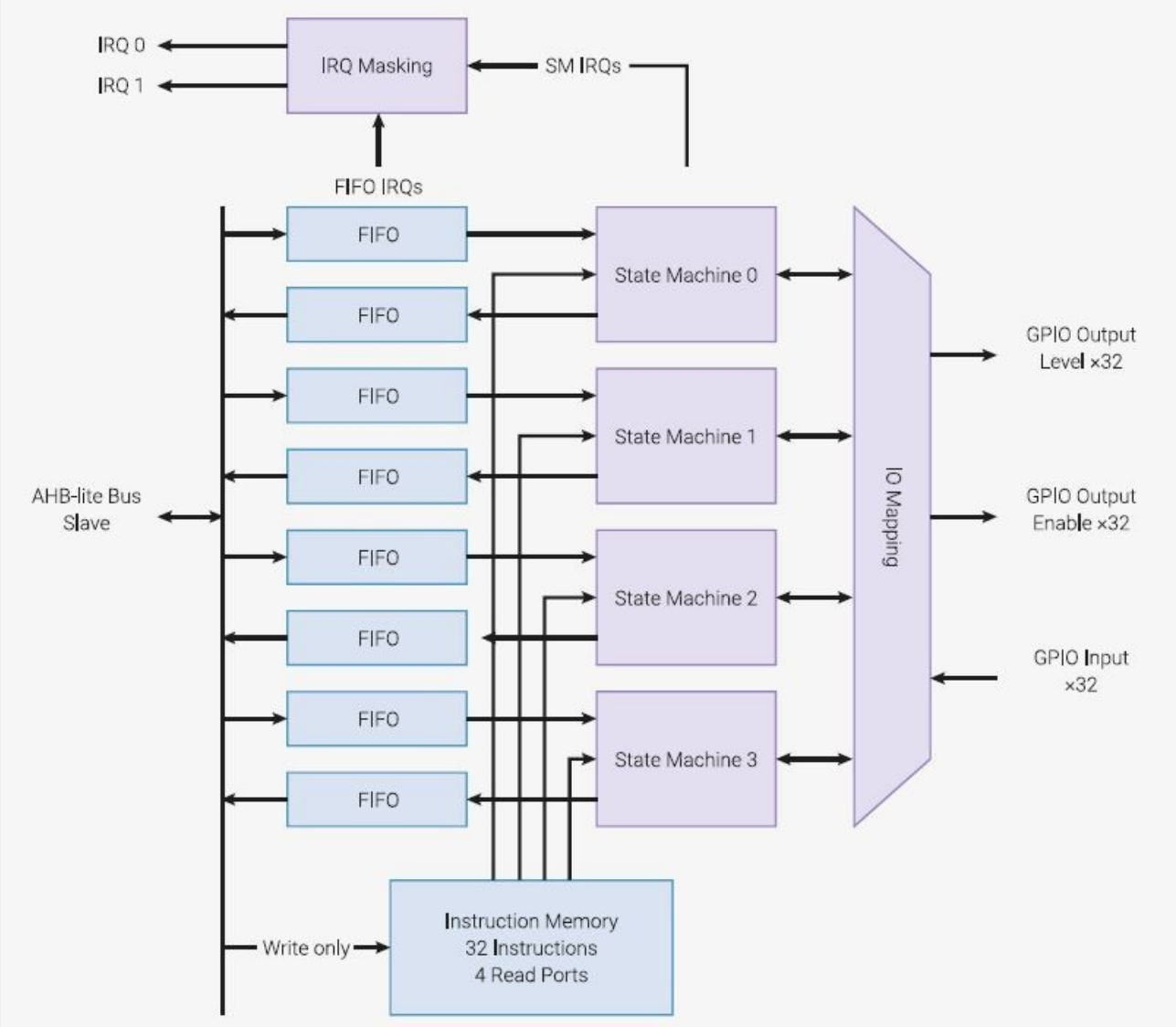
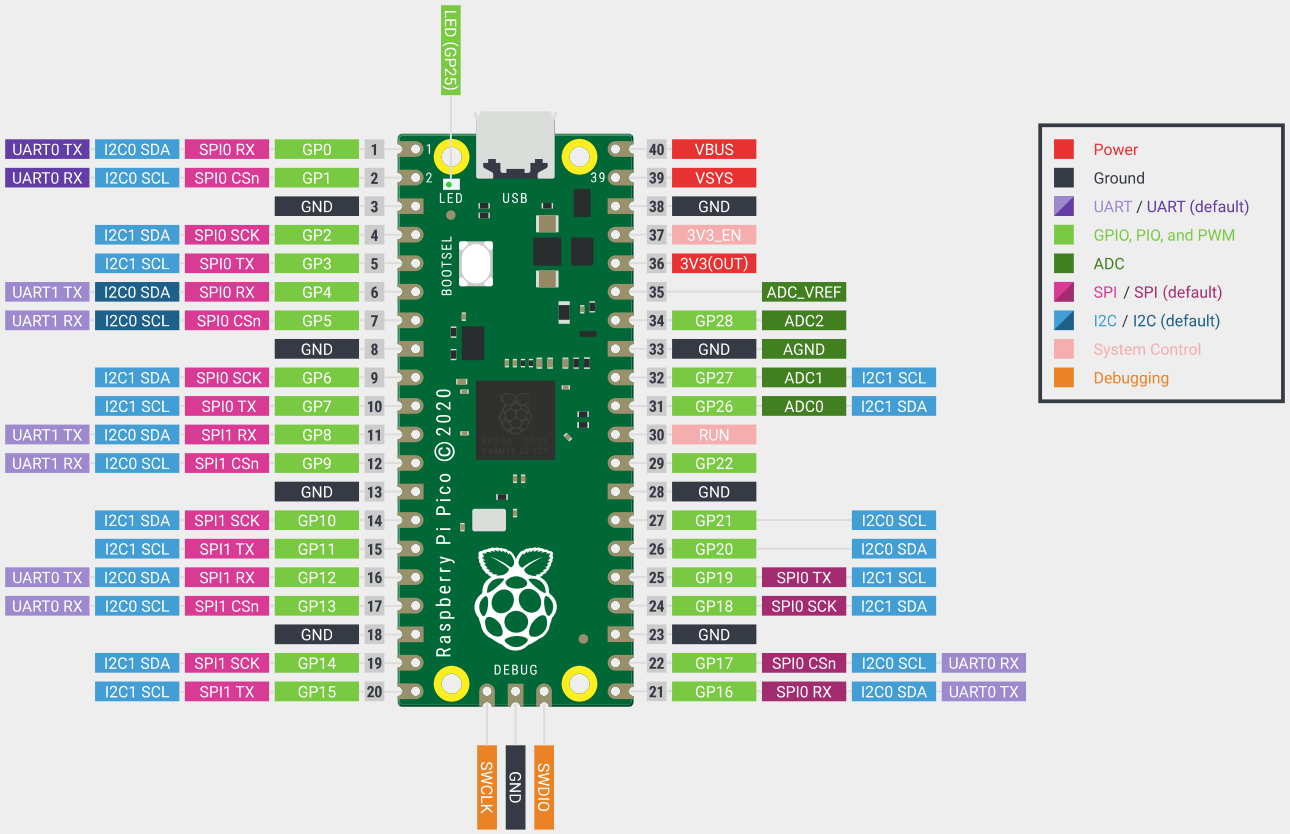
RP2040
PIO assembler



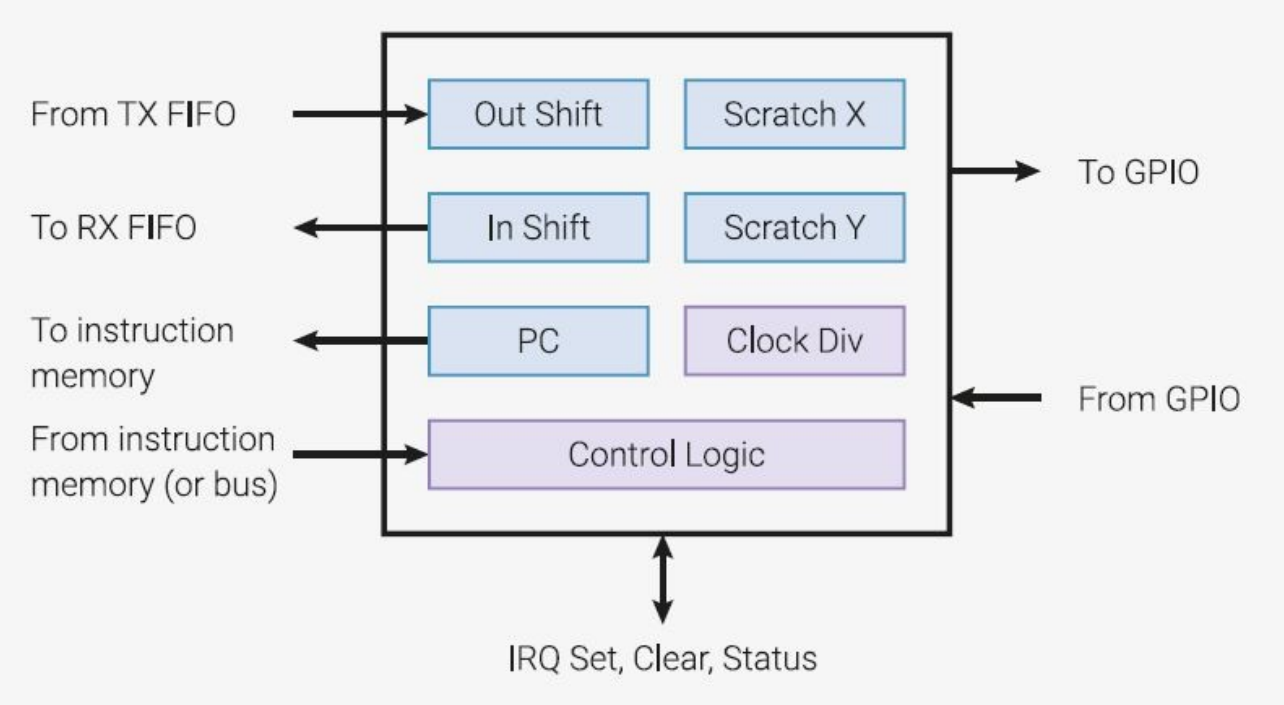
vsn 0.6 © 2023, Willem Ouwerkerk



RPi PICO board connections

PIO introduction

PIO stands for **P**rogrammable **I**nput & **O**utput unit. There are two PIO's in each core. Each PIO has four state machines (see cover). A state machine diagram is shown below. Each PIO has a program memory for 32 opcodes. That looks small, but PIO code is very compact. More than one programme can be stored in this memory. Programmes can be shared by multiple state machines.



A state machine

Starting and finishing a PIO program.

`{PIO ( sm pio – pa s )`	Start new PIO assembly program for state machine 'sm' of PIO block 'pio' and stop it
`PIO} ( pa s -- )`	Close PIO program. When the program is correctly finished, start it

```
clean-pio                   \ Empty code space, start at 0, stop all state machines

0 0 {pio                    \ I/O using sm-0 & PIO-0
    2000 =freq              \ 2kHz clock
    24 =in-pin              \ GPIO 24 as IN pin base
    1 24 1 =inputs          \ GPIO 24 with pull-up
    25 1 =out-pins          \ GPIO 25 as OUT pin base
    24 2 =set-pins          \ Administer both used pins for SET
    2 pindirs set,          \ Pin 24 is input, Pin 25 is output
    wrap-target             \ Copy Pin-24 inverted to Pin-25 & debounce 16 millisec.
        31 []  pins inv pins mov,
    wrap
    0 =exec                 \ Start SM-0 at address 0
pio}
```

PIO opcodes:

IN,	(cnt src --)	Shift 'cnt' bits from source 'src' to ISR Args: PINS X Y NULL ISR OSR
OUT,	(cnt dest --)	Shift 'cnt' bits out of OSR to destination. Args: PINS X Y NULL PC ISR EXEC PINDIRS
PUSH,	(?full ?block --)	Push ISR word to fifo. Optional arguments: BLOCK NOBLOCK IFFULL
PULL,	(?empty ?block --)	Pull word from fifo to OSR. Optional args: BLOCK NOBLOCK IFEMPTY
MOV,	(src ?op dest --)	Move 'src' data to 'dest' modified by optional. Arguments: **Src:** PINS X Y NULL STATUS ISR OSR **Dest:** PINS X Y EXEC PC ISR OSR **?Op:** NORM INV REV
WAIT,	(pol pin arg --)	Wait for a pin or IRQ to become high or low. Arguments: GPIO PIN IRQ LOW HIGH
IRQ,	(?rel irq arg --)	Wait for an IRQ or change the IRQ level using. Arguments:REL SET CLR WAIT NOWAIT
SET,	(value dest --)	Set 'value' to 'dest', 'dest' maybe pins or a register. Arguments: PINS X Y PINDIRS
NOP,	(--)	Pseudonym for: x x MOV,
PIO,	(opc --)	Manually assemble opcode 'opc'

Optional arguments for each opcode:

[]	(+n --)	Add 0 to 31 delay cycles to this opcode
SIDE	(msk --)	Add 'msk' bits as side-set controlled bits

Code example using SIDE & []

```
    begin,                \ Generate WS2812 pulse train
        1 x out,          \   0     0
        1 side  x0<>? if, \   1     1
        1 side  else,     \   1
            nop,          \         0
        then,
    y--? until,           \   0     0
    15 []  31 x set,      \ Long delay: x = 31
```

Control structures:

This PIO assembler has jump back labels & Forth control structures.
Shortcuts: ‘ps’ stands for: PIO-address & security
‘ct’ stands for: Conditional & argument type

ONE (--)	Save code pointer for use with ONE>
TWO (--)	Ditto with TWO>
ONE> (-- ps)	Leave address noted by ONE
TWO> (-- ps)	Ditto with TWO
IF, (ct -- ps)	Forth like control structures
ELSE, (ps0 -- ps1)	
THEN, (ps --)	
BEGIN, (-- ps)	
UNTIL, (ps ct --)	
WHILE, (ps0 ct -- ps1 ps0)	
REPEAT, (ps1 ps0 --)	
AGAIN, (ps --)	
NEVER? (-- ct) X0<>? (-- ct) X--? (-- ct) Y0<>? (-- ct) Y--? (-- ct) X=Y? (-- ct) PIN? (-- ct) OSRE? (-- ct)	Conditionals for PIO control structures Always false True when X <> 0 True when X = 0 before decrease True when Y <> 0 True when Y = 0 before decrease True when X = Y True when PIN? Is low True when OSR is empty

Examples:

```
    7 x set,          \ Shift 8 bits
    begin,
        1 pins out,   \ Shift one bit out
    x--? until,       \ Until 8 bits are done

    begin, again,     \ Endless wait loop

    x0<>? if,         \ X not empty?
        osr isr mov,  \ Save OSR in ISR
    else,
        31 x set,     \ X empty, new value to ISR using X
        x isr mov,
    then,
```

Directives-1:

Setup & initialisation		
CLEAN-PIO	(--)	Prepare program area, filling it with all zero's Set origin to zero & stop all state machines
=PIO	(pio --)	Active PIO block ‘pio’ for all other directives
SM	(f --)	(De)activate current state machine
SYNC	(mask --)	Restart & sync. clock of all sm’s in mask
RESTART	(mask --)	Restart all state machines in mask
=ORG	(pa --)	Set start addres for PIO code
Controlling pins		
=SET-PINS	(pin #p --)	Set base pin for SET and number of addressed pins ‘#p’
=OUT-PINS	(pin #p --)	Set base pin for OUT and number of addressed pins ‘#p’
=SIDE-PINS	(pin #p --)	Set base pin for Side-set and number of addressed pins ‘#p’
OPT	(--)	Use side-set function optional
=IN-PIN	(pin --)	Set base pin for IN
=JMP-PIN	(pin --)	Set input pin for PIN? used with JMP opcode
=INPUTS	(n pin #p --)	Set input type ‘n ‘to' ‘#p' pins from 'pin'
=STRENGTH	(+n pin #p --)	Set output strength ‘+n ‘ to 'pin' & ‘#p’ pins
SIDE-PINDIRS	(--)	Use side-set on PINDIRS instead of pins

Initialising I/O pins:

There is **=INPUTS** for initialising the inputs with pull-ups, etc. For setting the output driver current there is **=STRENGTH**

=INPUTS		=STRENGTH	
n	**Input type**	**+n**	**Output current**
0 (default)	Float	0	2 mA
1	Pull-up	1 (default)	4 mA
-1	Pull-down	2	8 mA
		3	12 mA

```
24 =in-pin              \ GPIO 24 as input pin base
24 2 =set-pins          \ GPIO 24 & 25 are used pins
1 24 1 =inputs          \ Add pull-up to GPIO 24
2 25 1 =strength        \ Set GPIO 25 to maximal 8 mA
```

Directives-2:

Wrap		
WRAP-TARGET	(--)	After reaching WRAP address execution continues here
WRAP	(--)	Go on executing at WRAP-TARGET address
Fifo		
=STEAL	(+n --)	Steal fifo space from TX=1 or RX=2 fifo, 0= no steal
=AUTOPUSH	(+n f --)	When ‘f’ is true, push ISR to fifo after +n bits are shifted in, when false only set count +n
=AUTOPULL	(+n f --)	When ‘f’ is true, pop fifo to OSR after +n bits are shifted out, when false only set count +n
=IN-DIR	(0|1 --)	ISR shift direction, 1 = right, 0 = left
=OUT-DIR	(0|1 --)	OSR shift direction, 1 = right, 0 = left
Special functions		
CLONE	(sm --)	Copy the first three state machine registers from 'sm' to the current state machine. See example file: **uart-1.f**
=EXEC	(opc --)	Execute opcode on current state machine
=CLOCK-DIV	(div --)	Set clock divider, 251 means divide sysclock by 2.51
=FREQ	(Hz --)	Set the clock to this frequency, give an error message if this is not possible
EXPORT	(--)	Export current PIO code into loadable ASCII

Special functions:

Three functions deal with setting the clock frequency for a state machine. First set the correct system clock using **=SYSCLOCK** The **=CLOCK-DIV** function gets the divisor as an integer on the stack, 3750 means a divisor of 37.5!

Use **CLONE** if the same programme is to run on multiple state machines. It copies the first three registers from the specified state machine to the current state machine. In doing so, it copies the main settings.

The word **=EXEC** allows you to execute any opcode directly on the current state machine of the active PIO. Finally an example of shift register settings.

```
    8 1 =autopush  8 1 =autopull \ Push/pull data after 8-bits are done
    0 =out-dir 0 =in-dir    \ Both shift registers shift data to left

    27 1 =out-pins          \ GPIO 27 output for shifted data
    28 =in-pin              \ GPIO 28 input for shifted data
```

Decompiler & PIO-info:

MPSEE	(pa --)	Show PIO assembly from address 'pa'
PSEE	(--)	Show PIO assembly from address zero
.FIFO	(--)	Show fifo status of all fifo's
.SM	(sm --)	Show condition of state machine 'sm'

PSEE example:

```
psee
 0: E083 set  pindirs 3
 1: E003 set  pins 3
 2: 0002 jmp  to: 2
 3: FF01 set  pins 1  side 1  [7]
 4: E75F set  y 1F  [7]
 5: E700 set  pins 0  [7]
 6: 0785 jmp  y-- to: 5  [7]
 7: 0003 jmp  to: 3   ok.
```

PIO external programming tools:

TX-DEPTH	(sm -- +n)	Leave used TX-fifo space, +n=3 when full
RX-DEPTH	(sm -- +n)	Leave used RX-fifo space, +n=3 when full
>TXF	(u sm --)	Push 'u' to TX fifo
RXF>	(sm -- u)	Pull 'u' from RX-fifo
SET-PIO	(pio --)	Select used PIO (0 or 1)
SM-ON	(f sm --)	(De) activate state machine ‘sm’
EXEC	(opc sm --)	Execute ‘opc’ on state machine ‘sm’
FREQ	(freq sm --)	Set state machine ‘sm’ to ‘freq’ in Hz
CLOCK-DIV	(u sm --)	Set clock divider of state machine ‘sm’
SYNC	(sms --)	Sync. clocks of all state machines ‘sms’
RESTART	(sms --)	Restart all state machines ‘sms’

Feeding the fifo with data in a safe way:

```
: PEMIT ( ch -- )   \ Character to PIO UART
    begin  0 tx-depth  3 < until \ Ready when TX fifo-0 not full?
    0 >txf ;        \ Push ‘ch’ to fifo-0
```

Note! When writing PIO programmes, be aware that the **SET,** opcode is usually used to initialise the direction for IO pins using **PINDIRS**. So do not forget the **SET** initialisation for all used pins. An example:

```
    26 1 =side-pins          \ GPIO 26 for side-set
    27 1 =out-pins           \ GPIO 27 for OUT
    28 =in-pin               \ GPIO 28 for IN
    26 3 =set-pins           \ We may use pin 26 to 28 for SETDIRS !!
```

Mini PIO:

When you generate a PIO-program using the **EXPORT** function, you may load the file '**mini-PIO.f**' in advance. This gives you the most important external PIO control functions!

These functions are shown in the table 'PIO external programming tools'.
Here is a program that toggles a LED, it can be controlled using the definitions below it.

```
hex
: TOGGLE1   \ GPIO-25 LED control
    0000 50200000 !
    1F000 502000CC !
    14000000 502000DC !
    F4240000 502000C8 !
    14000320 502000DC !
    8000320 502000DC !
    0006 400140CC !
    0006 400140D4 !
    E083 50200048 !
    0001 5020004C !
    1F100 502000CC !
    E703 50200050 !
    E75F 50200054 !
    E700 50200058 !
    0784 5020005C !
    5100 502000CC !
    0000 502000D8 !
    0001 50200000 !
    0000 set-pio ;
toggle1

: FLASH    2 0 exec ;      \ Jump to address 2, start flasher
: LED-OFF  1 0 exec  E000 0 exec ;  \ Pin 25 off, jump to wait loop
: LED-ON   1 0 exec  E001 0 exec ;  \ Pin 25 on, jump to wait loop
: TEMPO    7D0 max  0 freq  flash ; \ Change flasher frequency
```

Programming example-1: UART output

```
: =BAUD ( b -- )    8 *  =freq ; \ Set baudrate for UART

clean-pio                   \ Empty code space, start at zero
0 0 {pio                    \ Use state machine-0 on PIO-0
    115200 =baud            \ 115k2
    26 1 =side-pins  opt    \ GPIO 26 for optional SIDE
    26 1 =out-pins          \ GPIO 26 for OUT & SET
    26 1 =set-pins
    1 pindirs set,              \ Pin is output!
    wrap-target
        7 []  1 side  pull,     \ Stop bit, get data byte
        7 []  0 side 7 x set,   \ Start bit
        begin,
            1 pins out,         \ Shift 8 bits out
        6 []  x--? until,       \ Until 8 bits are done
    wrap
    0 =exec                 \ Start SM-0 code at address 0
pio}
```

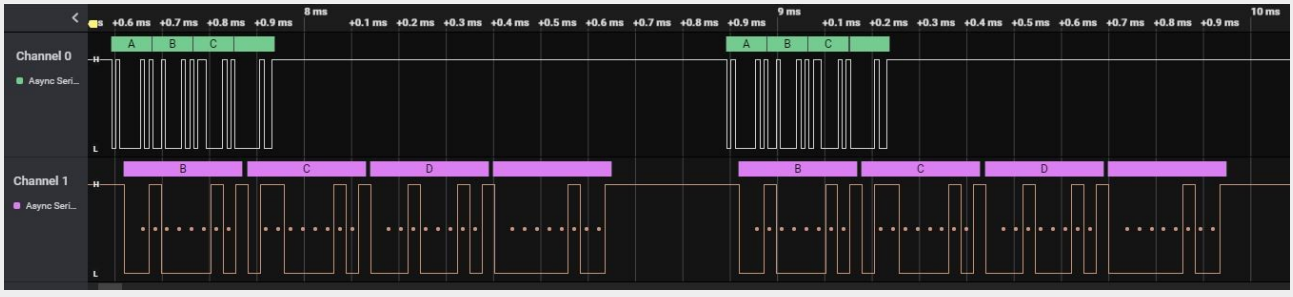
Forth UART control code:

```
: PEMIT ( ch -- )   begin  0 tx-depth  3 < until  0 >txf ;
: PTYPE ( a u -- )  0 ?do  count pemit  loop  drop ;
: PICO  ( -- )      s" RP2040 " ptype ;
```

Decompiled UART:

```
psee
 0: E081 set  pindirs 1
 1: 9FA0 pull block  side 1  [7]
 2: F727 set  x 7  side 0  [7]
 3: 6001 out  pins 1
 4: 0643 jmp  x-- to: 3  [6]
```

Dual UART output with PIO:



Programming example-2: Controlling an output pin:

```
\ Slow pulse on the LED mounted on GPIO 25 & 26 with wrapping
clean-pio                   \ Erase code space, start at zero
0 0 {pio                    \ Use state machine-0 on PIO-0
    2000 =freq              \ On 2000 Hz frequency
    25 2 =set-pins          \ GPIO 25 & 26 for SET

    3 pindirs set,              \ Both pins are outputs
    one  begin, again,          \ Wait loop, remember address
    two  wrap-target
        7 [] 3 pins set,        \ LED on, pin 25 & 26 on
        7 [] 31 y set,          \ Max. delay using Y
        begin,
            7 [] 0 pins set,    \ LED & pin off
        7 [] y--? until,        \ Wait longer
    wrap
    0 =exec                 \ Start SM-0 at address 0
pio}
```

Program control code:

This example uses the direct execute function built into every state machine.

```
hex
: FLASH    two> 0 exec ;      \ Jump to address 2, start flasher
: LED-OFF  one> 0 exec  E000 0 exec ; \ Pin 25 off, jump to wait loop
: LED-ON   one> 0 exec  E001 0 exec ; \ Pin 25 on, jump to wait loop
```

Decompiled LED control:

```
psee
 0: E083 set  pindirs 3
 1: 0001 jmp  to: 1
 2: E703 set  pins 3  [7]
 3: E75F set  y 1F  [7]
 4: E700 set  pins 0  [7]
 5: 0784 jmp  y-- to: 4  [7]
```

Slow (3.8 Hz) pulses read from GPIO 26:

Programming example-3: Reading a rotary encoder

One readout takes 2.5 milliseconds, the pins 26 to 28 are used as inputs with pull-up resistors. Pin 28 is the press action! It can be coded much shorter in Forth, however, it is a great example of using one or more inputs. Also it shows alternative use of the shift register to isolate a bit field.

```
clean-pio                   \ Empty code space, etc.
0 0 {pio                    \ Encoder readout on sm-0 & PIO-0
    2000 =freq              \ 2 kHz clock
    26 =in-pin              \ GPIO 26 as input pin base
    1 26 3 =inputs          \ Pull-up on all inputs
    26 3 =set-pins          \ Three pins used
    0 =out-dir              \ Shift OSR left
    0 pindirs set,          \ Only 26 to 28 are  input
    wrap-target
        7 y set,            \ Set no input value
        begin,
            pins osr mov,   \ Read inputs to OSR            .5
            29 null out,    \ Shift result to high 3 bits   .5
            3 x out,        \ Move to low 3 bits of X       .5
        x=y? while,         \ No change?                    .5
        repeat,             \ Read again                    .5
        x isr mov,          \ Change noticed, X to ISR
        push,               \ Output data to RX-fifo
    wrap
    0 =exec                 \ Start SM-0 at address 0
pio}
```

Rotary encoder:

```
: ENCODER?  ( -- f )   0 rx-depth  0= 0= ; \ True when encoder is used

\ 0=No movement, 1=Forward, -1=Backward, Press = true (Knob pressed)
: ENCODER   ( -- 0|1|-1 press )
    begin  encoder? until   \ Knob moved
    0 rxf>                  \ Read data
    dup 4 and 0= >r         \ Save knob pressed
    3 and  dup 3 <> if      \ Knob turned?
        2 <>  -2 and 1+  r> exit \ 1 or -1  and press
    then
    dup -  r> ;             \ 0 and press
```

Example program:

```
: DEMO      ( -- )          \ Show pulses up, down & knob pressed
    0  begin                \ Start at zero
        encoder if ." Press " then  + \ Show knob pressed
        dup .               \ Show counted pulses
    key? until  drop ;
```

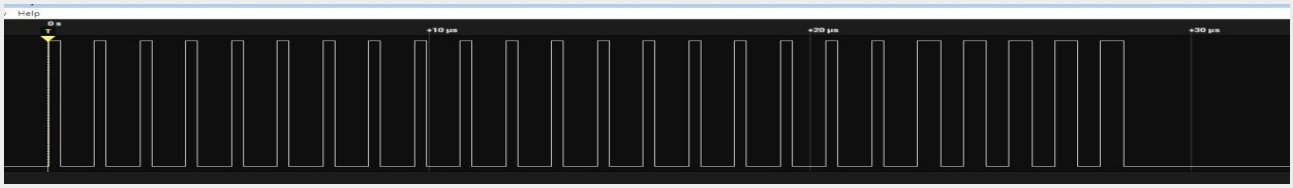
Programming example-4: Single WS2812 LED controller

This program uses a 3.333MHz clock for efficiënt WS2812 pulse train generation, and has three parts:

Part 1, Generates a new 24-bits LED color, uses ISR for temporary storage.
Part 2, Outputs the 24-bits pulse pattern (color code)
Part 3, A long delay using **12 lshift** with the OSR!

```
clean-pio                   \ Empty code space, etc.
0 0 {pio                    \ Use state machine-0 on PIO-0
    3333333 =freq           \ On 3.333.333 Hz frequency
    23 1 =side-pins         \ GPIO 23 for SIDE-SET & SET
    23 1 =set-pins  0 =out-dir \ OSR shift left
    1 pindirs set,
    begin,
        isr osr mov,        \ (1) Copy LED color to OSR
        8 x out,            \ Shift left one byte
        osr x mov,
        x0<>? if,           \ OSR not empty?
            osr isr mov,    \ Save new color in ISR
        else,
            31 x set,       \ OSR empty, start color in ISR
            x isr mov,
        then,
        23 y set,           \ (2) Output 24 bits to WS2812!
        begin,              \ With a specific pattern
\                              One   Zero
            1 x out,        \   0     0
            1 side  x0<>? if, \ 1     1
            1 side  else,   \   1
                nop,        \         0
            then,
        y--? until,         \   0     0
        15 [] 31 x set,     \ (3) Long delay: x = 31
        x osr mov,          \ OSR = 31
        12 x out,           \ OSR = 31 x 4096 = 126976
        osr x mov,          \ Delay = 126796x16 = 2028736/3333 ~ .6 sec.
        begin,  15 []  x--? until,
    again,
    0 =exec                 \ Start SM-0 at address 0
pio}
```

One 24-bits WS2812 bit stream:



Programming example-5: SPI in & out

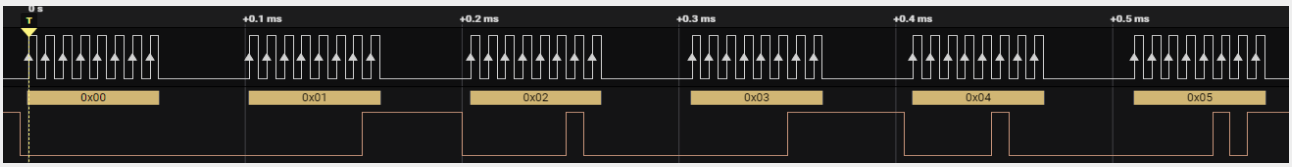
This program does bidirectional SPI with a zero clock phase. Auto-push and auto-pull are enabled after 8-bits are done. Calculating the bitrate:

500 kHz/4 PIO clocks = 125 kBit

```
\ Single SPI on state machine 0 and PIO 0, Clock phase = 0
\ SCK  = Side-set pin 0, SPI clock = 125 kHz
\ MOSI = OUT pin 0
\ MISO = IN pin 0
clean-pio                   \ Empty code space, etc.
0 0 {pio                    \ Use state machine-0 on PIO-0
    500000 =freq            \ 500 kHz
    8 1 =autopush 8 1 =autopull \ 8-bits data records
    0 =out-dir  0 =in-dir   \ OSR & ISR shift left
    26 1 =side-pins         \ GPIO 26 for SCK (SIDE)
    27 1 =out-pins          \ GPIO 27 for OUT
    28 =in-pin              \ GPIO 28 for IN
    1 28 1 =inputs          \ Pull-up on input
    26 3 =set-pins          \ Three pins used

    3 pindirs set,                  \ Pin 26 & 27 are outputs
    wrap-target
        1 []  0 side  1 pins out,   \ Output one bit, clock low
        1 []  1 side  1 pins in,    \ Input one bit, clock high
    wrap

    0 =exec                 \ Start SM-0 at address 0
pio}

: >SPI  ( b -- )    \ Byte to SPI
    begin  0 tx-depth  3 < until \ Space in fifo?
    24 lshift  0 >txf ;     \ Yes, move data to highest byte & send

: SPI>  ( -- b )    \ Byte from SPI
    begin  0 rx-depth until \ Received data present in fifo?
    0 rxf> ;                \ Yes, read out
```

Example program:

```
\ Show SPI transport, connect pin 27 to pin 28 for this test
: DEMO  ( u -- )
    0 ?do  cr i .  i >spi  spi> .  loop ;
```



Programming example-6: Using IRQ to transfer data

Program-1, reads a switch and sets IRQ-1 low when the switch is pressed. It adds delays for debouncing of the switch.

```
clean-pio                   \ Empty code space, etc.
0 0 {pio                    \ Use state machine-0 on PIO-0
    2000 =freq              \ On 2000 Hz frequency
    24 =jmp-pin             \ GPIO 24 as input pin base
    24 1 =set-pins
    1 24 1 =inputs          \ Pull-up on input

    0 pindirs set,              \ Pin 24 is input
    begin,
        31 [] pin? if,          \ Pin 24 low?
            1 clr irq,          \ Clear interrupt
            31 [] high 24 gpio wait, \ Wait until pin 24 high
        then,
    again,

    over =exec              \ Start sm-0 code at address 0
pio}
```

Program-2, the opcode **1 wait irq,** sets IRQ-1 high. It then waits for IRQ-1 to become low. After it becomes low GPIO 25 is toggled.

```
1 0 {pio    \ Toggle LED after an IRQ
    2000 =freq          \ On 2000 Hz frequency
    25 =in-pin          \ GPIO 25 as input & output pin base
    25 1 =out-pins
    25 1 =set-pins      \ GPIO 25 for SET

                            \ Program starts here
    1 pindirs set,          \ Pin 25 is output
    begin,
        1 wait irq,         \ Wait until IRQ 1 is low
        pins inv pins mov,  \ Invert pin 25, LED on/off
    again,

    over =exec          \ Start sm-1 at start address of this code
pio}
```

Links to interesting documents:

RP2040-datasheet.pdf	Datasheet
Getting Acquainted with PIO	Tutorial website
PIO examples	Examples on Github

Programming example-7: Using EXPORT

```
clean-pio  decimal          \ Empty code space mirror, etc.
\ Slow pulse on the LED mounted on GPIO 25 & 26 with wrapping
0 0 {pio                    \ Use state machine-0 on PIO-0
    2000 =freq              \ On 2000 Hz frequency
    25 2 =set-pins          \ GPIO 25 & 26 for SET
    3 pindirs set,          \ Both pins are outputs
    begin,
        7 [] 3 pins set,    \ LED on, pin 25 & 26 on
        7 [] 31 y set,      \ Max. delay using Y
        begin,
           7 [] 0 pins set, \ LED & pin off
        7 [] y--? until,    \ Wait longer
    again,
    0 =exec                 \ Start code at address 0
pio}
```

Using the **EXPORT** function we get a stand alone PIO program. You have to catch the code in your serial terminal!
Then first load the file '**mini-PIO.f**', after that load the code below en we get a flashing LED on GPIO-25 and a pin toggle on GPIO-26.

```
hex
: PIO-PROG
    0000 50200000 !
    1F000 502000CC !
    14000000 502000DC !
    F4240000 502000C8 !
    14000320 502000DC !
    8000320 502000DC !
    0006 400140CC !
    0006 400140D4 !
    E083 50200048 !
    E703 5020004C !
    E75F 50200050 !
    E700 50200054 !
    0783 50200058 !
    0001 5020005C !
    0000 502000D8 !
    0001 50200000 !
    0000 set-pio ;

pio-prog
```

When you want a flashing LED on noForth just type after loading the previous:

‘ pio-prog to app freeze

And noForth starts up next time with this LED flashing...