Appendix F – The PIO Programming Package

The PIO Programming Package

**Introduction to the PIO**

The RP2040 and RP2350 have many built in peripherals such as PWM, UART, ADC, SPI. Using PIOs it is possible to add specialised functions/peripherals such as high accuracy serial data interfaces and bit streams.

PIOs can be thought of as cut-down, highly specialised CPU cores. The RP2040 contains two PIO blocks while the RP2350 has three blocks. MMBasic refers to them as PIO0, PIO1 and PIO2 in line with the Raspberry Pi documentation. The PIOs run completely independently of the main system and of each other and run extremely fast, with a throughput of up to 32 bits during every clock cycle.

PIOs implement state machines. Before a state machine can execute it's program, the program needs to be written to PIO memory, and the state machine needs to be configured.

This appendix describes the support MMBasic can give in using PIO. It does not contain an explanation how to write PIO state machine programs. For better understanding how the PIO state machines work refer to the following thread "PIO explained PICOMITE" on the thebackshed.com forum: <https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=15385>

**Overview of PIO**

A single PIO block has four independent state machines. All four state machines share a single 32 instruction program area of flash memory. This memory has write-only access from the main system, but has four read ports, one for each state machine, so that each can access it independently at its own speed. Each state machine has its own program counter.

Each state machine also has two 32-bit "scratchpad" registers, X and Y, which can be used as temporary data stores.

I/O pins are accessed via an input/output mapping module that can access 32 pins (but limited to 30 for the RP2040). All state machines can access all the pins independently and simultaneously.

The system can write data into the input end of a 4-word 32-bit wide TX FIFO buffer. The state machine can then use pull to move the output word of the FIFO into the OSR (Output Shift Register). It can also use out to shift 1-32 bits at a time from the OSR into the output mapping module or other destinations. AUTOPULL can be used to automatically pull data until the TX FIFO is empty or reaches a preset level.

The system can read data from the output end of a 4-word 32-bit wide RX FIFO buffer. The state machine can then use in to shift 1-32 bits of data at a time from the input mapping module into the ISR (Input Shift Register). It can also use push to move the contents of the ISR into the FIFO. AUTOPUSH can be used to automatically push data until the RX FIFO is full or reaches a preset level.

The FIFO buffers can be reconfigured to form a single direction 8-word 32-bit FIFO in a single direction. The buffers allow data to be passed to and from the state machines without either the system or the state machine having to wait for the other.

Each of the four state machines in the PIO has four registers associated with it:

* CLKDIV is the clock divider, which has a 16-bit integer divider and an 8-bit fractional divider. This sets how fast the state machine runs. It divides down from the main system clock.
* EXECCTRL holds information controlling the translation and execution of the program memory
* SHIFTCTRL controls the arrangement and usage of the shift registers
* PINCTRL controls which and how the GPIO pins are used.

The four state machines of a PIO have shared access to its block of 8 interrupt flags. Any state machine can use any flag. They can set, reset or wait for them to change. In this way they can be made to run synchronously if required. The lower four flags are also accessible to and from the main system, so the PIO can be controlled or pass interrupts back.

DMA can be used to pass information to and from the PIO block via its FIFO from the RP2040's memory

A PIO has nine possible programming instructions, but there can be many variations on each one. For example, Mov can have up to 8 sources, 8 destinations, 3 process operations during the copy, with optional delay and/or side set operations!

* Jmp Jump to an absolute address in program memory if a condition is true (or instantly).
* Wait Stall operation of the state machine until a condition is true.
* In Shift a number of bits from a source into the ISR.
* Out Shift a number of bits out of the OSR to a destination.
* Push Push the contents of the ISR into the RX FIFO as a single 32-bit word.
* Pull Load a 32-bit word from the TX FIFO into the OSR.
* Mov Copy date from a source to a destination.
* Irq Set or clear an interrupt flag.
* Set Immediately write data to a destination.

Instructions are all 16-bit and contain both the instruction and all data associated with it. All instructions operate in 1 clock cycle, but it is possible to introduce a delay of several idle clock cycles between an instruction and the next.

Additionally, there is a facility called "side-set" which allows a value to be written to some pre-defined output pins while an instruction is being read from memory. This is transparent to the program.

**Programming PIO**

PicoMite firmware programs the PIO state machine memory using one of 3 methods. Each method will be explained with an example of the exact same program that toggles one of the GPIO pins of the Raspberry Pi Pico. Which GPIO pin is toggled, is determined by the configuration, not in the program itself.

PIO ASSEMBLE

This command is used to use the build in assembler to generate the program from mnemonics, then write it directly into PIO memory.

PIO ASSEMBLE 1,".program test" 'a program has to have a name

PIO ASSEMBLE 1,".line 0" 'start the program at line 0

PIO ASSEMBLE 1,"SET PINDIRS,1" 'SET the GPIO line to output

PIO ASSEMBLE 1,"label:" 'define a label called "label"

PIO ASSEMBLE 1,"SET PINS,1" 'SET the GPIO pin high

PIO ASSEMBLE 1,"SET PINS,0" 'SET the GPIO pin low

PIO ASSEMBLE 1,"JMP label" 'JuMP to "label"

PIO ASSEMBLE 1,".end program list" 'end program, list=show result

PIO PROGRAM LINE

This command can be used to program 16bit values to indidual lines (locations) in the PIO memory.

PIO PROGRAM LINE 1,0,&hE081 'SET pin output

PIO PROGRAM LINE 1,1,&hE001 'SET pin high

PIO PROGRAM LINE 1,2,&hE000 'SET pin low

PIO PROGRAM LINE 1,3,&h0001 'JMP to line 1

PIO PROGRAM

This command writes all 32 lines in one PIO from an array. This is useful once a PIO program is debugged. It is extremely compact.

DIM a%(7)=(&h0001E0000E001E081,0,0,0,0,0,0,0)

PIO PROGRAM 1,a%()

**Configuring PIO**

The PicoMite firmware can configure each state machine individually. Configuration allows 2 state machines to run the exact same program lines (e.g. an SPI interface) but operate with different GPIO pins and at different speeds. There are several configuration fields.

FREQUENCY

PicoMite firmware contains a default configuration for each configuration field, except for the frequency. The frequency is set by a 16 bit CLKDIV divider from the ARM clock. Example: when OPTION CPUSPEED 126000 is set the PIO can run at speeds between 126MHz and 1.922kHz (126000000 / 65536). Be aware that higher CPU speeds (overclocking) directly impact the state machine frequency.

PIN CONTROL

PicoMite firmware defaults the GPIO pins for use by MMBasic. For the PIO to take ownership of a GPIO pin MMBasic needs to assign it to PIO as below.

SETPIN GPxx,PIOx (e.g. SETPIN gp0,pio1)

A state machine can SET the state of a pin (SET is a state machine instruction), but can also output serial data to one or more GPIO pins using the OUT instruction. Or read serial data using the IN instruction. And GPIO pins can be set as a side effect of any state machine instruction (SIDE SET). For each method of interfacing, different pins can be mapped to the state machine.

It is important to understand is that these instructions work on consecutive pins. This means that there is a range of pins that can be controlled, starting at the lowest GPx pin number (e.g. GP0), and pins next to it can be included (up to 5 pins in total). So GP0,GP1,GP2 is a valid set of IO pins. GP0,GP1,GP6 is not. Consider this when designing a PIO application.

Assigning GPIO pins to a state machine uses the PIO helper function PINCTRL:

PIO(PINCTRL a,b,c,d,e,f,g)

a/ the number of SIDE SET pins (0...5), SIDE SET can write 5 pins at once

b/ the number of SET pins (0...5), SET can write 5 pins at once

c/ the number of OUT pins (0...31), OUT can write 32 pins at once

d/ the lowest pin for IN pins (GP0.....GP31) IN can read up to 32 pins at once

e/ the lowest pin for SIDE SET (GP0.....GP31)

f/ the lowest pin for SET (GP0.....GP31)

g/ the lowest pin for OUT (GP0.....GP31)

Ranges for the different functions can overlap, be identical, or adjacent.

EXECUTE CONTROL

The execute control register EXECCTRL configures the program flow. There is a field that connects a GPIO pin to a conditional jump (JMP instruction), and fields that hold the line address of the main program loop begin (.WRAP TARGET) and end (.WRAP).

If we want the program flow to change in response of a GPIO pin state, a JMP PIN is used. The JMP pin is assigned in the execute control configuration (there can only be 1 pin per state machine) and the JMP happens only when the pin is high).

The state machine program starts at the beginning and runs until it reaches the end. In above demo program, the program loops from the end to beginning using a (unconditional) JMP instruction. An alternative way to using the JMP instruction is defining the beginning of the loop (WRAP TARGET = line 1) and end of the loop (WRAP = line 2) and configure the state machine to only execute these instructions in between. The JMP instruction in line 3 is obsolete when WRAP/WRAP TARGET is used.

PIO(EXECCTRL a,b,c)

a/ the GPIO pin for conditional JMP (e.g. GP0)

b/ the WRAP TARGET line number (e.g. 1)

c/ the WRAP line number (e.g. 2)

SHIFT CONTROL

The IN and OUT instructions shift data from the FIFO register to the GPIO pins. In between MMBasic and the PIO, 32bit words can be communicated. Since both the ARM cores and the PIO microcontrollers operate independently, the data is exchanged through FIFO's. The ARM (MMBasic) puts data in the FIFO, PIO reads it. This uses the TX FIFO. The other way around uses the RX FIFO. The FIFO's are normally 4 words deep but can be configured to a single 8 word deep RX or TX FIFO.

The PIO can "shift" data IN the RX FIFO from the MSB side, or from the LSB side. That is set with the IN SHIFTDIR bit. Similar the OUT SHIFTDIR bit for OUT data. The autopull and autopush flags in combination with the pull and push thresholds determine when FIFO is replenished.

In RP2350 the FIFO’s can also be used as individual registers, allowing more flexible communication between MMBasic and the state machines. This is achieved by FJOIN\_RX\_GET and FJOIN\_RX\_PUT in the SHIFTCTRL register.

PIO(SHIFTCTRL a,b,c,d,e,f,g,h)

a/ push threshold (leave 0 for now)

b/ pull threshold (leave 0 for now)

c/ autopush (leave 0 for now)

d/ autopull (leave 0 for now)

e/ IN-shiftdir (1 = shift MSB, 0 = shift LSB)

f/ OUT-shiftdir (1 = shift MSB, 0 = shift LSB)

g/ fjoin\_rx (join TX and RX fifo to 1 RX fifo)

h/ fjoin\_tx (join TX and RX fifo to 1 TX fifo)

i/ fjoin\_rx\_get (1=ARM write individual FIFO registers, 2350 only)

j/ fjoin\_rx\_put (1=ARM read individual FIFO registers, 2350 only)

WRITING THE STATE MACHINE CONFIGURATION

A state machine configuration is written using the command:

PIO INIT MACHINE a,b,c,d,e,f,g

a/ the PIO (0 or 1)

b/ the state machine number (0...3)

c/ frequency (CPUSPEED/65536...CPUSPEED in Hz)

d/ pincontrol value (PIO(PINCTRL ......))

e/ execture control value (PIO(EXECCTRL......))

f/ shiftcontrol value (PIO(SHIFCTRL......))

g/ start address (0....31, the line at which the state machine starts executing, can be a label)

STARTING AND STOPPING A STATE MACHINES

Once the PIO is configured, you can start and stop the state machine using:

PIO START a,b

PIO STOP a,b

a/ the PIO number (0 or 1)

b/ the state machine (0...3)

Note that when stopping a state machine, it stops right where it is. To restart the state machine it is advisable to PIO INIT MACHINE first.

EXAMPLE PROGRAM 1

A complete PIO implementation that toggles a GPIO pins can be implemented in MMBasic as shown below. Connect a buzzer to GP0, and hear the audio tone generated by the PIO.

'disconnect ARM from GP0

setpin gp0,pio1 'use GP0 as output pin for PIO 1

'pio program used

'0 E081 'SET pin output

'1 E001 'SET pin high

'2 E000 'SET pin low

'3 0001 'jmp 1

'program pio 1 using an array to write the program in PIO memory, and start

Dim a%(7)=(&h0001E000E001E081,0,0,0,0,0,0,0)

PIO program 1,a%()

'configure pio 1 statemachine 0

p=Pio(pinctrl 0,1,,,,gp0,) ‘define SET uses 1 pin, and that is GP0

f=2029 ‘2029 Hz is lowest frequency CPUSPEED 133000

PIO init machine 1,0,f,p ‘use default for execctrl, shiftctrl, start ‘address(=0)

'start the PIO 1 state machine 0

PIO start 1,0

Note that the MMBasic program ends, but the sound on the buzzer continues. PIO is independent of the ARM microcontroller and continues until it is stopped. Entering the MMBasic editor will stop the PIO.

**FIFO's**

MMBasic and the PIO exchange information using FIFO's. The PIO's PUSH data into the RX FIFO (MMBasic is the receiver), or PULL data from the TX FIFO (MMBasic is the transmitter).

When PIO is fetching data from the FIFO the data is transferred to the OSR (Output Shift Register), from there is can be processed. The PIO can push the data from the ISR (Input shift register) into the FIFO. Additionally, the PIO has 2 registers X and Y that can be used for storage, or counting. PIO cannot add or subtract or compare.

Data flow:

MMBasic -> FIFO -> OSR -> PIO (or pins)

PIO (or pins) -> ISR -> FIFO -> MMBasic

MMBasic can write data into the TX FIFO and read data from the RX FIFO using:

PIO READ a,b,c,d

PIO WRITE a,b,c,d

a/ PIO number (0 or 1)

b/ state machine number (0...3)

c/ number of 32 bit words (1...4)

d/ integer variable name (i.e. variable% or array%())

PIO CLEAR clears all the PIO FIFO's, as does PIO START and PIO INIT MACHINE.

The MMBAsic program doesn't need to wait for data in the FIFO to appear since the RX FIFO can be assigned an interrupt. The MMBasic interrupt routine can fetch the data from the FIFO.

Similar for TX interrupt in which case MMBasic gets an interrupt when data is needed for the TX FIFO.

PIO INTERRUPT a,b,c,d

a/ PIO (0 or 1)

b/ state machine (0...3)

c/ Name of RX interrupt handler (i.e. "myRX\_Interrupt" or 0 to disable)

d/ Name of TX interrupt handler (i.e. "myTX\_Interrupt" or 0 to disable)

EXAMPLE PROGRAM 2

Below program explains many of the above presented MMbasic functions and commands. The program reads a NES controller (SPI) connected to the Raspberry Pi Pico. The NES controller consists of a HEF4021 shift register connected to 8 push button switches.

Program uses: **wrap** and **wrap target**, IN, side set and delay, PUSH, PIO READ. GP0 and GP1 are in SET for pin direction, and in side set for compact code.

The wiring is as defined in the code:

'disconnect ARM from GP0/1/2

setpin gp0,pio1 'clock out

setpin gp1,pio1 'load out

setpin gp2,pio1 'data in

'PIO program

PIO assemble 1,".program NES" 'a program needs a name

PIO assemble 1,".side\_set 2" 'use 2 bits for side set, 3 for ‘delay

PIO assemble 1,".line 0" 'start code at line 0

PIO assemble 1,"SET pindirs,&b11" 'set GP0,GP1 output, side GP0,GP1 ‘low

PIO assemble 1,".wrap target" 'wrap target = top of the loop

PIO assemble 1,"IN null,32 side 2" 'set ISR to 0, GP1 high (load), ‘GP0 low

PIO assemble 1,"SET X,7 side 0" 'set X counter to 7, GP0,GP1 low

PIO assemble 1,"loop:" 'inner loop label

PIO assemble 1,"IN pins,1 side 0" 'shift 1 databit in, keep GP0,GP1 ‘low

PIO assemble 1,"JMP X-- loop side 1" 'jmp to loop, dec. X, GP0 ‘high(clock)

PIO assemble 1,"PUSH side 0 [7]" 'now X=0, PUSH result into FIFO, ‘delay 7

PIO assemble 1,".wrap" 'end outer loop, repeat

PIO assemble 1,".end program list" 'end of program, list result

'configure pio1

p=Pio(pinctrl 2,2,,gp2,gp0,gp0,) 'GP0,GP1 out (SET and SIDE

‘ SET), GP2 IN

f=1e5 '100kHz

s=PIO(shiftctrl 0,0,0,0,0,0) 'shift in from LSB for IN (and

‘OUT)

e=PIO(execctrl gp0,PIO(.wrap target),PIO(.wrap) 'wrap and wrap target

'write the configuration

PIO init machine 1,0,f,p,e,s,0

'start the pio1 code

PIO start 1,0

'Check the the read data in MMBasic and print

dim d%

do

pio read 1,0,1,d%

print bin$(d%)

pause 200

loop

END

**DMA To and From the FIFOs**

The way that DMA works is as follows:

When reading from the FIFO the DMA controller waits on data being in the FIFO and when it appears transfers that data into microcontroller memory. Each time it reads it increments the pointer into the microcontroller memory so that it can, for example, incrementally fill an array as each and every data item is made available.

When writing to the FIFO the DMA controller writes data from microcontroller memory to the FIFO automatically waiting whenever the FIFO is full. Thus, data can be prepared in an array and the DMA controller will stream that data to the PIO FIFO as fast as the PIO program requires it.

DMA can transfer a 32-bit word, a 16-bit short, or an 8-bit byte and when setting up DMA you need to tell it the size of the tranfer and how many transfers to make. Because each transfer will increment the memory pointer by 1,2, or 4 bytes MMBasic must deal with the data packed into memory rather than the 64-bits used for MMbasic integers and floats. Luckily MMBasic implements two commands MEMORY PACK and MEMORY UNPACK to do this very efficiently but it could equally be done using standard BASIC arithmetic.

The DMA can be configured to repeatedly loop data into or out of a section of memory (a ring buffer)

The commands are:

PIO DMA\_IN a,b,c,d,e,f,g

PIO DMA\_OUT a,b,c,d,e,f,g

a/ pio (0 or 1)

b/ state machine (0...3)

c/ nbr (number of words to be transferred)

d/ data%() (interger array name)

e/ completioninterrupt (where to go when done, optional)

f/ transfersize (8/16/32, optional)

g/ loopbackcount (used data%() as a ring buffer, optional, loopbackcount = 2^n)

The DMA will start the state machine automatically and there is no need for a PIO START command. But, before starting the transfer make sure a fresh PIO INIT MACHINE is done, so the state machine starts at the required start address.

When a ring buffer is used, it requires special preparation:

PIO MAKE RING BUFFER a,b

a/ name of integer buffer

b/ size of the array in bytes

Example :

DIM packed%

PIO MAKE RING BUFFER packed%,4096

packed% will then be an integer array holding 4096/8=512 integers

This can then be used by the DMA for a loopbackcounter with DMA of 1024 32-bit words, 2048 16-bit shorts or 4096 8-bit bytes

EXAMPLE PROGRAM 3

This program brings everything together and uses DMA to read 128 samples from the PIO RX FIFO. For the demonstration, GP0 to GP5 are outputs of 3 PWMS, and are ,at the same time, sampled by the PIO as a 6 channel logic analyser or oscilloscope. The 128 samples are sent to the serial port as waveforms.

This Logic Analyzer program also demonstrates PIO DMA RX, MEMORY UNPACK, the use of buffers. It uses PWM’s to generate a test signal on gp0..gp5 for demo purpose. These same pins are read by the logic analyzer and output to the console. To use this Logic Analyzer normally comment out first 14 lines.

'generate a 50Hz 3 phase test signal to demonstrate the DMA on 6 GPIO pins.

SetPin gp0,pwm 'CH 0a

SetPin gp1,pwm 'CH 0b

SetPin gp2,pwm 'CH 1a

SetPin gp3,pwm 'CH 1b

SetPin gp4,pwm 'CH 2a

SetPin gp5,pwm 'CH 2b

Fpwm = 50: PW = 100 / 3

PWM 0, Fpwm, PW, PW - 100, 1, 1

PWM 1, Fpwm, PW, PW - 100, 1, 1

PWM 2, Fpwm, PW, PW - 100, 1, 1

PWM sync 0, 100/3, 200/3

'------------------------------- LA code PIO --------------------------

'PIO code to sample GP0..GP6 as elementary logic analyser

PIO clear 1

'in this program the PIO reads GP0..GP5 brute force

'and pushes data into FIFO. The clock speed determines the

'sampling rate. There are 2 instructions per cycle

'taking 10000/2 / 50 = 100 samples per 50Hz cycle.

PIO assemble 1,".program push"

PIO assemble 1,".line 0"

PIO assemble 1,".wrap target"

PIO assemble 1,"IN pins,6" ‘get 6 bits from GP0..GP5

PIO assemble 1,"PUSH block" 'push data when FIFO has room

PIO assemble 1,".wrap"

PIO assemble 1,".end program list"

'configuration

f=1e4 'PIO run at 10kHz

p=Pio(pinctrl 0,0,0,gp0,,,) 'IN base = GP0

e=Pio(execctrl gp0,PIO(.wrap target),PIO(.wrap)) 'wrap 1 to 0, gp0 is

‘default

s=Pio(shiftctrl 0,0,0,0,0,0) 'shift in through LSB, out is not used

'write the configuration, speed 10kHz (data in FIFO 10us after edge GP0)

PIO init machine 1,0,f,p,e,s,0 'start address = 0

'-------------------------- LA code MMBasic ----------------------------

'define memory buffers

Dim a$(1)=("\_","-") 'characters for the printout

length%=64 'size of the packed array

Dim data%(2\*length%-1) 'array to put the 32 bit samples

‘FIFO format

Dim packed%(length%-1) 'DMA array to pack 32 bit samples ‘in 64 bit integers

'let the DMA machine run, and repeat at will

Do

PIO DMA RX 1,0,2\*length%,packed%(),ReadyInt

print "press any key to restart sampling"

do:loop while inkey$=""

Loop

End

'--------------------------------SUBS MMBasic --------------------------

Sub ReadyInt

'stop the PIO and re-init for next run

PIO stop 1,0

PIO init machine 1,0,f,p,e,s,0 'start address = 0

'get the data from the packed DMA buffer and unpack to original 32 bit ‘format

Memory unpack packed%(),data%(),2\*length%,32

'Serial output as if logic analyzer traces

For j=0 To 5

mask%=2^j

For i=0 To 2\*length%-1

If i<106 Then Print a$(((data%(i) And mask%)=mask%));

Next i

Print : Print

Next j

End Sub

EXAMPLE PROGRAM 4

This program runs on RP2350 only and demonstrates the use of the FIFO’s as individual registers. The PIO of the RP2350 has specific instructions to support this MOV RXFIFO[y],ISR and MOV OSR,RXFIFO[y].

MMBasic can read/write the 4 individual FIFO registers by use of:

PIO WRITEFIFO a, b, c, d

PIO READFIFO( a, b, c)

a/ pio (0 or 1)

b/ state machine (0...3)

c/ nbr (FIFO register 0…3)

d/ data% (32 bit integer value)

The program configures the FIFO for individual reads, then writes some values in these register. It starts the PIO that updates 2 of the 4 individual FIFO registers. Then MMBasic reads the values to show only 2 register have changes, and no data shifted (as would happen in RP2040 FIFO).

'only for RP2350 assembler. this does not work on RP2040

pio clear 1

pio assemble 1,".program test"

pio assemble 1,".line 0"

pio assemble 1,"set y,4" 'just a value 4 in Y

pio assemble 1,"mov isr,y" 'copy Y to isr

pio assemble 1,"jmp y--,next" 'y=y-1 always to next

pio assemble 1,"next:" ‘label

pio assemble 1,"mov rxfifo[y],isr" 'should program 4 in fifo [3]

pio assemble 1,"mov isr,y" 'copy Y to isr

pio assemble 1,"mov rxfifo[2],isr" 'should program 3 in fifo [2]

pio assemble 1,"jmp 0" 'repeat

pio assemble 1,".end program"

'pio assemble 1,".end program list"

f=1e6 '1MHz

'PIO(EXECCTRL a,b,c)

e=pio(execctrl gp0,0,31) 'default value, not actually changed

'PIO(SHIFTCTRL a,b,c,d,e,f,g,h,i,j)

sr=pio(shiftctrl 0,0,0,0,0,0,0,0,0,1) 'read individual RX, RX=4 deep

sw=pio(shiftctrl 0,0,0,0,0,0,0,0,1,0) 'write individual RX, RX=4 deep

'PIO(PINCTRL a,b,c,d,e,f,g)

p=pio(pinctrl 0,0,0,gp0,gp0,gp0,gp0) 'defaultvalue , not actually changed

'test the use of FIFO as individual registers

'fill the fifo with pre-determined values

pio init machine 1,0,f,p,e,sw,0 'init machine for writing to RX fifo

for i=0 to &h3 'write the 4 RXFIFO registers

pio writefifo 1,0,i,&h100\*(i+1) 'write values &h100, &h200, &h300, &h400

next

'check the values are written correctly

print "3 RXFIFO registers before running the program"

pio init machine 1,0,f,p,e,sr,0 'init machine for reading the RX fifo

for i=0 to 3 'read the 4 RXFIFO registers

print i,hex$(pio(readfifo 1,0,i)) 'verify they are correctly written

next

print

'run the PIO program. That should (continuously) write to reg 2 and 3 in the FIFO, but not alter register 0 and 1

pio start 1,0

'show the updated FIFO registers

print "3 RXFIFO registers after running the program"

for i=0 to 3 'read the 4 FIFO registers to see if the program works

print i,hex$(pio(readfifo 1,0,i))

next