

8 UART's mit asm\_pio / PIO / DMA / Micropython auf dem PI PICO

8 UART-Schnittstellen mit asm\_pio / PIO / DMA / Micropython auf dem PI PICO

Worum geht es hier ?

Der PI PICO verfügt von Haus aus über 2 integrierte UART-Schnittstellen, von denen eine meist für die Programmierung genutzt wird.

Mit den PIO's kann man bis zu 8 zusätzliche UART-Schnittstellen (alternativ als Transmitter oder Receiver) an frei wählbaren Pins des PI PICO einrichten, zusätzlich zu den vorhandenen beiden Schnittstellen.

Es folgen drei Versionen der Schnittstelle, die ersten beiden entstanden bereits im Zusammenhang mit meinen "Spielereien mit asm\_pio + Micropython auf dem PI PICO".

Hier ging es darum auszuprobieren, was man mit den PIO's anstellen kann.

Die dritte und neue Version nutzt DMA für den Datentransfer mit den PIO's.

Version 1 sendet bis zu 9 Byte nonblocking.

Der Empfang wird per Interrupt abgewickelt:

nach jedem empfangenen Byte wird eine ISR ausgeführt, die das empfangene Byte abholt.

Die Software muss selbst erkennen, wann ein Datentransfer abgeschlossen ist.

Eine Variante erkennt das Übertragungsende (am Ausbleiben eines weiteren Startbits).

Das Problem dieser Varianten des Receivers ist, dass viele Interrupts ausgelöst werden.

Micropython kann diese aber bei höheren Baud-Raten nicht mehr schnell genug bedienen, so dass irgendwann Datenverluste drohen und auch das laufende Programm ausgebremst wird.

Für moderate Baud-Raten und wenig zeitkritische Anwendungen funktioniert das aber zufrieden stellend.

Ansatz der 2. Version war, die 32-Bit-Fifos der Statemachines als Puffer zum Senden und Empfangen besser zu nutzen und damit die Anzahl der erforderlich Interrupts zu reduzieren.

Die Beschränkung liegt hier im 32-Byte-Limit.

Der asm\_pio-Code des Receivers belegt das gesamte Instruktion-Memory. Und die Nachbereitung der empfangenen Daten ist etwas mühsam.

Zwischenzeitlich gibt es diverse Beiträge im Web, die aufzeigen, wie PICO's DMA auch unter Micropython nutzbar ist.

DMA bietet den Vorteil, dass nicht nur die Statemachines, sondern auch der Datentransfer im Hintergrund ablaufen.

Der Programmcode sieht auf den ersten Blick aufwändiger aus - aber das täuscht. Er ist nur länger - wodurch er aber übersichtlicher wird.

## 8 UART's mit asm\_pio / PIO / DMA / Micropython auf dem PI PICO

--- Version 1 - PIO mit 8 Bit pro Fifo

uart\_tx.py

- tx = UART\_PIO\_TX(statemachine, tx\_pin)
- tx.write(buffer [,start\_adr = 0[,size = None]])-> Anzahl zu schreibender Bytes

Der Transmitter erwartet ein beliebiges Iterable mit zu sendenden Bytes.  
Er puffert die Daten (8 Byte) in seinen 8 Fifos, wobei aber jedes pro Fifo nur 1 Byte aufnimmt.

Beim Versand von bis zu (8 + 1) Byte ist das nonblocking.

uart\_rx1.py

- rx = UART\_PIO\_RX(statemachine, rx\_pin, buffer\_size)
- rx.any() -> aktuelle Anzahl empfangener Bytes
- rx.ready() -> userdefined
- rx.get\_buffer((buffer, bytes)) -> Referenz auf Empfangsbuffer, Anzahl Bytes)

Bei der Instantiierung muss die gewünschte Buffergröße angegeben werden.

Nach jedem empfangenen Byte wird ein IRQ ausgelöst.

In der ISR wird das empfangene Byte im vorbereiteten Array abgelegt und ein Index incrementiert.

Die Anzahl der bereits eingelesenen Bytes kann über rx.any() abgerufen werden.  
Eine benutzerdefinierte Methode muss die Kriterien beschreiben, die einen vollständigen Datensatz charakterisieren (eine bestimmte Datenlänge, ein Stopbyte ... ).

Das heißt, das Ende der Übertragung muss die Software selbst erkennen.

uart\_rx2.py

- rx = UART\_PIO\_RX(statemachine, rx\_pin, buffer\_size)
- rx.ready() -> Übertragungsende erkannt ?
- rx.get\_buffer((buffer, bytes)) -> Referenz auf Empfangsbuffer, Anzahl Bytes)

Die Statemachine des Receivers erkennt hier das Ende der Übertragung daran, dass nach einem Stopbit innerhalb von 2 Bitzeiten kein weiteres Startbit folgt.

In beiden Varianten wird bei einem Überlauf des Empfangs-Arrays ein IndexError ausgelöst.

--- Version 2 - PIO mit 32 Bit pro Fifo

uart\_tx32.py

Verpackt bis zu 32 8-Bit Sendedaten in eine 32-Bit-Variable, die in die 8 Fifos geschrieben und anschließend nonblocking gesendet werden.  
Ist schnell, wenn bis zu 32 Byte versandt werden sollen.

uart\_rx32.py

Speichert bis zu 32 der empfangenen 8-Bit Daten in den 8 32-Bit-Fifos zwischen,  
Das Übertragungsende erkennt die Statemachine (am ausbleibenden Startbit des nicht folgenden Bytes).

Daraufhin wird ein Interrupt ausgelöst, die Daten innerhalb der ISR ausgelesen

und in die richtige Reihenfolge gebracht. Das erfordert einige Byte-Schiebereien.

Auch dieser Ansatz ist schnell, aber wieder auf Übertragungen bis zu 32 Byte beschränkt.

Der asm\_pio-Code belegt leider fast den gesamten verfügbaren 32-Instruktions-Bereich.

-- Version 3 - PIO + DMA

Transmitter und Receiver erhalten bzw. schreiben die Daten per DMA von bzw. auf die Statemachine.

Das Prinzip des DMA-Transfers ist einfach:

Um einen Transfer zu starten, müssen lediglich 4 Register geschrieben werden:

- Read\_Adress : Adresse des zu schreibenden Datenarrays bzw. Registers
- Write\_Adress : Ziel-Adresse der Daten (Registeradresse bzw. Datenarray)
- Counter : Anzahl der zu schreibenden Daten
- Ctrl/Trigger : Konfiguration und Start des Transfers

Die Funktion der drei ersten Register ist selbsterklärend, für das Verständnis des Control + Trigger-Registers muss man tiefer in das Datenblatt einsteigen.

Es finden sich einige Diskussionen mit Programmbeispielen im Web, Aber Copy / Paste reichten nicht ganz aus, ein paar Hürden waren beim Receiver noch zu nehmen.

#### 1. Umfang der Übertragung

Beim DMA-Transfer muss der Umfang des Transfers angegeben und im Counter-Register übergeben werden. Beim Senden stellt das kein Problem dar.

Beim Empfangen dagegen ist (oft) nicht bekannt, wie viele Bytes eingeht werden (etwa bei einem GPS-Datensatz).

Daher muss einerseits der Receiver immer über einen überdimensionierten Puffer verfügen.

Andererseits kann der DMA-Controller aber dadurch nicht mehr erkennen, wann die Übertragung abgeschlossen ist:

Ist der Datensatz kleiner als die angemeldete Übertragungslänge, dann bleibt der Transfer offen (der DMA-Controller erwartet nach weitere Daten).

Daher muss das Signal für das Übertragungsende von der empfangenden Statemachine kommen.

Sie muss erkennen, ob nach dem letzten Stopbit ein weiteres Startbit folgt - oder nicht.

Im letzterem Falle ist die Übertragung beendet und ein Interrupt wird ausgelöst. In der ISR wird der DMA-Transfer beendet (das ist Voraussetzung dafür, dass er neu gestartet werden kann).

Eine kleine ASM-Routine ("dma\_abort()") übernimmt diese Aufgabe

#### 2.) Bufferüberlauf

Die Anzahl der zu übertragenden Bytes wird beim Triggern des DMA-Channels festgelegt.

Der Buffer selbst darf natürlich größer sein.

Treffen mehr Bytes ein als dem DMA-Controller mitgeteilt, dann werden die nicht mehr in das Empfangsarray übernommen.

Die Folge ist, dass die Statemachine beim push() blockt und vergeblich darauf wartet, dass das anstehende Byte abgeholt werden.

Die Statemachine kann daher kein Übertragungsende mehr erkennen, es gehen

(vermutlich) Daten verloren und der DMA-Transfer ist blockiert.

Um diese Problem zu vermeiden, muss der Buffer und der ins Counter-Register geschriebene Wert immer mindestens so groß sein wie der größte zu empfangene Datensatz.

Um zu prüfen, ob so ein Überlauf stattgefunden hat, gibt es die Methode `check()`.

Wenn das Busy-Bit im Controll/Trigger-Register nicht mehr gesetzt ist, dann hat der DMA-Controller registriert, dass die erwarteten Daten vollständig eingegangen sind. Weitere Daten werden nicht mehr übernommen.

`check()` kann aber in dem kurzen Zeitfenster zwischen dem Ende eines Stopbits und dem Erkennen eines Transfer-Endes nicht entscheiden, ob die UART-Übertragung abgeschlossen ist oder ob noch weitere Bytes folgen werden.

Daher wertet `check()` bereits ein nicht mehr gesetztes Busy-Flag als einen Überlauf und wirft einen `IndexError`.  
Der verfügbare Buffer muss also (sofern mit `check()` geprüft werden soll) um mindestens 1 Byte größer sein als der größte erwartete Datensatz.

### 3.) Reduzierung des Speicherbedarf

Die Statemachine shiftet die empfangenen 8-Bits in das 32-Bit breite `InputShiftRegister` der Statemachine.

Die "natürliche" Shiftichtung ist von links nach rechts (`SHIFT_RIGHT`).

Damit stehen die 8-Bit im höchstwertigen Byte - und müssen mit einem `uint32` abgeholt werden.

Indem die Bits von rechts nach links in das ISR geschiftet werden, stehen sie im niedrigstwertigen Byte und können mit einem `uint8` ausgelesen werden.

Das reduziert den Speicherbedarf um 75% - allerdings ist auch die Reihenfolge der Bits nun invertiert !

Darum wird in der ISR eine kleine Assembler-Routine ("`bit_reverse8()`") ausgeführt, in der die Bitfolge aller Bytes im Empfangsarray "reversed" wird.

### 4.)

Nachdem durch 3.) der Speicherbedarf reduziert werden konnte, habe ich dem Receiver einen zweiten Buffer gegönnt. Der Receiver speichert die Empfangsdaten abwechselnd in einem der beiden gleich großen Pufferbereiche und schaltet nach dem Abschluss eines Transfers sofort auf den jeweils anderen Puffer um.

`uart_dma_tx.py`

```
tx = PIO_UART_DMA_TX(statemachine, dmaChannel, tx_pin, baud)
tx.write(data, cnt = None) -> type data == array !!
tx.busy()                  -> DMA-Transfer noch aktiv ?
tx.out_waiting()           -> wie viele Bytes müssen noch gesendet werden ?
```

`uart_dma_rx.py`

```
rx = PIO_UART_DMA_RX(statemachine, dmaChannel, rx_pin, buffer_size, callback,
baud)
rx.ready()                  -> Übertragung abgeschlossen ?
rx.get_buffer(buffer, bytes) -> Referenz auf Empfangsbuffer, Anzahl Bytes
rx.check()                  -> bei Bufferoverflow -> IndexError
```

## 8 UART's mit asm\_pio / PIO / DMA / Micropython auf dem PI PICO

Transmitter und Receiver jeweils in der Kombination von PIO plus DMA ist wohl der effektivste Ansatz, um zusätzliche UART-Schnittstellen am PI PICO einzurichten.

Wobei manchmal der Speicherbedarf des asm\_pio-Codes eine Rolle bei der Entscheidung spielen kann.

Wenn auf den PIO\*s noch weitere Anwendungen ausgeführt werden sollen, dann muss man sehen, was man in den insgesamt verfügbaren 32 Instruktionen unterbringen kann.

Transmitter	asm_pio-Instruktionen
uart_tx.py	5
uart_tx32.py	12
uart_dma_tx.py	5

Receiver	asm_pio-Instruktionen
uart_rx_1.py	10
uart_rx_2.py	19
uart_rx_32.py	30
uart_dma_rx.py	18

Michael S.

22.07.2022