmcode wird dadurch länger, aber ich hoffe, dass Bit-Fehler besser erkannt werden.
Nach dem Eingang eines Bytes wird ein Interrupt ausgelöst und im IRQ-Handler ein Zähler für die eingegangenen Bytes inkrementiert.

main() muss diesen Zähler abfragen, und - wenn > 0 - die aufgelaufenen Daten abholen.
Alternativ könnte man auf den Interrupt verzichten und die Statemachine regelmäßig mit sm.rx_fifo() auf empfangenen Daten pollen.

Der (Hinter-) Gedanke bei de Verwendung der Interrupts ist, dass dieser möglicherweise auch aus einem "lightsleep()" wecken könnte - was zur Zeit aber nicht funktioniert.

Dadurch, dass eine Statemachine in seinem Fifo 8 Words puffern kann, verfügt dieser Receiver automatisch über einen 8-Byte-Buffer.

pio'uart_rx_4a.py

Nun ist es eigentlich schade, wenn von den 32 Bit der Fifo-Register nur 8 Bit als Puffer genutzt werden.
Darum schreibt die nächste Version bis zu 4 Byte in ein Word der Fifos, bevor es einen Interrupt auslöst.
Der IRQ-Handler inkremntiert nun einen Word-Counter. Main() muss die Daten rechtzeitig abholen, bevor alle Fifo-Register voll sind und der Receiver blockt.

Ein Problem entsteht dann, wenn eine Sendung aus einer nicht durch 4 teilbaren Anzahl von Bytes besteht - dann steckt das Ende der Sendung noch im Puffer.

pio_uart_rx_4b.py

Da man die Länge der Sendung nicht immer auf eine ohne Rest durch 4 teilbare Anzahl von Bytes beschränken kann, muss das Ende der Übertragung erkannt werden, können, damit auch ein nur teilweise gefülltes ISR-Register exportiert wird:

Das Ende wird erkannt, wenn nach einem Stopbit für mehr als 1 Bit-Zeit kein neues Startbit folgt.
Dann werden die bislang empfangenen Bytes aus dem ISR ins Fifo-Register geschrieben. Zusätzlich wird die Anzahl der Bytes im letzten Word (das X-Register zählte die Bytes) in ein weiteres Fifo-Register geschrieben und

anschließend ein Interrupt ausgelöst.
Im Normalfall wird nur 1 Word (= die letzten 4 Bytes) gesendet.
Der Interrupt-Handler kann am Eingang von 2 Words das Ende die Übertragung erkennen und schreibt die beiden letzten Words in separate Variablem (last_word und last_cnt).

Die Anzahl der Bytes ist im X-Register ablesbar:
Byte_Anzahl = 3 - X-Register

Die empfangenen Bytes wurden in der Statemachine bitweise nach rechts ins ISR geshiftet.
Am rechten Ende stehen daher unbenutzte Bytes. Wieviel, das verrät das X-Register:

```
wenn x = 3 : __ __ __ __ # alle 4 Bytes frei
     x = 2 : 11 __ __ __ # nur das höchstwertige Byte benutzt
     x = 1 : 22 11 __ __ # die beiden höchstwertigen Bytes benutzt
     x = 0 : 33 22 11 __ # nur das niedrigstwertige Byte frei
```

Die "leeren" Bytes werden verworfen, die verbleibenden Datenbytes werden in das abholende Array geschrieben und zusammen mit der Anzahl der empfangenen Bytes zurückgegeben.

Das klingt etwas kompliziert, ist es leider auch.
Der Vorteil: solange die Daten schneller abgeholt werden, als sie empfangen wurden, kann die Übertragung beliebig lang sein.

Von den 8 Fifo-Registern werden nur 2 genutzt, statt dessen in ein Ringbuffer von 16 Words eingerichtet, der die Daten zwischenspeichern kann, bevor sie von main() abgeholt werden.

pio_uart_rx_32_byte.py
Da jede Statemachine über 8 * 32 Bit-Fifos verfügt, sollte es möglich sein, 32 Bytes in den Fifo-Registern zu puffern und erst am Ende der Übertragung einen Interrupt auszulösen.
Voraussetzung ist allerdings, dass nicht mehr als 32 Byte pro Übertragung gesendet werden - sonst gehen die überzähligen Bytes verloren.

Auch bei dieser Variante muss das Ende der Übertragung durch eine "Sendepause" von 1 Bitzeit erkannt werden können.

Die Auswertung der Daten ist ähnlich wie beim Vorgänger und sieht so aus:

Solange weniger als 29 Byte empfangen wurden, sind weniger als 8 Fifo-Register mit Daten belegt.
Es bleibt mindestens 1 Register frei, um die Gesamtzahl der empfangenen Bytes zu übermitteln.
Die steht zunächst im X-Register der Statemachine und wird zusammen mit dem Y-Register in das nächste freie Fifo-Register kopiert.

Sind zwischen 29 und 32 Byte übermittelt worden, dann sind bereits alle Register mit Daten belegt.
Das X- und Y-Register werden blockend gepushed (warten darauf, dass Register frei werden) und müssen nach dem Leeren der Fifos ausgelesen werden.

Wenn genau 32 Byte empfangen wurden, dann ist es wichtig zu wissen, ob nach dem letzten Stopbit noch ein Startbit folgte - das wäre dann das 33.Byte und das ginge verloren.

Wenn das Y-Register (das die Dauer des Stopbits indirekt misst) übergelaufen ist, dann ist das ein gutes Zeichen:
es wurde innerhalb der festgelegten Wartezeit kein neues Startbit erkannt, es sind also genau 32 Byte empfangen worden.
Ein Wert von Y >= 0 dagegen bedeutet, dass ein Startbit erkannt wurde.
Warum ist das so?
Weil die Warteschleife auf das n#chste Startbit verlassen wurde, bevor der Zähler übergelaufen ist (der Überlauf ist am Zählerstand Y = 0xFFFF_FFFF erkannbar).

Gleichgültig, wie viele Bytes empfangen wurden, im allerletzten Register steht dann: 0x0000_YYXX
Wobei 0xYY der Stand des Y-Registers, 0xXX der Stand des X-Registers ist.

Wenn 0xYY nicht = 0xFF, dann liegt ein Fehler vor (neues Startbit erkannt).
Die Anzahl der empfangenen Bytes errechnet sich so:
Anzahl_Bytes = 32 - (0xXX + 1)

Bei dieser Receiver-Variante setzt der IRQ-Handler lediglich ein Flag, das von main() aus getestet werden muss.
Sind Daten empfangen worden, dann übergibt main() ein Array, in das der komplette "Datensatz" geschrieben wird.

UART-Fazit

Die PIOs des Pico lassen sich in verschiedener Form als UART nutzen.
Welche Variante für eine konkrete Anwendung geeignet ist, hängt von diversen Faktoren ab:
- Übertragungsrate (Python ist keine Rennmaschine),
- Anzahl der pro Sendung zu übertragenden Bytes im Verhältnis zur Buffergröße
- Pausen zwischen den Sendungen / Auslastung des Microcontrollers

Bestehen die Übertragungen aus weniger als 10 Byte, dann eignet sich pio_uart_rx_1.py, für Daten bis 32 Byte pio_uart_rx_32.py.
Beide sollten mit 115200 Baud arbeiten können.
Bei den beiden anderen Varianten können beliebig viele Bytes empfangen werden, allerdings ist MicroPython hier der die Performance begrenzende Flaschenhals.

pio_irq_test.py
Eine andere Frage ist, inwieweit die Interrupts geeignet sind, mehrere parallel arbeitende Receiver zu unterstützen.
Das Konzept der Interrupts in asm_pio ist nur spärlich dokumentiert und war (mir) nicht auf Anhieb verständlich.

Zitat aus der SDK für C:

1.) *If the MSB is set, the state machine ID (0…3) is added to the IRQ index, by way of modulo-4 addition on the two LSBs.*
2.) *For example, state machine 2 with a flag value of 0x11 will raise flag 3, and a flag value of 0x13 will raise flag 1.*
3.) *IRQ flags 4-7 are visible only to the state machines;*
4.) *IRQ flags 0-3 can be routed out to system level interrupts, on either of the PIO’s two external interrupt request lines, configured by IRQ0_INTE and IRQ1_INTE.*
5.) *The modulo addition bit allows relative addressing of 'IRQ' and 'WAIT' instructions, for synchronising state machines which are running the same program.*

6.) Bit 2 (the third LSB) is unaffected by this addition. If Wait is set, Delay cycles do not begin until after the wait period elapses.

Die Sätze 3 und 4 sind nachvollziehbar.

Satz 1 bezieht sich auf rel(irq), denn rel() setzt lediglich das Bit.4 (=MSB des Index) der Instruktion (der Index sind die 5 LSB der asm_pio-Instruktion)
-> {"rel": lambda x: x | 0x10}.

Aus den Sätzen 1.), 2.), 5.) und 6.) geht hervor:

Rel(irq) addiert zur IRQ-Nummer (aber nur zu Bit.0 + Bit.1) die Statemachine-Nummer [0-3] und maskiert den Überlauf, Bit.2 bleibt unverändert:

IRQ = (#SM + #IRQ) & 0x03

Der IRQ wird also letztlich durch die jeweils aktive Statemachine eindeutig festgelegt.
Wenn rel(0), dann entspricht die IRQ-Nummer immer der Nummer der Statemchine.

Die Interrupts 0-3 führen innerhalb MicroPythons einen irq_handler aus, sofern der mit sm.irq("irq_handler") eingerichtet wurde.
Das IRQ-Flag wird nach dem Ausführen des Irq-Handlers automatisch gelöscht.

Ist das Bit.2 in der Interrupt-Nr. gesetzt, dann sind Interruptflags nur innerhalb der Statemaachines des zugehörigen PIO-Blockes sichtbar.

Am Beispiel der "pio_buzzer.py", bei dem 2 Statemachines miteinander kommunizieren, kann man die Zusammenhänge nachvollziehen:
Jeder der Interrupts von 0–7 funktioniert – allerdings nur solange, wie beide Statemachines auf dem gleichen PIO-Block laufen !

Verwendet man aber die Interrups 0-3 UND hat einen Irq-Händler installiert, dann werden der Irq-Handler ausgeführt (wenn Interrupt_Nr. == Statemachine_Nr).
Der Intervallgeber im Beispiel versagt nun, da das Interupt-Flag vom Handler gelöscht wird und den Buzzer nicht mehr abschaltet.

Die UART-Receiver nutzen den selben asm_pio-Code und lösen jeweils einen Systeminterrupt aus.
Wenn die Interrupt-Nummer im asm_pio-Code fest vorgegeben wäre, könnte der Code nicht von mehreren Statemchine ausgeführt werden.
Durch Einsatz von irq(rel(0)) führt jede Statemachine einen anderen IRQ aus.
Diese Zusammenhänge sollen in pio_irq_test.py überprüft werden.

So habe ich getestet:

Es führen 8 Statemachines den gleich asm_pio-Code aus, benutzen aber jeweils individuelle IRQ-Handler.
Jeder Statemachine wird ein anderes Datenbyte übergeben, das sie unverändert wieder per ISR und push() ins TX_Fifo schreibt und einen irq(rel(0)) auslöst.

Wenn jeder IRQ-Handler genau den Wert zurückliefert, der an die Statemachine geschickt wurde, dann arbeiten die Interrupts unabhängig voneinadner.

Das Ergebnis ist wie erwartet.

Es sollte also möglich sein, 8 UART-Receiver einzurichten, die interrupt-gesteuert unabhängig voneinander Daten empfangen können

Sonstiges ----------

Anmerkung:
Ziel meiner Übungen war zu sehen, was ich für meine Zwecke mit den PIOs anstellen kann, unabhängig davon, ob das selbe auf anderem Weg auch oder gar besser realisierbar ist.

pio_buzzer_1.py
In der Python-SDK wird ein Beispiel für eine blinkende LED geliefert.
Erhöht man den Blinktakt und tauscht die LED gegen ein Piezo-Summer, dann erhält man einen - naja - eben einen Summer.
Eine 2.Statemachine sorgt dafür, dass daraus ein Intervall-Summer mit einer wählbaren "Speed" und Anzahl von "Beeps" wird.
Dies war meine allererste Übung in asm_pio, in der ich sehen wollte, wie ich für einen IR-Remote-Transmitter einen modulierbaren, taktgenauen Träger von 38KHz generieren kann.

pio_buzzer_2.py
Am Ende meiner Übungen habe ich dann noch andere Methoden kennengelernt, den Summer umzusetzen.
Hier wird die Modulation durch das Shiften eines 32-Bit-Musters gesteuert.
Jedes Bitmuster beschreibt den Schaltzustand des Tongenerators über einen Zeitraum von ca 1 Sekunde.
Bis 8 (blockend 9) Bitmuster können in einem "Rutsch" in die Fifos geschrieben werden.
Der Tongenerator selbst konnte auf 2 asm_pio-Instruktionen reduziert werden.

pio_buzzer_3.py
Hier wird der Tongenerator über externe Pins gesteuert.
Der Programmcode wird um einige Instruktionen kürzer, aber es werden zwei zusätzliche Pins benötigt.

pio_ir_nec_tx.py
Erhöht man die "Summer-Frequenz" weiter auf 38 KHz, tauscht den Summer gegen eine IR-LED und taktet die LED gemäß NEC-Protokoll, dann erhält man einen IR-Remote-Transmitter.

pio_ir_nec_rx.py
Ein dazu passender IR-Remote-Receiver ist ebenfalls via asm_pio realisiert.

pio_pwm.py
MicroPython bietet von Haus aus eine 16-Bit-PWM mit bis zu 2 KHz Wiederholfrequenz.

In den Beispielen zur SDK findet sich eine Version, die auf den PIO's läuft.
Diese Version ist hier erweitert und man kann experimentieren mit
- einstellbarer PWM-Tiefe (8-32 Bit),
- unterschiedlicher Wiederholfrequenz und
- Anzahl der Zwischenschritte zwischen 0 und maximaler PWM-Tiefe,

Die Zwischenschritte (Compare-Werte) werden exponentiell interpoliert, sollten also eine optisch "linear" empfundene Ansteuerung von LED's ermöglichen.

Ok, was nun noch kommt, das kann man natürlich auch Python ausreichend genau realisieren. Aber auf den PIO's läuft das völlig unabhängig von Rest des Pico.

pio_taster.py
Ein entprellter Taster (Schließer gegen high) schaltet einen Pin abwechselnd ein bzw. aus.

pio_off_delay.py
Ein entprellter Taster schaltet einen Pin ein und anschließend mit einer einstellbaren Verzögerungszeit wieder aus (Treppenhausautomat).
Wird der Taster innerhalb der Verzögerungszeit erneut betätigt, startet die Verzögerungszeit neu.

pio_onoff_delay.py
Ein entprellter Taster schaltet einen Pin abwechselnd ein und aus, wobei getrennte Verzögerungszeit für das Ein- und das Ausschalten möglich sind.
Während des Ablaufes der Verzögerungszeit signalisiert ein Pin, dass Verzögerungen aktiv sind.

pio_not_gate.py
Invertiert ein Signal per Hardware innerhalb eines PIO-Taktes.
Kann vor einen Counter geschaltet werden, wenn der Ruhepegel invertiert werden muss.

Anmerkung.
Für meine "Spielereien" hatte ich bislang noch keine praktische Anwendung.
Die Funktionstests habe ich entweder nach "Augenschein" (= blinkende LED) oder mit einem LA durchgeführt.
Es können daher durchaus noch unerkannte Fehler vorliegen.
Und der Code lässt sich vermutlich auch kompakter und eleganter formulieren.

Hilfreiche Informationen zu asm_pio habe ich hier gefunden:

- https://www.youtube.com/watch?v=yYnQYF_Xa8g
- https://github.com/GitJer/Some_RPI-Pico_stuff

In meinen Programmbeispielen habe ich häufig den asm_pio-Code als @staticmethod in die jeweilige Klasse eingebunden.

Die IDE Thonny wirft seltsame und aussagelose Fehlermeldungen, wenn innerhalb dieser eingebundenen Routinen Fehler erkannt werden.
Und verweigert eine weitere Zusammenarbeit. Nur ein Reset hilft dann weiter.

Für die Phase der Entwicklung ist es daher sinnvoll, die Routinen außerhalb der Klassen als "normale" Funktion einzusetzen.

Michael S.