TAQOZ Reloaded v2.8 - Writing Inline Assembly Code

TAQOZ Reloaded is a powerful language, so why inline assembly code?

- When your TAQOZ Reloaded application still runs too slowly e.g. when streaming audio or video. After measuring execution times of words in the critical path with LAP and .LAP, converting some of those words to assembly language may get you within spec. Aim to do the minimum necessary.
- TAQOZ Reloaded with it's inline assembler makes a highly interactive sandbox to experiment with various functions, reducing development time.
- There is often no need to write a test harness around the function under test - just manual entry from the TAQOZ command line is enough to check the function is working. Your edit-compile-test cycle is much quicker than other languages - as each word comes to life. More fun means you may be more productive.

There is a downside - each assembly language instruction consumes 32 bits of memory whereas TAQOZ high level words only need 16 bits. The assembler takes up precious space in memory.

The TAQOZ Interactive Assembler (TIA) in TAQOZ Reloaded is intended for creating relatively short pieces of assembly code, in line with good forth practice. No awful 'War and Peace' sized functions, thank you very much.

The following notes were enough baby steps to get me writing inline code for a dsp project.

Obtaining the Assembler

Check that the assembler is already present in your tool set. If the TAQOZ Reloaded start up message includes something like...

4890 *P2ASM*   TAQOZ INTERACTIVE ASSEMBLER for the PARALLAX P2 - 210124-1200

...then the assembler is already there. Otherwise, from the Tachyon Forth dropbox, find P2ASM.FTH in folder Tachyon/P2/TAQOZ/Forth. Upload this file to TAQOZ and check it compiles with no errors. You can optionally keep the assembler resident in the tool set by typing BU <enter> to back up the complete system.

P2 Reference Documents

The P2 instruction set, Assembly Language Manual and the Propeller 2 Documentation are essential aids to inline assembly programming. A paper Tachyon Forth Model by Peter Jakacki is also useful.

Instruction Syntax

P2 instructions in general are written as:-

{_RET_} or { conditional }	ROR	Destination,	{#}Source	{wc/wz/wcz}
optional return to TAQOZ or a conditional	Instruction name		Optional # denotes constant value	Optional flags affected as a result of the instruction

Any instruction may be made conditional with:-

<table>
<tr><td>Conditional name including aliases</td><td>Flags</td></tr>
<tr><td>if_nz_and_nc
if_a
if_00
if_nc_and nz</td><td>nc & nz</td></tr>
<tr><td>if_z_and_nc
if_01
if_nc_and_z</td><td>nc & z</td></tr>
<tr><td>if_ae
if_0x
if_nc</td><td>nc</td></tr>
<tr><td>if_nz_and_c
if_10
if_c_and_nz</td><td>c & nz</td></tr>
<tr><td>if_ne
if_x0
if_nz</td><td>nz</td></tr>
<tr><td>if_z_ne_c
if_diff
if_c_ne_z</td><td>c <> z</td></tr>
<tr><td>if_nz_or_nc
if_not_11
if_nc_or nz</td><td>nc | nz</td></tr>
<tr><td>if_11
if_z_and_c
if_c_and_z</td><td>c & z</td></tr>
<tr><td>if_same
if_z_eq_c
if_c_eq_z</td><td>c = z</td></tr>
<tr><td>if_e
if_x1
if_z</td><td>z</td></tr>
</table>

if_z_or_nc if_not_10 if_nc_or_z	nc | z
if_b if_1x if_c	c
if_not_01 if_nz_or_c if_c_or _nz	c | nz
if_not_00 if_be if_z_or_c if_c_or_z	c | z

N.B. Totally blank lines are not permitted within an assembly definition - Christof Eb reports they cause 'stack mismatch' error messages from the Assembler.

Defining an inline assembly word

A simple inline assembly word is bracketed with **code** and **end** like this:-

```
code MYCODE                    'keyword 'code' followed by the name of the new word
        ( assembly words )
        ret                    'return to the TAQOZ interpreter
end                            'keyword 'end' signals the end of the assembly word
```

Back and Forth within the Definition

The words **ASM:** and **FORTH:** make possible the mixing of forth and assembly code in the same definition. e.g.

```
pub MYWORD ( ch -- )           --- N.B. This currently does not work – word FORTH: is buggy
                ... forth words ...
        ASM:
                ... assembly language
        FORTH:
                .. forth words ...
;
```

or

```
pub MYWORD ( ch -- )
                ... forth words ...
        ASM:
                ... assembly language
                jmp #\@<forth word name> or ret
end
```

Here's a working example of this, created by Christof Eb on the Parallax forum, which should get you going with your own 'mixed' words:-

```
FORTH ASSEMBLER
pri PCNEXT  _pc @ 12 + ;
' CALL 8 + W@ 1- := doNEXT            --- points to the start addr of the forth interpreter

pub FORTH:                            --- bug fix for FORTH: word
  end [C] ] ;
FORTH


: testD              ( – )            --- test word for mixed assembler and forth
  1                                   --- initialise a counter with the value 1 on the data stack
  begin
    ASM:
       add a,#1                       --- counter now 1 + 1 = 2
       mov PTRA,##PCNEXT
       jmp doNEXT
    FORTH:
      crlf dup .
    ASM:
       add a,#1                       --- add 1 to the counter
       mov PTRA,##PCNEXT
       jmp doNEXT
    FORTH:
      crlf dup .                      --- copy and display the counter
      1000 ms                         --- do nothing for 1 s
  dup 10 > until                      --- and loop to begin until counter = 11
  drop                                --- clean up the data stack
;
```

TAQOZ register usage

The following registers are available for use within inline assembly language:-

xx
yy
zz also known as r0 or r1
r2
r3
r4
acc also known as ac

ptra useful for indexing arrays and strings
ptrb useful for indexing arrays and strings - used by TAQOZ so value needs preserving

During the reading or writing of data to hub memory, ptra and ptrb can incremented and decremented by the correct number of bytes using the following keywords:-

ptra points to the address, no pointer increment or decrement
ptra++ after the instruction has executed, ptra is incremented by byte, word or long as appropriate
++ptra before the instruction executes, ptra is incremented by byte, word or long as appropriate
ptra-- after the instruction has executed, ptra is decremented by byte, word or long as appropriate
--ptra before the instruction executes, ptra is decremented by byte, word or long as appropriate

The same grammar applies to ptrb e.g ptrb++ or --ptrb

Kernel registers

Defined registers in the forth kernel are:-

_depth, _pin, sck, mosi, miso, ss, reg4, reg5, reg6

Labels

To enable jumps in an assembly based word, TIA provides 8 fixed name labels **l0** to **l7** (lowercase L) which is plenty, because they can all be reused once each definition is complete. This is the syntax for labels:-

```
code MYWORD
        ... assembly language ..
.l2     ... assembly language ...               ' this line has a label l2
        ... assembly language ...
        djnz    r1,#l2                          ' this line includes a jump to l2
        ret
end
```

N.B. **Backward jumps** can be made using any type of jump instruction. **Forward jumps** are only supported for the DJNZ instruction. Thanks for that tip, Christof Eb. Christof also mentions that the SKIP instruction can be useful for forward jumps e.g.

```
cmp d,#2 wz
if_z skip #%11111111                    ' skips next 8 instructions
```

Looping Macros

As an aid to creating loops in a more high level fashion, these macros are aimed at easing a new assembly language user into simple looping without fuss. They're not as versatile as the various branching assembly language instructions and they don't translate to other languages that support inline coding.

The FOR: NEXT: macros

Example code:-

```
code ffibo      ( n1 -- n2 )            ' n2=n1'th fibonacci number
                        mov     xx,#0
                        mov     yy,#1
                        FOR:
                                        add     yy,xx
                                        djnz    a,#l2
                                _ret_   mov     a,yy
                                .l2     add     xx,yy
                        NEXT:   a
                _ret_   mov     a,xx
end
```

So the '**NEXT: a'** statement decrements a (the top of data stack) and jumps back to the '**FOR:**' statement if a is non-zero.

The BEGIN: AGAIN: UNTIL: macros

BEGIN: ... AGAIN: is an endless loop and is used like BEGIN ... AGAIN in high level Taqoz.
BEGIN: .... UNTIL: <condition> is a loop that ends, where <condition> is one of WC , WZ or WCZ

The Data Stack

top of stack is named as	a
tos + 1 is named as	b
tos + 2 is named as	c
tos + 3 is named as	d

Words that don't alter the stack depth

The less the stack is disturbed, the better from a speed point of view. If it can be arranged that the stack depth doesn't change, then the inline word will run the fastest. So, a word to double the value on the top of stack is:-

```
code MYDOUBLE     ( n1 -- n2 )      'n2 = n1 * 2
        _ret_  shl a,#1             'shift tos one place left and return to TAQOZ
end
```

Notice that the **ret** to the TAQOZ interpreter is combined with the last instruction.

Here's a more complicated example that multiplies two 32 bit unsigned values together to produce a 64 bit result. Notice there are two input parameters and one output parameter. However the inputs take up the same space as the output, so the stack depth doesn't need to change:-

```
code MUL                                        ( b a -- a*b as a double )
        getword     xx,a,#1
        getword     yy,b,#1
        mov         zz,xx
        mul         zz,yy
        mul         xx,b
        mul         yy,a
        add         xx,yy  wc
        getword     yy,xx,#1
        bitc        yy, #16
        shl         xx,#16
        mul         a,b
        add         xx,a  wc
        addx        yy,zz
        mov         b,xx
 _ret_  mov         a,yy
end
```

Dropping values from the stack

This example makes a jump to @DROP to drop the top value from the stack and return to the TAQOZ interpreter. This example also shows two TAQOZ constants being used in the inline word:-

```
56 := SCL
57 := SDA

code MYTX16                     '( word -- )
        rev a
        shr a,#16
        wypin a,#SDA
        wypin #32,#SCL
        jmp #@DROP              'an equivalent macro is DROP;
end
```

To drop the top two values, use relative address jmp #@2DROP
To drop the top three values use jmp #@3DROP
The @ is interpreted as "find the address of <forth definition>", so very useful.

To jump to an absolute address write this as "#\@DROP" :-

```
code WSTX
        ... assembler language ...
        jmp             #\@2DROP
end
```

These endings are used often, so there are three macros **DROP; 2DROP;** and **3DROP;** that can be substituted for the 'jmp #@DROP end' lines.

Pushing new values to the stack

When your word needs more space on the stack for results e.g. Here's a function that needs no inputs, but just pushes the value 1 to top of stack:-

```
code ONEOFAKIND
        >PUSHX                  ' make room for a new tos entry
        _ret_ mov a,#1          ' and set that entry = 1
end
```

Other convenience words

Other convenience words used like >PUSHX are:-

>ROT	(a b c -- b c a)	move the third entry to top of stack
>SWAP	(n1 n2 -- n2 n1)	swap the top two stack entries
>I	(-- n1)	returns the current loop index n1
>SPIRD	(n1 -- n2)	Read 8-bits left into n1 so that n2 = n1<<8+new. Four successive SPIRDs will receive 32-bits
>SPIWB	(byte --)	Shift 8 bits from data[0..7] out and leave data on stack (restored with other bytes zeroed)
>SPICE	(--)	

Reading and Writing to hub Memory

There are three memory regions: cog RAM, lookup RAM, and hub RAM. Each cog has its own cog RAM and lookup RAM, while the hub RAM is shared by all cogs. (RAM - random access memory)

Memory Region	Memory Width	Memory Depth	Instruction D/S Address Ranges	Program Counter Address Ranges
COG	32 bits	512	$000..$1FF	$00000..$001FF
LOOKUP(LUT)	32 bits	512	$000..$1FF	$00200..$003FF
HUB	8 bits	1,048,576 (*)	$00000..$FFFFF	$00400..$FFFFF

So this is writing and reading to a long variable in hub memory (read and write word or byte would be very similar):-

```
long MYVARI

code MYWRITE       ( n1 -- )          ' writes n1 to variable MYVARI
        wrlong a,#MYVARI
        jmp #\@DROP                    ' you can just use DROP; here
end

code MYREAD ( -- n1 )
        >PUSHX                         'make room for the result
_ret_   rdlong a,#MYVARI               ' set tos=MYVARI
end
```

Using indirect addressing to make a more useful function

Reading and writing to a specifically named variable is OK for accessing global variables, but more usual is to read and write to an address supplied as an input on the stack. This makes for a more useful word to access any variable:-

```
code MYINC          ( adr --- )     ' increment long at adr
        mov PTRA,a                  ' PTRA = adr
        rdlong a,ptra               ' read long at adr
        add a,#1                    ' increment the long
        wrlong a,ptra               ' write it back to the adr
        DROP;
end
```

Using indirect addressing to work on arrays

This is where assembly language really starts to pay off in speed: e.g. A dsp application requires numerous looped functions to work on arrays of signal data. These looped functions are written in assembly code. The result is that most of the processors' time is spent in the loops doing the dsp maths and very little time is spent in the TAQOZ interpreter between the dsp words. The application is written in TAQOZ, but runs nearly as fast and efficiently as an all-Assembly application. BUT, it's been written and tested one function at a time - so much less of a monster to commission.

Here's an inline assembly word and identical forth word, working on a byte array:-

```
100 bytes KAKA

' increment all elements of byte array at adr
code MYINC                          '( arraysize array -- )
        mov PTRA,a                  ' PTRA = adr
        .l0                         ' start our array processing loop
                rdbyte a,ptra       ' read byte at adr
                add a,#1            ' increment the byte
                wrbyte a,ptra++     ' write it back to the adr and increment PTRA
                djnz b,#l0          ' and do above for all array elements
        2DROP;
end

' increment all elements of byte array at adr
pub MYINCFORTH                      ( arraysize array -- )
        SWAP
        FOR
                DUP  I +            --- calculate the address
                C++                 --- increment array elements
        NEXT
        DROP                        --- and clean up
;
```

Compare the speed of the forth versus inline assembly:-

```
TAQOZ# 100 KAKA LAP MYINC LAP .LAP --- 5,096 cycles= 25,480ns @200MHz ok
TAQOZ# 100 KAKA LAP MYINCFORTH LAP .LAP --- 38,136 cycles= 190,680ns @200MHz ok
TAQOZ# 38136 5096 / . --- 7  ok
```

So, around a 7x improvement in speed achieved using only 'rooky' code knowledge.

Running code from COG RAM

Christof Eb from the Parallax forum provided this very useful coding tip: If you really want a TAQOZ word to run as fast as possible, then it can be run from COG Ram. This executes faster than Hub Ram. Here we define a 'test' word to try it out:-

```
code test ( value n -- value+3*n 0 )
        add b,#3
        sub a,#1 wz
        if_nz jmp #\@COGMOD     ' loop from start
        ret
end
```

Now we copy this 'test' word from Hub Ram to COG Ram with the command:-

```
AT test 2+ 4 LOADMOD          \ loads the asm instructions of 'test' into COG Ram
```

Now we can execute the code from COG Ram with:-

```
10 5 COGMOD . .               \ executes the loaded code
```

Nice one!

Currently, LOADMOD / COGMOD uses COG Ram starting at $1CB. When tested, a code word containing up to 40 instructions can be transferred to COG Ram. Any more than that and TAQOZ stops working.

COG Ram memory map

Christof Eb supplied this map of the Cog Ram, which applies to the current version *TAQOZ RELOADED sIDE* V2.8 'CHIP' Prop_Ver G 200MHz 210401-1230 :-

COG Ram Address		Name	Comment
Dec	Hex	(from source code)	
0	0	reset vector	
1	1	regpointer	points to user variables
2	2	sck	SPI Pins
3	3	mosi	
4	4	miso	
5	5	ss=spi_cs	
6	6	clkdly	delay for SPI
7	7	sddly	
8	8	sdhl	
9	9	sclpin	I2C Pins
10	A	sdapin	
11	B	i2cdly	delay for I2C
12	C	txpin	serial
13	D	pinreg	for PIN operation
14	E	u / ACC	registers
15	F	xx / x	
16	10	R0 / yy / y	
17	11	R1 / zz	
18	12	R2	
19	13	R3	
20	14	R4	
21	15	auxptr	L stack Pointer (LUT)
22	16	lpptr	Loop Stack (LUT)
23	17	retptr	Return Stack (LUT)
24	18	depth	Parameter Stack Depth
25	19	lap1	timing
26	1A	lap2	timing
27	1B	seed	random?
28	1C	deltaR	delay count
29	1D	tflgs	flags
30	1E	rxlong	' last 4 characters for KEY@'
31	1F	rxwrC	' RX write index - updated by ISR'
32	20	rxrdC	' RX read index - updated by READRX

33	21	rxdat	' temp rx isr data'
34	22	rxptr	' temp rx isr buffer ptr'
35	23	a	' tos'
36	24	b	' 2nd'
37	25	c	' 3rd'
38	26	d	' 4th'
39	27	loop index	' loop index I '
40	28	loop limit	' limit of DO loop '
41	29	loopip	' branch back address'
42	2A	ffffffff M1 long -1	' -1 or $FFFF_FFFF ' - constant
43 .. 458	2B .. 1CA	Cog Kernel Code	
459 .. 489	1CB .. 1E9	COGMOD space	
490	1EA	dbtab	for TRACING
491	1EB	hr2	for TRACING
492	1EC	hr1	for TRACING
493	1ED	hr0	for TRACING
494	1EE	xreg	for TRACING
495	1EF	traceL	for TRACING
496	1F0	$1F0 RAM / IJMP3	interrupt call address for INT3
497	1F1	$1F1 RAM / IRET3	interrupt return address for INT3
498	1F2	$1F2 RAM / IJMP2	interrupt call address for INT2
499	1F3	$1F3 RAM / IRET2	interrupt return address for INT2
500	1F4	$1F4 RAM / IJMP1	interrupt call address for INT1
501	1F5	$1F5 RAM / IRET1	interrupt return address for INT1
502	1F6	$1F6 RAM / PA CALLD-imm return	CALLPA parameter, or LOC address
503	1F7	$1F7 RAM / PB CALLD-imm return,	CALLPB parameter, or LOC address
504	1F8	$1F8 PTRA	pointer A to hub RAM
505	1F9	$1F9 PTRB	pointer B to hub RAM
506	1FA	$1FA DIRA	output enables for P31..P0
507	1FB	$1FB DIRB	output enables for P63..P32
508	1FC	$1FC OUTA	output states for P31..P0
509	1FD	$1FD OUTB	output states for P63..P32
510	1FE	$1FE INA	* input states for P31..P0
511	1FF	$1FF INB	** input states for P63..P32

* also debug interrupt call address
** also debug interrupt return address

Other Links

- MEDIA.FTH located in folder TAQOZ/Forth in the Tachyon Forth Files makes quite a lot of use of inline code and is worth looking at for further examples.
- There is a TAQOZ Interactive Assembler thread in the Parallax forum. Be warned, early examples of code won't run without editing, as the TIA syntax evolved.
- My CREATE DOES> inline code included in this SI5351 driver
- My Locks and WAITATN inline code

Conclusion

The above has been a quick introduction to creating inline assembly words using TIA in TAQOZ Reloaded. The paper was written as reminder notes whilst the author was getting familiar with the subject. This will be revised from time to time, to jot down handy techniques as more is learned.

Bob Edwards, retired EMC engineer in SW U.K., ham radio call G4BBY April 2022

Appendix 1 CREATE ... DOES>

These two words allow the user to expand the TAQOZ compiler – often useful around assembly language:-

```
 --- CREATE ... DOES> FOR TAQOZ V2.8 VER 2
{
CREATE / DOES> is the pearl of the Forth programming language, enabling the definition of new 'defining' words,
thus extending the compiler to suit the application - a very powerful feature. Here we test the new words out with a new data type,
WARRAY, a single dimension word array
}

--- create new dev with dummy cfa (save ptr to it)
pub CREATE( -- )
	[C] GRAB
	[C] CREATE:		--- Using the next word in the input stream as the name, create a VARIABLE type dictionary entry
	[C] GRAB		--- make sure CREATE: has run before anything more
	HERE 2- 0 REG W! --- save the address of the code after DOES> in the REG scratchpad area
;

--- set new cfa to point back to DOES: code (skipped by DOES: itself)
pub DOES>  ( -- )
	R>			--- the first word location in the new word being defined
	0 REG W@		--- retrieve the address stored on scratchpad
	W!			--- set the first word to execute as the address of the code after DOES>
;
```

```
--- example definition of a new 'array of words' data type - no bounds checking
pre WARRAY
   CREATE ( cnt -- )
       FOR
          0 [C] ||
      NEXT                    --- Create cnt bytes set to 0
   DOES> ( index -- addr )
      2* R> +                 --- the address of the first byte + index = the entry reqd
;

--- Create a new array MYARRAY1 which can hold 10 word sized values
10 WARRAY MYARRAY1

--- now lets check the array addresses are formed correctly
0 MYARRAY1 .
1 MYARRAY1 .
2 MYARRAY1 .
3 MYARRAY1 .

--- now check write and read data works
: TEST1  ---
10 FOR I I MYARRAY1 W! NEXT ---
10 FOR I MYARRAY1 W@ . SPACE NEXT ---
;
```

Appendix 2 - Cog Synchronisation

```
--- Extension to Synchronising Execution between COGS ver1 for Taqoz Reloaded v2.8 - Bob Edwards May 2021

--- Pause execution waiting for an ATN flag from another cog
--- The cog waits for the ATN flag for up to 'clocks' ticks.
--- If ATN occurs before timeout, 'flag' = 1, else if timedout, 'flag' = 0
code WAITATN                        ( clocks -- flag )
            getct xx                ' read the bottom half of the 64 bit system counter
            add a,xx
            setq a                  ' timeout set for when the system counter = a
            waitatn wc              ' wait for atn flag
_ret_       wrnc a                  ' a = carry flag
end

--- Just loop until an ATN flag is received, using POLLATN
pub SLAVE1      ( -- )
      BEGIN
            POLLATN
            IF
                  ." Slave1 received ATN, thanks!" CRLF
            ELSE
                  250 ms
                  ." Slave1, no ATN seen this time" CRLF
            THEN
      AGAIN
;
```

```
pub SLAVE2          ( timeout -- )
        BEGIN
                200000000 WAITATN
                IF
                        ." Slave2 received ATN, thanks!" CRLF
                ELSE
                        ." Slave2 timed out!" CRLF
                THEN
        AGAIN
;


--- Output a message to show the MASTER looping. Set cog 5 AND 6
 ATN flag on each pass
: MASTER            ( -- )
        BEGIN
                200 ms
                ." Hello from the Master, wake up cog 5 & 6
                " CRLF
                %1100000 COGATN                         --- Send ATN to cog 5 and 6
                KEY
        UNTIL
;
```

```
--- set master and both slaves going
pub DEMO    ( -- )
        %ERSCN %HOME
        %BOLD ." Press any key to stop" %PLAIN CRLF
        ' SLAVE2 5 RUN
        ' SLAVE1 6 RUN
        MASTER
        5 s
        5 COGSTOP
        6 COGSTOP
        %BOLD ." The Slave loops were synchronised to ATN flags from the Master, until it was stopped" CRLF
        ." after which, they free-ran because they were no longer receiving those flags" %PLAIN CRLF
;
```

Appendix 3 - Patch for SETEDG and POLLEDG

```
--- SETEDG and POLLEDG use setse1, but this is already in use by the terminal input, so these two words don't work properly
--- this patch redefines the two words to use setse2 and pollse2  - Bob Edwards May 2021

--- useful edge definitions
1 := rising
2 := falling
3 := changing

--- sets event for 'edge' = rising, falling, changing, on SmartPin 'pin'
--- original SETEDG used se1 which is already used in the serial port
code SETEDG                ( edge pin -- )
        shl    b,#6
        add    a,b
        setse2  a
        2DROP;
end
--- e.g. use as 'rising 6 SETEDG' etc

--- redefinition of POLLEDG - polls for the SETEDG event
--- flag = TRUE if event occurred, else flag = FALSE
code POLLEDG      ( -- flag )
                >PUSHX
                pollse2 wc
                wrnc a
        _ret_   sub a,#1
end
```

Appendix 4 - Locks

--- LOCKS v1 for Taqoz Reloaded v2.8 - Bob Edwards May 2021

--- If two or more cogs are writing to the same data in hub memory, that data would be in jeopardy from race conditions as each cog performs
--- its read-modify-write cycle. The outcome of two cogs writing to the same address at very nearly the same time is unknown - which of the
--- two values ends up at the address?

--- To fix that, the P2 has a pool of 16 semaphore bits called locks....

--- Allocate a lock, returning the lock number n. If n = 0-15, lock was allocated, if n=-1 then all locks were already allocated

```
code LOCKNEW ( -- n )
                >PUSHX              'make space on the stack
                locknew a wc        't.o.s. new lock allocation number
        if_nc   ret                 'successful allocation
        _ret_   mov a,#-1           'else signal all locks taken
end
```

--- Return lock number n (0-15) to the pool

```
code LOCKRET ( n -- )
                lockret a
                DROP;
                ret
end
```

--- test LOCKNEW & LOCKRET

```
CRLF 17 FOR LOCKNEW . CRLF NEXT                          --- try allocating one too many locks
13 LOCKRET CRLF ." Did we release lock 13? - " LOCKNEW   --- check we can release a lock
CRLF 16 FOR I LOCKRET NEXT                               --- return all locks
```

```
--- Attempt to 'take' Lock n, flag = 0 if successful, else flag = 1
code LOCKTRY                                ( n -- flag )
        locktry a wc
        _ret_   wrnc a                      ' a = carry flag
end

--- Release Lock n (0-15) - only the cog that took the lock is permitted to do this
code LOCKREL( n -- )
        lockrel a
        DROP;
end

--- Read lock n status, lock_status 1 = unlocked, 0 = locked - N.B. if lock is not owned, results invalid
code LOCK?                                  ( n -- lock_owner lock_status  )
        >PUSHX                              ' make room for status
        lockrel b wc                        ' t.o.s.+1   = cog no. whose lock it is
_ret_   wrnc a                              ' t.o.s.     = lock status
end

long LOCKNUM1
long LOCKNUM2

pub SLAVE1 ( -- )
        BEGIN
                BEGIN
                        LOCKNUM1 @ LOCKTRY
                        30 ms
                0= UNTIL
                ." Slave acquired lock" CRLF
                500 ms
                LOCKNUM1 @ LOCKREL
                ." Slave released lock" CRLF
                500 ms
        AGAIN ;
```

```
pub SLAVE2     ( -- )
        BEGIN
                BEGIN
                        LOCKNUM2 @ LOCKTRY
                        30 ms
                0= UNTIL
                500 ms
                LOCKNUM2 @ LOCKREL
                500 ms
        AGAIN
;

pub SLAVE3 ( -- )
        BEGIN
                BEGIN
                        LOCKNUM1 @ LOCKTRY
                        30 ms
                0= UNTIL
                750 ms
                LOCKNUM1 @ LOCKREL
                750 ms
        AGAIN
;

pub MASTER ( -- )
        BEGIN
                BEGIN
                        LOCKNUM1 @ LOCKTRY
                        30 ms
                0= UNTIL
                ." Master acquired lock" CRLF
                1000 ms
                LOCKNUM1 @ LOCKREL
                ." Master released lock" CRLF
```

```
            1000 ms
        KEY UNTIL ;pub .LOCKS           ( -- )
        OFF %CURSOR
        BEGIN
            %ERSCN %HOME
            %BOLD ." Press any key to stop scanning..." %PLAIN CRLF
            16 FOR
                ." Lock number " I . SPACE
                I LOCK? SWAP
                ." owned by cog #" . SPACE
                ." status is " . CRLF
            NEXT
            20 ms
        KEY UNTIL
        ON %CURSOR
;


--- Demo the use of LOCK? to monitor Lock status in the P2
pub DEMO1       ( -- )
        LOCKNEW
        LOCKNUM1 !
        LOCKNEW
        LOCKNUM2 !
        ' SLAVE2 5 RUN
        ' SLAVE3 6 RUN
        .LOCKS
        5 COGSTOP
        6 COGSTOP
        %BOLD ." So Cog 5 and 6 were monitored, by means of the LOCK? word, taking and releasing two locks" %PLAIN CRLF
        LOCKNUM1 @ LOCKRET
        LOCKNUM2 @ LOCKRET
;
```

```
--- Demo two cogs using a lock - each waits to take the lock in turn, then releases it
pub DEMO2      ( -- )
        LOCKNEW
        LOCKNUM1 !
        %ERSCN %HOME
        %BOLD ." Press any key to stop the Master - 5s later the Slave will stop too" %PLAIN CRLF CRLF
        ' SLAVE1 5 RUN %ERLINE
        MASTER
        %BOLD ." Master stopped " CRLF %PLAIN
        5 s
        5 COGSTOP
        %BOLD ." Slave also stopped" CRLF
        ." Any code sequence protected by the lock could only have been run BY ONE COG AT A TIME" CRLF
        ." so any read-modify-writes would run undamaged by the other cog. Enables race-free data" CRLF
        ." or orderly sharing of a subsystem - e.g. the console port" CRLF
        %PLAIN
        LOCKNUM1 @ LOCKRET
;
```

Appendix 5 - Output to a Pin - high level code versus assembly language

With the cpu clock set to 200MHz this is high level code to toggle pin 10 high and low :-

```
: TEST BEGIN 10 HIGH 10 LOW AGAIN ;
```

Pin 10 goes high for 485nS and low for 675nS.

The equivalent in assembly language might be:-

```
code CTEST
        mov a,#10       ' pin 10
.l1     drvh a
        drvl a
        drvh a
        drvl a          ' output two +ve pulses
        jmp #l1
        ret
end
```

Pin 10 goes high for ~10nS and low for the same time. The jump however takes around 120nS, so quite slow by comparison with the 'linear' instructions.