SPI-Slave mit asm_pio + MicroPython auf dem PI PICO

Für den PI PICO und die anderen MicroPython Ports kenne ich für die TWI- und die SPI-Schnittstelle keine Slave-Implementierungen.
Mit Hilfe der PIO's des PICO sollten beide realisierbar sein.

Wenn man die USI-TWI-Schnittstelle auf den ATtinys kennt, dann scheint ein ähnlicher Ansatz auch mit den PIO's machbar.
Wenn da nicht MicroPython zu langsam wäre, um den Software-Part dieser Schnittstelle zu meistern (siehe IRQ-Speed-Test am Ende).

An der SPI-Schnittstelle geht es zwar noch flotter voran als am TWI-Bus.
Dafür ist das Protokoll aber so einfach, dass die Übertragung ohne die Mithilfe von MicroPython möglich ist.
Zumindest so lange, wie in den Statemachines genug Platz zum Speichern der Daten vorhanden ist.
Diese Einschränkung stellen die Fifos der Statemachines dar, mit der Konsequenz, dass bei der vorgestellt Umsetzung Transaktionen nicht umfangreicher als 32 Byte sein können.

Damit kann man aber leben, wenn man nur Sensordaten etc. übertragen will.

SPI-Slave

Der PIO-Part des SPI-Slaves ist recht einfach:
3 Statemachines teilen sich die Arbeit, insgesamt ca. 15 asm_pio-Instruktionen sind erforderlich.

Statemachine 0
überwacht die Chip-Select Leitung. Bei einer fallenden Flanke (Beginn der Übertragung) werden die Statemachines 1 (Receiver) und 2 (Transmitter) aktiviert (sie beginnen, das sck-Signal zu pollen).
Bei einer steigenden Flanke (Ende der Übertragung) wird ein Interrupt ausgelöst, der über ein Flag MicroPython das Ende der Übertragung signalisiert.

Für die eigentliche Transaktion werden zwei getrennte Statemachines für RX und TX eingesetzt, damit jeweils für beide Übertragungsrichtungen 8 Fifo-Register zur Verfügung stehen.

Statemachine 1
Wartet auf den Interrupt der Statemachine 0 und überwacht dann das sck-Signal.
Liest bei einer steigenden Flanke das Daten-Bit am mosi-Pin ein und schreibt es in das ISR (InputShiftRegister).
Nach jeweils 32 Bit wird der Inhalt des ISR automatisch ("autopush") in ein RX-Fifo-Register "gepushed".

Statemachine 2
Während der Initialisierung wird in das Y-Register die Anzahl der zu sendenden Bits eingetragen.
Am Beginn jeder Transaktion wird dieser Wert ins X-Register kopiert und nach jedem ausgegebenen Bit decrementiert.

Auch die Statemachine 2 wartet auf den Interrupt der Statemachine 0 und überwacht dann das sck-Signal, schreibt bei einer fallenden Flanke ein Daten-Bit aus dem OSR (OutputShiftRegister) auf dem miso-Pin raus.

Nach jeweils 32 Bit wird der Inhalt des OSR automatisch ("autopull") aus einem TX-Fifo-Register in das OSR "gepulled".

Ein Bit-Zähler (voreingestellt auf die Anzahl der zu schreibenden Bits) lauft mit. Ist die Anzahl der zu schreibenden Bits erreicht, dann
- wird ein Interrupt ausgelöst, der den vollständigen Versand der Daten signalisiert und
- die Statemachine springt an den Programmanfang und wartet auf einen neuen Interrupt, der durch eine fallende Flanke an Chip-Select ausgelöst werden wird.

Es werden immer Gruppen von 4 Byte übertragen, da die Fifos des PICO 32-Bit breit sind.
Bei der Instantiierung / Initialisierung des SPI-Slave muss die Anzahl der zu übertragenden Words (32-Bit -> 4 Byte) vereinbart werden.
Dazu dient der Parameter "spi_words" [1 ... 8].

Der SPI-Master muss nun immer genau diese Anzahl von Words senden und die gleiche Anzahl erhält er vom Slave zurück geliefert.

Das war auch schon das ganze Geheimnis des SPI-Slave.

Für die Kommunikation mit der Anwendung kennt der SPI-Slave folgende Methoden:

received()
: prüft, ob eine Interrupt-Routine das Flag "irq_flag" gesetzt hat. Wenn Daten empfangen wurden, dann werden die aus den RX-Fifos-Registern in das Array "rx_data[]" kopiert.
Liefert True zurück, wenn Daten empfangen wurden, sonst False.
Received() muss in der mainloop() hinreichend regelmäßig aufgerufen werden.
Nach dem Empfang der Daten muss der TX-Puffer möglichst schnell wieder mit neuen Daten gefüllt werden, damit der Slave die nächste Anfrage des Masters beantworten kann.

rx_words()
: Übergibt eine Referenz auf das Array "rx_data[]" mit den empfangenen Daten.
Das Array enthält Words, die bei Bedarf in Bytes aufgelöst werden müssen.
Der Anwender kann zu sendende Daten im Array "tx_data[]" ablegen und anschließend "put_words()" aufrufen, dann werden diese Daten in die TX-Fifo-Register des Slave übertragen und beim nächsten Datentransfer vom Master abgeholt.
Alternativ kann ein beliebiges Datenarray (der gleichen Länge) an "put_words()" übergeben werden.

tx_words()
: Übergibt eine Referenz auf das Array "rx_data[]", in dem die zu sendenden Words abgelegt werden können.
"put_words()" kopiert die Daten aus dem Array "tx_data[]" in die TX-Fifo-Register des SPI-Slaves.

put_words(data = None)
: Schreibt die Words im Array "tx_data[]" in die TX-Fifo-Register.
Wird optional ein Array als Parameter übergeben, dann werden dessen Daten geschrieben.
Voraussetzung ist, dass das übergebene Array die Länge entsprechend der Anzahl der zu übertragenden Words aufweist.
Wird diese Methode aufgerufen, dann prüft sie, ob die TX-Fifo-

Register frei sind.
Wenn nicht, dann werden deren aktuelle Inhalte überschrieben.

Hinweis
put_words() muss möglichst schnell nach dem Empfang von Daten ausgeführt werden, da zu diesem Zeitpunkt die TX-Fifo-Register ausgelesen sind und der Transmitter des SPI-Slaves beim Empfang neuer Daten blocken würde (er würde weder Daten empfangen noch senden).
Der SPI-Master würde in diesem Fall entweder "0x00" oder "0xFF", empfangen, abhängig vom zufälligen Pin-Status des miso-Pins.

Der grundsätzliche Programmablauf könnte so aussehen:

```
mainloop
      # Sind Daten empfangen worden ?
      if slave.received():
            # Wenn ja, dann stehen die empfangenen Daten in rx_data[],
            # slave.rx_words() liefert eine Referenz auf rx_data[].
            # Nun die Daten bearbeiten, zu sendende Daten in tx_data[]
            # schreiben,
            # slave.tx_eords() liefert eine Referenz auf tx_data[].

            user_func()   # irgendetwas mit den Daten machen

            # Zum Abschluss die Daten aus tx_data (oder andere Daten) in die
            # TX-Fifo-Register kopieren.

            slave.put_words()
      ....
```

Die Anwendung auf dem SPI-Slave muss in ihrer mainloop() prüfen, ob Daten eingegangen sind, damit die RX-Fifo-Register ausgelesen werden und Platz für neue Daten gemacht wird.
Die Daten in "rx_data{[" können ignoriert werden, in jedem Fall aber müssen die TX-Fifo-Register des SPI-Slave zügig mit neuen Daten beschicken werden.
Denn auch leere TX-Register führen zum Blocken des Transmitters und damit des gesamten Slaves.

SPI-Master

Zum Testen diente der PI PICO gleichzeitig als SPI-Master.
Wichtig ist, dass der SPI-Master genau die Anzahl von Words bzw. Bytes sendet, die der SPI-Slave erwartet.

Der SPI-Master kennt folgende Methoden:

read()
schreibt Dummy-Daten 0xAAAA_AAAA (weil die für den Slave ohne Bedeutung sind) und holt die Daten aus den TX-Fifo-Registern des SPI-Slaves ab.

write(data)
Schreibt die übergebenen Daten über seine SPI-Schnittstelle byteweise heraus.
Die Daten müssen als Array, Bytearray oder Liste übergeben werden.
Wenn die Anzahl der Elemente = (Anzahl der zu übertragenden Words), dann werden die Words in ein Array von Bytes umkopiert.

Ist die Anzahl = (spi_word * 4), dann kann das Array ohne weitere Konvertierung direkt versandt werden (geht deutlich schneller !).

Write() schreibt das Array über die SPI-Schnittstelle byteweise raus und holt vom Slave den Inhalt des TX-Fifo-Register ab.

Die Verbindung zwischen SPI-Master und SPI-Slave sieht so aus:

Die Pins auf dem SPI-Master sind (auf dem PI PICO) nicht frei wählbar.
Beim SPI-Slave dagegen sind die Pins flexibel, mit der einzigen Ausnahme, dass der sck-Pin direkt auf den mosi-Pin folgen muss
(weil das im asm_pio-Code von _spi_in() so erwartet wird).

```
SPI-Master      SPI-Slave
miso    <---    miso
mosi    --->    mosi
sck     --->    sck     # sck muss der auf mosi folgende Pin sein !!!
csel    --->    csel
```

Die Pin-Belegung im Test ist der Datei "pins.txt" beschrieben.

Sonstiges

Die erzielbare SPI-Baudrate ist u.a. abhängig von der Qualität der Verbindung zwischen Master und Slave.
Bei meinen Tests waren Master und Slave auf ein und dem selben PICO installiert, der PICO steckte in einem Breadboard, die Pins waren "fliegend" verdrahtet und nur die internen PullUps waren gesetzt.

Eine SPI-Baudrate von 8 MHz führte dabei zu keinen Problemen.
Im Zweifelsfall sollte man mit kleineren Wert arbeiten, weil die Baudrate auf die erreichbare Brutto-Geschwindigkeit nur einen geringen Einfluss hat, der bremsende Faktor ist MicroPython.
F_PIO der Statemachine muss deutlich höher sein als F_SPI. Pro SPI-Takt sind theoretisch mindestens 4 PIO-Takte erforderlich (wait, out, wait, in).
Praktisch sollte der Faktor eher bei 8 - 12 liegen.

SPI Speed Test

In der Datei "spi_slave_test_5.py" habe ich einige Tests durchgeführt und die erreichbare F_SPI ausgelotet:
12 MHz waren bei meinem Testaufbau möglich, allerdings nur bei maximalem F_PIO-Takt (125 MHz).

Dadurch, dass Master und Slave nacheinander arbeiten müssen (weil auf dem selben Controller installiert), ist die Kommunikation ausgebremst:
Die gemessenen Zeiten für einen SPI-Write lagen bei minimal 1200 us:

```
Master                          Slave
Daten aufbereiten, Array mit
32-Byte füllen
Daten senden          ---->     Daten empfangen
Daten empfangen       <----     Daten senden
                                Daten aus den RX-Fifos abholen, invertieren
                                und in die TX-Fifo-Register schreiben.
                                Master zum Senden auffordern.
-----
```

Für jede zusätzliche Operation gönnt sich MicroPython gerne mal einige zusätzliche Millisekunden.

Die Reduktion des SPI-Taktes von 8 MHz auf 1 MHz zeigt das erwartete Ergebnis:

Die Übertragung verlängert sich von 32 * 1us = 32 us auf 32 * 8us = 256 us.
Der gemessene Zeitunterschied lag bei 227us, theoretisch sollte die Differenz 256 - 32 = 224 us betragen.

IRQ Speed Test

Dieser Test stellte sich im Zusammenhang mit der Frage, ob auch ein TWI-Slave mit MicroPython realisierbar wäre.

Um abzuschätzen, wie lange MicroPython für die Ausführung eines Interrupts benötigt, hatte ich folgenden Test ausgeführt ("pio_irq_test.py"):

Ein Rechteckgenerator erzeugt 100 Takte einer definierbaren Frequenz.
An jeder steigenden Flanke wird ein Pin-Interrupt ausgelöst,
Der IRQ-Handler incrementiert dabei lediglich einen (globalen) Zähler.
Am Ende des Tests sollte der Zähler also genau die Anzahl der steigenden Flanken gezählt haben, also "100".
Wenn weniger gezählt wurde, dann war die Ausführungszeit des IRQ-Handlers länger als die Dauer eines Taktes.

Bis zu einer Taktfrequenz von ca. 15.000 KHz war das Ergebnis positiv, bei einem höheren Takt gehen Flanken verloren.
Das bedeutet, dass die gesamte ISR etwa 66 us zur Ausführung benötigt.
Ersetzt man den Pin-Interrupt durch einen Interrupt, der im pio_asm-Code des Generators ausgelöst wird, dann scheint die Ausführung etwas flotter zu werden, gemessen habe ich ca. 55 us.
Der Decorator "@mircorpython.native" beschleunigt noch einmal um wenige Prozentpunkte.

Um einen TWI-Slave nach dem Vorbild der USI-TWI-Slaves aus der 8-Bit-AVR-Welt umzusetzen ist das alles viel zu langsam, um bei einem TWI-Takt von 100 Khz den TWI-Slave zu bedienen (100KHz = 10us / Takt).

Folglich wird man für einen TWI-Slave wohl oder übel auf C zurückgreifen müssen.

Michael S.